Complete Guide to Enterprise Container Security

Version 1.0
Released: February 20, 2018

Securosis, L.L.C. 515 E. Carefree Highway Suite #766 Phoenix, AZ 85085 **T** 602-412-3051
info@securosis.com www.securosis.com

Author's Note

The content in this report was developed independently of any sponsors. It is based on material originally posted on the Securosis blog, but has been enhanced, reviewed, and professionally edited.

Special thanks to Chris Pepper for editing and content support.

This report is licensed by Aqua Security.

Aqua Security enables enterprises to secure their container and cloud-native applications from development to production, accelerating application deployment and bridging the gap between DevOps and IT security. Aqua's Container Security Platform provides full visibility into container activity, allowing organizations to detect and prevent suspicious activity and attacks in real time. Integrated with container lifecycle and orchestration tools, the Aqua platform provides transparent, automated security while helping to enforce policy and simplify regulatory compliance. Aqua was founded in 2015 and is backed by Lightspeed Venture Partners, Microsoft Ventures, TLV Partners, and IT security leaders, and is based in Israel and Boston, MA. For more information, visit www.aquasec.com or follow us on twitter.com/AquaSecTeam.

Copyright

Assembling a Container Security Program

Table of Contents

The Need for Container Security

The explosive growth of containers is not surprising — technologies such as Docker alleviate several problems for developers deploying applications. Developers need simple packaging, rapid deployment, reduced environmental dependencies, support for micro-services, generalized management, and horizontal scalability — all of which containers help provide. When a single technology enables us to address several technical problems at once, it's very compelling. But this generic model of packaged services, where the environment is designed to treat each container as a "unit of service", sharply reduces transparency and audit-ability (by design), and gives security pros nightmares. We run more code and faster, but must accept a loss of visibility inside the container. It begs the question, "How can we introduce security without losing the benefits of containers?"

Containers scare the hell out of security pros because they are so opaque. The burden of securing containers falls across Development, Operations, and Security teams — but none of these groups always knows how to tackle their issues. Security and development teams may not even be fully aware of the security problems they face, as security is typically ignorant of the tools and technologies developers use, and developers don't always know what risks to look for. Container security extends beyond containers to the entire build, deployment, and runtime environments.

And the container security space has changed substantially since our initial research 18-20 months back. Security of the orchestration manager has become a primary concern, as organization rely more heavily on the eco-systems to deploy and manage applications at scale. We have seen a sharp increase in adoption of container services (PaaS) from various cloud vendors, which changes how organizations need to approach security. We reached forward a bit in our first container security paper, covering build pipeline security issues because we felt that was a hugely under-served area, but over the last 18 months DevOps practitioners have taken note, and this has become the top question we get. The rapid change of pace in this market means it's time for a refresh.

We get a ton of calls from people moving towards — or actively engaged in — DevOps, so we will target this research at both security practitioners and developers & IT operations. We will cover some reasons containers and container orchestration managers create new security concerns, as well as how to go about creating security controls across the entire spectrum os OS's, containers, orchestration managers, registries, build tools and so on. We will not go into great detail on how to secure apps in general here — instead we will focus on build, container management, deployment, platform, and runtime security which arise with containers.

Threats to Containers

To better understand which container security areas you should focus on, and why we recommend particular controls, it helps to understand which threats need to be addressed and which areas containers affect most. Some threats and issues are well-known, some are purely lab proofs of concept, and others are threat vectors which attackers have yet to exploit — typically because there is so much low-hanging fruit elsewhere.

So what are the primary threats to container environments?

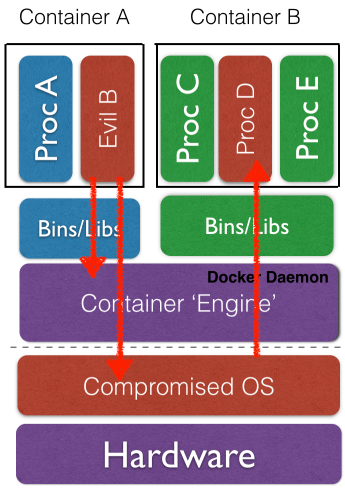
Threats to the Build Environment

The first area which needs protection is the build environment. It's not first on most people's lists for container security, but I start here because it is typically the least secure, and the easiest place to insert malicious code. Developers tend to loathe security in development because it slows them down. That is why there is an entire industry dedicated to test data management and data asking: because developers tend to end-run around security whenever it slows their build and testing processes.

What kinds of threats are we talking about, specifically? Things like malicious or moronic source code changes. Malicious or mistaken alterations to automated build controllers. Configuration scripts with errors, or which expose credentials. The addition of insecure libraries or down-rev/insecure versions of existing code. We want to know whether runtime code has been scanned for vulnerabilities. And we worry about failures to audit all the above and catch any errors.

Runtime Behavior

What the hell is in the container? What does it do? Is that even the correct version? These are common questions from operations folks. They have no idea. Nor do they know whether developers included tools like `ssh` in a container so they can alter its contents on the fly. Just as troubling is the difficulty of mapping access rights to OS and host resources by a container, which can break operational security and open up the entire stack to various attacks. Security folks are typically unaware of what — if any — container hardening may have been performed. You want to know each container's contents have been patched, vetted, hardened, and registered prior to deployment. If a container have open ports for administration, these interfaces may be compromised, and used to attack the container engine, the host OS, or other



containers.

Operating System Security

The underlying operating system's security is a concern. The key question is whether it is configured correctly to restrict each container's access to the subset of resources it needs, and to effectively block everything else. Customers worry that a container will attack the underlying host OS or the container engine. They worry that the container engine may not sufficiently shield the underlying OS. If an attack on the host platform succeeds it's pretty much game over for that cluster of containers, and may give malicious code sufficient access to pivot and attack other systems.

Orchestration Manager Security

A key reason to update and reissue this report is this change in the container landscape, where focus has shifted to orchestration managers which control containers. It sounds odd, but as containers have become a commodity unit of application delivery, organizations have begun to feel they understand containers, and attention has shifted to container management. Attention and innovation have shifted to focus on cluster orchestration, with Kubernetes the poster child for optimizing value and use of containers. But most of the tools are incredibly complex. And like many software products, the focus of orchestration tools is scalability and ease of management — not security. As you probably suspected, orchestration tools bring a whole new set of security issues and vulnerabilities. Insecure default configurations, as well as permission escalation and code injection vulnerabilities, are common. What's more, most organizations issue certificates, identity tokens and keys *from the orchestration manager* as containers are launched.

Securing the Build Pipeline

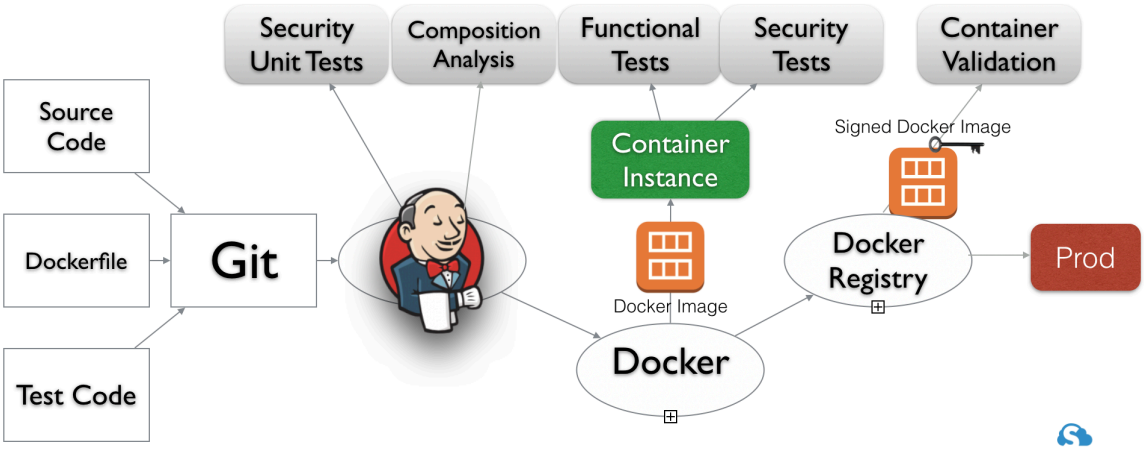
Most people fail to consider the build environment when thinking about container security, but it is critical. With more and more of out infrastructure defined as code, targeting code development is a playground for attackers. The build environment is traditionally the domain of developers, who don't share much detail with outsiders (meaning security teams). But with Continuous Integration (CI) or full Continuous Deployment (CD), we're shooting new code into production, potentially several times a day. An easy way for an attacker to hack an application is get into its development or build environment — usually far less secure than production — and alter code or add new code to containers. The risk is aggravated by DevOps rapidly breaking down barriers between groups, and operations and security teams given access so they can contribute to the process. Collaboration demands a more complex and distributed working environment, with more stakeholders. Better controls are needed to restrict who can alter the build environment and update code, and an audit process to validate who did what.

It's also prudent to keep in mind the reasons developers find containers so attractive, lest you try to adopt security controls which limit their usefulness. First, a container simplifies building and packaging application code — abstracting the app from its physical environment — so developers can worry about the application rather than its supporting systems. Second, the container model promotes lightweight services — breaking large applications down into small pieces, easing modification and scaling... especially in cloud and virtual environments. Finally, a very practical benefit is that container startup is nearly instant, allowing agile scaling up and down in response to demand. It is important to keep these feature in mind when considering security controls, because any control that reduces one of these core advantages is likely to be rejected or ignored.

Build pipeline security breaks down into two basic areas. The first is application security: essentially testing your code and its container to ensure it conforms to security and operational practices. This includes tools such as static analysis, dynamic analysis, composition analysis, scanners built into the IDE, and tools which monitor runtime behavior. We will cover these topics in the next section. The second area of concern is the tools used to build and deploy applications — including source code control, build tools, the build controller, container registries, container management facilities, and runtime access control. At Securosis we often call this the "management plane", as these interfaces — whether API or GUI — are used to set access policies, automate behaviors, and audit activity. Let's dive into build tool security.

Securing The Build

The problem is *conceptually* simple, but there are many tools used for building software, and most have several plug-ins which alter how data flows, so environments can get complicated. You can call this Secure Software Delivery, Software Supply Chain Management, or Build Server Security — take your pick, because these terms are equivalent for our purpose. Our goal is to shed light on the tools and processes developers use to build application, so you can better gauge the threats, as well as security measures to secure these systems.



Following is a list of recommendations for securing platforms in the build environment to ensure secure container construction. We include tools from Docker and others to automate and orchestrate source code, building, the Docker engine, and the repository. For each tool you select some combination of identity management, roles, platform segregation, secure storage of sensitive data, network encryption, and event logging.

- **Source Code Control:** Stash, Git, GitHub, and several variants are common. Source code control has a wide audience because it is now common for Security, Operations, and Quality Assurance to all contribute code, tests, and configuration data. Distributed access means all traffic should run over SSL or VPN connections. User roles and access levels are essential for controlling who can do what, but we recommend requiring token-based or certificate-based authentication, or **two-factor authentication at a minimum**, for all administrative access. This is good housekeeping whether you are using containers or not, but containers' lack of transparency, coupled with automated processes pushing them into production, amplifies the need to protect the build.

- **Build Tools and Controllers:** The vast majority of development teams we speak with use build controllers like Bamboo and Jenkins, with these platforms becoming an essential part of their automated build processes. They provide many pre-, post-, and intra-build options, and can link to a myriad of other facilities. This is great for integration flexibility but can complicate security. We suggest full network segregation of the build controller system(s), and locking network connections to limit what can communicate with them. If you can

deploy build servers as on-demand containers without administrative access to ensure standardization of the build environment and consistency of new containers. Limit access to the build controllers as tightly as possible, and leverage built-in features to restrict capabilities when developers need access. We also suggest locking down configuration and control data to prevent tampering with build controller behavior. Keep any sensitive data, including `ssh` keys, API access keys, database credentials, and the like in a secure database or data repository (such as a key manager, encrypted `.dmg` file, or vault) and pulling credentials on demand to ensure sensitive data never sits on-disk unprotected. Finally, enable the build controller's built-in logging facilities or logging add-ons, and stream output to a secure location for auditing.

- **Container Platform Security:** Whether you use Docker or another tool to compose and run containers, your container manager is a powerful tool which controls what applications run. As with build controllers like Jenkins, you'll want to limit access to specific container administrator accounts. Limit network access to only build controller systems. Make sure Docker client access is segregated between development, test, and production, to limit who and which services can launch containers in production.

- **Container Registry Security:** We need to discuss container registries, because developers and IT teams often make the same two mistakes. The first is to allow anyone to add containers to the registry, regardless of whether they have been vetted. In such an environment it's all too easy to insert an insecure container into production. It's common for developers to leverage open source tools and platforms to speed development, some of which are available as pre-built containers. But it's also common for attackers to create 'pre-pwned' containers loaded with malware, which probe and attack their host and other accessible containers. You'll want to ensure that your registry only accepts containers from trusted sources, and not just whatever some developer grabbed on a whim. Ensuring containers are vetted, and that only containers signed with a trusted key can be launched, helps protect from these problems. You'll also want to ensure that your registries and clients require IAM credentials, to limit who can control the build or add images, and that images loaded into production musts come from your approved registry.

Securing Container Contents

Testing the code and supplementary components which will execute within containers, and verifying that everything conforms to security and operational practices, is core to any container security effort. One of the major advances over the last year or so is the introduction of security features for the software supply chain, from container packaging and runtime environments including Docker, CoreOS's Rocket, OpenShift and so on. We also see a number of third-party vendors helping to validate container content, both before and after deployment. Each solution focuses on slightly different threats to container construction — Docker, for example, offers tools to certify that a container has gone through your process without alteration, using digital signatures and container repositories. Third-party tools focus on security benefits outside what runtime environment providers offer, such as examining commonly used open-source libraries for known flaws. So while things like process controls, digital signing services to verify chain of custody, and creation of a bill of materials based on known trusted libraries are all important, you'll need more than what is packaged with your base container management platform. You will want to consider third-party to help harden your container inputs, analyze resource usage, analyze static code, analyze library composition, and check for known malware signatures. In a nutshell, you need to look for risks which won't be caught by your base platform.

Container Validation and Security Testing

- **Runtime User Credentials:** We could go into great detail here about user IDs, namespace views, and resource allocation; but instead we'll focus on the most important thing: don't run container processes as `root`, because that would provide attackers too-easy access to the underlying kernel and a direct path to attack other containers and the Docker engine itself. We recommend using specific user ID mappings with restricted permissions for each class of container. We understand roles and permissions change over time, which requires ongoing work to keep kernel views up to date, but user segregation offers a failsafe to limit access to OS resources and virtualization features underlying the container engine.

- **Security Unit Tests:** Unit tests are a great way to run focused test cases against specific modules of code — typically created as your development teams find security and other bugs — without needing to build the entire product every time. They cover things such as XSS and SQLi testing of known attacks against test systems. As the body of tests grows over time it provides an expanding regression testbed to ensure that vulnerabilities do not

creep back in. During our research we were surprised to learn that many teams run unit security tests from Jenkins. Even though most are moving to micro-services, fully supported by containers, they find it easier to run these tests earlier in the cycle. We recommend unit tests somewhere in the build process to help validate the code in containers is secure.

- **Code Analysis:** A number of third-party products perform automated binary and white box testing, rejecting builds when critical issues are discovered. We also see several new tools available as plug-ins to common Integrated Development Environments (IDE), where code is checked for security issues prior to check-in. We recommend you implement some form of code scanning to verify the code you build into containers is secure. Many newer tools offer full RESTful API integration within the software delivery pipeline. These tests usually take a bit longer to run but still fit within a CI/CD deployment framework.

- **Composition Analysis:** Another useful security technique is to check libraries and supporting code against the CVE (Common Vulnerabilities and Exposures) database to determine whether you are using vulnerable code. Docker and a number of third parties — including some open source distributions — provide tools for checking common libraries against the CVE database, and can be integrated into your build pipeline. This is important as containers are built in layers, including many open source libraries and tools along with proprietary application code. Developers are not typically security experts, and new vulnerabilities are discovered in common open-source libraries every week, so an independent checker to validate components of your container stack is both simple and essential.

- **Hardening:** Over and above making sure what you use is free of known vulnerabilities, there are other tricks for securing containers before deployment. This type of hardening is similar to OS hardening, which will we discuss in the next section; removal of libraries and unneeded packages reduces attack surface. There are several ways to check for unused items in a container, and you can then work with the development team to verify and remove unneeded items. A 'lean base image' approach to only keep the bare minimum of what is needed for the application to run, which you can enforce via manual reviews of through build scripts, helps reduce attack surface. Another hardening technique is to check for hard-coded passwords, keys, and other sensitive items in the container — these breadcrumbs makes things easy for developers, but help attackers even more. Some firms use manual scanning for this, while others leverage security tools to automate it.

- **Container Signing and Chain of Custody:** How do you know where a container came from? Did it complete your build process? These techniques address "image to container drift": addition of unwanted or unauthorized items. You want to ensure your entire process was followed, and that nowhere along the way did a well-intentioned developer subvert your process with untested code. You can accomplish this by creating a cryptographic digest of all image contents, and then track it though your container lifecycle to ensure that no unapproved images run in your environment. Digests and digital fingerprints help you detect

code changes and identify where each container came from. Some container management platforms offer tools to digitally fingerprint code at each phase of the development process, alongside tools to validate the signature chain. But these capabilities are seldom used, and platforms such as Docker may only optionally produce signatures. While all code should be checked prior to being placed into a registry or container library, signing images and code modules happens during building. You will need to create specific keys for each phase of the build, sign code snippets on test completion but before code is sent on to the next step in the process, and (most important) keep these keys secured so attackers cannot create their own trusted code signatures. This offers some assurance that your vetting process proceeded as intended.

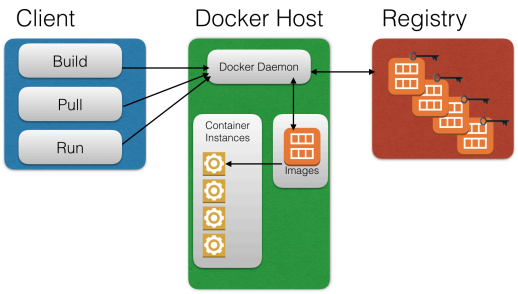
Container Runtime Security

After the focus on tools and processes in previous sections, we can now focus on containers in production systems. This includes which images are moved into production registries, selecting and running containers, and the security of underlying host systems.

Runtime Security

- **The Control Plane:** Our first order of business is ensuring the security of the control plane: tools for managing host operating systems, the scheduler, the container client, engine(s), the registires, and any additional deployment tools. As we advised for container build environment security, we recommend limiting access to specific administrative accounts: one with responsibility for operating and orchestrating containers, and another for system administration (including patching and configuration management). On-premise we recommend network and physical segregation, and for cloud and virtual systems we prefer logical segregation. The good news is that several third-party tools offer full identity and access management, LDAP/AD integration, and token-based SSO (*i.e.:* SAML) across systems.
- **Resource Usage Analysis:** Many readers are familiar with this for performance, but it can also offer insight into basic code security. Does the container allow port 22 (SSH for administration) access? Does the container try to update itself? What external systems and utilities does it depend upon? Any external resource usage is a potential attack point for attackers, so it's good hygiene to limit ingress and egress points. To manage the scope of what containers can access, third-party tools can monitor runtime access to environment resources — both inside and outside the container. Usage analysis is basically automated review of resource requirements. This is useful in a number of ways — especially for firms moving from a monolithic architecture to micro-services. Analysis can help developers understand which references they can remove from their code, and help operations narrow down roles and access privileges.
- **Selecting the Right Image:** We recommend establishing a trusted image repository and ensuring that your production environment can only pull containers from that trusted source. *Ad hoc* container management makes it entirely too easy for engineers to bypass security controls, so we recommend establishing trusted central repositories for production images. We also recommend scripting deployment to avoid manual intervention, and to ensure the latest certified container is always selected. This means checking application signatures in

your scripts before putting containers into production, avoiding manual verification overhead or delay. Trusted repository and registry services can help by rejecting containers which are not properly signed. Fortunately many options are available, so pick one you like. Keep in mind that if you build many containers each day, a manual process will quickly break down. It is okay to have more than one image repository — if you are running across multiple cloud environments there are advantages to leveraging the native registry in each one.



- **Immutable Images:** Developers often leave shell access to container images so they can log into containers running in production. Their motivation is often debugging and on-the-fly code changes, both bad for consistency and security. Immutable containers — which do not allow `ssh` connections — prevent interactive real-time manipulation. In the general case this forces developers to fix code in the development pipeline, and removes a principal attack path. And it ensures you do not lose track of changes or lose version control over containers in production. But from a specific threat perspective, attackers routinely scan for `ssh` access to take over containers, and leverage them to attack underlying hosts and other containers. We strongly suggest use of immutable containers without 'port 22' access, and making sure that all container changes take place (with logging) in the build process, rather than in production.

- **Time To Live:** How long ago did you build this container image? How long has this container been running? In case of container compromise a very practical question is: how many containers are currently running this software bundle? We recommend that you do not allow containers to ‘live’ for weeks or months, as it is entirely possible that a new vulnerability has been discovered since one of them was approved and launched. It is now common for firms to set a maximum ‘time to live’ for containers in production, limiting them to a couple hours or perhaps a couple days — at most — before they are replaced. As a practical matter, one image may instantiate into a thousand running containers. To address problems you need to update the image and replace running containers using the old image. You want to get into the habit of regularly replacing running containers as a simple methodology regularly patching and replacing to keep containers in synch with current versions.

- **Input Validation:** At startup containers accept parameters, configuration files, credentials, JSON, and scripts. In more aggressive scenarios 'agile' teams shove new code segments into containers as input variables, making existing containers behave in fun new ways. Validate that all input data is suitable and complies with policy, either manually or using a third-party security tool. You must ensure that each container receives the correct user and

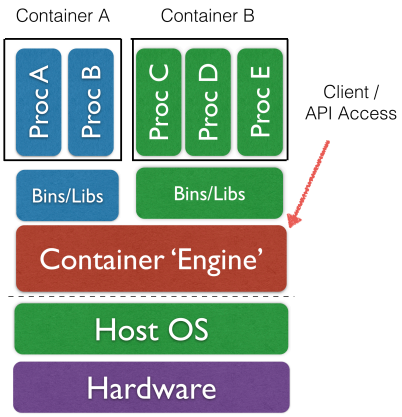
group IDs to map to the assigned view at the host layer. This can prevent someone from forcing a container to misbehave, or simply prevent dumb developer mistakes.

- **Blast Radius:** The cloud enables you to run different containers under different cloud user accounts, limiting the resources available to any given container. If an account or container set is compromised, the same cloud service restrictions which prevent tenants from interfering with each other will limit damage between your different accounts and projects. For more information see our reference material on [limiting blast radius with user accounts] (https://securosis.com/blog/cloud-security-best-practice-limit-blast-radius-with-multiple-accounts).
- **Container Group Segmentation:** One of the principal benefits of container management systems is help scaling tasks across pools of shared servers. Each management platform offers a modular architecture, with scaling performed on node/minion/slave sub-groups, which in turn include a set of containers. Each node forms its own logical subnet, limiting network access between sets of containers. This segregation limits 'blast radius' by restricting which resources any container can access. It is up to application architects and security teams to leverage this construct to improve security. You can enforce this with network policies on the container manager service, or network security controls provided by your cloud vendor. Over and above this orchestration manager feature, third-party container security tools — whether running as an agent inside containers, or as part of underlying operation systems — can provide a type of logical network segmentation which further limits network connections between groups of containers. All together this offers fine-grained isolation of containers and container groups from each another.

Platform Security

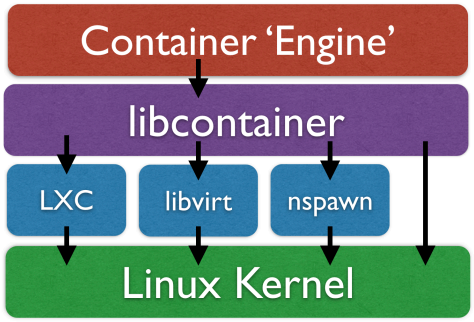
Until recently, when someone talked about container security, they were really talking about how to secure the hypervisor and underlying operating system. So most articles and presentations on container security focuses on this single — admittedly important — facet. But we believe runtime security needs to encompass more than that, and we break the challenge into three areas: host OS hardening, isolation of namespaces, and segregation of workloads by trust level.

- **Host OS/Kernel Hardening:** Hardening is how we protect a host operating system from attacks and misuse. It typically starts with selection of a hardened variant of the operating system you will use. But while these versions come with secure variants of both libraries and features, you will still have work to leverage your baseline configuration and remove unneeded features. There are good benchmarks out



there, such as the CIS Docker Benchmark, which provides a list of checks to give you a good baseline for the do's and do-nots around hardening. At minimum you'll want to ensure user authentication and access roles are set, that permissions for binary file access are properly set, logging of audit data is enabled, and the base OS bundle is fully patched. Review patch and configuration status of the virtualization libraries (such as `libcontainer`, `libvirt`, and LXC) your container engine relies on to protect itself.

- **Resource Isolation and Allocation:** A critical element of container security is limiting container access to underlying operating system resources, particularly to prevent a container from snooping on — or stealing — data from other containers. The first step is making sure container privileges are assigned to a role. The container engine must run at the host operating system's `root` user, but your containers must not, so set up user roles for your container groups. Next up is the resource isolation model for containers, which is built on two concepts: `cgroups` and namespaces. A namespace creates a virtual map of the resources any given task will be provided. It maps specific users and groups to subsets of resources (*e.g.:*, networks, files, IPC, etc.) within their namespace. We recommend default deny on inbound requests, and only allow containers with a legitimate need to communicate on open network channels. And you not mix container and non-container services on a single machine. You will create specific user IDs for containers and/or group IDs for different classes of containers, then assign IDs to containers at runtime. A container is then limited in how much of a resource it is allocated by a Control Group (*i.e.:* `cgroup`). The `cgroup` provides a mechanism to partition tasks into hierarchical groups, and control how much of any particular resource (such as memory or CPU cycles) a task can use. This helps protect one group of containers from being starved of resources by another.



- **Segregate Workloads:** We discussed resource isolation at the kernel level, but you should also isolate container engine/OS groups and their containers at the network layer. For container isolation we recommend mapping groups of mutually trusted containers to separate machines and/or network security groups. For containers running critical services or management tools, consider running a limited number of containers per VM and grouping them by trust level/workload or into a dedicated cloud VPC, to limit attack surface and minimize an attacker's ability to pivot in case of service or container compromise. When using orchestration managers, at the very least, ensure that critical apps run in their own node, or in extreme cases, their own cluster. As we mentioned above under Container Group Segregation, orchestration manager features and third-party products can help with segmentation. For on-premise or 'private cloud' you can consider one container per VM or

physical server for on-premise applications, but this sacrifices some benefits of using containers on virtual infrastructure.

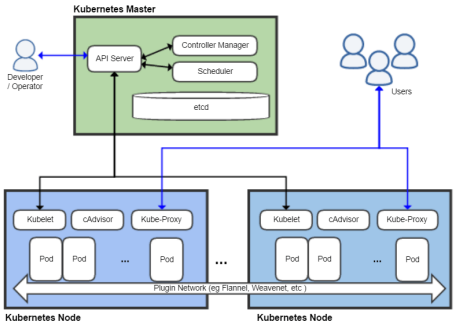
Platform security and container isolation are both huge fields of study, and we have only scratched the surface. Operating system providers, Docker, and third-party security providers offer best practices, research papers, and blogs with great detail, often detailing issues with specific operating systems.

Orchestration Manager Security

This research effort is focused on container security, but any discussion of container security now must address securing containers *within* a specific orchestration management framework. There are many orchestration managers in active use: Kubernetes, Mesos, Swarm; as well as cloud-native container management systems offered as part of AWS, Azure, and GCP. And each offers a unique, and complex, security challenge. Kubernetes is the dominant tool for managing clusters of containers, and with its rapid surge in popularity came many additional concerns for container security programs — both thanks to added environmental complexity and because its default security can generously be described as 'poor'. There are public demonstrations of how to gain unauthorized `root` access on nodes; escalate privileges; bypass identity checks; and exfiltrate code, keys, and credentials. Our point is that many container managers need considerable tuning to be secure.

We have already discussed security aspects such as OS hardening, image safety, namespaces, and network isolation to limit potential 'blast radius'. And we addressed hardening container code, trusted image repositories to keep administrators from accidentally running malicious containers, and immutable container images to prevent direct shell access. Now we can address specific orchestration manager areas you should secure.

- **Management Plane Security:** Cluster management — whether you're using Swarm, Kubernetes, or Apache Mesos/ Marathon — will be handle via command line and APIs. For example the `etcd` key-value store and `kubectl` controller are fundamental to managing a Kubernetes cluster, but these tools can be misused a variety of ways by an attacker. In fact the graphical user interface on some platforms does not require user authentication, so disabling them is a common best practice. You'll want to limit who can access administrative features, but developers or attackers can import their own command-line tools, so simple access controls are insufficient. Network isolation can help protect the master management server and control where administrative



commands can run. Use network isolation, leverage more recent IAM services built into your cluster manager (RBAC for Kubernetes), and set up least privilege for node service accounts.

- **Segregate Workloads:** We have already discussed namespaces and network isolation, and we have mentioned the need to keep different types of containers segregated into their own nodes/minions, not just pods. But there are other, more basic controls which should be in place. With both on-premise Kubernetes and cloud container deployments, we often find a flat network architecture. We also find developers and QA personnel have direct access to production accounts and servers. We strongly recommend segregating development and production resources in general, but particularly within production orchestration systems, to segregate partition sensitive workloads to their own nodes or even cluster instances. Additionally, set network security policies ("security groups" in AWS) to 'default deny' inbound connections as a good starting point, and only add specific exceptions as needed by applications. This is the default network policy for most cloud services, because it reduces attack surface effectively. Default deny also reduces the likelihood of containers automatically updating themselves from external sources, and can prevent attackers from uploading new attack tools should they gain a foothold in your environment.

- **Limit Discovery:** Cluster management tools collect a broad assortment of metadata on cluster configuration, containers, and nodes. This data is essential for cluster management, but can also offer a map to attackers probing your system. Limiting which services and users can access metadata, and ensuring requesting parties are fully authorized, helps reduce attack surface. Many platforms offer metadata proxies to filter and validate requests.

- **Upgrade and Patch:** The engineers behind most container managers have responded well to known security issues, so newer versions tend to be much more secure. Virtualization is key to any container cluster, and these platforms build redundancy in, so you can leverage cluster management features to quickly patch and replace both cluster services and containers.

- **Logging:** We recommend collecting logs from all containers and nodes. Many attacks focus on privilege escalation and obtaining certificates, so we also recommend monitoring all identity modification API calls and failures, to highlight attacks.

- **Test Yourself**: Security checkers and CIS security benchmarks are available for containers and container orchestration managers, which you can use to get an idea of how well your baseline security stacks up. These provide a good initial step for validating cluster security. Unfortunately default container manager configurations tend to be insecure, and most administrators are not fully aware of all the features of any given cluster manager, so these checkers are a great way to get up to speed on appropriate security controls. You can also engage pen testers to provide an idea where poor configurations, insecure plug-ins or poor identity management controls may have left you vulnerable.

Keep in mind that these are very basic recommendations — we cannot do this topic justice within the scope of this paper. That said, we really want to raise reader awareness of proven attacks on all existing open source container management systems, and the considerable amount of work needed to secure a cluster.

Secrets Management

When you start up a container or orchestration manager it needs permissions to communicate with other containers, nodes, databases, or network resources. In a highly modular service-oriented architecture a container without credentials can't make API calls, leverage data encryption, establish identity with other containers or really get much real work done. But we *don't* want engineers hard-coding secrets into containers, nor do we want secrets sitting in files on servers. Provisioning ephemeral machine identities — on-demand and at scale — is tricky: we need to securely make secrets available to containers when they need them, but expose only the 'right' secrets to the containers that are supposed to get them.

The new products to address this issue are called 'Secrets Management' platforms. And as you might imagine this is not purely a container issue, but a general IT security issue that has grown exponentially with the use of cloud services, containers and orchestration frameworks like Kubernetes and Swarm. For those reasons, secrets management products securely store a wide range of secrets including encryption keys, API certificates, identity tokens, SSL certificates, and passwords. They can share secrets across groups of trusted services and users, leveraging existing directory services to determine who has access to what.

Solutions are widely available, including commercial tools, and many orchestration managers and container ecosystem providers (most notably Docker) offer secrets management built-in. But before you run off and try them out, some words of caution. Most built-in secrets management tools have some serious pitfalls, with some forcing specific technology choices on you, and others fail to get even basic basic security right. Either of which may be a non-starter for you. There are general use secrets management platforms do not have these short-comings, but may be more difficult to integrate, may not meet the speed and scalability needs, and still others simply do not 'do' provisioning of machine identities. It's going to take a little research to find a solution that gives you container-level support for the platforms you use.

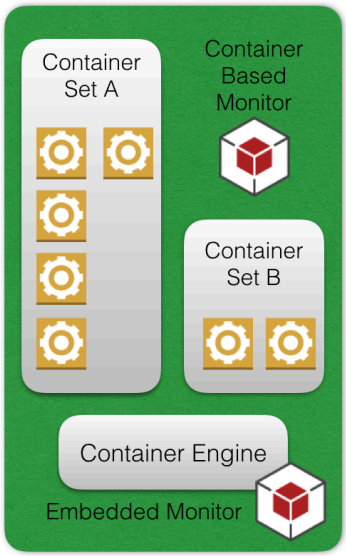
We cannot fully address this complex topic within this paper either, so if you'd like more information, please see our paper on Secrets Management.

Monitoring and Auditing

We close out this research paper with two key areas to highlight: Monitoring and Auditing. We want to draw additional attention to these items because they are essential to security programs, but with sporadic coverage on security blogs and in the press. Additionally, most Development and Security teams are not aware of the variety of monitoring options, and we have seen many misconceptions and downright fear about auditing.

Monitoring

Every security control we have discussed so far has to do with preventative security. Essentially these are security efforts that remove vulnerabilities, or make it hard from anyone to exploit them. We address known attack vectors with well-understood responses such as patching, secure configuration, and encryption. But vulnerability scans can only take you so far. What about issues you are not expecting? What if a new attack variant gets by your security controls, or a trusted employee makes a mistake? This is where monitoring comes in: it's how you discover the unexpected stuff. Monitoring is critical to a security program — it's how you learn what is effective, track what's really happening in your environment, and detect what's broken.



For container security it is no less important, but today it's not something you get from Docker or any other container provider.

Monitoring tools work by first collecting events, and then examining them in relation to security policies. The events may be requests for hardware resources, IP-based communication, API requests to other services, or sharing information with other containers. Policy types are varied. We may have deterministic policies, such as which users and groups can terminate resources, which containers are disallowed from making external HTTP requests, or what services a container is allowed to run. Or we may have dynamic — also called 'behavioral' — policies, which prevent issues such as containers calling undocumented ports, using 50% more memory resources than typical, or uncharacteristically exceeding runtime parameter thresholds. Combining deterministic white and black list policies with dynamic behavior detection provides the best of both worlds, enabling you to detect both simple policy violations and unexpected variations from the ordinary.

We strongly recommend that your security program include monitoring container activity. Today, a couple container security vendors offer monitoring products. Popular evaluation criteria for differentiating products and determining suitability include:

- **Deployment Model:** How does the product collect events? What events and API calls can it collect for inspection? Typically these products use either of two models for deployment: an agent embedded in the host OS, or a fully privileged container-based monitor running in the Docker environment. How difficult is it to deploy collectors? Do the host-based agents require a host reboot to deploy or update? You will need to assess what type of events can be captured.

- **Policy Management:** You will need to evaluate how easy it is to build new policies — or modify existing ones — within the tool. You will want to see a standard set of security policies from the vendor to help speed up deployment, but over the lifetime of the product, you will stand up and manage your own policies, so ease of management is key to your long-term happiness.

- **Behavioral Analysis:** What, if any, behavioral analysis capabilities are available? How flexible are they, meaning what types of data can be used in policy decisions? Behavioral analysis requires starting with system monitoring to determine 'normal' behavior. The pre-built criteria for detecting aberrations are often limited to a few sets of indicators, such as user ID or IP address, but more advanced tools offer a dozen or more choices. The more you have available — such as system calls, network ports, resource usage, image ID, and inbound and outbound connectivity — the more flexible your controls can be.

- **Activity Blocking:** Does the vendor provide the capability to block requests or activity? It is useful to block policy violations and in order to ensure containers behave as intended. Some provide 'blunt force blocking', which means they drop application requests or even pausing a container to ensure a request is not processed. Others allow for granular control level policies on what actions are to be blocked, or even modify behavior when these conditions are encountered. Care is required, as blocking can disrupt application functionality, causing friction between Development and Security, but blocking is invaluable for maintaining Security's control over what containers can do.

- **Platform Support:** You will need to verify your monitoring tool supports the OS platforms you use (CentOS, CoreOS, SUSE, Red Hat, or even Windows) and the orchestration tool (such as Swarm, Kubernetes, Mesos, or ECS) of your choice.

Audit and Compliance

Developers are interested in what happened with the last build. Security teams may want to know if `sshd` was removed from the new container, or if a specific set of security tests were run. IT wants to know the version of the latest build in the repository. Audit and compliance teams are not interested in those questions. They want to know what administrators have access to management functions, or which containers have access to regulated data? How are those containers segregated from other containers? Can you demonstrate your process for addressing common vulnerabilities? To satisfy questions of this type you're going to need operational logs, configuration data and process documents.

During our investigation for this series we did not speak with any firms which did not have a SIEM system, log capture or big data platform like Splunk for event capture in place. They have already created controls and reports to support regulatory and contractual commitments. The challenge is now to map those into new environments like cloud and container orchestration managers where applications now exist as micro-services on dozens of servers that appear and disappear regularly.

The good news is that the vast majority of code repositories, build controllers, and container management systems — specifically the Docker runtime and Docker Trusted Registry — produce event logs, in formats which can be consumed by various log management and Security Information and Event Management (SIEM) systems without modification. Many third-party security tools for image validation and behavioral monitoring do as well. Additionally, most auditors are well versed in the requirements for Payment Card Industries Data Security Standard (PCI-DSS), or the data privacy provisions to the Gramm-Leach-Bliley Act (GLBA) so they understand the requirements to be met.

The bad news is mapping the events and — more importantly — data flows to existing requirements takes some work. Using IP addresses, application event logs or cloud services or do not always provide needed reference points when using virtual networks, micro-service architectures, ephemeral servers and externally managed identities. There is some mapping, filtering and — in some cases — enrichment to the logs in order to make use of them. Given the virtual, on-demand nature of these environments you'll need to adjust many reports to reflect the changes in the environment, and you'll leverage monitoring activity at the API/application layer to gain a complete picture of activity.

About the Analyst

Adrian Lane, Analyst and CTO

Adrian Lane is a Senior Security Strategist with 25 years of industry experience. He brings over a decade of C-level executive expertise to the Securosis team. Mr. Lane specializes in secure application development, database and data security. With extensive experience as a member of the vendor community (including positions at Ingres, Unisys and Oracle), in addition to time as an IT customer in the CIO role, Adrian brings a business-oriented perspective to security implementations. Prior to joining Securosis, Adrian was CTO at database security firm IPLocks, Vice President of Engineering at Touchpoint, and CTO of the secure payment and digital rights management firm Transactor/Brodia. Adrian also blogs for Dark Reading and is a regular contributor to Information Security Magazine. Mr. Lane is a Computer Science graduate of the University of California at Berkeley with post-graduate work in operating systems at Stanford University.

If you have any questions on this topic, or want to discuss your situation specifically, feel free to send us a note at info@securosis.com.

About Securosis

Securosis, LLC is an independent research and analysis firm dedicated to thought leadership, objectivity, and transparency. Our analysts have all held executive level positions and are dedicated to providing high-value, pragmatic advisory services. Our services include:

- **Primary research publishing**: We currently release the vast majority of our research for free through our blog, and archive it in our Research Library. Most of these research documents can be sponsored for distribution on an annual basis. All published materials and presentations meet our strict objectivity requirements and conform to our Totally Transparent Research policy.
- **Research products and strategic advisory services for end users**: Securosis will be introducing a line of research products and inquiry-based subscription services designed to assist end user organizations in accelerating project and program success. Additional advisory projects are also available, including product selection assistance, technology and architecture strategy, education, security management evaluations, and risk assessment.
- **Retainer services for vendors**: Although we will accept briefings from anyone, some vendors opt for a tighter, ongoing relationship. We offer a number of flexible retainer packages. Services available as part of a retainer package include market and product analysis and strategy, technology guidance, product evaluation, and merger and acquisition assessment. Even with paid clients, we maintain our strict objectivity and confidentiality requirements. More information on our retainer services (PDF) is available.
- **External speaking and editorial**: Securosis analysts frequently speak at industry events, give online presentations, and write and speak for a variety of publications and media.
- **Other expert services**: Securosis analysts are available for other services as well, including Strategic Advisory Days, Strategy Consulting engagements, and Investor Services. These tend to be customized to meet a client's particular requirements.

Our clients range from stealth startups to some of the best known technology vendors and end users. Clients include large financial institutions, institutional investors, mid-sized enterprises, and major security vendors.

Additionally, Securosis partners with security testing labs to provide unique product evaluations that combine in-depth technical analysis with high-level product, architecture, and market analysis. For more information about Securosis, visit our website: <http://securosis.com/>.