Security in Quick-Deploy Software Schemes

Ian Altgilbers

December 12, 2014

Abstract

Virtual appliances, containers, recipes and bundled installers are common shortcuts to delivering application stacks and associated services. These approaches often do most of the work for you, leaving the administrator minimal location-specific configuration. However, such shortcuts present tradeoffs in security. I take a survey of the options in this field (Virtual Appliances, Docker, Vagrant, etc.) to weigh the pros and cons of each. Next I will look at the security issues related to these containers, whether they be baked in rootkits with malicious intent, dangerously open default settings that leave you open to attack, or other default settings that could make you an unwitting attacker. In the end I hope to raise awareness of the security issues inherent to using this type of software and to provide some ways to minimize the potential threats.

Contents

1 Introduction

Technology in modern software development, especially web application development, advances at a blistering pace. A hot, new, must-use language, framework, application, or service hits the shelves every week. As a developer, integrator, or systems engineer you are pressured to adopt these new technologies as quickly as possible to keep from falling behind. This need to keep up has led to the rise a variety of quick deploy solutions for getting applications and their building blocks up and running in short order. The shortcuts employed to simplify the process are not free... They often come at the expense of security.

2 To the Community

I chose this topic to research because it is something that I sometime deal with professionally. The department I work for is often tasked with standing up proof-of-concept systems connecting existing technologies in (hopefully) interesting ways. We often do not have the time or resources to give the systems and applications the security audits they probably deserve. Outside my little world, I see a broader trend toward using ”shortcut schemes” to commoditize the underlying systems. While I appreciate the efficiencies these approaches can yield, I worry that people and organizations use them blindly and do not fully realize the level of trust they are putting in the systems and processes they use.

3 Trust

Any time you use software written, bundled, distributed, or configured by someone else, you are trusting them and the processes they use. You are trusting that they are honest and are not going to be doing anything malicious or negligent. This trust is transitive. If party A trusts party B and party B trusts party C, then party A implicitly trusts party C. This chain of trust can extend countless levels through numerous businesses, committees, consortia, and individuals. Transitive trust is a great time saver, so long as you can be comfortable trusting everyone trusted by your ”trustees.”

While trust is something to be careful with, it is something that cannot be practically avoided. There is no way to ”roll your own” everything from the ground up and maintain any reasonable level of productivity. Even building all software components from source that you have read and analyzed involves trusting your compiler. Ken Thompson wrote an article in the early 80’s demonstrating that you can’t really trust anything you didn’t write yourself. He cleverly outlines an exploit of baking in a trojan horse into a compiler. From there, even if you rebuilt the compiler from source, your new compiler would still have the exploit baked in [1]. The only way to recover would be to rewrite your compiler in assembly, bypassing any possibly contaminated compiler. Even then you are trusting that the hardware you are using faithfully executes the instructions issued. There are there are plenty of ways your hardware can work against you. Unless you mine your own silicon and copper and build your computer from scratch, you are trusting someone.

Since trust is unavoidable, we need to do our best to build and extend it. Even if you explicitly trust a party, you still need to do some verification to know you’re actually getting what they’re providing. This can be done with checksums, signed checksums, signed code, etc. The PKI system is one way we can leverage an existing trust network.

4 Best Practices

4.1 Least Privilege

Most of the security best practices are related to trust. The Principle of Least Privilege, attributed to Jerome Saltzer, is one of the key tenets of computer security. ”Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.”[2] While it may be convenient to use a single MySQL account for administration and application purposes, limiting the access granted to an application user limits the damage a bad actor could do. Similarly, running application processes as the system’s root user can simplify file permissions problems, but it can also escalate an application vulnerability into a system vulnerability.

4.2 Isolation

Isolation is closely related to the principle of least privilege. It is the concept of putting barriers between different components, at various levels of the technology stack, to keep one component from affecting another component in an unintended fashion. Isolation can happen at many different levels. Systems can be isolated at the network level by firewalls and private networks. Applications can be isolated from each other by running in separate systems.

5 Types of deployments

There are many ways any given application can be deployed, depending on the requirements to be met and the resources which are available. Most of these tools and methods and be slotted into one of the following categories.

5.1 Physical Appliances

Some vendors provide physical appliances which live in your datacenter and are often black boxes, often requiring the bare minimum site-specific configuration. These devices sometimes run proprietary software, where the vendor alone has knowledge. These devices were more popular ten years ago, before virtual machines supplanted them. With a physical appliance, you are fully trusting the vendor to have secured the product. All you may be able to do is isolate it appropriately on your network to protect against it being vector to attack other parts of your infrastructure.

5.2 Virtual Appliances

With the spread of virtualization platforms, many software providers began offering pre-packaged virtual machines with their product preconfigured and ready to go. On VMWare's Virtual Appliance Marketplace alone there are almost 1800 prepackaged virtual machines.[3] With minimal configuration and minimal physical resources, an application can be brought online. Virtual appliances bring many of the same risks as physical along with a different set of concerns arising from the virtual environment.

5.3 Bundled Installers

Here we have an application that is distributed with all of its dependencies. MAMP, Atlassian Confluence, and Internet2's Grouper are examples of products in this category. Often a particular version of a bundled dependencies is included to simplify compatibility issues. Problems arise when a vulnerability is found in one of the bundled components. Depending on the particulars of the deployment, patching one of the dependencies may be difficult or impossible, leaving you at the mercy of the bundler to release a patch or updated version.

5.4 Recipes

A step up from bundled installers are the recipe based solutions. Products like Puppet, Chef, and Ansible use a collection of "recipes" which programmatically define software dependencies and configurations. These recipes can be modularized for easier maintenance and included in larger recipes. They are an extension to the idea of a kickstart script for system builds. Some applications are distributed via these recipes as an alternative to an installer. An advantage to recipe based deployment is that is readable/auditable in advance. You can know what you are getting.

5.4.1 Vagrant

Vagrant is product in this class that goes a step further by allowing users to precompute many of the dependencies and configurations, distilling them into virtual machines bundles called "boxes". Boxes are used in conjunction with vagrantfiles to bring an application online. The vagrantfile can do about anything, from installing packages to configuring system parameters.[4] The time savings here is in using one of the hundreds of base images that others have built and shared on vagrantcloud.com or vagrantbox.es. The boxes available in the vagrantcloud.com repository are somewhat curated, but many are just created and shared by ostensibly well meaning members of the community. The vagrantbox.es repository seems to be a little less managed, with the actual box files being served from all corners of the internet.

In using boxes to simplify a deployment, you give back part of the advantage of readability/discoverability of the recipe. Often Vagrant box publishers will document their build process, but you're trusting that those documents are updated and faithfully reflect the state of the box. If you have to resort to using other system auditing tools and procedures to validate the box, what are you really saving?

5.5 Containers

One of the hottest technologies these days is Docker. In many ways Docker is similar to other recipe based deployments like Vagrant, however it builds on a fundamentally different foundation. Docker uses LXC linux containers instead of discrete VMs. This strategy can provide a number of advantages. First, there is a performance advantage, since multiple containers can share one kernel, reducing overhead. Docker also provides a different level of isolation, since each container is responsible for fulfilling its own dependencies. Containers use a couple mechanisms for isolation that make them rival VMs in isolation.

5.5.1 Namespaces

First are namespaces. The idea behind namespaces is to encapsulate a specific system-wide resource such that makes it appear that processes in each namespace have full access to the global resource. There are several different types of namespaces for the different types of resources.[5]

1. Mount namespaces isolate processes at the filesystem level. Filesytems mounted inside a container are not visible to other containers or the host. Similarly, host file systems are not visible to processes in the container, unless explicitly allowed.

2. UTS namespaces allow a container to have its own identity. Different containers can be identified with their own specific hostnames and domains.

3. IPC namespaces separate interprocess communication, like POSIX message queues, semaphores and signals.

4. PID namespaces isolate the process tree. Each container gets its own unique process ids, including its own init process.

5. Network namespaces allow each container to have its own IP address(es), routing tables, DNS servers, etc. This allows for conveniences like having applications in multiple containers all bound to the same port (i.e. 80).

6. User namespaces give a container its own perspective on user and group ids, separate from the host system. This allows for process in a container to run will full root privileges while being unprivileged beyond the container.

It also allows for a container to have users and groups that conflict with users and groups in the host system.

5.5.2 Control Groups

Similar to how hypervisors ration resources to hosted virtual machines, resources available to containers can be controlled with control groups (cgroups). Cgroups are a resource metering mechanism built in to the linux kernel that apply to groups of processes. It is like rlimit extended to a group of processes. Cgroups can limit the number of CPUs available and the number of shares it has, similar to how VMWare handles resource contention. They can also apportion disk IO, network IO and memory usage[6]. While these limits may seem like they are just performance related, they can be used to prevent one container from monopolizing resources and DOSing other containers via resource starvation.

5.5.3 Container Linking

Segregating application components into their own containers is a big plus for security, but components need to interact with each other and the outside world to actually deliver a service. This could be done by binding to ports on the network and communicating as if you were talking to any other service. Fortunately, Docker provides a more scoped and slightly more automated way of handling these interactions. Container linking allows containers on the same host to communicate securely with each other without opening up access to any other systems (or containers). Linking goes a step further by injecting information about the source container into the environment variables and /etc/hosts file of the recipient container. This allows for programmatic discovery and autoconfiguration of the software on the receiving side of the link.[7] For example, a web application container could link to a database container and the administrator would not have to know anything about the actual IPs and port numbers in use.

5.5.4 Vulnerabilites

One potential risk when using containers arises from the fact that all containers on a particular host share a kernel. If there happens to be a vulnerability in the kernel, a process in a container could potentially exploit it to break out of the container. However, the fact that containers are abstracted away from the kernel means that patching the kernel for the host fixes it for all containers on the host, whereas with virtual machines you'd have to patch the kernel in each individual VM against the vulnerability.

Another risk in using Docker is the docker daemon. This daemon is the control center for all the containers running on a host and is somewhat analogous to the hypervisor in the realm of virtual machines. Any security issue here can be leveraged to exploit all containers on the host. To guard against this a container host should run as few services as possible alongside the docker daemon to reduce the attack surface. [8][9]

5.5.5 Dockerfiles

The heart of Docker's success is the registry and the hundreds of repositories and thousands of dockerfiles.[10] Like vagrantfiles or any of the other recipe based systems, Docker uses a script (dockerfile) to define and configure what is in a container. The dockerfiles build off existing dockerfiles and layer on new packages and configurations. This modularity allows for some amazing efficiencies in building containers, but transitive trust raises its head again. There are dockerfiles published that build on dockerfiles down several levels. If you use them blindly, you're trusting everyone in the that chain. Docker has an advantage over Vagrant here, because you can trace back through the chain to audit everything that is going into a container.

5.6 Platform as a Service

Another quick-deploy avenue takes automation near the limits. Platform as a Service (PaaS) providers, like Heroku, provide the whole application environment and run the code you specify. Here you are taking trust to the extreme and assuming that the service provider is doing all the right things on the backend to protect you from vulnerabilities in their infrastructure and dependency stack.

6 Types of Vulnerabilities

Each deployment strategy has its own combination of weaknesses and vulnerabilities. Some of these apply almost universally, while others are more applicable to particular tools. You always have to be concerned about backdoors and rootkits being baked into what you are deploying, especially when you use tools that leverage pre-built VMs. Using components/dependencies with known vulnerabilities can be just as dangerous. Quick deploys solutions also often forgo some common hardening like firewalling and scoped binding (i.e. only binding to interface you intend to use, instead of binding to 0.0.0.0). One of the biggest issues with such systems is the default configurations. From username and passwords, to database names, to TCP port numbers used, using known defaults opens you up to automated attacks and attacks from less sophisticated attackers.

7 False Sense of Security

Many feel that open, community sourced deployments provide a safe haven against many attackers, since ostensibly, a large community of experts has vetted the software's source code and configurations. In recent months a couple of high profile vulnerabilities have surfaced in software that is very widely distributed that should give us pause in blindly trusting even the most foundational software.. Both HeartBleed and ShellShock were catastrophic failures in a software that millions of people use and have access to source.

8 Mitigation

There are limitless ways for things to go wrong with systems built around shortcuts, but you can still use these systems effectively if you follow a few guidelines.

1. Know your sources. Anyone can publish software these days, from the established kernel guru to the script kiddies looking to make a mark. There over 3 million user accounts and over 7 millions repos on github. Sourceforge has also millions of users and hundreds of thousands of projects. It is up to you to know who writes and maintains the software you are using.

2. Verify your sources. Use checksums and cryptographic signatures to make sure you're getting the payload your requested. Using https sources gives you another level of assurance that you are downloading software from the legitimate repository.

3. Know your tool chain. The tools you use are the glue that binds all of building blocks together into a useable application/service. The better you know the tools, the better off you will be a spotting inconsistencies and possible problems.

4. Isolate at an appropriate level. All systems can be secure if they are unplugged and stored in a vault, but to be of value an application needs to be accessible. Use the isolation features on your platform of choice to allow only that communication which is needed to render the service.

5. Least privilege is your friend. Use the tools at your disposal to restrict unnecessary access. Whether it be file level permission, database grants, SSH access, or commit rights to a source repo, loose permissions can be used by attackers to quickly escalate a minor application flaw into a full system com-

promise and even onto an enterprise level disaster.

9 Conclusion

Security is and always will be the process of managing risk. To effectively manage risk you must know the risk. In the realm of quick-deploy, the goal is to get something running as fast as possible with as little local configuration as possible is at odds with the goal of deploying secure systems, since the shortcuts used can lead to trusting countless unknown parties. The key is to recognize and be conscious of the shortcuts you are taking and weigh the potential risk incurred against the time/complexity saved. You will never eliminate all risks, but you can sleep well at night knowing you've done your best to understand them.

References

[1] Ken Thompson, Reflections on Trust,
http://cm.bell-labs.com/who/ken/trust.html

[2] The Protection of Information in Computer Systems,
http://web.mit.edu/Saltzer/www/publications/protection/

[3] Virtual Appliances — Solution Exchange,
https://solutionexchange.vmware.com/store/category_groups/virtual-appliances

[4] Vagrantfile - Vagrant Documentation https://docs.vagrantup.com/v2/vagrantfile/index.html

[5] Namespaces in operation, part 1: namespaces overview,
http://lwn.net/Articles/531114/

[6] PaaS Under the Hood, Episode 2: cgroups,
http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-24-c

[7] Linking containers together - Docker Documentation,
https://docs.docker.com/userguide/dockerlinks

[8] Docker Security,
https://docs.docker.com/articles/security/

[9] Containers & Docker: How Secure Are They?,
http://blog.docker.com/2013/08/containers-docker-how-secure-are-they/

[10] Docker Registry Search,
https://registry.hub.docker.com

[11] http://www.vagrantbox.es

[12] http://www.vagrantcloud.com