RC25651 (WAT1702-031) February 15, 2017
Computer Science

IBM Research Report

Understanding Security Implications of Using Containers in the Cloud

Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila,
Shripad Nadgowda, James Doran
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

Understanding Security Implications of Using Containers in the Cloud

Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, James Doran
IBM TJ Watson Research Center, Yorktown Heights, NY USA

Abstract

Container technology is being adopted as mainstream platform for IT solutions because of high degree of agility, reusability and portability it offers. It is well-suited for achieving faster release cycle through DevOps model. However, there are challenges to address for successful adoption. First, it is difficult to establish the full pedigree of images downloaded from public registries. Some might have vulnerabilities introduced unintentionally through rounds of updates by different users. Second, non-conformance to immutable software deployment policies, such as those promoted by DevOps principles, introduces vulnerabilities and loss of control over deployed software. In this study, we investigate containers deployed in a production cloud to derive a set of recommended approaches to address these challenges. Our analysis reveals evidence that (i), images of unresolved pedigree have introduced vulnerabilities to containers belonging to third parties; (ii), updates to live public containers are common, defying the tenet that deployed software is immutable; and (iii), scanning containers and images alone is insufficient to eradicate vulnerabilities from public containers. We advocate for better systems support for tracking image provenance and resolving undesired changes to containers, and propose practices that container users should adopt to limit the vulnerability of their containers.

1 Introduction

Containers are expanding their adoption in the IT arena rapidly as evidenced by recent launches of IBM Bluemix [9], Amazon ECS [1], Azure Container Service [4] and Google Container Engine [6]. Reasons are plentiful. The motto of '*Build, Ship and Run*', easy reusability of images, easy distribution of code, and simplicity of *pick and run* are all attractive because they align well with the agility, portability, visibility goals of modern software development and DevOps principles [14]. Ultimately the goal is to shorten release cycles, and thus time-to-market, as much as possible.

However, with the mainstream adoption of containers, new challenges emerge. Among them, we focus on the following two that we believe are among the most critical. First, the ease of distribution and reuse of containers make it difficult to fully understand the origin and pedigree of images we use. Consider this scenario where two benign development actions can lead to a serious security exposure. A developer builds an image with password-based authentication enabled and pushes it to an image registry. Another developer, unaware of this, pulls this image and builds a database application on top, where the database application adds a default user ID and password during its installation. This new image is now pushed back to the registry. As a result of these independent actions we end up with an image with a high-risk security exposure that is pulled and deployed by several unsuspecting users. Anyone can freely use this image to deploy the same database application in the cloud, and it could be one of yours as well. Unintended vulnerabilities could be introduced this way to an image and can quickly spread in the cloud [8].

A second challenge arises where the expectations from the employed DevOps practices do not match reality with containerized application deployments. Modern DevOps practices advocate an "immutable architecture" model, where all software, system and infrastructure requirements of an application are expressed as code. This gives the developers the ability to re-create the infrastructure and applications in a repeatable and agile way. Containers, with their ability to package all system and software requirements, are a key enabler for this immutable architecture model. However, there is a commonly observed mismatch or "drift" between the declared architecture and the actual deployed application instance in the cloud. This deviation stems from several facts such as in-place updates (e.g., manual configuration change), dynamic configuration and application updates. Such drift can introduce unexpected exposures and side effects on

deployed applications and can go unnoticed for a long time with image-centric validation processes.

In this paper we present real examples to these challenges based on our experiences with an internal,production cloud. We demonstrate an actual case study on image provenance and its implications. We present a detailed data analysis on the extent of the observed drift in cloud, its root causes and mitigation techniques. We demonstrate the value of automated and systematic scanning of container images and live instances to address these challenges, emerging solutions in this space [7, 15, 2, 3, 10], their scope and limitations. We present key insights derived from observing aggregate cloud data on security, provenance and drift. Our analysis shows that drift exists in 5% of our scanned images, it has diverse causes, and in some cases can lead to significant vulnerabilities. Our analysis shows that image-centric security solutions are insufficient, and continuous scanning of images and live containers, coupled with good DevOps practices are required to ensure high level of cloud security.

Overall, our contribution can be summarized as: *(i)* Sharing of our experiences of analyzing the security states of containers and images from a production-level container cloud, *(ii)* Detailed drift analysis to understand to what extent it occurs in the production cloud and what the common characteristics are. Based on the analysis, we describe comprehensive list of causes of drift, *(iii)* Lessons and suggestions of approaches we should take to continue to improve the safety of container cloud.

2 Image & Container Checking

Our security scanning mechanism is fully integrated into a production-level container cloud used internally. It is automatically triggered upon new image pushes and new container launches. It extracts various features such as list of files with attributes, list of installed packages, OS information, docker inspect, and docker history as presented in [11]. It then feeds the extracted features to compliance and package vulnerability checkers, and persists the output of these checkers into store for aggregate and historical analysis.

Compliance Checking: Compliance rules used in our anlysis are based on set of best practices recommended to minimize the chances of compromise. Complete list of rules we use in the scan is described in Table 1. Rules are largely categorized into (i) password restrictions (Rules 2B-D), (ii) file system integrity and (iii) remote access packages (Rules 9A-G). Of particular interest is SSH-related rules - 9A, 9E, 9F and 9G.

Package Vulnerability Checking: Vulnerabilities in software are announced via the National Vulnerability Database (NVD) [12]. Each vulnerability is assigned a unique id known as Common Vulnerability Exposure (CVE) ID [5], and given a score to communicate the impact of the vulnerability. In addition, it also lists specific versions of the products affected by the given vulnerability. Our vulnerability checker uses above information to determine vulnerability status of images and containers. Container images and running instances are scanned periodically to determine their vulnerabilities status. One of the consequences of repeated scanning is an image that has no vulnerabilities in a given scan may turn out to be vulnerable later.

ID	Rules
1A	Each UID must be used only once.
2B	Maximum password age must be set to 90 days.
2C	Minimum password length must be 8.
2D	Minimum days that must elapse between user-initiated password changes should be 1.
5A,B	RD/WR access of root/.rhosts,.netrc only by root
5D,E	Permission of /usr,/etc must be r-x or more restrictive.
5F	The file /etc/security/opasswd must exist and the permission must be rw——- or more restrictive.
5J	Permission settings of /var for other must be r-x or more restrictive.
5K	Permission of /var/tmp must be rwxrwxrwt.
5L	Permission setting of /var/log for other must be r-x or more restrictive.
5M	Permission check of /var/log/faillog
5N	Permission check of /var/log/tallylog
5S	Permission check of snmpd.conf
6D,E,F	wtmp/faillog/tallylog must file exist.
8O	no_hosts_equiv must be present
9A	SSH server must not have been installed.
9B	Telnet server must not have been installed.
9C	Rsh server must not have been installed.
9D	Ftp server must not have been installed.
9E	SSH server must not be enabled.
9F	SSH password authentication should not be enabled.
9G	All passwords must not be weak.

Table 1: List of Compliance Rules.

3 Image Security: Unsafe Pedigree

The foremost challenge of adopting the container cloud identified earlier is the difficulty with grasping the full history of what updates have been applied to the image to be in current state. This means that the base image you pull may contain unidentified vulnerabilities whether it was crafted or inadvertent. Even worse, multiple series of modifications and re-push by different users, including yours, may jointly create unexpected vulnerabilities. Thus, it is naive to expect that images would stay clean even if users strictly follow best practice guidlines. In this section we make a case for the importance of systematic and automated image scan to deal with such issues. We drive our discussion using one actual scenario we encountered.

Case Study: Recently we have come across a puzzling pattern in one of the analytics data. We were looking at the list of about 50 containers that were classified as 'high-risk' that violated SSH-related rules 9A, 9F and 9G. They all had SSH server running, password-authentication enabled and the an ID with weak pass-

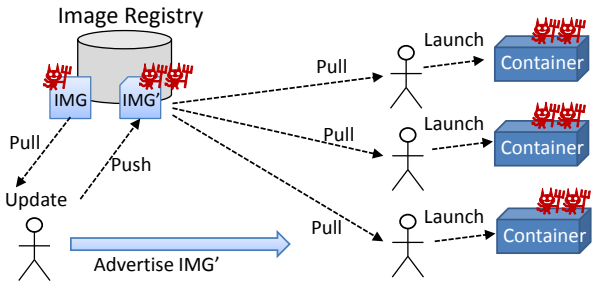

Figure 1: Scenario of Vulnerability Spread

word. What was most peculiar was that the source image names of all of them had common part, say "*myappsrv*" [1], as if they were all created by one owner. But all of them belonged to different users. How can we explain this phenomenon in which all different users launched containers whose images seemed to have derived from the same source at the same time?

To find an answer we started with searching the Docker Hub for the image that contained "*myappsrv*". We found a candidate, but lacked description. The 'docker inspect' output had the postgres start up commands as the entry point. And, several ports (tcp 22, 5432, 7276, 7286, 9080, 9443) were open. List of packages installed in the image also indicated that it was a postgresql database with SSH server. With further investigation we finally learned that this image was used in an online course. Students were instructed to pull, customize and launch a container from it.

Figure 1 illustrates the spread process. The instructor pulls the image that already had a postgres server with a default ID of 'postgres' with weak password. Without knowing the existence of this ID, he installs SSH server packages. This image is pushed to the registry and advertised to all the online students. Students pull it and launch containers of their own, resulting in large number of high-risk containers. The instructor was unaware that the original image had an ID with weak password. Also, when installing the SSH server, the intention was to allow only the key-based authentication. In the config file, this line was commented out.

```
\#PasswordAuthentication yes
```

However, if it is commented out the default behavior of sshd is to enable it. It is easy to be misled to believe that the password authentication is disabled. As a result of all of these, high-risk containers became wide-spread.

Discussion: It is worthwhile making a few points from this case study. First, vulnerabilities can creep in through accumulation of innocuous updates and it is difficult to foresee. Second, we started out with noticing a common pattern in image names across group of vulnerable containers. This exemplifies the advantage of analyzing the

[1] It is not the actual name.

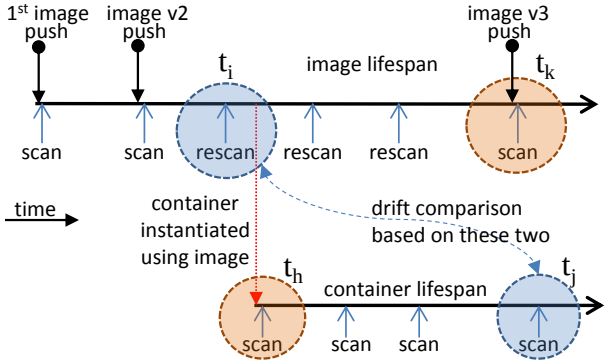

Figure 2: Our definition of drift

aggregate data as a whole which may lead to useful information that eventually leads to the root cause. Based on the observation we make the following statement. *Ensuring the safety of the container cloud should not solely be dependent on users behaving in responsible manner. We must rely on the automated solutions that performs security scans frequently and analyzes the data as a while.*

4 Container Security: Drift Analysis

Here, we analyze the data collected from production-level container cloud, used internally, to understand the drift behavior. The data is collected from two instances of the contained cloud operating independently of each other for about two week period in Oct, 2016.

The questions we are interested in are: Does drift exist? If so, how many containers exhibit the drift? Between the compliance and vulnerability, which is the cause of drift? In compliance, which rules in specific are causing the drift? Does drift always increase, or is there a case where it decreases as well and what are they?

4.1 Definition of Drift

We specifically consider the *drift* in terms of the compliance and vulnerability violations. The drift in compliance is defined as the difference between the number of compliance rules violated in a running container and in its corresponding image. Likewise the drift in vulnerability is defined as the difference of the number of vulnerable packages.

Figure 2 illustrates the time aspect of comparison in determining the drift. The upper horizontal arrow indicates the life span of an image. From the moment it is pushed, it is scanned for the compliance and vulnerability. Push of newer version also triggers a new scan. There are scans not triggered by a push as labeled as 'rescan'. Rescans are to ensure that results are up-to-date with respect to the updated compliance rules and vulnerable package definitions. The lower horizontal arrow represents the lifespan of a container.

We compare the scan result between t_i and t_j to obtain the drift, denoted as $D(t_i,t_j)$. $D(t_k,t_j)$ was dismissed be-

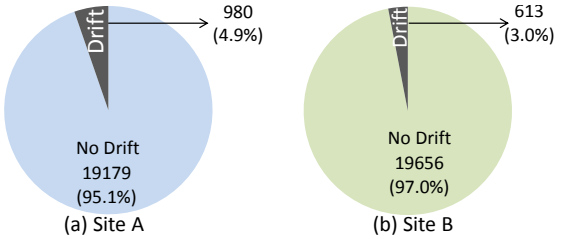

Figure 3: Proportion of containers having drifts.

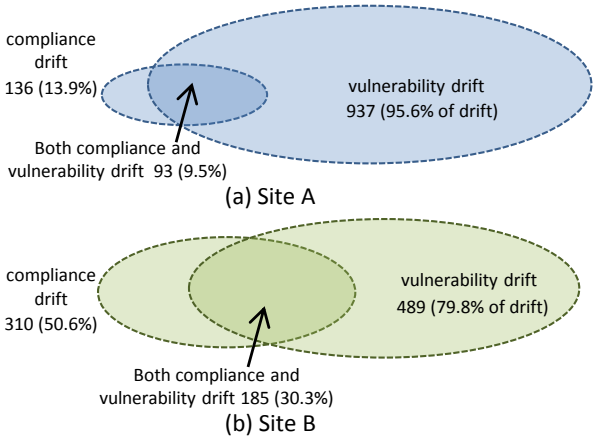

Figure 4: Break-down of drift into compliance and vulnerability.

cause t_k might contain new updates that are independent of the container and, thus, making it difficult to identify true divergence of states. Also, $D(t_h, t_j)$ is inappropriate since it may miss $D(t_i, t_h)$ that occurs at the launch time.

4.2 Analysis and Highlights of Findings

We find that drifts do exist in the production containers and the magnitude is less than 5%. As shown in Figure 3, 4.9% and 3.0% of the containers exhibit drifts in both Site A and B, respectively. The existence of drifts, even as small as 5%, is intriguing because ideally the drift is not expected to occur. One harmless cause of the drifts would be the increased number of vulnerable packages in containers not because they actually increased, but because the list of known vulnerable packages grew over time. This raises a question as to how many drifts fall under such category. Also, the site difference of 1.9% seems to be meaningful to deserve a closer look. To find the explanation, we look at the break-down of drift.

Figure 4 is the Venn diagram of drifts broken down into compliance and vulnerability. The existence of compliance drifts tells us that there are other types of drifts than the ones due to the growing definition of vulnerable packages. What is common for both sites is that vulnerability drifts dominates. However, the proportion of vulnerability vs. compliance drifts shows notable difference. Site A has much smaller ratio of compliance drifts (13.9%) than the Site B (50.6%). This may be the indication that the characteristics of the containers from both

		Site A		Site B	
Vulnerability	Increased	913	93.2%	295	48.1%
	Decreased	24	2.4%	194	31.6%
Compliance	Increased	72	7.3%	223	36.4%
	Unchanged	2	0.2%	13	2.1%
	Decreased	62	6.3%	74	12.1%

Table 2: Drift categorization in terms of the direction of changes. Percentage is based on the total drifts per site.

Site A			Site B		
Rule	Count	Pct	Rule	Count	Pct
9A	34	47.2%	1A	134	60.1%
1A	25	34.7%	2B	95	42.6%
9F	24	33.3%	2C	90	40.4%
2B	18	25.0%	2D	50	22.4%
2C	18	25.0%	9A	19	8.5%
2D	6	8.3%	5L	11	4.9%
9G	4	5.6%	9F	8	3.6%
5S	1	1.4%	5S	1	0.4%
			9G	1	0.4%

Table 3: Compliance rules violated in drift cases. Refer to Table 1 for the description of rule codes.

are intrinsically different in regard to compliance rules. Also, it is interesting that the absolute number of vulnerability drift at Site A is twice as many as that of Site B. Note that it does not necessarily imply that containers at Site B are more secure. This means that the vulnerability status does not change as much irrespective of how secure the images and containers are.

Table 2 provides the break-down of drifts in terms of whether the drift count increases or decreases. One example of a decrease is when the user logs in and manually patches vulnerable packages in the container. According to the Table 2, significant portion (31.6%) of the vulnerability drift at Site B is in the '*Decreased*' category. This contrasts with Site A's number which has only 24 (2.4%). In case of the compliance drift, the proportion of the '*Increased*' category for Site B is much larger than that of Site A. Table 3 explains the cause of the difference. It is because of the high proportion of violations of rule 1A (60.1%) which is twice as large in proportion compared to Site A(34.7%). In addition, password-related rules, 2B-D, rank high in the table for Site B whereas SSH-related rules, 9A and 9F, are towards top of the list for Site A. It is interesting to see that, at Site B, violations of password rules occur more than the violation of SSH rules to the running containers. Similarly, the reverse holds for the Site A. Table 4 also shows the composition of rules that are fixed. We can see that there is a tendency of fixing SSH-related violation within containers at Site B. *Although site differences exist, majority of the drifts are due to the changes of vulnerability status. Also, data shows that 'in-place' updates to the containers, both benign and undesirable, are taking place.*

Focus on SSH rules: In this part we specifically study the drift of SSH related rules among the rules in Table 1. Compliance to the SSH related rules is of particular interest because it is one of the most exploited vulnerabilities. Once compromised, the consequence could be

Site A			Site B		
Rule	Count	Pct	Rule	Count	Pct
2C	28	45.2%	9A	33	44.6%
9G	26	41.9%	9F	32	43.2%
2B	26	41.9%	9G	30	40.5%
9F	19	30.6%	2D	25	33.8%
9A	8	12.9%	2B	12	16.2%
5B	1	1.6%	2C	11	14.9%
			5S	1	1.4%

Table 4: Compliance rules fixed in drift cases. Refer to Table 1 for the description of rule codes.

Category	Site A		Site B	
No SSH, Password become weak	1	1.3%	1	1.2%
SSH installed	31	39.2%	19	23.5%
SSH installed with weak password	3	3.8%		
Password become weak	1	1.3%		
Sum	**36**	**45.6%**	**20**	**24.7%**
Password become strong	21	26.6%	26	32.1%
Password Auth disabled	13	16.5%	2	2.5%
No SSH, password become strong	1	1.3%		
SSH gets removed	8	10.1%	33	40.7%
Sum	**43**	**54.4%**	**61**	**75.3%**
Total	79	100%	81	100%

Table 5: Classification of SSH-related compliance rule drifts.

deadly. But, in many cases this vulnerability is exposed out of neglect, and most of the attacks can be prevented even with small awareness.

Table 5 summarizes the findings related to the SSH rules. It classifies the SSH-related drifts into 8 categories and presents the statistics. Proportion of containers with drifts of SSH rules are about 0.4% for both Site A (79/20K) and Site B (81/20K). The upper half of the table represents drift categories that negatively impacts the SSH vulnerabilities. The lower half shows the drifts that strengthen it. Although magnitude differs, there exist drifts that increases the SSH vulnerabilities in both sites. The highest risk arises when all SSH rules gets violated as a result of these drifts whether it was through manual human actions or automated scripts.

Overall, our study of drifts suggests that the security scanning of images is insufficient to eliminate the vulnerabilities. Since security status changes while containers are running, it is critical that containers be scanned periodically.

4.3 Discussion

Why Does Drift Happen? While industry thinking coalesces around the belief that containers should be immutable [13], our findings have shown that containers deployed in a cloud drift from their original configuration. Drift occurs for several reasons.

• *Update via Remote Shell Access:* Users of Docker containers are able to login into their containers and execute local commands that alter the state of the containers. Containers offer two shell access modalities: native Docker daemon commands (e.g., *exec*, *attach*) and user installed remote shell servers (e.g., SSH login).

• *Automated Software Update:* Owing to a long history of bug and vulnerability discovery in software long after they ship, software often install with default options to automatically install updates as they become available. As developers build container images they often neglect to changing such defaults.

• *Software configured at runtime:* To aid with usability, popular server applications offer Web admin front ends that allow novice and expert users alike to change their configurations long after they have been deployed.

What can we do about drift? Both the systems and container user communities must work together to realize the promise of an immutable infrastructure. Systems must provide better mechanisms to version and track changes and automate detection of drift from desired container state. Container users must also adopt practices that lead to immutability.

• *Systems support:* Disallowing changes altogether on containers is untenable. Applications, even if stateless, often write cache data or logs to the local file system. First we should track all changes made to containers and give users visibility into these changes. Second, systems must recognize benign changes to the container file systems and memory from undesired changes.

• *Best practices:* Users must adopt practices that contribute to immutable infrastructures. The first step is to discontinue bad habits from the time-sharing era of logging in to manually effect changes. DevOps practices require changes to exist as versioned code that is systemically validated before delivery to production environments. Delivery is the replacement of the live container with a new instance containing versioned code.

Some configurations are bound to the application at run time and cannot be built into the container image. One such example is environment specific variables such as the hostname of a service that the container software depends on. For these configurations, developers must rely on configuration management systems that track changes rather than manually feeding the container with arguments in an ad-hoc manner.

5 Conclusion

In this paper, we first established the importance of DevOps as a standard software delivery practice for container-based micro-service architecture. And as an underlying principle DevOps requires security assurance over the pedigree of images along with operational immutability for containers instantiated from these images. To substantiate the extent to which these principles are currently violated, we presented our study on analysis of images and containers in production-level container cloud. We also discussed common characteristics and causes of drifts. Thus, there is an increasing need to have a regulatory protocol and enforcement engine in the platform to curb such non-conformity to ensure security and success of DevOps.

References

[1] AMAZON EC2 CONTAINER SERVICE. https://aws.amazon.com/ecs/.

[2] AMAZON INSPECTOR. https://aws.amazon.com/inspector/.

[3] AQUA. https://www.aquasec.com/.

[4] AZURE CONTAINER SERVICE. https://azure.microsoft.com/en-us/services/container-service/.

[5] COMMON VULNERABILITIES AND EXPOSURES. https://cve.mitre.org/.

[6] CONTAINER ENGINE. https://cloud.google.com/container-engine/.

[7] DOCKER SECURITY SCANNING. https://docs.docker.com/docker-cloud/builds/image-scan/.

[8] GUMMARAJU, JAYANTH AND DESIKAN,TARUN AND TURNER, YOSHIO. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities. https://www.banyanops.com/blog/analyzing-docker-hub/.

[9] IBM BLUEMIX. https://console.ng.bluemix.net/.

[10] IBM VULNERABILITY ADVISOR. https://www.ibm.com/blogs/bluemix/tag/ibm-vulnerability-advisor/.

[11] KOLLER, R., ISCI, C., SUNEJA, S., AND DE LARA, E. Unified monitoring and analytics in the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (Santa Clara, CA, 2015), USENIX Association.

[12] NATIONAL VULNERABILITY DATABASE. https://nvd.nist.gov/.

[13] RAFAEL BENEVIDES. 10 things to avoid in docker containers. https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/.

[14] ROCHE, J. Adopting devops practices in quality assurance. *Commun. ACM 56*, 11 (Nov. 2013), 38–43.

[15] TWISTLOCK. https://www.twistlock.com/.