Building a Virtually Air-gapped Secure Environment in AWS*

with Principles of DevOps Security Program and Secure Software Delivery

Erkang Zheng
LifeOmic, Inc.
Morrisville, North Carolina
erkang.zheng@lifeomic.com

Phil Gates-Idem
LifeOmic, Inc.
Morrisville, North Carolina
phil.gates-idem@lifeomic.com

Matt Lavin
LifeOmic, Inc.
Morrisville, North Carolina
matt.lavin@lifeomic.com

ABSTRACT

This paper presents the development and configuration of a virtually air-gapped cloud environment in AWS, to secure the production software workloads and patient data (ePHI) and to achieve HIPAA compliance.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**; *Trust frameworks*; *Software security engineering*; • **Software and its engineering** → Risk management;

KEYWORDS

cybersecurity, security, trust, DevOps, cloud, AWS, risk management

ACM Reference Format:
Erkang Zheng, Phil Gates-Idem, and Matt Lavin. 2018. Building a Virtually Air-gapped Secure Environment in AWS: with Principles of DevOps Security Program and Secure Software Delivery. In *HoTSoS '18: Hot Topics in the Science of Security: Symposium and Bootcamp, April 10–11, 2018, Raleigh, NC, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3190619.3190642

1 INTRODUCTION

With all the latest attacks on Health IT, especially ransomware, we need a security operating model that requires minimal resources to achieve not only compliance, but more importantly, real security to protect the confidentiality, integrity, availability and privacy of patient data and sensitive personal information.

Cloud is no longer a fringe idea or for just shadow IT. Not only has it been embraced by the vast technology startup community, cloud adoption has become a top priority of many large enterprises. The perspective and consensus around the security of cloud has quickly taken a 180. It has quickly shifted from "hell no" to "cloud can be more secure than your own infrastructure". That's true, but how?

LifeOmic was founded at the end of 2016, building a secure precision medicine platform utilizing cloud technologies. We implemented a "zero trust", data-centric security model with and a virtually air-gapped production environment by harnessing the power of cloud infrastructure, platform and services in Amazon Web Services (AWS).

1.1 Structure of this paper

In this paper, we discuss these four important concepts and implementations:

- First, the principles, assumptions, and assurances of an effective security program for a cloud DevOps organization.
- Second, how LifeOmic built a virtually "air-gapped" environment in AWS to secure its production workload and resources by prohibiting any internal network connectivity or user access.
- Third, how software is securely delivered to this "air-gapped" environment without VPN or SSH access.
- Finally, how to ensure the secure software delivery process is followed using production change management automation.

2 BUILDING AN EFFECTIVE SECURITY PROGRAM, THE RIGHT WAY

The design and implementation of a security program has direct and long-term impact to the effectiveness and efficiency of an organization's security operations. Many security programs are designed by adopting one of more industry frameworks, such as ISO 27001, NIST Cybersecurity Framework (CSF). Being in the healthcare industry, LifeOmic has chosen to align its security program to the Health Information Trust Alliance Cybersecurity Framework (HITRUST CSF) as well as the HIPAA data privacy and security provisions. HITRUST CSF was developed in collaboration with healthcare and security experts and has become the de facto standard for healthcare compliance. The framework covers extensive controls and requirements across 19 security domains.

Additionally, a risk-based approach is taken, following a set of assumptions, assurances, and principles described below.

2.1 Assumptions and Assurances

First and foremost, we must recognize and accept that cyberattacks are inevitable. No individual or organization, no matter how large or small, is immune to cyberattacks. At some point, a malware will infect one of the systems. An attacker will somehow gain unauthorized access to a resource. The recent breach of a Tesla-owned Amazon Web Services (AWS) account hijacked to mine cryptocurrency[1] is a good example and reminder of that fact.

Therefore, we first need to **assume compromise but expose no single point of compromise**. We must limit the attack blast

*copyright 2018 LifeOmic, Inc. and authors

[1] http://fortune.com/2018/02/20/tesla-hack-amazon-cloud-cryptocurrency-mining/

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HoTSoS '18, April 10–11, 2018, Raleigh, NC, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6455-3/18/04.
https://doi.org/10.1145/3190619.3190642

radius such that each attack is contained as tightly as possible. This ensures that the compromise of one system or user does not easily propagate or escalate.

We must also **track everything since you cannot protect what you can't see**. This involves maintaining an always up-to-date view of all users, assets, resources, compute instances, and data repositories whether they are on premise or in the cloud, as well as the real time analysis of events and alerts across the entire environment.

In order to achieve the above, **automation is key because people don't scale and changes are constant**. The amorphous nature of the cloud and speed of DevOps presents unprecedented challenges in security. Security operations will never be able to keep up unless we take a DevSecOps approach to automate as much as possible.

Automation is not everything. We must also **build products that are secure by design and secure by default**. Software development shall conform to the highest standard of security throughout its development lifecycle, from security design and threat modeling, to code scanning and reviews, to secure deployment and penetration testing. A sound secure development process ensures security is built-in, not bolt-on to every component of every software.

Last but not least, we must **engage everyone in security**. It is often said that security is everyone's responsibility. This includes sharing the responsibility and ownership of security with the entire organization, especially the DevOps teams, and leveraging crowd-sourced services such as a public bug bounty program. The level of engagement will be meaningful if security is not driven by rewarding the right behaviors instead of fear and roadblocks.

In a nutshell, a culture shift is required for the security in modern SaaS operations to **favor transparency over obscurity, practicality over process, and usability over complexity**.

3 THE VIRTUALLY AIR-GAPPED ENVIRONMENT

Many attacks start by taking advantage of vulnerabilities in a relatively low risk system – usually an end-user device. This gives attackers "a way in" to the corporate environment. After gaining a foothold, the attacker can now try to escalate privilege and move laterally to compromise other devices on the internal network, until the final targets are reached.

Let's ask ourselves this question:

How do you ensure, at any given time and at all times, that none of the user systems on the internal network are compromised and therefore cannot further infect and impact others on the same network?

The answer is simple – you can't. The probability of someone's system getting compromised at some point is practically guaranteed.

Instead, we must take a data-centric approach to build on a zero-trust security architecture. In this architecture, granular security policies are applied to small segregated environments based on criticality of resources, and to reduce the blast radius to an absolute minimum.

3.1 Zero-trust architecture

"Zero Trust" is a data-centric security design that puts micro-perimeters around specific data or assets so that more granular rules can be enforced. It remedies the deficiencies with perimeter-centric strategies and the legacy devices and technologies used to implement them. It does this by promoting *"never trust, always verify"* as its guiding principle. This differs substantially from conventional security models which operate on the basis of *"trust but verify"*.

In particular, with Zero Trust there is no default trust for any entity – including users, devices, applications, and packets – regardless of what it is and its location on or relative to the corporate network. In addition, verifying that authorized entities are always doing only what they're allowed to do is no longer optional; it's now mandatory.

3.2 Segregated environments meet short-lived processes

We extend the zero-trust security model with a *"Minimal Infrastructure"* approach, where we use *"Anything-as-a-Service"* whenever possible, to harness the full power of the cloud. This includes the use of **Amazon Web Services (AWS)** as our main infrastructure as well as other cloud services such as **Okta** for identity and access management (i.e. the cloud identity provider/IdP).

The suite of cloud services allows us to contain and control access at a much more granular level, compared to operating on-premise infrastructure. We are able to more easily integrate and automate security operations via access to the extensive APIs provided by the cloud services.

In AWS, there are separate accounts for development, test, DevOps infrastructure, security and production. They connect to an organizational master account for centralized billing. Each account is self-contained with its own security policies for access control. Additionally, minimizing infrastructure significantly reduces always-on attack surfaces. Services that are not used are turned off, instead of being idly available which opens itself up to attacks.

LifeOmic platform is designed on a microservices architecture, heavily leveraging **Docker** containers and **AWS Lambda** functions. The containers and Lambda functions are short-lived – they are spun up as soon as a request comes in and are terminated right after their job is complete. Each Lambda function is active for no more than five minutes. Each container or function operates in an individually isolated processing environment.

The ephemeral nature of our computational instances not only makes our services extremely scalable, but also virtually impenetrable. This operating model minimizes persistent attack surface and blast radius, making it virtually impossible for any Advanced Persistent Threat (APT) – the main culprit of most high profile cyber attacks – to gain a foothold, replicate in the environment, and exfiltrate data.

3.3 Need-based temporary access

It is a commonly known security best practice to provision access following the least-privilege principle. We follow that approach but additionally extend it to allow only temporary access to resources in our AWS environments.

LifeOmic employees and the underlying LifeOmic platform microservices are restricted to just the capabilities that they need to do their job.

For employees, least-privileged access means that their access to production systems is extremely limited or non-existent. If given access to production, then it is for a short duration with minimal capabilities. For example, when troubleshooting a production problem then only read-only access to the infrastructure (not customer data) is given. An employee's identity and access permissions are centrally managed via a **single sign on (SSO)** solution. During emergency operational issues, an employee can be quickly given escalated privileged by highly trusted individuals. All activity related to this process is captured and available for future audits.

For services, least-privileged access means that the role of a given microservice is configured to be as restrictive as possible without impeding its intended functionality. For example, if a microservice uses DynamoDB as a storage backend then it only has access to the DynamoDB tables that it owns. Furthermore, a microservice should only be allowed to communicate with another microservice when required.

Access to LifeOmic AWS accounts are permissible through temporary credentials / sessions only. No persistent users, passwords or access keys are allowed in AWS IAM configurations for end-user access, either to the **AWS console** or **AWS CLI**. The access is also protected by **multi-factor authentication (MFA)** at least once every eight hours or with every high-risk access (e.g. access from a new location, unknown device, or to access privileged resources). This is achieved using the Assume Role capability in AWS Identity and Access Management (IAM) service.

AWS Console Access.

- An organization master account in AWS is configured with IAM roles such as **Developer** and **Security**.
- SAML based Single Sign On (SSO) and a trust relationship is established between the pre-defined roles in *lifeomic-master* AWS account and an "AWS application" provisioned in Okta.
- Users are assigned their corresponding roles through application and role assignment in Okta.
- Via SSO, Users authenticate through **Okta** by using their Okta username, password, and MFA.
- Upon successful authentication and MFA validation, users are logged into the *lifeomic-master* AWS account using AWS *Assume Role* capability.
- The roles in master by default has highly restricted access. For example, the **Developer** role does not have access to any services and resources in the master account.
- The user is required to Assume a Role in a sub-account, connected via cross-account trust policy defined at account bootstrap or through an approved change management process. For example, a Developer can assume the **Administrator** role in *lifeomic-dev* AWS account, which is the sandboxed development environment in a separate AWS account.

AWS CLI/SDK Access.

- A command line tool, **Okta AWS-CLI**[2], is used to obtain temporary credentials (access keys) for developers to connect to AWS using the CLI or SDK.
- By running the **Okta AWS-CLI** tool, developers are prompted to authenticate to Okta using their Okta credentials and MFA token/app.
- Upon successful authentication and MFA validation, a temporary access key and session token is inserted into the local configuration file (e.g. `/.aws/credentials` and `/.aws/config`).
- These temporary credentials expire after one hour and a new temporary credential must be obtained for access.
- Through this method, developers are granted the same permission as they would by assuming the **Developer** role through AWS console.

3.4 Watch everything, even the watchers

You can't protect what you can't see. As the famous strategist, Sun Tzu, once said, *"Know thy self, know thy enemy. A thousand battles, a thousand victories."* It all starts with knowing ourselves. This applies to the infrastructure, environments, operations, users, systems, resources, and most importantly, data. It is important to inventory all assets, document all operations, identify all weaknesses, and visualize/understand all events.

This includes conducting various risk analysis, threat modeling, vulnerability assessments, application scanning, and penetration testing. Not only that, this requires security operations to keep an eye on everything, and someone should also "watch the watchers".

As part of LifeOmic security operations, all environments are monitored, all events are logged, all alerts are analyzed, all assets are tracked. There is no privileged access without prior approval or full auditing. For our critical environments in AWS, we integrated two independent solutions to monitor the infrastructure and account configurations, user activities, and security events to ensure that one solution would cover anything that might be missed by the other, and that if one fails or is compromised, the security team is alerted via the second monitoring solution.

3.5 Establishing the "air gap" to production

For the production environments in AWS, we want to provide an even higher level of security assurance, in a way such that

- There is no internal network connectivity into the environment such as VPN, SSH, or AWS DirectConnect.
- LifeOmic engineers can only access applications logs in production for troubleshooting and support, but have no access to systems, configurations, resources, workloads or any customer data.

Any privileged access into production environment requires an approved changed management ticket and passing four security gates:

- The elevated role must be assigned to the approved individual in the centralized IdP (Okta);
- The user must authenticate and pass MFA validation;

[2] https://github.com/oktadeveloper/okta-aws-cli-assume-role

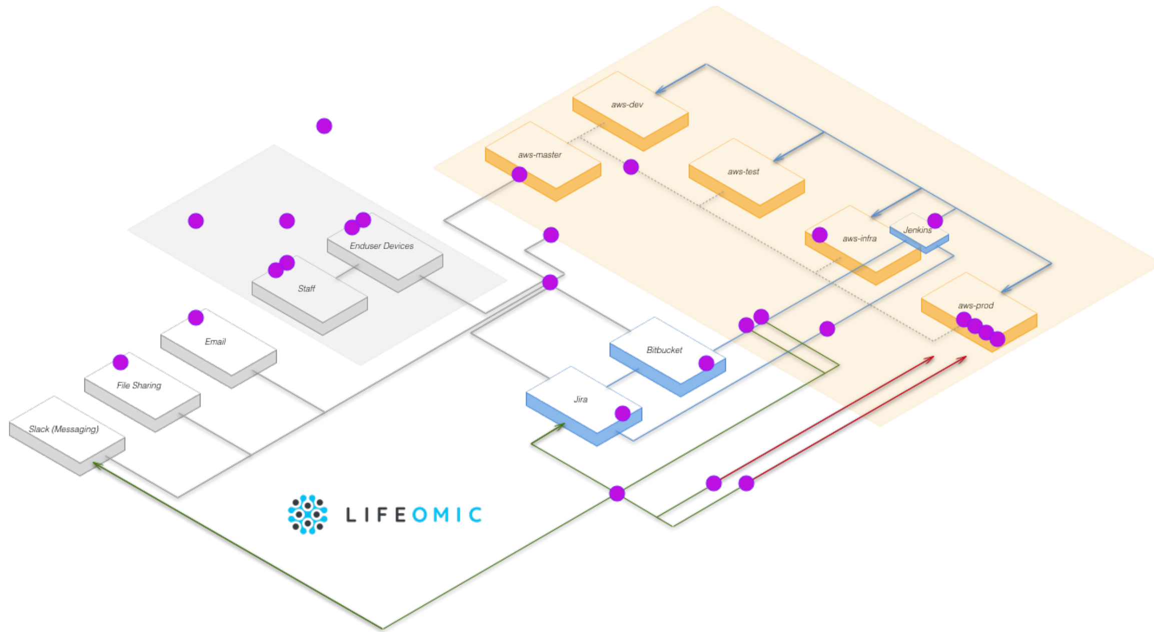

Figure 1: LifeOmic Cloud DevSecOps Blueprint (each purple dot represents a unique security control)

- An explicit deny access rule to production must be temporarily lifted for the user to assume a privileged role in production; and
- Even with the privileged access, certain risky actions such as making changes to IAM policies, users, roles or groups and accessing customer data are explicitly denied.

Privileged access role does not provide the ability to access production workloads at the network layer, such as SSH.

Now, the question is, how do we get software deployed into such environments?

4 SECURE SOFTWARE DELIVERY PIPELINE

Security of a production software system does not start when the software is running in the production environment. Security starts from the very beginning – when the software is written – and continues throughout the deployment process. Mature software delivery pipelines leverage numerous security checks. At each stage in the process, risks should be assessed, and countermeasures should be implemented.

4.1 Automation

A fast software delivery pipeline is not the only benefit of automation; automation minimizes human errors and human intervention which greatly improves security.

LifeOmic relies on a fully automated *Continuous Integration / Continuous Delivery (CI/CD)* software delivery pipeline. The software delivery pipeline handles running tests, scanning code for vulnerabilities, building software, deploying changes, and monitoring production systems.

Humans make mistakes. For this reason, LifeOmic requires all changes to the production platform to be automated. This alleviates the need for anyone to have production access to do their job.

4.2 Infrastructure as code

LifeOmic closely adheres to the *infrastructure as code* policy. This policy dictates that all infrastructure should be described in code. Because infrastructure *is* code, it can be versioned, tested, built, and deployed just like normal application code. At LifeOmic, we use terraform to declaratively describe and deploy the infrastructure.

4.3 Immutable builds

A *build* produces artifacts that encapsulate the logic of the software and the instructions for how to deploy it. These build artifacts do not change as they are used to deploy the changes to each environment.

4.4 Developer friendly

Happy developers are typically more productive. One way of keeping developers happy is to remove the manual work and automate as much as possible. When developers are annoyed by manual work they will start trying to circumvent the process which is exactly the opposite of what a company wants to achieve when they implement new security measures. At LifeOmic, we want developers to focus on writing code. Beginning with code review and tooling, developers are trained to follow the process from the very beginning which greatly simplifies the work that is required by them downstream.

4.5 Deep Dive: LifeOmic software delivery pipeline

To fully describe the LifeOmic software delivery pipeline, we will break it down to the following stages:

(1) Code and Test
(2) Build
(3) Deploy
(4) Monitor

When developing software for a multi-tenant cloud platform, most companies aim for Continuous Integration / Continuous Delivery (CI/CD).

Typically, deploying numerous small changes quickly is less risky than deploying a lot of changes infrequently. When the production system starts to diverge drastically from the latest code, it is harder to anticipate some of the problems that will arise when changes are finally integrated into the production system. Automation is essential for an effective CI/CD process because automation allows for rapid validation and deployment.

Stage 1: Code and Test. Every company wants to trust their developers but writing code for large distributed systems is complicated, so automated and manual reviews are needed to identify potential vulnerabilities or defects in the source code.

Modern source code management systems make it easy for teams to collaborate on code. Code reviews via a *pull request* are an essential step in the coding process. When developers are ready to integrate their code changes into the mainline code (or *master* branch) then they should push their code to a branch and open a pull request. The pull request gives other developers and security experts a chance to review the code before it is merged. Each reviewer should provide feedback and approve the pull request once all feedback has been addressed. This code review process makes it possible for teams to share knowledge and provides an additional security safeguard.

The LifeOmic platform consists of code from numerous projects that can be built, tested, and deployed independently. Each project is contained within its own **Bitbucket** git repo, and each project typically corresponds to a single micro-service, a portion of the infrastructure, or an internal tool.

Every software delivery pipeline should also include multiple levels of testing. The first level of testing should be *unit testing* which involves isolating small portions of the code and testing it independently by closely controlling the input and verifying that actual output matches the expected output. The packaged code should ideally contain *integration tests* and *smoke tests*. Integration tests run in a target environment before being deployed. These tests ensure that the target environment is in a state that is ready for the given changes. Smoke tests run in a target environment after changes are deployed. These tests provide a "sanity check" to make sure the system is still operating normally after the changes have been applied.

Bitbucket repos are configured to ensure that developers adhere to a prescribed process.

The prescribed process for merging code to the *master* branch requires the following:

- Pull request must be approved by someone other than author before it can be merged.
- Pull request branch must have a passing build before it can be merged.
- All tests on the pull request branch must be passing.

Once code is merged to *master*, it is built, and the build artifacts are published.

Before code is ever deployed it should be analyzed in a sandbox using static and dynamic code access. The software delivery pipeline should verify that all code scans have been completed before accepting the new code.

Security measures in the Code and Test phase:

- Source code "linting" (enforcement of code style and best practices)
- Comprehensive test suite development
- Secure coding best practices
- Local secure code analysis
- Code review / pull requests
- Source control versioning
- Infrastructure as code

Stage 2: Build. Code that was written on developer workstations should be built in a secure build environment. While developers should be able to build their software locally to test the build and deploy process, only official builds should be deployed to production environments. Builds should be signed by the trusted build system. Builds that haven't been properly signed should never be deployed to production.

Official builds of the software happen via a single managed instance of Jenkins. Each build job for code merged to *master* branch produces one or more artifacts that are stored in **AWS S3**. The build artifacts are organized by the build ID and are immutable as they move through the deploy process.

A typical LifeOmic project will produce the following build artifacts:

- **Build manifest**: The build manifest describes the contents of the build as well as metadata about the build (build date, build number, git commit, etc.)
- **Deploy docker image**: This docker image encapsulates the instructions for deploying the build. When deploying to a target environment, this docker image is used to launch a container that deploys the software. This container contains terraform, AWS CLI, and other tools that perform the deployment via a repeatable process.
- **Runtime docker image**: For services that run in **AWS Elastic Container Service (ECS)**, one or more runtime docker images will be produced during the build. At LifeOmic, most services deploy to **AWS Lambda** instead of **AWS ECS**, so these services do not produce a runtime docker image.

Security measures in the Build phase:

- Code signing to ensure immutable builds
- Perform automated testing and verify passing results
- Perform automated security scans and ensure no vulnerability is introduced, including
  * Open source dependency security analysis
  * Static application security testing
  * Automated baseline dynamic scanning
  * Docker image vulnerability scanning

Stage 3: Deploy. The deploy process is perhaps the most challenging stage of the software delivery pipeline because it involves applying changes to a production system while trying to minimize

the impact to end-users. The deploy needs to be secure, controlled, and repeatable. The deploy process is especially challenging when aiming for zero downtime.

Immutable builds should be deployed and tested in "lower" environments, such as dev and test, before being promoted to production.

At LifeOmic, deploys are fully automated to minimize human error and to ensure that all deploys are repeatable from development up to the final production environment. Each environment lives in isolation within its own AWS account. Furthermore, each environment is "virtually air-gapped" which means there is no private network connectivity between environments. Because there is no private network connectivity deploys are orchestrated via the AWS APIs. The IAM role given to the Jenkins service (but not developers) allows the Jenkins service to assume a role in each target environment that gives it a very narrow interface for initiating the deployment of new changes. The process to initiate a deploy involves invoking an **AWS Lambda** function that accepts the following input arguments:

- **Deploy Manifest URI**: The deploy manifest URI points to a file stored in **S3** bucket that describes the deploy request.
- **Deploy Manifest Signature**: A cryptographic signature of the *deploy manifest* file is generated by a Jenkins build job using a pre-shared secret. The pre-shared secret used to generate the signature is known only by **Jenkins** (stored securely as a *Jenkins secret*) and by the target environment (stored using **AWS Systems Manager Parameter Store** as an encrypted secure string).

The **AWS Lambda** function that receives the deploy request validates the signature, and, if valid, launches a worker in **AWS ECS**. The worker then fetches the build artifacts as described in the deploy manifest and launches the deploy docker image that encapsulates the deployment job. Because the worker is not human, the system is less prone to human error. Also, by not allowing humans to apply the changes, it is much less likely that a human will have intentional or unintentional access to production data. This design choice thus makes it easier for the platform to adhere to privacy requirements as prescribed by HIPAA or other certifications.

Security measures in the Deploy phase:

- Fully automated builds ("hands-free deployments")
- Verify that the original source code was approved by multiple people
- Verify that change management ticket has been approved
- Verify build signatures
- Verify security scans
- Verify that tests passed
- Automated analysis of infrastructure code
- Automated analysis of changes
- Run integration tests
- Run smoke tests

Stage 4: Monitor. Once software is running in a production environment, it is important to monitor the health and safety of the system. Various proprietary and third-party tools are used to monitor the system and send alerts. These alerts are then curated so that the security operations team is not flooded with noise or false positives. The security response team has tools available that allow them to investigate incidents.

Security measures in the Monitor phase:

- Tooling for recognizing, analyzing, and mitigating vulnerabilities
- Health checks
- Application metrics
- Anomaly detection
- Security audit logs
- Continuous penetration testing

The next big question becomes, how do we ensure that this process has been followed with each production deploy? What type of reviews and approvals are required and how does it scale with CI/CD in a Cloud DevOps operating environment?

5 PRODUCTION CHANGE MANAGEMENT AUTOMATION

5.1 Goals

The goals of the change management process for our organization is to document what is being changed and, in the case of changes to software that we develop, that the changes were developed using the processes documented to ensure high quality products. We require all software changes to undergo peer review and security scanning before the code is put into production. The requirements for production deployments will evolve over time but for the sake of this paper, those are the two initial requirements.

Our initial implementation of the change management process was for developers to propose a change and write out a list of changes being made, ideally with references to automated builds and individual code changes. A security team member would review the list of changes, manually looking for the details of each change and checking that each change was done following the correct practices. Needless to say, the approach of using humans to document the changes and to review the practices was both time consuming and prone to oversights. It was possible for the person requesting the change to forget to list a change that was made, and it was possible for the security review to incorrectly review the changes that were listed. With people focused on different tasks at different times, it was common for the change requester to be stuck waiting on a human reviewer and for the human reviewer to be waiting for more details from the requester.

Change Management becomes easier because it's easier to verify that the process was followed via automation. With high levels of automation, you have less reliance on a "paper pusher" that needs to click approve on every change.

Given our goal of quickly improving our production systems while maintaining a very high level of functional and security quality, it seemed like a better change management process was possible. The tools that we use for software development, **Jenkins**, **Bitbucket**, and **Jira**, provide a good history of what happened, and it was possible to automatically collect both details of what was being changed and proof that the correct processes were followed. In the end, a system that was both more thorough than humans and allowed for faster change in production was created.

5.2 Implementation

The first step to an automated approval process was to more accurately capture what code changes were being requested for deployment to production. In our system, each deployment is defined by a specific **Jenkins** project and build identifier. All of our code is stored in git repositories in **Bitbucket** and for each **Jenkins** build, it is possible to list which versions of the code are included in the build. When reviewing change requests, the list of code changes, not just the code version, is desired. All of our change management requests are stored in **Jira** and each request includes the project and build details. By using Jira's search capabilities, it is possible to find the previous version of the project that was approved for production. With the previously approved and the newly proposed versions, git can be used to complete the changes between the two versions. That list of changes is a huge step forward from a human created list because it is consistently created and it always accurate, never accidentally forgetting a change or adding an extra one.

With a list of changes since the last production approval, the next step is to review the changes for the required peer code reviews. The **Bitbucket** *Pull Request* feature is used by developers to propose code changes, to collect peer feedback and to integrate the changes after approval. The **Bitbucket** service allows the data about pull requests to be extracted and analyzed through an API they provide. For each change that is being proposed for production, the pull requests for the project are reviewed to make sure each change has an associated pull request. For each pull request, the peer reviews are captured by the reviewer giving approval in the request. The change management review tool will ensure that every change in the pull request has been approved by somebody other than the author. If all changes in the proposed deployment have an associated pull request and each pull request is marked as approved by somebody other than the original developer, then the change management review tool will proceed to checking for rest of the required processes.

After reviewing the changes for peer reviews, the next requirement in automatic change management approval is that a security scan has been run for the code. For this check, the **Jenkins** build associated with the proposed change is reviewed again. When a Jenkins build is executed, logs for the build are stored for later review and those logs contain the details of whether a security scan was run and whether any vulnerabilities were found. If the logs for the build can be found, and if a successful security scan is detected in the logs, then the change management review tool will proceed to building a final assessment of the proposed production change.

Assuming the code review and security scan processes have been followed, the automated change management tool will leave a comment in in the change management request with the details of each change that was reviewed and details of the detected security scan. If all processes have been followed correctly, then the change management tool will add itself to the list of approvers and will mark the change management request as approved.

If at any time in the automated review process a deviation from our required process is found, the review automation will collect the relevant information and leave a comment in the change management request ticket. In the case of a process deviation, the ticket is left in an unapproved state and it is up to a human to review what was found and to ensure that the changes for production are acceptable before approving the deployment.

5.3 Integrating change management into the software delivery pipeline

This automated change review process is integrated into LifeOmic continuous delivery pipeline in 3 steps:

1. Create/Validate Change Request Ticket. **Jenkins** is used for continuous delivery (build and deploy), and a *Product Change Manage project (PRODCM)* is created in **Jira** to track the change request tickets and approvals. We developed a *Jenkins-Jira* automation such that:

- Whenever deployment to a controlled environment (e.g. production accounts and infrastructure account) is requested, the Jenkins job will check for an approved PRODCM ticket, or create a new ticket if not found.
- The automation code will attempt to automatically populate the required data for the **PRODCM** ticket, such as build number, deploy action, target environment, etc.
- If the data cannot be automatically populated, the job is paused to prompt an engineer for manual input.
- Job will be paused until the request is approved or canceled (rejected). Before continuing to deployment, **Jenkins** will validate the change request's build job identifier, build number and source code branch.

2. Detect Change Details and Obtain Approval. The aforementioned review automation is implemented as in such way (the code is named **change-management-bot**):

- Whenever a PRODCM ticket is created, the **bot** is triggered via a **Jira** *webhook*.
- The bot is configured to examine the following:
  * Look for all the code changes since the last approved **PRODCM** ticket.
  * Check that all code commits have been approved by a reviewer other than the author, except for a version bump.
  * Ensure that security scanning has been completed for this build and no blocking issue is found.
- Details of the analysis are posted to the **PRODCM** Jira ticket. An example is shown in the screenshot figure.
- When all the required checks pass validation, the bot recommends approval. The **change-management-bot** may be configured to automatically approve the ticket if all of the required conditions above (and future ones) are met. Additionally, a manual review / approval is always required in the following conditions:
  * This is the first prod deploy with no prior approval history
  * A related CM ticket / deploy of the same project is pending
  * If human approvals are needed, the required approvers will review the details and approve/decline accordingly.
- Random inspections of automatically approved tickets are performed by the security team monthly to ensure the automation functions properly.

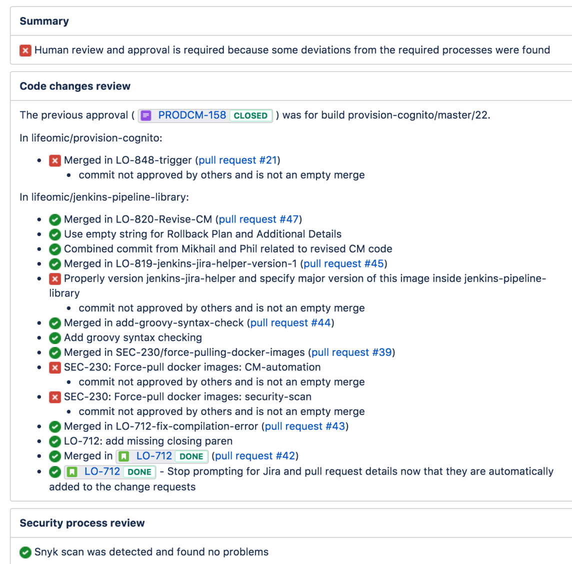

Figure 2: An example screenshot of the comments posted on a PRODCM ticket by change-management-bot.

3. Detect Risky Changes, Deploy and Close. Jenkins job proceeds only with an approved and validated **PRODCM** ticket.

During production deploys, a `terraform plan` is always performed first to detect risky changes. Examples of security-related or risky changes include:

- Change to the "policy" attribute of an AWS resource
- Change to IAM policy, role, user or group
- Attach/detach policy
- Change/delete to security group
- Deletion of production resources

If risky changes are detected, the deploy is paused and the PRODCM ticket is updated to require manual review before continuing. Once a deploy is completed, the **PRODCM** ticket is automatically resolved and closed.

5.4 Impact

After months of running the automated change management review tool there was a clear improvement in the documentation about what changed in each change to production and a more complete review of the processes followed for each of those changes. The time between requesting a change to production and the request being approved has dramatically reduced allowing faster improvement to production systems. Equally important, less time from the security team is spent performing mechanical reviews of changes and the teams time can be spent on more valuable work of reviewing complicated changes or doing proactive architectural reviews.

One design principal of the automated reviews was that the tool would never automatically reject a change and would, in the worst case, leave a comment about any process deviation that were found and defer the approval to a human. Because humans are slower than computers, the automated review tool has led to some cultural improvements from the development team. Developers choose to follow processes that allow for better auditing in exchange for faster production changes and they push each other to follow the correct process for all changes so that deployments to production are not held up by the need for a human review. The automation caused the incentive of fast production deployments to be self-motivate development to improve security process compliance.

6 CONCLUSIONS

Implementing this virtually "air-gapped" environment in AWS required additional efforts in the software delivery pipeline and change management automation. This upfront effort was significant, but it enabled a much more delightful developer experience while providing maximum security assurance to our customers.