O'REILLY®

DevOpsSec

Delivering Secure Software Through Continuous Delivery

Jim Bird

3 Easy Ways to Stay Ahead of the Game

The world of security is constantly changing. Here's how you can keep up:

1. **Download free reports** on the current and trending state of security. **oreil.ly/Security_reports**

2. **Subscribe** to the weekly Security newsletter. **oreil.ly/Security_news**

3. **Attend the O'Reilly Security Conference,** the must attend conference for security professionals. **oreil.ly/Security_conf**

For more information and additional Security resources, visit **oreil.ly/Security_topics**.

DevOpsSec

Securing Software through Continuous Delivery

Jim Bird

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

DevOpsSec
by Jim Bird

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Courtney Allen
Production Editor: Shiny Kalapurakkel
Copyeditor: Bob & Dianne Russell, Octal Publishing, Inc.
Proofreader: Kim Cofer
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

June 2016: First Edition

Revision History for the First Edition
2016-05-24: First Release

978-1-491-95899-5

[LSI]

Table of Contents

CHAPTER 1

DevOpsSec: Delivering Secure Software through Continuous Delivery

Introduction

Some people see DevOps as another fad, the newest new-thing overhyped by Silicon Valley and by enterprise vendors trying to stay relevant. But others believe it is an authentically disruptive force that is radically changing the way that we design, deliver, and operate systems.

In the same way that Agile and Test-Driven Development (TDD) and Continuous Integration has changed the way that we write code and manage projects and product development, DevOps and Infrastructure as Code and Continuous Delivery is changing IT service delivery and operations. And just as Scrum and XP have replaced CMMi and Waterfall, DevOps is replacing ITIL as the preferred way to manage IT.

DevOps organizations are breaking down the organizational silos between the people who design and build systems and the people who run them—silos that were put up because of ITIL and COBIT to improve control and stability but have become an impediment when it comes to delivering value for the organization.

Instead of trying to plan and design everything upfront, DevOps organizations are running continuous experiments and using data from these experiments to drive design and process improvements.

DevOps is finding more effective ways of using the power of automation, taking advantage of new tools such as programmable configuration managers and application release automation to simplify and scale everything from design to build and deployment and operations, and taking advantage of cloud services, virtualization, and containers to spin up and run systems faster and cheaper.

Continuous Delivery and Continuous Deployment, Lean Startups and MVPs, code-driven configuration management using tools like Ansible and Chef and Puppet, NoOps in the cloud, rapid self-service system packaging and provisioning with Docker and Vagrant and Terraform are all changing how everyone in IT thinks about how to deliver and manage systems. And they're also changing the rules of the game for online business, as DevOps leaders use these ideas to out-pace and out-innovate their competitors.

The success that DevOps leaders are achieving is compelling. According to Puppet Labs' 2015 State of DevOps Report:

- High-performers deploy changes 30 times more often than other organizations, with lead times that are 200 times shorter.
- Their change success rate is 60 times higher.
- And when something goes wrong, they can recover from failures 168 times faster.

What do you need to do to achieve these kinds of results? And, how can you do it in a way that doesn't compromise security or compliance, or, better, in a way that will actually improve your security posture and enforce compliance?

As someone who cares deeply about security and reliability in systems, I was very skeptical when I first heard about DevOps, and "doing the impossible 50 times a day."[1] It was too much about how to "move fast and break things" and not enough about how to build systems that work and that could be trusted. The first success stories

1 From an early post on Continuous Deployment: *http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/*

were from online games and social-media platforms that were worlds away from the kinds of challenges that large enterprises face or the concerns of small businesses.

I didn't see anything new or exciting in "branching in code" and "dark launching" or developers owning their own code and being responsible for it in production. Most of this looked like a step backward to the way things were done 25 or 30 years ago, before CMMi and ITIL were put in to get control over cowboy coding and hacking in production.

But the more I looked into DevOps, past the hype, I found important, substantial new ideas and patterns that could add real value to the way that systems are built and run today:

Infrastructure as Code
: Defining and managing system configuration through code that can be versioned and tested in advance using tools like Chef or Puppet dramatically increases the speed of building systems and offers massive efficiencies at scale. This approach to configuration management also provides powerful advantages for security: full visibility into configuration details, control over configuration drift and elimination of one-off snowflakes, and a way to define and automatically enforce security policies at runtime.

Continuous Delivery
: Using Continuous Integration and test automation to build pipelines from development to test and then to production provides an engine to drive change and at the same time a key control structure for compliance and security, as well as a safe and fast path for patching in response to threats.

Continuous monitoring and measurement
: This involves creating feedback loops from production back to engineering, collecting metrics, and making them visible to everyone to understand how the system is actually used and using this data to learn and improve. You can extend this to security to provide insight into security threats and enable "Attack-Driven Defense."

Learning from failure
: Recognizing that failures can and will happen, using them as learning opportunities to improve in fundamental ways through

blameless postmortems, injecting failure through chaos engineering, and practicing for failure in game days; all of this leads to more resilient systems and more resilient organizations, and through Red Teaming to a more secure system and a proven incident response capability.

About DevOps

This paper is written for security analysts, security engineers, pen testers, and their managers who want to understand how to make security work in DevOps. But it also can be used by DevOps engineers and developers and testers and their managers who want to understand the same thing.

You should have a basic understanding of application and infrastructure security as well as some familiarity with DevOps and Agile development practices and tools, including Continuous Integration and Continuous Delivery. There are several resources to help you with this. Some good places to start:

- *The Phoenix Project* by Gene Kim, Kevin Behr, and George Spafford is a good introduction to the hows and whys of DevOps, and is surprisingly fun to read.
- Watch "10+ Deploys per Day," John Allspaw and Paul Hammond's presentation on Continuous Deployment, which introduced a lot of the world to DevOps ideas back in 2009.[2]
- And, if you want to understand how to build your own Continuous Delivery pipeline, read *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Jez Humble and Dave Farley.

The more I talked to people at organizations like Etsy and Netflix who have had real success with DevOps at scale, and the more I looked into how enterprises like ING and Nordstrom and Capital One and Inuit are successfully adopting DevOps, the more I started to buy in.

And the more success that we have had in my own organization with DevOps ideas and tools and practices, the more I have come to

2 Velocity 2009: "10+ Deploys per Day." *https://www.youtube.com/watch?v=LdOe18KhtT4*

understand how DevOps, when done right, can be used to deliver and run systems in a secure and reliable way.

Whether you call it SecDevOps, DevSecOps, DevOpsSec, or Rugged DevOps, this is what this paper will explore.

We'll begin by looking at the security and compliance challenges that DevOps presents. Then, we'll cover the main ideas in secure DevOps and how to implement key DevOps practices and workflows like Continuous Delivery and Infrastructure as Code to design, build, deploy, and run secure systems. In Chapter 4, we'll map security checks and controls into these workflows. Because DevOps relies so heavily on automation, we'll look at different tools that you can use along the way, emphasizing open source tools where they meet the need, and other tools that I've worked with and know well or that are new and worth knowing about. And, finally, we'll explore how to build compliance into DevOps, or DevOps into compliance.

CHAPTER 2

Security and Compliance Challenges and Constraints in DevOps

Let's begin by looking at the major security and compliance challenges and constraints for DevOps.

Speed: The Velocity of Delivery

The velocity of change in IT continues to increase. This became a serious challenge for security and compliance with Agile development teams delivering working software in one- or two-week sprints. But the speed at which some DevOps shops initiate and deliver changes boggles the mind. Organizations like Etsy are pushing changes to production 50 or more times each day. Amazon has thousands of small ("two pizza") engineering teams working independently and continuously deploying changes across their infrastructure. In 2014, Amazon deployed 50 million changes: that's more than one change deployed every second of every day.[1]

So much change so fast...

How can security possibly keep up with this rate of change? How can they understand the risks, and what can they do to manage

1 "AWS re:Invent 2015 | (DVO202) DevOps at Amazon: A Look at Our Tools and Processes." *https://www.youtube.com/watch?v=esEFaY0FDKc*

them when there is no time to do pen testing or audits, and no place to put in control gates, and you can't even try to add a security sprint or a hardening sprint in before the system is released to production?

Where's the Design?

DevOps builds on Agile development practices and extends Agile ideas and practices from development into operations.

A challenge for many security teams already working in Agile environments is that developers spend much less time upfront on design. The Agile manifesto emphasizes "working software over documentation," which often means that "the code is the design," and "Big Design Up Front" is an antipattern in most Agile shops. These teams want to start with a simple, clean approach and elaborate the design in a series of sprints as they get more information about the problem domain. The principle of YAGNI ("You Ain't Gonna Need It") reminds them that most features specified upfront might never be used, so they try to cut the design and feature-set to a minimum and deliver only what is important, as soon as they can, continuously refactoring as they make changes.

Many DevOps teams take all of these ideas even further, especially teams following a Lean Startup approach. They start with a Minimum Viable Product (MVP): the simplest and cheapest implementation of an idea with a basic working feature-set, which is delivered to real users in production as soon as possible. Then, using feedback from those users, collecting metrics, and running A/B experiments, they iterate and fill out the rest of the functionality, continuously delivering changes and new features as quickly as possible in order to get more feedback to drive further improvements in a continuous loop, adapting and pivoting as needed.

All of this makes it difficult for security teams to understand where and how they should step in and make sure that the design is secure before coding gets too far along. How do you review the design if design isn't done, when there is no design document to be handed off, and the design is constantly changing along with the code and the requirements? When and how should you do threat modeling?

Eliminating Waste and Delays

DevOps is heavily influenced by Lean principles: maximizing efficiency and eliminating waste and delays and unnecessary costs. Success is predicated on being first to market with a new idea or feature, which means that teams measure—and optimize for—the cycle-time-to-delivery. Teams, especially those in a Lean Startup, want to fail fast and fail early. They do rapid prototyping and run experiments in production, with real users, to see if their idea is going to catch on or to understand what they need to change to make it successful.

This increases the tension between delivery and security. How much do engineers need to invest—how much can they afford—in securing code that could be thrown away or rewritten in the next few days? When is building in security the responsible thing to do, and when is it wasting limited time and effort?

In an environment driven by continuous flow of value and managed through Kanban and other Lean techniques to eliminate bottlenecks and using automation to maximize efficiency, security cannot get in the way. This is a serious challenge for security and compliance, who are generally more concerned about doing things right and minimizing risk than being efficient or providing fast turnaround.

It's in the Cloud

Although you don't need to run your systems in the cloud to take advantage of DevOps, you probably do need to follow DevOps practices if you are operating in the cloud. This means that the cloud plays a big role in many organizations' DevOps stories (and vice versa).

In today's cloud—Infrastructure as a Service (IaaS) and Platform as a Service (PaaS)—platforms like Amazon AWS, Microsoft Azure, and Google Cloud Platform do so much for you. They eliminate the wait for hardware to be provisioned; they take away the upfront cost of buying hardware and setting up a data center; and they offer elastic capacity on demand to keep up with growth. These services hide the details of managing the data center and networks, and standing up and configuring and managing and monitoring servers and storage.

There are so many capabilities included now, capabilities that most shops can't hope to provide on their own, including built-in security and operations management functions. A cloud platform like AWS offers extensive APIs into services for account management, data partitioning, auditing, encryption and key management, failover, storage, monitoring, and more. They also offer templates for quickly setting up standard configurations.

But you need to know how to find and use all of this properly. And in the shared responsibility model for cloud operations, you need to understand where the cloud provider's responsibilities end and yours begin, and how to ensure that your cloud provider is actually doing what you need them to do.

The Cloud Security Alliance's "Treacherous Twelve" highlights some of the major security risks facing users of cloud computing services:

1. Data breaches
2. Weak identity, credential, and access management
3. Insecure interfaces and APIs
4. System and application vulnerabilities
5. Account hijacking
6. Malicious insiders
7. Advanced Persistent Threats (APTs)
8. Data loss
9. Insufficient due diligence
10. Abuse and nefarious use of cloud services
11. Denial of Service
12. Shared technology issues

Microservices

Microservices are another part of many DevOps success stories. Microservices—designing small, isolated functions that can be changed, tested, deployed, and managed completely independently—lets developers move fast and innovate without being held up by the rest of the organization. This architecture also encourages developers to take ownership for their part of the system, from design to delivery and ongoing operations. Amazon and Netflix have had remarkable success with building their systems as well as their organizations around microservices.

But the freedom and flexibility that microservices enable come with some downsides:

- Operational complexity. Understanding an individual microservice is simple (that's the point of working this way). Understanding and mapping traffic flows and runtime dependencies between different microservices, and debugging runtime problems or trying to prevent cascading failures is much harder. As Michael Nygard says: "An individual microservice fits in your head, but the interrelationships among them exceed any human's understanding."
- Attack surface. The attack surface of any microservice might be tiny, but the total attack surface of the system can be enormous and hard to see.
- Unlike a tiered web application, there is no clear perimeter, no obvious "choke points" where you can enforce authentication or access control rules. You need to make sure that trust boundaries are established and consistently enforced.
- The polyglot programming problem. If each team is free to use what they feel are the right tools for the job (like at Amazon), it can become extremely hard to understand and manage security risks across many different languages and frameworks.
- Unless all of the teams agree to standardize on a consistent activity logging strategy, forensics and auditing across different services with different logging approaches can be a nightmare.

Containers

Containers—LXC, rkt, and (especially) Docker—have exploded in DevOps.

Container technologies like Docker make it much easier for developers to package and deploy all of the runtime dependencies that their application requires. This eliminates the "works on my machine" configuration management problem, because you can ship the same runtime environment from development to production along with the application.

Using containers, operations can deploy and run multiple different stacks on a single box with much less overhead and less cost than using virtual machines. Used together with microservices, this makes it possible to support microsegmentation; that is, individual microservices each running in their own isolated, individually managed runtime environments.

Containers have become so successful, Docker in particular, because they make packaging and deployment workflows easy for developers and for operations. But this also means that it is easy for developers —and operations—to introduce security vulnerabilities without knowing it.

The ease of packaging and deploying apps using containers can also lead to unmanageable container sprawl, with many different stacks (and different configurations and versions of these stacks) deployed across many different environments. Finding them all (even knowing to look for them in the first place), checking them for vulnerabilities, and making sure they are up-to-date with the latest patches can become overwhelming.

And while containers provide some isolation and security protection by default, helping to reduce the attack surface of an application, they also introduce a new set of security problems. Adrian Mouat, author of *Using Docker*, lists five security concerns with using Docker that you need to be aware of and find a way to manage:

Kernel exploit
: The kernel is shared between the host and all of the kernels, which means that a vulnerability in the kernel exposes everything running on the machine to attack.

Denial of Service attacks
: Problems in one container can DoS everything else running on the same machine, unless you limit resources using cgroups.

Container breakouts
: Because isolation in containers is not as strong as in a virtual machine, you should assume that if an attacker gets access to one container, he could break into any of the other containers on that machine.

Poisoned images
: Docker makes it easy to assemble a runtime stack by pulling down dependencies from registries. However, this also makes it easy to introduce vulnerabilities by pulling in out-of-date images, and it makes it possible for bad guys to introduce malware along the chain. Docker and the Docker community provide tools like trusted registries and image scanning to manage these risks, but everyone has to use them properly.

Compromising secrets
: Containers need secrets to access databases and services, and these secrets need to be protected.

You can lock down a container by using CIS guidelines and other security best practices and using scanning tools like Docker Bench, and you can minimize the container's attack surface by stripping down the runtime dependencies and making sure that developers don't package up development tools in a production container. But all of this requires extra work and knowing what to do. None of it comes out of the box.

Separation of Duties in DevOps

DevOps presents some challenges to compliance. One of the most difficult ones to address is Separation of Duties (SoD).

SoD between ops and development is designed to reduce the risk of fraud and prevent expensive mistakes and insider attacks by ensuring that individuals cannot make a change without approval or transparency. Separation of Duties is spelled out as a fundamental control in security and governance frameworks like ISO 27001, NIST 800-53, COBIT and ITIL, SSAE 16 assessments, and regulations such as SOX, GLBA, MiFID II, and PCI DSS.

Auditors look closely at SoD, to ensure that requirements for data confidentiality and integrity are satisfied; that data and configuration cannot be altered by unauthorized individuals; and that confidential and sensitive data cannot be viewed by unauthorized individuals. They review change control procedures and approval gates to ensure that no single person has end-to-end control over changes to the system, and that management is aware of all material changes before they are made, and that changes have been properly tested and reviewed to ensure that they do not violate regulatory requirements. They want to see audit trails to prove all of this.

Even in compliance environments that do not specifically call for SoD, strict separation is often enforced to avoid the possibility or the appearance of a conflict of interest or a failure of controls.

By breaking down silos and sharing responsibilities between developers and operations, DevOps seems to be in direct conflict with SoD. Letting developers push code and configuration changes out to production in Continuous Deployment raises red flags for auditors. However, as we'll look at in Chapter 5, it's possible to make the case that this can be done, as long as strict automated and manual controls and auditing are in place.

Another controversial issue is granting developers access to production systems in order to help support (and sometimes even help operate) the code that they wrote, following Amazon's "You build it, you run it" model. At the Velocity Conference in 2009, John Allspaw and Paul Hammond made strong arguments for giving developers access, or at least limited access, to production:[2]

> Allspaw: "I believe that ops people should make sure that developers can see what's happening on the systems without going through operations... There's nothing worse than playing phone tag with shell commands. It's just dumb.
>
> Giving someone [i.e., a developer] a read-only shell account on production hardware is really low risk. Solving problems without it is too difficult."
>
> Hammond: "We're not saying that every developer should have root access on every production box."

2 *http://www.kitchensoap.com/2009/06/23/slides-for-velocity-talk-2009/*

Any developer access to a regulated system, even read-only access, raises questions and problems for regulators, compliance, infosec, and customers. To address these concerns, you need to put strong compensating controls in place:

- Limit access to nonpublic data and configuration.
- Review logging code carefully to ensure that logs do not contain confidential data.
- Audit and review everything that developers do in production: every command they execute, every piece of data that they looked at.
- You need detective change control in place to track any changes to code or configuration made outside of the Continuous Delivery pipeline.
- You might also need to worry about data exfiltration: making sure that developers can't take data out of the system.

These are all ugly problems to deal with, but they can be solved.

At Etsy, for example, even in PCI-regulated parts of the system, developers get read access to metrics dashboards (what Etsy calls "data porn") and exception logs so that they can help find problems in the code that they wrote. But any changes or fixes to code or configuration are reviewed and made through their audited and automated Continuous Deployment pipeline.

Change Control

How can you prove that changes are under control if developers are pushing out changes 10 or 50 times each day to production? How does a Change Advisory Board (CAB) function in DevOps? How and when is change control and authorization being done in an environment where developers push changes directly to production? How can you prove that management was aware of all these changes before they were deployed?

ITIL change management and the associated paperwork and meetings were designed to deal with big changes that were few and far between. Big changes require you to work out operational dependencies in advance and to understand operational risks and how to mitigate them, because big, complex changes done infrequently are

risky. In ITIL, smaller changes were the exception and flowed under the bar.

DevOps reverses this approach to change management, by optimizing for small and frequent changes—breaking big changes down to small incremental steps, streamlining and automating how these small changes are managed. Compliance and risk management need to change to fit with this new reality.

CHAPTER 3

Keys to Injecting Security into DevOps

Now let's look at how to solve these problems and challenges, and how you can wire security and compliance into DevOps.

Building Security into DevOps: Etsy's Story

Etsy, a successful online crafts marketplace, is famous for its Continuous Deployment model, where engineers (and managers and even pets) push changes out 50 times or more every day. It is also known for its blameless, "Just Culture," in which engineers are taught to embrace failure, as long as they learn from their mistakes.

Etsy's security culture is built on top of its engineering culture and connects with the wider culture of the organization. Some of its driving principles are:

- Trust people to do the right thing, but still verify. Rely on code reviews and testing and secure defaults and training to prevent or catch mistakes. And if mistakes happen in production, run postmortems to examine and understand what happened and how to fix things at the source.
- "If it Moves, Graph it." Make data visible to everyone so that everyone can understand and act on it, including information about security risks, threats, and attacks.
- "Just Ship It." Every engineer can push to production at any time. This includes security engineers. If something is broken

and you can fix it, fix it and ship the fix out right away. Security engineers don't throw problems over the wall to dev or ops if they don't have to. They work with other teams to understand problems and get them fixed, or fix the problem themselves if they can. Everyone uses the Continuous Deployment pipelines and the same tools to push changes out to production, including the security team.

- Security cannot be a blocker. The word "No" is a finite resource—use it only when you must. Security's job is to work with development and operations to help them to deliver, but deliver safely. This requires security to be practical and make realistic trade-offs when it comes to security findings. Is this problem serious enough that it needs to block code from shipping now? Can it be fixed later? Or, does it really need to be fixed at all? Understand the real risk to the system and to the organization and deal with problems appropriately. By not crying wolf, the security team knows that when serious problems do come up, they will be taken seriously by everyone.

Etsy's security team takes a number of steps to build relationships between the security team and engineering.

"Designated Hackers" is a system by which each security engineer supports four or five development teams across the organization and are involved in design and standups. The security engineer tries to understand what these teams are doing and raise a signal if a security risk or question comes up that needs to be resolved. They act as a channel and advocate between security and the development teams. This helps to build relationships, and builds visibility into design and early stage decisions—when security matters most.

Every new engineering hire participates in one-week boot camps where they can choose to work with the security team to understand what they do and help to solve problems. And each year every engineer does a senior rotation where they spend a month with another team and can choose to work with the security team. These initiatives build understanding and relationships between organizations and seed security champions in engineering.[1]

1 Rich Smith, Director of Security Engineering, Etsy. "Crafting an Effective Security Organization." QCon 2015 *http://www.infoq.com/presentations/security-etsy*

Shift Security Left

To keep up with the pace of Continuous Delivery, security must "shift left," earlier into design and coding and into the automated test cycles, instead of waiting until the system is designed and built and then trying to fit some security checks just before release. In DevOps, security must fit into the way that engineers think and work: more iterative and incremental, and automated in ways that are efficient, repeatable, and easy to use. See Figure 3-1 for a comparison between waterfall delivery and the DevOps cycle.

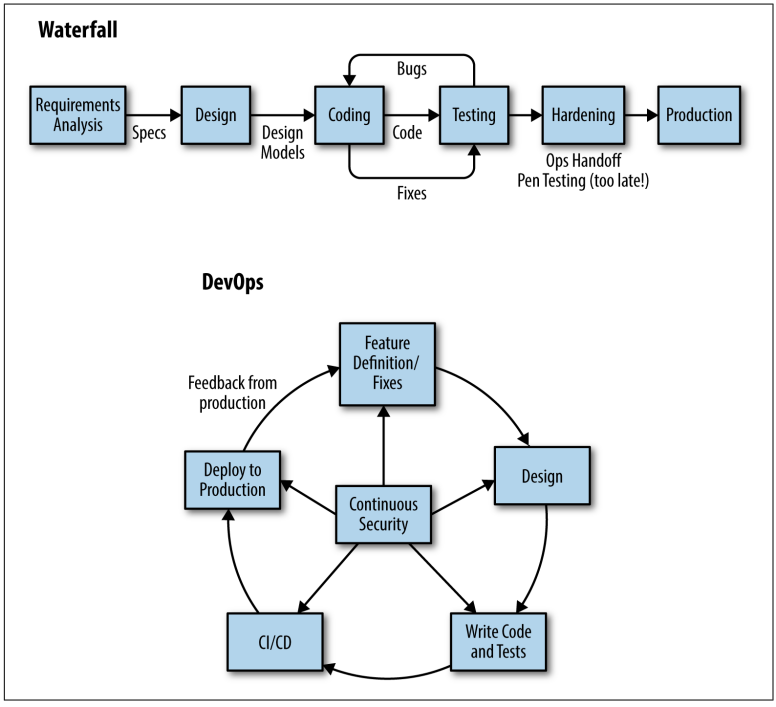

Figure 3-1. The waterfall cycle versus the DevOps cycle

Some organizations do this by embedding infosec specialists into development and operations teams. But it is difficult to scale this way. There are too few infosec engineers to go around, especially ones who can work at the design and code level. This means that developers and operations need to be given more responsibility for security, training in security principles and practices, and tools to help them build and run secure systems.

Developers need to learn how to identify and mitigate security risks in design through threat modeling (looking at holes or weaknesses in the design from an attacker's perspective), and how to take advantage of security features in their application frameworks and security libraries to prevent common security vulnerabilities like injection attacks. The OWASP and SAFECode communities provide a lot of useful, free tools and frameworks and guidance to help developers with understanding and solving common application security problems in any kind of system.

OWASP Proactive Controls

The OWASP Proactive Controls is a set of secure development practices, guidelines, and techniques that you should follow to build secure applications. These practices will help you to shift security earlier into design, coding, and testing:

Verify for security early and often
: Instead of leaving security testing and checks to the end of a project, include security in automated testing in Continuous Integration and Continuous Delivery.

Parameterize queries
: Prevent SQL injection by using a parameterized database interface.

Encode data
: Protect the application from XSS and other injection attacks by safely encoding data before passing it on to an interpreter.

Validate all inputs
: Treat all data as untrusted. Validate parameters and data elements using white listing techniques. Get to know and love regex.

Implement identity and authentication controls
: Use safe, standard methods and patterns for authentication and identity management. Take advantage of OWASP's Cheat Sheets for authentication, session management, password storage, and forgotten passwords if you need to build these functions in on your own.

Implement appropriate access controls
: Follow a few simple rules when implementing access control filters. Deny access by default. Implement access control in a central filter library—don't hardcode access control checks throughout the application. And remember to code to the activity instead of to the role.

Protect data
: Understand how to use crypto properly to encrypt data at rest and in transit. Use proven encryption libraries like Google's KeyCzar and Bouncy Castle.

Implement logging and intrusion detection
: Design your logging strategy with intrusion detection and forensics in mind. Instrument key points in your application and make logs safe and consistent.

Take advantage of security frameworks and libraries
: Take advantage of the security features of your application framework where possible, and fill in with special-purpose security libraries like Apache Shiro or Spring Security where you need to.

Error and exception handling
: Pay attention to error handling and exception handling throughout your application. Missed error handling can lead to runtime problems, including catastrophic failures. Leaking information in error handling can provide clues to attackers; don't make their job easier than it already is.

Secure by Default

Shifting Security Left begins by making it easy for engineers to write secure code and difficult for them to make dangerous mistakes, wiring secure defaults into their templates and frameworks, and building in the proactive controls listed previously. You can prevent SQL injection at the framework level by using parameterized queries, hide or simplify the output encoding work needed to protect applications from XSS attacks, enforce safe HTTP headers, and provide simple and secure authentication functions. You can do all of this in ways that are practically invisible to the developers using the framework.

Making software and software development secure by default is core to the security programs at organizations like Google, Facebook, Etsy, and Netflix. Although the pay back can be huge, it demands a fundamental change in the way that infosec and development work together. It requires close collaboration between security engineers and software engineers, strong application security knowledge and software design, and coding skills to build security protection into frameworks and templates in ways that are safe and easy to use. Most important, it requires a commitment from developers and their managers to use these frameworks wherever possible.

Most of us aren't going to be able to start our application security programs here; instead, we'll need to work our way back to the beginning and build more security into later stages.

Making Security Self-Service

Engineers in DevOps shops work in a self-service environment. Automated Continuous Integration servers provide self-service builds and testing. Cloud platforms and virtualization technologies like Vagrant provide self-service, on-demand provisioning. Container technologies like Docker offer simple, self-service packaging and shipping.

Security needs to be made available to the team in the same way: convenient, available when your engineers need it, seamless, and efficient.

Don't get in their way. Don't make developers wait for answers or stop work in order to get help. Give them security tools that they can use and understand and that they can provision and run themselves. And ensure that those tools fit into how they work: into Continuous Integration and Continuous Delivery, into their IDEs as they enter code, into code pull requests. In other words, ensure that tests and checks provide fast, clear feedback.

At Twitter, security checks are run automatically every time a developer makes a change. Tools like Brakeman check the code for security vulnerabilities, and provide feedback directly to the developer if something is found, explaining what is wrong, why it is wrong, and how to fix it. Developers have a "bullshit" button to reject false posi-

tive findings. Security checks become just another part of their coding cycle.[2]

Using Infrastructure as Code

Another key to DevOps is *Infrastructure as Code*: managing infrastructure configuration in code using tools like Ansible, Chef, Puppet, and Salt. This speeds up and simplifies provisioning and configuring systems, especially at scale. It also helps to ensure consistency between systems and between test and production, standardizing configurations and minimizing configuration drift, and reduces the chance of an attacker finding and exploiting a one-off mistake.

Treating infrastructure provisioning and configuration as a software problem and following software engineering practices like version control, peer code reviews, refactoring, static analysis, automated unit testing and Continuous Integration, and validating and deploying changes through the same kind of Continuous Delivery pipelines makes configuration changes repeatable, auditable, and safer.

Security teams also can use Infrastructure as Code to program security policies directly into configuration, continuously audit and enforce policies at runtime, and to patch vulnerabilities quickly and safely.

2 "Put your Robots to Work: Security Automation at Twitter." OWASP AppSec USA 2012, *https://vimeo.com/54250716*

UpGuard: From Infrastructure to Secure Code

UpGuard is a service that helps you to automatically capture configuration information into tests and code, and then establish policies and monitor compliance against those policies. UpGuard discovers configuration details from Linux and Windows servers, network devices and cloud services, and tracks changes to this information over time. You can use it as a Tripwire-like detective change control tool to alert you to unauthorized changes to configuration or to audit configuration management activities.

You can also visualize the configuration of your systems and identify inconsistencies between them. And you can establish policies for different systems or types of systems by creating fine-grained automated tests or assertions to ensure that the proper versions of packages are installed, that specific files and users and directories are set up correctly, that ports are opened or closed, and that processes are running.

UpGuard automatically creates directives for configuration management tools, including Ansible, Chef, Puppet, Microsoft Windows PowerShell DSC, and Docker, to bring your infrastructure configuration into code with a prebuilt test framework.

It continuously assesses the risk of your configuration, assigning a score based on compliance (test coverage and pass ratios as well as compliance with external benchmarks like CIS), integrity (tracking of unauthorized changes to configuration), and security (based on scanning for vulnerabilities—UpGuard includes a community-based vulnerability scanner and can integrate with other scanners).

Iterative, Incremental Change to Contain Risks

DevOps and Continuous Delivery reduce the risk of change by making many small, incremental changes instead of a few "big bang" changes.

Changing more often exercises and proves out your ability to test and successfully push out changes, enhancing your confidence in your build and release processes. Additionally, it forces you to automate and streamline these processes, including configuration man-

agement and testing and deployment, which makes them more repeatable, reliable, and auditable.

Smaller, incremental changes are safer by nature. Because the scope of any change is small and generally isolated, the "blast radius" of each change is contained. Mistakes are also easier to catch because incremental changes made in small batches are easier to review and understand upfront, and require less testing.

When something does go wrong, it is also easier to understand what happened and to fix it, either by rolling back the change or pushing a fix out quickly using the Continuous Delivery pipeline.

It's also important to understand that in DevOps many changes are rolled out *dark*. That is, they are disabled by default, using runtime "feature flags" or "feature toggles". These features are usually switched on only for a small number of users at a time or for a short period, and in some cases only after an "operability review" or pre-mortem review to make sure that the team understands what to watch for and is prepared for problems.[3]

Another way to minimize the risk of change in Continuous Delivery or Continuous Deployment is *canary releasing*. Changes can be rolled out to a single node first, and automatically checked to ensure that there are no errors or negative trends in key metrics (for example, conversion rates), based on "the canary in a coal mine" metaphor. If problems are found with the canary system, the change is rolled back, the deployment is canceled, and the pipeline shut down until a fix is ready to go out. After a specified period of time, if the canary is still healthy, the changes are rolled out to more servers, and then eventually to the entire environment.

Use the Speed of Continuous Delivery to Your Advantage

The speed at which DevOps moves can seem scary to infosec analysts and auditors. But security can take advantage of the speed of delivery to respond quickly to security threats and deal with vulnerabilities.

3 *http://www.slideshare.net/jallspaw/go-or-nogo-operability-and-contingency-planning-at-etsycom*

A major problem that almost all organizations face is that even when they know that they have a serious security vulnerability in a system, they can't get the fix out fast enough to stop attackers from exploiting the vulnerability.

The longer vulnerabilities are exposed, the more likely the system will be, or has already been, attacked. WhiteHat Security, which provides a service for scanning websites for security vulnerabilities, regularly analyzes and reports on vulnerability data that it collects. Using data from 2013 and 2014, WhiteHat found that 35 percent of finance and insurance websites are "always vulnerable," meaning that these sites had at least one serious vulnerability exposed every single day of the year. The stats for other industries and government organizations were even worse. Only 25 percent of finance and insurance sites were vulnerable for less than 30 days of the year. On average, serious vulnerabilities stayed open for 739 days, and only 27 percent of serious vulnerabilities were fixed at all, because of the costs and risks and overhead involved in getting patches out.

Continuous Delivery, and collaboration between developers and operations and infosec staff working closely together, can close vulnerability windows. Most security patches are small and don't take long to code. A repeatable, automated Continuous Delivery pipeline means that you can figure out and fix a security bug or download a patch from a vendor, test to make sure that it won't introduce a regression, and get it out quickly, with minimal cost and risk. This is in direct contrast to "hot fixes" done under pressure that have led to failures in the past.

Speed also lets you make meaningful risk and cost trade-off decisions. Recognizing that a vulnerability might be difficult to exploit, you can decide to accept the risk temporarily, knowing that you don't need to wait for several weeks or months until the next release, and that the team can respond quickly with a fix if it needs to.

Speed of delivery now becomes a security advantage instead of a source of risk.

The Honeymoon Effect

There appears to be another security advantage to moving fast in DevOps. Recent research shows that smaller, more frequent changes can make systems safer from attackers by means of the "Honey-

moon Effect": older software that is more vulnerable is easier to attack than software that has recently been changed.

Attacks take time. It takes time to identify vulnerabilities, time to understand them, and time to craft and execute an exploit. This is why many attacks are made against legacy code with known vulnerabilities. In an environment where code and configuration changes are rolled out quickly and changed often, it is more difficult for attackers to follow what is going on, to identify a weakness, and to understand how to exploit it. The system becomes a moving target. By the time attackers are ready to make their move, the code or configuration might have already been changed and the vulnerability might have been moved or closed.

To some extent relying on change to confuse attackers is "security through obscurity," which is generally a weak defensive position. But constant change should offer an edge to fast-moving organizations, and a chance to hide defensive actions from attackers who have gained a foothold in your system, as Sam Guckenheimer at Microsoft explains:

> If you're one of the bad guys, what do you want? You want a static network with lots of snowflakes and lots of places to hide that aren't touched. And if someone detects you, you want to be able to spot the defensive action so that you can take countermeasures.... With DevOps, you have a very fast, automated release pipeline, you're constantly redeploying. If you are deploying somewhere on your net, it doesn't look like a special action taken against the attackers.

CHAPTER 4

Security as Code: Security Tools and Practices in Continuous Delivery

Security as Code is about building security into DevOps tools and practices, making it an essential part of the tool chains and workflows. You do this by mapping out how changes to code and infrastructure are made and finding places to add security checks and tests and gates without introducing unnecessary costs or delays.

Security as Code uses Continuous Delivery as the control backbone and the automation engine for security and compliance. Let's begin by briefly defining Continuous Delivery, and then walk through the steps on how to build security into Continuous Delivery.

Continuous Delivery

Agile ideas and principles—working software over documentation, frequent delivery, face-to-face collaboration, and a focus on technical excellence and automation—form the foundation of DevOps. And Continuous Delivery, which is the control framework for DevOps, is also built on top of a fundamental Agile development practice: *Continuous Integration*.

In Continuous Integration, each time a developer checks in a code change, the system is automatically built and tested, providing fast and frequent feedback on the health of the code base. Continuous Delivery takes this to the next step.

Continuous Delivery is not just about automating the build and unit testing, which are things that the development team already owns. Continuous Delivery is provisioning and configuring test environments to match production as closely as possible—automatically. This includes packaging the code and deploying it to test environments; running acceptance, stress, and performance tests, as well as security tests and other checks, with pass/fail feedback to the team, all automatically; and auditing all of these steps and communicating status to a dashboard. Later, you use the same pipeline to deploy the changes to production.

Continuous Delivery is the backbone of DevOps and the engine that drives it. It provides an automated framework for making software and infrastructure changes, pushing out software upgrades, patches, and changes to configuration in a way that is repeatable, predictable, efficient, and fully audited.

Putting a Continuous Delivery pipeline together requires a high degree of cooperation between developers and operations, and a much greater shared understanding of how the system works, what production really looks like, and how it runs. It forces teams to begin talking to one another, exposing and exploring details about how they work and how they want to work.

There is a lot of work that needs to be done: understanding dependencies, standardizing configurations, and bringing configuration into code; cleaning up the build (getting rid of inconsistencies, hard-coding, and jury rigging); putting everything into version control—application code and configuration, binary dependencies, infrastructure configuration (recipes, manifests, playbooks, CloudFormation templates, and Dockerfiles), database schemas, and configurations for the Continuous Integration/Continuous Delivery pipeline itself; and, finally, automating testing (getting all of the steps for deployment together and automating them carefully). And you may need to do all of this in a heterogeneous environment, with different architectures and technology platforms and languages.

Continuous Delivery at London Multi-Asset Exchange

The London Multi-Asset Exchange (LMAX) is a highly regulated FX retail market in the United Kingdom, where Dave Farley (coauthor of the book *Continuous Delivery*) helped pioneer the model of Continuous Delivery.

LMAX's systems were built from scratch following Agile best practices: TDD, pair programming, and Continuous Integration. But they took this further, automatically deploying code to integration, acceptance, and performance testing environments, building up a Continuous Delivery pipeline.

LMAX has made a massive investment in automated testing. Each build runs through 25,000 unit tests with code coverage failure, simple code analysis (using tools like Findbugs, PMD, and custom architectural dependency checks) and automated integration sanity checks. All of these tests and checks must pass for every piece of code submitted.

The last good build is automatically picked up and promoted to integration and acceptance testing, during which more than 10,000 end-to-end tests are run on a test cluster, including API-level acceptance tests, multiple levels of performance tests, and fault injection tests that selectively fail parts of the system and verify that the system recovers correctly without losing data. More than 24 hours' worth of tests are run in parallel in less than 1 hour.

If all of the tests and reviews pass, the build is tagged. All builds are kept in a secure repository, together with dependent binaries (like the Java Runtime). Code and tests are tracked in version control.

QA can take a build to conduct manual exploratory testing or other kinds of tests. Operations can then pull a tagged build from the development repository to their separate secure production repository and use the same automated tools to deploy to production. Releases to production are scheduled every two weeks, on a Saturday, outside of trading hours.

This is Continuous Delivery, not Continuous Deployment as followed at Amazon or Etsy. But it still takes advantage of the same type of automation and controls, even though LMAX created a lot of

the tooling on its own using scripts and simple workflow conventions, before today's DevOps tools were available.

Injecting Security into Continuous Delivery

Before you can begin adding security checks and controls, you need to understand the workflows and tools that the engineering teams are using:

- What happens before and when a change is checked in?
- Where are the repositories? Who has access to them?
- How do changes transition from check-in to build to Continuous Integration and unit testing, to functional and integration testing, and to staging and then finally to production?
- What tests are run? Where are the results logged?
- What tools are used? How do they work?
- What manual checks or reviews are performed and when?

And how can you take advantage of all of this for security and compliance purposes?

Let's map out the steps involved from taking a change from check-in to production and identify where we can insert security checks and controls. See Figure 4-1 for a model that explains how and where to add security checks into a Continuous Delivery workflow.

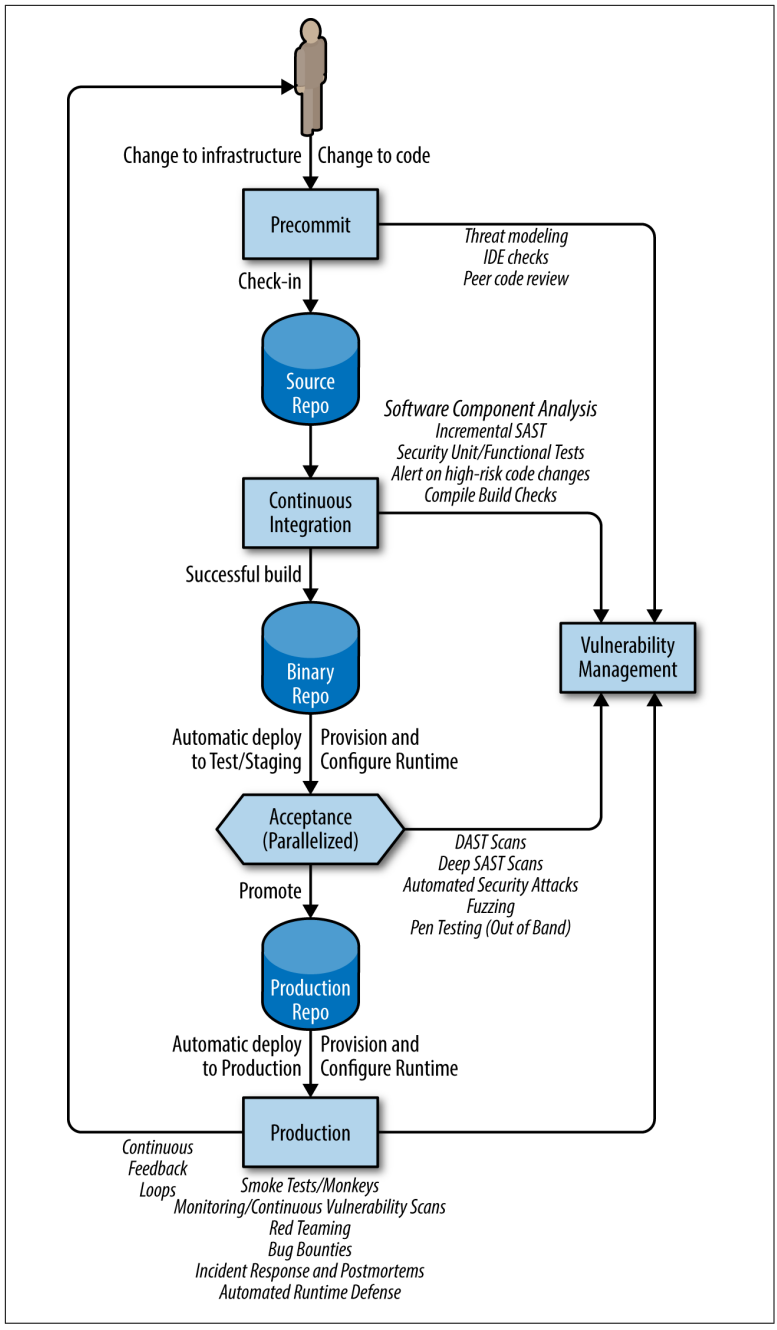

Figure 4-1. Security checks and controls in engineering workflows

Precommit

These are the steps before and until a change to software or configuration is checked in to the source code repo. Additional security checks and controls to be added here include the following:

- Lightweight, iterative threat modeling and risk assessments
- Static analysis (SAST) checking in the engineer's IDE
- Peer code reviews (for defensive coding and security vulnerabilities)

Commit Stage (Continuous Integration)

This is automatically triggered by a check in. In this stage, you build and perform basic automated testing of the system. These steps return fast feedback to developers: did this change "break the build"? This stage needs to complete in at most a few minutes. Here are the security checks that you should include in this stage:

- Compile and build checks, ensuring that these steps are clean, and that there are no errors or warnings
- Software Component Analysis in build, identifying risk in third-party components
- Incremental static analysis scanning for bugs and security vulnerabilities
- Alerting on high-risk code changes through static analysis checks or tests
- Automated unit testing of security functions, with code coverage analysis
- Digitally signing binary artifacts and storing them in secure repositories[1]

1 For software that is distributed externally, this should involve signing the code with a code-signing certificate from a third-party CA. For internal code, a hash should be enough to ensure code integrity.

Acceptance Stage

This stage is triggered by a successful commit. The latest good commit build is picked up and deployed to an acceptance test environment. Automated acceptance (functional, integration, performance, and security) tests are executed. To minimize the time required, these tests are often fanned out to different test servers and executed in parallel. Following a "fail fast" approach, the more expensive and time-consuming tests are left until as late as possible in the test cycle, so that they are only executed if other tests have already passed.

Security controls and tests in this stage include the following:

- Secure, automated configuration management and provisioning of the runtime environment (using tools like Ansible, Chef, Puppet, Salt, and/or Docker). Ensure that the test environment is clean and configured to match production as closely as possible.
- Automatically deploy the latest good build from the binary artifact repository.
- Smoke tests (including security tests) designed to catch mistakes in configuration or deployment.
- Targeted dynamic scanning (DAST).
- Automated functional and integration testing of security features.
- Automated security attacks, using Gauntlt or other security tools.
- Deep static analysis scanning (can be done out of band).
- Fuzzing (of APIs, files). This can be done out of band.
- Manual pen testing (out of band).

Production Deployment and Post-Deployment

If all of the previous steps and tests pass, the change is ready to be deployed to production, pending manual review/approvals and scheduling (in Continuous Delivery) or automatically (in Continuous Deployment). Additional security checks and controls are needed in production deployment and post-deployment:

- Secure, automated configuration management and provisioning of the runtime environment
- Automated deployment and release orchestration (authorized, repeatable, and auditable)
- Post-deployment smoke tests
- Automated runtime asserts and compliance checks (monkeys)
- Production monitoring/feedback
- Runtime defense
- Red Teaming
- Bug bounties
- Blameless postmortems (learning from failure)

Depending on the risk profile of your organization and systems, you will need to implement at least some of these practices and controls. Leaders in this space already do most of them.

Now, let's look more closely at these security controls and practices and some of the tools that you can use, starting with design.

Secure Design in DevOps

Secure design in DevOps begins by building on top of secure libraries and frameworks—building security in upfront and trying to make it invisible to developers. Security risk assessments also need to be integrated into design as it changes and as part of managing the software supply chain: the open source and third-party components and frameworks that teams use to assemble important parts of any system.

Risk Assessments and Lightweight Threat Modeling

We've already looked at the essential problem of design in rapidly moving DevOps environments. These teams want to deliver to real users early and often so that they can refine the feature set and the design in response to production feedback. This means that the design must be lightweight at the outset, and it is constantly changing based on feedback.

In Continuous Deployment, there is no Waterfall handoff of design specifications to coders—there may not be any design specifications

at all that can be reviewed as part of a risk assessment. When there is minimal design work being done, and "the code is the design," where and how do you catch security problems in design?

You begin upfront by understanding that even if the design is only roughed out and subject to change, the team still needs to commit to a set of tools and the runtime stack to get going. This is when threat modeling—looking at the design from an attacker's perspective, searching for gaps or weaknesses in security controls and defenses—needs to start.

At PayPal, for example, every team must go through an initial risk assessment, filling out an automated risk questionnaire whenever it begins work on a new app or microservice.[2] One of the key decision points is whether the team is using existing languages and frameworks that have already been vetted.[3] Or, are they introducing something new to the organization, technologies that the security team hasn't seen before? There is a big difference in risk between "just another web or mobile app" built on an approved platform, and a technical experiment using new languages and tools.

Here are some of the issues to understand and assess in an upfront risk review:

- Do you understand how to use the language(s) and frameworks safely? What security protections are offered in the framework? What needs to be added to make it simple for developers to "do the right thing" by default?
- Is there good Continuous Delivery toolchain support for the language(s) and framework, including SAST checking or IAST tooling, and dependency management analysis capabilities to catch vulnerabilities in third-party and open source libraries?
- Is sensitive and confidential data being used? What data is needed, how is it to be handled, and what needs to be audited? Does it need to be stored, and, if so, how? Do you need to make

2 "Agile Security – Field of Dreams." Laksh Raghavan, PayPal, RSA Conference 2016. *https://www.rsaconference.com/events/us16/agenda/sessions/2444/agile-security-field-of-dreams*

3 At Netflix, where they follow a similar risk-assessment process, this is called "the paved road," because the path ahead should be smooth, safe, and predictable.

considerations for encryption, tokenization and masking, access control, and auditing?

- Do you understand the trust boundaries between this app/service and others: where do controls need to be enforced around authentication, access control, and data quality? What assumptions are being made in the design?

These questions are especially important in microservices environments, in which teams push for flexibility to use the right tool for the job: when it is not always easy to understand call-chain dependencies—when you can't necessarily control who calls you and what callers expect from your service, and when you can't control what downstream services do, or when or how they will be changed. For microservices, you need to understand the following:

- What assumptions are you making about callers? Where and how is authentication and authorization done? How can you be sure?
- Can you trust the data that you are getting from another service? Can other services trust the data that you are providing to them?
- What happens if a downstream service fails, or times out, or returns an incomplete or inconsistent result?

After the upfront assessment, threat modeling should become much simpler for most changes, because most changes will be small, incremental, and therefore low risk. You can assess risk inexpensively, informally, and iteratively by getting the team to ask itself a few questions as it is making changes:

- Are you changing anything material about the tooling or stack? Are you introducing or switching to a new language, changing the backend store, or upgrading or replacing your application framework? Because design is done fast and iteratively, teams might find that their initial architectural approach does not hold up, and they might need to switch out all or part of the technology platform. This can require going back and reassessing risk from the start.
- How are you affecting the attack surface of the system? Are you just adding a new field or another form? Or, are you opening up

new ports or new APIs, adding new data stores, making calls out to new services?

- Are you changing authentication logic or access control rules or other security plumbing?
- Are you adding data elements that are sensitive or confidential? Are you changing code that has anything to do with secrets or sensitive or confidential data?

Answering these questions will tell you when you need to look more closely at the design or technology, or when you should review and verify trust assumptions. The key to threat modeling in DevOps is recognizing that because design and coding and deployment are done continuously in a tight, iterative loop, you will be caught up in the same loops when you are assessing technical risks. This means that you can make—and you need to make—threat modeling efficient, simple, pragmatic, and fast.

Securing Your Software Supply Chain

Another important part of building in security upfront is to secure your software supply chain, minimizing security risks in the software upon which your system is built. Today's Agile and DevOps teams take extensive advantage of open source libraries to reduce development time and costs. This means that they also inherit quality problems and security vulnerabilities from other people's code.

According to Sonatype, which runs the Central Repository, the world's largest repository for open source software

> 80 percent of the code in today's applications comes from libraries and frameworks

and a lot of this code has serious problems in it. Sonatype looked at 17 billion download requests from 106,000 different organizations in 2014. Here's what it found:

> Large software and financial services companies are using an average of 7,600 suppliers. These companies sourced an average of 240,000 software "parts" in 2014, of which 15,000 included known vulnerabilities.

More than 50,000 of the software components in the Central Repository have known security vulnerabilities. One in every 16 download requests is for software that has at least one known security vulnera-

bility. On average, 50 new critical vulnerabilities in open source software are reported every day.

Scared yet? You should be. You need to know what open source code is included in your apps and when this changes, and review this code for known security vulnerabilities.

Luckily, you can do this automatically by using Software Component Analysis (SCA) tools like OWASP's Dependency Check project or commercial tools like Sonatype's Nexus Lifecycle or SourceClear. You can wire these tools into your build or Continuous Integration/Continuous Delivery pipeline to automatically inventory open source dependencies, identify out-of-date libraries and libraries with known security vulnerabilities, and fail the build automatically if serious problems are found. By building up a bill of materials for every system, you can prepare for vulnerabilities like Heartbleed or DROWN—you can quickly determine if you are exposed and what you need to fix.

These tools also can alert you when new dependencies are detected so that you can create a workflow to review them.

OWASP Dependency Check

OWASP's Dependency Check is an open source scanner that catalogs open source components used in an application. It works for Java, .NET, Ruby (gemspec), PHP (composer), Node.js and Python, and some C/C++ projects. Dependency Check integrates with common build tools (including Ant, Maven, and Gradle) and CI servers like Jenkins.

Dependency Check reports on any components with known vulnerabilities reported in NIST's National Vulnerability Database and gets updates from the NVD data feeds.

Other popular open source dependency checking tools include the following:

- Bundler Audit for Ruby
- Retire.js for Javascript
- SafeNuGet for NuGet libraries

If you are using containers like Docker in production (or even in development and test) you should enforce similar controls over dependencies in container images. Even though Docker's Project Nautilus scans images in official repos for packages with known vulnerabilities, you should ensure that all Docker containers are scanned, using a tool like OpenSCAP or Clair, or commercial services from Twistlock, FlawCheck, or Black Duck Hub.

Your strategic goal should be to move to "fewer, better suppliers" over time, simplifying your supply chain in order to reduce maintenance costs and security risks. Sonatype has developed a free calculator that will help developers—and managers—understand the cost and risks that you inherit over time from using too many third-party components.[4].

But you need to recognize that even though it makes good sense in the long term, getting different engineering teams to standardize on using a set of common components won't be easy, especially for microservices environments in which developers are granted the freedom to use the right tools for the job, selecting technologies based on their specific requirements, or even on their personal interests.

Begin by standardizing on the lowest layers—the kernel, OS, and VMs—and on general-purpose utility functions like logging and metrics collection, which need to be used consistently across apps and services.

Writing Secure Code in Continuous Delivery

DevOps practices emphasize the importance of writing good code: code that works and that is easy to change. You can take advantage of this in your security program, using code reviews and adding automated static analysis tools to catch common coding mistakes and security vulnerabilities early.

4 Shannon Lientz, *http://www.devsecops.org/blog/2016/1/16/fewer-better-suppliers*

Using Code Reviews for Security

Peer code reviews are a common engineering practice at many Agile and DevOps shops, and they are mandatory at leading organizations like Google, Amazon, Facebook, and Etsy.

Peer code reviews are generally done to share information across the team and to help ensure that code is maintainable, to reinforce conventions and standards, and to ensure that frameworks and patterns are being followed correctly and consistently. This makes code easier to understand and safer to change, and in the process, reviewers often find bugs that need to be fixed.

You can also use code reviews to improve security in some important ways.

First, code reviews increase developer accountability and provide transparency into changes. Mandatory reviews ensure that a change can't be pushed out without at least one other person being aware of what was done and why it was done. This significantly reduces the risk of insider threats; for example, someone trying to introduce a logic bomb or a back door in the code. Just knowing that their code will be reviewed also encourages developers to be more careful in their work, improving the quality of the code.

Transparency into code reviews can be ensured using code review tools like

- Gerrit
- Phabricator
- Atlassian Crucible

Frameworks and other high-risk code including security features (authentication workflows, access control, output sanitization, crypto) and code that deals with money or sensitive data require careful, detailed review of the logic. This code must work correctly, including under error conditions and boundary cases.

Encouraging developers to look closely at error and exception handling and other defensive coding practices, including careful parameter validation, will go a long way to improving the security of most code as well as improving the runtime reliability of the system.

With just a little training, developers can learn to look out for bad practices like hardcoding credentials or attempts at creating custom crypto. With more training, they will be able to catch more vulnerabilities, early on in the process.

In some cases (for example, session management, secrets handling, or crypto), you might need to bring in a security specialist to examine the code. Developers can be encouraged to ask for security code reviews. You can also identify high-risk code through simple static code scanning, looking for specific strings such as credentials and dangerous functions like crypto functions and crypto primitives.

To identify high-risk code, Netflix maps out call sequences for microservices. Any services that are called by many other services or that fan out to many other services are automatically tagged as high risk. At Etsy, as soon as high-risk code is identified through reviews or scanning, they hash it and create a unit test that automatically alerts the security team when the code hash value has been changed.

Code review practices also need to be extended to infrastructure code—to Puppet manifests and Chef cookbooks and Ansible playbooks, Dockerfiles, and CloudFormation templates.

What About Pair Programming?

Pair programming, where developers write code together, one developer "driving" at the keyboard, and the other acting as the navigator, helping to guide the way and looking out for trouble ahead, is a great way to bring new team members up to speed, and it is proven to result in better, tighter, and more testable code. But pairing will miss important bugs, including security vulnerabilities, because pair programming is more about joint problem solving, navigating toward a solution rather than actively looking for mistakes or hunting for bugs.

Even in disciplined XP environments, you should do separate security-focused code reviews for high-risk code.

SAST: in IDE, in Continuous Integration/Continuous Delivery

Another way to improve code security is by scanning code for security vulnerabilities using automated static analysis software testing (SAST) tools. These tools can find subtle mistakes that reviewers

will sometimes miss, and that might be hard to find through other kinds of testing.

But rather than relying on a centralized security scanning factory run by infosec, DevOps organizations like Twitter and Netflix implement self-service security scanning for developers, fitting SAST scanning directly into different places along the engineering workflow.

Developers can take advantage of built-in checkers in their IDE, using plug-ins like FindBugs or Find Security Bugs, or commercial plug-ins from Coverity, Klocwork, HPE Fortify, Checkmarx, or Cigital's SecureAssist to catch security problems and common coding mistakes as they write code.

You can also wire incremental static analysis precommit and commit checks into Continuous Integration to catch common mistakes and antipatterns quickly by only scanning the code that was changed. Full system scanning might still be needed to catch interprocedural problems that some incremental scans can't find. You will need to run these scans, which can take several hours or sometimes days to run on a large code base, outside of the pipeline. But the results can still be fed back to developers automatically, into their backlog or through email or other notification mechanisms.

Different kinds of static code scanning tools offer different value:

- Tools that check for code consistency, maintainability, and clarity (PMD and Checkstyle for Java, Ruby-lint for Ruby) help developers to write code that is easier to understand, easier to change, easier to review, and safer to change.
- Tools that look for common coding bugs and bug patterns (tools like FindBugs and RuboCop) will catch subtle logic mistakes and errors that could lead to runtime failures or security vulnerabilities.
- Tools that identify security vulnerabilities through taint analysis, control flow and data flow analysis, pattern analysis, and other techniques (Find Security Bugs, Brakeman) can find many common security issues such as mistakes in using crypto functions, configuration errors, and potential injection vulnerabilities.

You should not rely on only one tool—even the best tools will catch only some of the problems in your code. Good practice would be to

run at least one of each kind to look for different problems in the code, as part of an overall code quality and security program.

There are proven SAST tools available today for popular languages like Java, C/C++, and C#, as well as for common frameworks like Struts and Spring and .NET, and even for some newer languages and frameworks like Ruby on Rails. But it's difficult to find tool support for other new languages such as Golang, and it's especially difficult to find for dynamic scripting languages. Most static analyzers, especially open source tools, for these languages are still limited to linting and basic checking for bad practices, which helps to make for better code but aren't enough to ensure that your code is secure.

Static analysis checking tools for configuration management languages (like Foodcritic for Chef or puppet-lint for Puppet) are also limited to basic checking for good coding practices and some common semantic mistakes. They help to ensure that the code works, but they won't find serious security problems in your system configuration.

SonarQube

SonarQube wraps multiple SAST scanning tools for multiple languages. It originally wrapped open source tools, and now includes proprietary checkers written by SonarSource. Some of these checkers, for languages like Objective-C and Swift, C/C++, and other legacy languages are only available in the commercial version. But there is good support in the open source version of SonarQube for Java, JavaScript, PHP, and other languages like Erlang.

One of SonarQube's main purposes is to assess and track technical debt in applications. This means that most of the code checkers are focused on maintainability (for style and coding conventions) as well as for coding correctness and common bug patterns. However, SonarSource has recently started to include more security-specific checkers, especially for Java.

SonarQube runs in Continuous Integration/Continuous Delivery, with plug-ins for Jenkins and GitHub. You can set quality gates and automatically notify the team when the quality gates fail. It collects metrics and provides reports and dashboards to analyze these metrics over time, to identify where bugs are clustered and to compare metrics across projects.

To ensure that the feedback loops are effective, it's important to tune these tools to minimize false positives and provide developers with clear, actionable feedback on real problems that need to be fixed. Noisy checkers that generate a lot of false positives and that need review and triage can still be run periodically and the results fed back to development after they have been picked through.

Security Testing in Continuous Delivery

Security testing needs to be moved directly into Continuous Integration and Continuous Delivery in order to verify security as soon as changes are made. This could mean wiring application scanning and fuzzing into the Continuous Delivery pipeline. It could also mean taking advantage of work that the development team has already done to create an automated test suite, adding security checks into unit testing, and automating security attacks as part of integration and functional testing.

Although you need to run penetration tests and bug bounty programs outside of Continuous Delivery, they still provide valuable feedback into the automated testing program. You need to track all of the vulnerabilities found in scanning and testing—inside and outside of the pipeline, in a vulnerability manager.

Dynamic Scanning (DAST)

Black box Dynamic Analysis Security Testing (DAST) tools and services are useful for testing web and mobile apps, but they don't always play nicely in Continuous Integration or Continuous Delivery. Most DAST tools are designed to be run by security analysts or pen testers, not a Continuous Integration engine like Jenkins or Bamboo.

You can use tools like OWASP ZAP to automatically scan a web app for common vulnerabilities as part of the Continuous Integration/Continuous Delivery pipeline. You can do this by running the scanner in headless mode through the command line, through the scanner's API, or by using a wrapper of some kind, such as the ZAProxy Jenkins plug-in or a higher-level test framework like BDD-Security (which we'll look at in a later section).

There is no definitive guidance (yet... the ZAP project team is working on some) on how to best integrate scanning into Continuous

Delivery—you'll need to explore this on your own. You can try to spider the app (if it is small enough), but it generally makes more sense in Continuous Integration and Continuous Delivery to target your scans in order to reduce the amount of time needed to execute the tests and minimize the amount of noise created. You can do this by proxying automated functional regression tests executed by tools like Selenium through the scanner in order to map out and navigate key forms and fields to be scanned. Then, invoke the scanner's API and instruct the scanner to execute its fuzzing attacks.

Then, you will need to pick through the results, filter out the background noise, and determine what results constitute a pass or fail and whether you need to stop the pipeline. As with static analysis tools, you will need to tune dynamic scans to minimize false positives. You will want to set "the bug bar" high enough to ensure that you are not wasting the development team's time.

You will also need to remove duplicate findings after each scan. Tools like Code Dx or ThreadFix (which we will also look at in a later section) can help you to do this.

Like static analysis scans, dynamic analysis checking takes time, and will probably need to be spun off to run in parallel with other tests, or even done out of band.

Fuzzing and Continuous Delivery

Another testing technique that can be valuable in finding security vulnerabilities (especially injection bugs) is *fuzzing*. Fuzzing is a brute-force reliability testing technique wherein you create and inject random data into a file or API in order to intentionally cause errors and then see what happens. Fuzz testing is important in embedded systems development for which the costs of mistakes are high, and it has been a fundamental part of application security testing programs at Microsoft, Facebook, Adobe, and Google.[5] Fuzz testing tools are also commonly used by security researchers to hunt for bugs.

5 "Fuzzing at Scale." Google Security Blog. *https://security.googleblog.com/2011/08/fuzzing-at-scale.html*

However, fuzzing, like scanning, doesn't fit nicely into Continuous Integration and Continuous Delivery automation for a number of reasons:

- Fuzz tests are generally not predictable or repeatable. The nature of fuzzing is to try random things and see what happens.
- The results of fuzz testing are also not predictable. The system might crash or some kind of exception might occur that can leave the system in an undefined state for future testing.
- The results of fuzzing can be difficult to assess and understand and might require a manual reviewer to identify and qualify problems and recognize when multiple problems have the same cause.
- Good fuzzing takes time—hours or days—to complete and lots of CPU cycles, which makes it difficult to fit into the time box of Continuous Delivery.

Some newer fuzzing tools are designed to run (or can be adapted to run) in Continuous Integration and Continuous Delivery. They let you to seed test values to create repeatable tests, set time boxes on test runs, detect duplicate errors, and write scripts to automatically set up/restore state in case the system crashes. But you might still find that fuzz testing is best done out of band.

Security in Unit and Integration Testing

Continuous Integration and Continuous Delivery and especially practices like Behavior-Driven Development (BDD) and Test-Driven Development (TDD)—wherein developers write tests before they write the code—encourage developers to create a strong set of automated tests to catch mistakes and protect themselves from regressions as they add new features or make changes or fixes to the code.

Most of these tests will be positive, happy-path tests which prove that features work as expected. This is the way that most developers think and it is what they are paid to do. But this means that they often miss testing for edge cases, exceptions, and negative cases that could catch security problems.

And most of these automated tests, following automated "testing pyramid" conventions, will be low-level unit tests, written by devel-

opers to validate detailed logic within a method or function. Unit tests are important in catching regressions as you refactor code or make other changes, but they won't find mistakes or misunderstandings made in calling functions or services, like not calling a function at all, which are a common cause of security vulnerabilities and can only be caught in higher-level functional or integration tests.

For security code and framework code and for other high-risk functions, you should convince developers to step off the happy path and write good unit and functional and integration tests around—and especially outside—boundary conditions. They need to test error handling and exception handling logic, and write negative tests: sanity tests that should never pass unless something has gone wrong. Insist on high levels of automated test coverage for high-risk code.

Spend some time with the team to come up with abuse(r) or "evil user" stories, "misuse cases" that explore how a bad user could try to take advantage of a feature, or what could happen if they stray off of the main success scenarios. This doesn't necessarily require specialized security knowledge; you can accomplish a lot by just asking

> But, what happens if the user doesn't...?

Then, write negative tests for these cases, tests which prove that unauthenticated users can't access admin functions, that a user can't see or change information for a different account, that they can't tamper with a return value, and so on. If these tests fail, something is seriously broken and you want to learn about it as early as possible.

Automated Attacks

Even with these tests in place, you should still go further and try to attack your system. Bad guys are going to attack your system, if they haven't done so already. You should try to beat them to it.

There are a few test frameworks that are designed to make this easy and that behave well in Continuous Integration and Continuous Delivery:

- Gauntlt
- Mittn
- BDD-Security

Using one of these tools, you will be able to set up and run a basic set of targeted automated pen tests against your system as part of your automated test cycle.

Gauntlt: Go Ahead, Be Mean to Your Code

Gauntlt is a popular open source test framework written in Ruby (and now also available in Golang) that uses Cucumber to describe simple security-related asserts or complex automated attack scenarios in a way that is easy for developers and auditors to follow.

Gauntlt wraps pen-testing and security-testing tools, abstracting the details of how they work and making them more accessible, and controlling them so that you can create repeatable, deterministic steps with clear pass/fail results.

You compose attack scenarios using Gherkin syntax, a high-level, English-like language that is essentially self-documenting, making it simple to describe and write tests:

Given
: Set up conditions

When
: Attack steps

Then
: Parse output, filter out noise, and check return values to determine pass/fail

Here is a simple attack from the Gauntlt site:

```
# nmap-simple.attack
Feature: simple nmap attack to check for open ports
   Background:
      Given "nmap" is installed
      And the following profile:
         | name | value |
         | hostname | example.com |
   Scenario: Check standard web ports
      When I launch an "nmap" attack with:
         """
         nmap -F <hostname>
         """
      Then the output should match /80.tcp\s+open/
      Then the output should not match:
         """
         25\/tcp\s+open
         """
```

The intention behind Gauntlt is to encourage developers, testers, and security specialists to collaborate on security testing, focusing more on what they want to test, and less on how to do the testing.

It comes prepackaged with attack adapters for different tools (curl, sslyze, sqlmap, garmr, arachni, dirb, heartbleed) that implement step definitions from Gherkin to executable code. It also includes sample attack files that you can customize and extend.

Some common tests that you can do using tools like Gauntlt include running nmap to check for open ports, checking that SSL is configured correctly, attempting SQL injections, and testing for high-severity vulnerabilities like Heartbleed.

Just as with automating integration testing or acceptance testing, it will take a while to build up a strong set of security tests in Continuous Delivery. Begin by building a post-deployment security smoke test, a basic regression test that will run in acceptance testing and in production to catch common and important security problems, and to ensure that security configurations are correct.

Pen Testing and Bug Bounties

Manual penetration testing is not effective as a control gate in Continuous Delivery or Continuous Deployment. The velocity of delivery is too fast, and pen tests take too long to set up, run, and review.

But there is still important value in pen testing out-of-band from the Continuous Delivery pipeline, not only to satisfy mandatory compliance requirements. More important, you can use the results of pen testing to validate your security program, highlighting strengths and weaknesses.

Good pen testing is exploratory and creative—unlike most of the automated testing in Continuous Delivery, which is intended to catch the same kinds of mistakes in design and coding and configuration, over and over. A good pen tester will help you to find problems that you wouldn't otherwise have known to look for or known how to find.

The real value in these tests is not in the bugs that they find; it's in the information that the bugs provide you, if you look deep enough. Where did the bug come from? Why did you miss finding it yourself? How did it get there in the first place? What do we need to

change or improve to prevent problems like this from happening again?

The same principle applies to Bug Bounties, which are part of the security programs at leading organizations like Google, Etsy, Netflix, and Facebook. Enlisting the community of security researchers to find security and reliability bugs in your software gives you access to creativity and skills that you couldn't afford otherwise. They will find some important bugs. Fixing these bugs will make your system safer.

But, more importantly, they will provide you with information on where you need to improve your design, coding, reviews, testing, and configuration management. This is information that you can use to get better as an organization and to build better and safer systems.

Using Pen Tests to Improve Automated Testing at LMAX

At The London Multi-Asset Exchange (LMAX), as part of its commitment to Continuous Delivery, one of the team's goals was to cover all of the system's behavior through automated testing, to the extent possible. This included not only functional and integration testing, but performance (scalability and latency), operational resiliency (through fault injection), and security.

As a regulated financial entity, LMAX underwent regular pen testing by external experts. The team saw pen tests not just as a checkmark, a compliance gate that they had to pass through, but also as a valuable learning opportunity—and a challenge. They tried to understand, and eventually to anticipate, what the pen testers were looking for and how they found security problems. Then, LMAX built this in to its own automated testing, to try to catch as many problems as possible on their own, before the system was deployed to production.[6].

6 Dave Farley (*http://www.continuous-delivery.co.uk/*), Interview March 17, 2016

Vulnerability Management

Infosec needs their own view into the pipeline and into the system, and across all of the pipelines and systems and portfolios, to track vulnerabilities, assess risk, and understand trends. You need metrics for compliance and risk-management purposes, to understand where you need to prioritize your testing and training efforts and to assess your application security program.

Collecting data on vulnerabilities lets you ask some important questions:

- How many vulnerabilities have you found?
- How were they found? What tools or testing approaches are giving you the best returns?
- What are the most serious vulnerabilities?
- How long are they taking to get fixed? Is this getting better or worse over time?

You can get this information by feeding security testing results from your Continuous Delivery pipelines into a vulnerability manager, such as Code Dx or ThreadFix.

ThreadFix

ThreadFix is a vulnerability management tool (available in open source and enterprise versions) that consolidates vulnerability information to provide a view into vulnerability risks and remediation across tools, pipelines, and apps—and over time. ThreadFix automatically takes vulnerability findings from SAST and DAST tools (and manual pen tests), deduplicates the results, and lets an analyst review and triage vulnerabilities and easily turn them into bug reports that can be fed back into a developer's bug tracking system or IDE. ThreadFix is designed to facilitate the feedback loop from testing to developers while providing analytical and reporting tools to security analysts so that they can compare vulnerability risks across a portfolio and track program performance such as time to remediation.

The open source engine includes integration with different testing tools, Continuous Integration/Continuous Delivery servers, and development toolsets. The commercial enterprise version offers

cross-portfolio analysis and reporting as well as scanner orchestration capabilities.

Securing the Infrastructure

In Continuous Delivery, the same practices, automated workflows, and controls that are used to build and deliver secure code are used to secure the infrastructure:

- Managing configuration as code (checking code into version control, ensuring that it is reviewed, scanning it for common mistakes)
- Building hardening policies into configuration code by default
- Using the Continuous Delivery pipeline to automatically test, deploy, and track configuration changes
- Securing the Continuous Delivery pipeline itself

Let's look at these ideas in some more detail.

Automated Configuration Management

Code-driven configuration management tools like Puppet, Chef, and Ansible make it easy to set up standardized configurations across hundreds of servers using common templates, minimizing the risk that hackers can exploit one unpatched server, and letting you minimize any differences between production, test, and development environments. All of the configuration information for the managed environments is visible in a central repository and under version control. This means that when a vulnerability is reported in a software component like OpenSSL, it is easy to identify which systems need to be patched. And it is easy to push patches out, too.

These tools also provide some host-based intrusion-detection capabilities and give you control over configuration drift: they continuously and automatically audit runtime configurations to ensure that every system matches the master configuration definition, issue alerts when something is missing or wrong, and can automatically correct it.

Security should be baked in to Amazon Machine Images (AMIs) and other configuration templates. Puppet manifests, Chef cook-

books, Ansible playbooks, and Dockerfiles should be written and reviewed with security in mind. Unit tests for configuration code should include security checks such as the following:

- Ensure that unnecessary services are disabled
- Ensure that ports that do not need to be open are indeed not open
- Look for hardcoded credentials and secrets
- Review permissions on sensitive files and directories
- Ensure that security tools like OSSEC or AIDE are installed and set up correctly
- Ensure that development tools are not installed in production servers
- Check auditing and logging policies and configurations

Build standard hardening steps into your recipes instead of using scripts or manual checklists. This includes minimizing the attack surface by removing all packages that aren't needed and that have known problems; and changing default configurations to be safe.

Security standards like the Center for Internet Security (CIS) benchmarks and NIST configuration checklists can be burned into Puppet and Chef and Ansible specifications. There are several examples of Puppet modules and Chef cookbooks available to help harden Linux systems against CIS benchmarks and the Defense Information Systems Agency Security Technical Implementation Guides.

Hardening.io

Hardening.io is an open source infrastructure hardening framework from Deutsche Telekom for Linux servers. It includes practical hardening steps for the base OS and common components such as ssh, Apache, nginx, mysql, and Postgres.

Hardening templates are provided for Chef and Puppet as well as Ansible (only base OS and ssh is currently implemented in Ansible). The hardening rules are based on Deutsche Telekom's internal guidelines, BetterCrypto, and the NSA hardening guide.

Securing Your Continuous Delivery Pipeline

It's important not only to secure the application and its runtime environment, but to secure the Continuous Delivery tool chain and build and test environments, too. You need to have confidence in the integrity of delivery and the chain of custody, not just for compliance and security reasons, but also to ensure that changes are made safely, repeatably, and traceably.

Your Continuous Delivery tool chain is also a dangerous attack target itself: it provides a clear path for making changes and pushing them automatically into production. If it is compromised, attackers have an easy way into your development, test, and production environments. They could steal data or intellectual property, inject malware anywhere into the environment, DoS your systems, or cripple your organization's ability to respond to an attack by shutting down the pipeline itself.

Continuous Delivery and Continuous Deployment effectively extend the attack surface of your production system to your build and automated test and deployment environment.

You also need to protect the pipeline from insider attacks by ensuring that all changes are fully transparent and traceable from end to end, that a malicious and informed insider cannot make a change without being detected, and that they cannot bypass any checks or validations.

Do a threat model on the Continuous Delivery pipeline. Look for weaknesses in the setup and controls, and gaps in auditing or logging. Then, take steps to secure your configuration management environment and Continuous Delivery pipeline:

- Harden the systems that host the source and build artifact repositories, the Continuous Integration and Continuous Delivery server(s), and the systems that host the configuration management, build, deployment, and release tools. Ensure that you clearly understand—and control—what is done on-premises and what is in the cloud.
- Harden the Continuous Integration and/or Continuous Delivery server. Tools like Jenkins are designed for developer convenience and are not secure by default. Ensure that these tools (and the required plug-ins) are kept up-to-date and tested frequently.

- Lock down and harden your configuration management tools. See "How to be a Secure Chef," for example.
- Ensure that keys, credentials, and other secrets are protected. Get secrets out of scripts and source code and plain-text files and use an audited, secure secrets manager like Chef Vault, Square's KeyWhiz project, or HashiCorp Vault.
- Secure access to the source and binary repos and audit access to them.
- Implement access control across the entire tool chain. Do not allow anonymous or shared access to the repos, to the Continuous Integration server, or confirmation manager or any other tools.
- Change the build steps to sign binaries and other build artifacts to prevent tampering.
- Periodically review the logs to ensure that they are complete and that you can trace a change through from start to finish. Ensure that the logs are immutable, that they cannot be erased or forged.
- Ensure that all of these systems are monitored as part of the production environment.

Security in Production

Security doesn't end after systems are in production. In DevOps, automated security checks, continuous testing, and monitoring feedback loops are integral parts of production operations.

Runtime Checks and Monkeys

If you are going to allow developers to do self-service, push-button deploys to production and you can't enforce detailed reviews of each change, you will need to add some runtime checking to catch oversights or shortcuts. This is what Jason Chan at Netflix calls moving "from gates to guardrails".

After each deploy, check that engineers used templates properly and that they didn't make a fundamental mistake in configuration or deployment that could open up the system to attack or make it less reliable under failure.

This is why Netflix created the Simian Army, a set of automated runtime checks and tests, including the famous Chaos Monkey.

Chaos Monkey, Chaos Gorilla, and Chaos Kong check that the system is set up and designed correctly to handle failures by randomly injecting failures into the production runtime, as part of Netflix's approach to *Chaos Engineering*.

The other monkeys are rule-driven compliance services that automatically monitor the runtime environment to detect changes and to ensure that configurations match predefined definitions. They look for violations of security policies and common security configuration weaknesses (in the case of Security Monkey) or configurations that do not meet predefined standards (Conformity Monkey). They run periodically online, notifying the owner(s) of the service and infosec when something looks wrong. The people responsible for the service need to investigate and correct the problem, or justify the situation.

Security Monkey captures details about changes to policies over time. It also can be used as an analysis and reporting tool and for forensics purposes, letting you search for changes across time periods and across accounts, regions, services, and configuration items. It highlights risks like changes to access control policies or firewall rules.

Similar tools include Amazon's AWS Inspector, which is a service that provides automated security assessments of applications deployed on AWS, scans for vulnerabilities, and checks for deviations from best practices, including rules for PCI DSS and other compliance standards. It provides a prioritized list of security issues along with recommendations on how to fix them.

Although checks like this are particularly important in a public cloud environment like Netflix operates in, where changes are constantly being made by developers, the same ideas can be extended to any system. Always assume that mistakes can and will be made, and check to ensure that the system setup is correct any time a change is made. You can write your own runtime asserts:

- Check that firewall rules are set up correctly
- Verify files and directory permissions
- Check sudo rules

- Confirm SSL configurations
- Ensure that logging and monitoring services are working correctly

Run your security smoke test every time the system is deployed, in test and in production.

Tools like Puppet and Chef will automatically and continuously scan infrastructure to detect variances from the expected baseline state and alert or automatically revert them.

Situational Awareness and Attack-Driven Defense

DevOps values production feedback and emphasizes the importance of measuring and monitoring production activity. You can extend the same approaches—and the same tools —to security monitoring, involving the entire team instead of just the SOC, making security metrics available in the context of the running system, and graphing and visualizing security-related data to identify trends and anomalies.

Recognize that your system is, or will be, under constant attack. Take advantage of the information that this gives you. Use this information to identify and understand attacks and the threat profile of the system.

Attacks take time. Move to the left of the kill chain and catch them in the early stages. You will reduce the Mean Time to Detect (MTTD) attacks by taking advantage of the close attention that DevOps teams pay to feedback from production, and adding security data into these feedback loops. You also will benefit by engaging people who are closer to the system: the people who wrote the code and keep the system running, who understand how it is supposed to work, what normal looks like, and when things aren't normal.

Feed this data back into your testing and your reviews, prioritizing your actions based on what you are seeing in production, in the same way that you would treat feedback from Continuous Integration or A/B testing in production. This is real feedback, not theoretical, so it should be acted on immediately and seriously.

This is what Zane Lackey at Signal Sciences calls "Attack-Driven Defense". Information on security events helps you to understand and prioritize threats based on what's happening now in production.

Watching for runtime errors and exceptions and attack signatures shows where you are being probed and tested, what kind of attacks you are seeing, where they are attacking, where they are being successful, and what parts of the code need to be protected.

This should help drive your security priorities, tell you where you should focus your testing and remediation. Vulnerabilities that are never attacked (probably) won't hurt you. But attacks that are happening right now need to be resolved—right now.

Signal Sciences

Signal Sciences is a tech startup that offers a next-generation SaaS-based application firewall for web systems. It sets out to "Make security visible" by providing increased transparency into attacks in order to understand risks. It also provides the ability to identify anomalies and block attacks at runtime.

Signal Sciences was started by the former leaders of Etsy's security team. The firewall takes advantage of the ideas and techniques that they developed for Etsy. It is not signature-based like most web application firewalls (WAFs). It analyzes traffic to detect attacks, and aggregates attack signals in its cloud backend to determine when to block traffic. It also correlates attack signals with runtime errors to identify when the system might be in the process of being breached.

Attack data is made visible to the team through dashboards, alert notifications over email, or through integration with services like Slack, HipChat, PagerDuty, and Datadog. The dashboards are built API-first so that data can be integrated into log analysis tools like Splunk or ELK, or into tools like ThreadFix or Jira.

The firewall and its rules engine are being continuously improved and updated, through Continuous Delivery.

Runtime Defense

If you can't successfully shift security left, earlier into design and coding and Continuous Integration and Continuous Delivery, you'll need to add more protection at the end, after the system is in production. Network IDS/IPS solutions tools like Tripwire or signature-based WAFs aren't designed to keep up with rapid system and technology changes in DevOps. This is especially true for cloud IaaS

and PaaS environments, for which there is no clear network perimeter and you might be managing hundreds or thousands of ephemeral instances across different environments (public, private, and hybrid), with self-service Continuous Deployment.

A number of cloud security protection solutions are available, offering attack analysis, centralized account management and policy enforcement, file integrity monitoring and intrusion detection, vulnerability scanning, micro-segmentation, and integration with configuration management tools like Chef and Puppet. Some of these solutions include the following:

- Alert Logic
- CloudPassage Halo
- Dome9 SecOps
- Evident.io
- Illumio
- Palerra LORIC
- Threat Stack

Another kind of runtime defense technology is Runtime Application Security Protection/Self-Protection (RASP), which uses run-time instrumentation to catch security problems as they occur. Like application firewalls, RASP can automatically identify and block attacks. And like application firewalls, you can extend RASP to legacy apps for which you don't have source code.

But unlike firewalls, RASP is not a perimeter-based defense. RASP instruments the application runtime code and can identify and block attacks at the point of execution. Instead of creating an abstract model of the code (like static analysis tools), RASP tools have visibility into the code and runtime context, and use taint analysis and data flow and control flow and lexical analysis techniques, directly examining data variables and statements to detect attacks. This means that RASP tools have a much lower false positive (and false negative) rate than firewalls.

You also can use RASP tools to inject logging and auditing into legacy code to provide insight into the running application and attacks against it. They trade off runtime overheads and runtime costs against the costs of making coding changes and fixes upfront.

There are only a small number of RASP solutions available today, mostly limited to applications that run in the Java JVM and .NET CLR, although support for other languages like Node.js, Python, and Ruby is emerging. These tools include the following:

- Immunio
- Waratek
- Prevoty

- Contrast Security (which we will look at in some more detail)

Contrast Security

Contrast is an Interactive Automated Software Testing (IAST) and RASP solution that directly instruments running code and uses control flow and data flow analysis and lexical analysis to trace and catch security problems at the point of execution. In IAST mode, Contrast can run on a developer's workstation or in a test environment or in Continuous Integration/Continuous Delivery to alert if a security problem like SQL injection or XSS is found during functional testing, all while adding minimal overhead. You can automatically find security problems simply by executing the code; the more thorough your testing, and the more code paths that you cover, the more chances that you have to find vulnerabilities. And because these problems are found as the code is executing, the chances of false positives are much lower than running static analysis.

Contrast deduplicates findings and notifies you of security bugs through different interfaces such as email or Slack or HipChat, or by recording a bug report in Jira. In RASP mode, Contrast runs in production to trace and catch the same kinds of security problems and then alerts operations or automatically blocks the attacks.

It works in Java, .NET (C# and Visual Basic), Node.js, and a range of runtime environments.

Other runtime defense solutions take a different approach from RASP or firewalls. Here are a couple of innovative startups in this space that are worth checking out:

tCell

tCell is a startup that offers application runtime immunity. tCell is a cloud-based SaaS solution that instruments the system at runtime and injects checks and sensors into control points in the running application: database interfaces, authentication controllers, and so on.

It uses this information to map out the attack surface of the system and identifies when the attack surface is changed. tCell also identifies and can block runtime attacks based on the following:

- Known bad patterns of behavior (for example, SQL injection attempts)—like a WAF.
- Threat intelligence and correlation—black-listed IPs, and so on.
- Behavioral learning—recognizing anomalies in behavior and traffic. Over time, it identifies what is normal and can enforce normal patterns of activity, by blocking or alerting on exceptions.

  tCell works in Java, Node.js, Ruby on Rails, and Python (.NET and PHP are in development).

Twistlock
: Twistlock provides runtime defense capabilities for Docker containers in enterprise environments. Twistlock's protection includes enterprise authentication and authorization capabilities—the Twistlock team is working with the Docker community to help implement frameworks for authorization (their authorization plug-in framework was released as part of Docker 1.10) and authentication, and Twistlock provides plug-ins with fine-grained access control rules and integration with LDAP/AD.
  Twistlock scans containers for known vulnerabilities in dependencies and configuration (including scanning against the Docker CIS benchmark). It also scans to understand the purpose of each container. It identifies the stack and the behavioral profile of the container and how it is supposed to act, creating a white list of expected and allowed behaviors.
  An agent installed in the runtime environment (also as a container) runs on each node, talking to all of the containers on the node and to the OS. This agent provides visibility into runtime activity of all the containers, enforces authentication and authorization rules, and applies the white list of expected behaviors for each container as well as a black list of known bad behaviors (like a malware solution).
  And because containers are intended to be immutable, Twistlock recognizes and can block attempts to change container configurations at runtime.

Learning from Failure: Game Days, Red Teaming, and Blameless Postmortems

Game Days—running real-life, large-scale failure tests (like shutting down a data center)—have also become common practices in

DevOps organizations like Amazon, Google, and Etsy. These exercises can involve (at Google, for example) hundreds of engineers working around the clock for several days, to test out disaster recovery cases and to assess how stress and exhaustion could impact the organization's ability to deal with real accidents.[7]

At Etsy, Game Days are run in production, even involving core functions such as payments handling. Of course, this begs the question, "Why not simulate this in a QA or staging environment?" Etsy's response is, first, the existence of any differences in those environments brings uncertainty to the exercise; second, the risk of not recovering has no consequences during testing, which can bring hidden assumptions into the fault tolerance design and into recovery. The goal is to reduce uncertainty, not increase it.[8]

These exercises are carefully tested and planned in advance. The team brainstorms failure scenarios and prepares for them, running through failures first in test and fixing any problems that come up. Then, it's time to execute scenarios in production, with developers and operators watching closely and ready to jump in and recover, especially if something goes unexpectedly wrong.

You can take many of the ideas from Game Days, which are intended to test the resilience of the system and the readiness of the DevOps team to handle system failures, and apply them to infosec attack scenarios through *Red Teaming*. This is a core practice at organizations like Microsoft, Facebook, Salesforce, Yahoo!, and Intuit for their cloud-based services.

Like operations Game Days, Red Team exercises are most effectively done in production.

The Red Team identifies weaknesses in the system that they believe can be exploited, and work as ethical hackers to attack the live system. They are generally given freedom to act short of taking the system down or damaging or exfiltrating sensitive data. The Red Team's success is measured by the seriousness of the problems that they find, and their Mean Time to Exploit/Compromise.

---

7 ACM: Resilience Engineering: Learning to Embrace Failure. *https://queue.acm.org/detail.cfm?id=2371297*

8 ACM: "Fault Injection in Production, Making the case for resilience testing." *http://queue.acm.org/detail.cfm?id=2353017*

The Blue Team is made up of the people who are running, supporting, and monitoring the system. Their responsibility is to identify when an attack is in progress, understand the attack, and come up with ways to contain it. Their success is measured by the Mean Time to Detect the attack and their ability to work together to come up with a meaningful response.

Here are the goals of these exercises:

- Identify gaps in testing and in design and implementation by hacking your own systems to find real, exploitable vulnerabilities.
- Exercise your incident response and investigation capabilities, identify gaps or weaknesses in monitoring and logging, in playbooks, and escalation procedures and training.
- Build connections between the security team and development and operations by focusing on the shared goal of making the system more secure.

After a Game Day or Red Team exercise, just like after a real production outage or a security breach, the team needs to get together to understand what happened and learn how to get better. They do this in *Blameless Postmortem* reviews. Here, everyone meets in an open environment to go over the facts of the event: what happened, when it happened, how people reacted, and then what happened next. By focusing calmly and objectively on understanding the facts and on the problems that came up, the team can learn more about the system and about themselves and how they work, and they can begin to understand what went wrong, ask why things went wrong, and look for ways to improve, either in the way that the system is designed, or how it is tested, or in how it is deployed, or how it is run.

To be successful, you need to create an environment in which people feel safe to share information, be honest and truthful and transparent, and to think critically without being criticized or blamed—what Etsy calls a "Just Culture." This requires buy-in from management down, understanding and accepting that accidents can and will happen, and that they offer an important learning opportunity. When done properly, Blameless Postmortems not only help you to learn from failures and understand and resolve important problems, but

they can also bring people together and reinforce openness and trust, making the organization stronger.[9]

Security at Netflix

Netflix is another of the DevOps unicorns. Like Etsy, Amazon, and Facebook, it has built its success through a culture based on "Freedom and Responsibility" (employees, including engineers, are free to do what they think is the right thing, but they are also responsible for the outcome) and a massive commitment to automation, including in security—especially in security.

After experiencing serious problems running its own IT infrastructure, Netflix made the decision to move its online business to the cloud. It continues to be one of the largest users of Amazon's AWS platform.

Netflix's approach to IT operations is sometimes called "NoOps" because they don't have operations engineers or system admins. They have effectively outsourced that part of their operations to Amazon AWS because they believe that data center management and infrastructure operations is "undifferentiated heavy lifting." Or, put another way, work that is hard to do right but that does not add direct value to their business.

Here are the four main pillars of Netflix's security program:[10]

Undifferentiated heavy lifting and shared responsibility
: Netflix relies heavily on the capabilities of AWS and builds on or extends these capabilities as necessary to provide additional security and reliability features. It relies on its cloud provider for automated provisioning, platform vulnerability management, data storage and backups, and physical data center protections. Netflix built its own PaaS layer on top of this, including an extensive set of security checks and analytic and monitoring services. Netflix also bakes secure defaults into its base infrastructure images, which are used to configure each instance.

9 "Blameless PostMortems and a Just Culture." *https://codeascraft.com/2012/05/22/blameless-postmortems/*

10 See "Splitting the Check on Compliance and Security: Keeping Developers and Auditors Happy in the Cloud." Jason Chan, Netflix, AWS re:Invent, October 2015. *https://www.youtube.com/watch?v=Io00_K4v12Y*

Traceability in development
: Source control, code reviews through Git pull requests, and the Continuous Integration and Continuous Delivery pipeline provide a complete trace of all changes from check-in to deployment. Netflix uses the same tools to track information for its own support purposes as well as for auditors instead of wasting time creating audit trails just for compliance purposes. Engineers and auditors both need to know who made what changes when, how the changes were tested, when they were deployed, and what happened next. This provides visibility and traceability for support and continuous validation of compliance.

Continuous security visibility
: Recognize that the environment is continuously changing and use automated tools to identify and understand security risks and to watch for and catch problems. Netflix has written a set of its own tools to do this, including Security Monkey, Conformity Monkey, and Penguin Shortbread (which automatically identifies microservices and continuously assesses the risk of each service based on runtime dependencies).

Compartmentalization
: Take advantage of cloud account segregation, data tokenization, and microservices to minimize the system's attack surface and contain attacks, and implement least privilege access policies. Recognizing that engineers will generally ask for more privileges than they need "just in case," Netflix has created an automated tool called Repoman, which uses AWS Cloudtrail activity history and reduces account privileges to what is actually needed based on what each account has done over a period of time. Compartmentalization and building up bulkheads also contains the "blast radius" of a failure, reducing the impact on operations when something goes wrong.

Whether you are working in the cloud or following DevOps in your own data center, these principles are all critical to building and operating a secure and reliable system.

CHAPTER 5

Compliance as Code

DevOps can be followed to achieve what Justin Arbuckle at Chef calls "Compliance as Code": building compliance into development and operations, and wiring compliance policies and checks and auditing into Continuous Delivery so that regulatory compliance becomes an integral part of how DevOps teams work on a day-to-day basis.

Chef Compliance

Chef Compliance is a tool from Chef that scans infrastructure and reports on compliance issues, security risks, and outdated software. It provides a centrally managed way to continuously and automatically check and enforce security and compliance policies.

Compliance profiles are defined in code to validate that systems are configured correctly, using InSpec, an open source testing framework for specifying compliance, security, and policy requirements.

You can use InSpec to write high-level, documented tests/assertions to check things such as password complexity rules, database configuration, whether packages are installed, and so on. Chef Compliance comes with a set of predefined profiles for Linux and Windows environments as well as common packages like Apache, MySQL, and Postgres.

When variances are detected, they are reported to a central dashboard and can be automatically remediated using Chef.

A way to achieve Compliance as Code is described in the "DevOps Audit Defense Toolkit", a free, community-built process framework written by James DeLuccia, IV, Jeff Gallimore, Gene Kim, and Byron Miller.[1] The Toolkit builds on real-life examples of how DevOps is being followed successfully in regulated environments, on the Security as Code practices that we've just looked at, and on disciplined Continuous Delivery. It's written in case-study format, describing compliance at a fictional organization, laying out common operational risks and control strategies, and showing how to automate the required controls.

Defining Policies Upfront

Compliance as Code brings management, compliance, internal audit, the PMO and infosec to the table, together with development and operations. Compliance policies and rules and control workflows need to be defined upfront by all of these stakeholders working together. Management needs to understand how operational risks and other risks will be controlled and managed through the pipeline. Any changes to these policies or rules or workflows need to be formally approved and documented; for example, in a Change Advisory Board (CAB) meeting.

But instead of relying on checklists and procedures and meetings, the policies and rules are enforced (and tracked) through automated controls, which are wired into configuration management tools and the Continuous Delivery pipeline. Every change ties back to version control and a ticketing system like Jira for traceability and auditability: all changes must be made under a ticket, and the ticket is automatically updated along the pipeline, from the initial request for work all the way to deployment.

Automated Gates and Checks

The first approval gate is mostly manual. Every change to code and configuration must be reviewed precommit. This helps to catch mistakes and ensure that no changes are made without at least one other person checking to verify that it was done correctly. High-risk code (defined by the team, management, compliance, and infosec)

1 *http://itrevolution.com/devops-and-auditors-the-devops-audit-defense-toolkit/*

must also have an SME review; for example, security-sensitive code must be reviewed by a security expert. Periodic checks are done by management to ensure that reviews are being done consistently and responsibly, and that no "rubber stamping" is going on. The results of all reviews are recorded in the ticket. Any follow-up actions that aren't immediately addressed are added to the team's backlog as another ticket.

In addition to manual reviews, automated static analysis checking is also done to catch common security bugs and coding mistakes (in the IDE and in the Continuous Integration/Continuous Delivery pipeline). Any serious problems found will fail the build.

After it is checked-in, all code is run through the automated test pipeline. The Audit Defense Toolkit assumes that the team follows Test-Driven Development (TDD), and outlines an example set of tests that should be executed.

Infrastructure changes are done using an automated configuration management tool like Puppet or Chef, following the same set of controls:

- Changes are code reviewed precommit
- High-risk changes (again, as defined by the team) must go through a second review by an SME
- Static analysis/lint checks are done automatically in the pipeline
- Automated tests are performed using a test framework like rspec-puppet or Chef Test Kitchen or ServerSpec
- Changes are deployed to test and staging in sequence with automated smoke testing and integration testing

And, again, every change is tracked through a ticket and logged.

Managing Changes in Continuous Delivery

Because DevOps is about making small changes, the Audit Defense Toolkit assumes that most changes can be treated as standard or routine changes that are essentially preapproved by management and therefore do not require CAB approval.

It also assumes that bigger changes will be made "dark." In other words, that they will be made in small, safe, and incremental

changes, protected behind runtime feature switches that are turned off by default. The feature will only be rolled out with coordination between development, ops, compliance, and other stakeholders.

Any problems found in production are reviewed through postmortems, and tests added back into the pipeline to catch the problems (following TDD principles).

Separation of Duties in the DevOps Audit Toolkit

In the DevOps Audit Toolkit, a number of controls enforce or support Separation of Duties:

- Mandatory independent peer reviews ensure that no engineer (dev or ops) can make a change without someone else being aware and approving it. Reviewers are assigned randomly where possible to prevent collusion.
- Developers are granted read-only access to production systems to assist with troubleshooting. Any fixes need to be made through the Continuous Delivery pipeline (fixing forward) or by automatically rolling changes back (again, through the Continuous Delivery pipeline/automated deployment processes) which are fully auditable and traceable.
- All changes made through the pipeline are transparent, published to dashboards, IRC, and so on.
- Production access logs are reviewed by IT operations management weekly.
- Access credentials are reviewed regularly.
- Automated detective change control tools (for example, Tripwire, OSSEC, UpGuard) are used to check for unauthorized changes.

These controls minimize the risk of developers being able to make unauthorized, and undetected, changes to production.

Using the Audit Defense Toolkit

The DevOps Audit Defense Toolkit provides a roadmap to how you can take advantage of DevOps workflows and automated tools, and some of the security controls and checks that we've already looked at, to support your compliance and governance requirements.

It requires a lot of discipline and maturity and might be too much for some organizations to take on—at least at first. You should examine the controls and decide where to begin.

Although it assumes Continuous Deployment of changes directly to production, the ideas and practices can easily be adapted for Continuous Delivery by adding a manual review gate before changes are pushed to production.

Code Instead of Paperwork

Compliance as Code tries to minimize paperwork and overhead. You still need clearly documented policies that define how changes are approved and managed, and checklists for procedures that cannot be automated. But most of the procedures and the approval gates are enforced through automated rules in the Continuous Integration/Continuous Delivery pipeline, leaning on the automated pipeline and tooling to ensure that all of the steps are followed consistently and taking advantage of the detailed audit trail that is automatically created.

In the same way that frequently exercising build and deployment steps reduces operational risks, exercising compliance on every change, following the same standardized process and automated steps, reduces the risks of compliance violations. You—and your auditors—can be confident that all changes are made the same way, that all code is run through the same tests and checks, and that everything is tracked the same way: consistent, complete, repeatable, and auditable.

Standardization makes auditors happy. Auditing makes auditors happy (obviously). Compliance as Code provides a beautiful audit trail for every change, from when the change was requested and why, to who made the change and what that person changed, who reviewed the change and what was found in the review, how and

when the change was tested, to when it was deployed. Except for the discipline of setting up a ticket for every change and tagging changes with a ticket number, compliance becomes automatic and seamless to the people who are doing the work.

Just as beauty is in the eye of the beholder, compliance is in the opinion of the auditor. Auditors might not understand or agree with this approach at first. You will need to walk them through it and prove that the controls work. But that shouldn't be too difficult, as Dave Farley, one of the original authors of *Continuous Delivery* explains:

> I have had experience in several finance firms converting to Continuous Delivery. The regulators are often wary at first, because Continuous Delivery is outside of their experience, but once they understand it, they are extremely enthusiastic. So regulation is not really a barrier, though it helps to have someone that understands the theory and practice of Continuous Delivery to explain it to them at first.
>
> If you look at the implementation of a deployment pipeline, a core idea in Continuous Delivery, it is hard to imagine how you could implement such a thing without great traceability. With very little additional effort the deployment pipeline provides a mechanism for a perfect audit trail. The deployment pipeline is the route to production. It is an automated channel through which all changes are released. This means that we can automate the enforcement of compliance regulations—"No release if a test fails," "No release if a trading algorithm wasn't tested," "No release without sign-off by an authorised individual," and so on. Further, you can build in mechanisms that audit each step, and any variations. Once regulators see this, they rarely wish to return to the bad old days of paper-based processes.[2]

2 Dave Farley (*http://www.continuous-delivery.co.uk/*), Interview July 24, 2015

CHAPTER 6

Conclusion: Building a Secure DevOps Capability and Culture

DevOps—the culture, the process frameworks and workflows, the emphasis on automation and feedback—can all be used to improve your security program.

You can look to leaders like Etsy, Netflix, Amazon, and Google for examples of how you can do this successfully. Or the London Multi-Asset Exchange, or Capital One, or Intuit, or E*Trade, or the United States Department of Homeland Security. The list is growing.

These organizations have all found ways to balance security and compliance with speed of delivery, and to build protection into their platforms and pipelines.

They've done this—and you can do this—by using Continuous Delivery as a control structure for securing software delivery and enforcing compliance policies; securing the runtime through Infrastructure as Code; making security part of the feedback loops and improvement cycles in DevOps; building on DevOps culture and values; and extending this to embrace security.

Pick a place to begin. Start by fixing an important problem or addressing an important risk. Or start with something simple, where you can achieve a quick win and build momentum.

Implementing Software Component Analysis to automatically create a bill of materials for a system could be an easy win. This lets you identify and resolve risks in third-party components early in the

SDLC, without directly affecting development workflows or slowing delivery.

Securing the Continuous Delivery pipeline itself is another important and straightforward step that you can take without slowing delivery. Ensuring that changes are really being made in a reliable, repeatable, and auditable way that you and the business can rely on the integrity of automated changes. Doing this will also help you to better understand the engineering workflow and tool chain so that you can prepare to take further steps.

You could start at the beginning, by ensuring that risk assessments are done on every new app or service, looking at the security protections (and risks) in the language(s) and framework(s) that the engineering team wants to use. You could build hardening into Chef recipes or Puppet manifests to secure the infrastructure. Or, you could start at the end, by adding runtime checks like Netflix's monkeys to catch dangerous configuration or deployment mistakes in production.

The point is to start somewhere and make small, meaningful changes. Measure the results and keep iterating. Take advantage of the same tools and workflows that dev and ops are using. Check your scripts and tools into version control. Use Docker to package security testing and forensics tools. Use the Continuous Delivery pipeline to deploy them. Work with developers and operations to identify and resolve risks. Use DevOps to secure DevOps.

Find ways to help the team deliver, but in a secure way that minimizes risks and ensures compliance while minimizing friction and costs. Don't get in the way of the feedback loops. Use them instead to measure, learn, and improve.

Working with dev and ops, understanding how they do what they do, using the same tools, solving problems together, will bring dev and ops and infosec together.

In my organization, moving to DevOps and DevOpsSec has been, and continues to be, a journey. We began by building security protection into our frameworks, working with a security consultancy to review the architecture and train the development team. We implemented Continuous Integration and built up our automated test suite and wired-in static analysis testing.

We have created a strong culture of code reviews and made incremental threat modeling part of our change controls. Regular pen tests are used as opportunities to learn how and where we need to improve our security program and our design and code. Our systems engineering team manages infrastructure through code, using the same engineering practices as the developers: version control, code reviews, static analysis, and automated testing in Continuous Integration. And as we shortened our delivery cycle, moving toward Continuous Delivery, we have continued to simplify and automate more steps and checks so that they can be done more often and to create more feedback loops. Security and compliance are now just another part of how we build and deliver and run systems, part of everyone's job.

DevOps is fundamentally changing how dev and ops are done today. And it will change how security is done, too. It requires new skills, new tools, and a new set of priorities. It will take time and a new perspective. So the sooner you get started, the better.

About the Author

Jim Bird is a CTO, software development manager, and project manager with more than 20 years of experience in financial services technology. He has worked with stock exchanges, central banks, clearinghouses, securities regulators, and trading firms in more than 30 countries. He is currently the CTO of a major US-based institutional alternative trading system.

Jim has been working in Agile and DevOps environments for several years. His first experience with incremental and iterative ("step-by-step") development was back in the early 1990s, when he worked at a West Coast tech firm that developed, tested, and delivered software in monthly releases to customers around the world—he didn't realize how unique that was at the time. Jim is active in the DevOps and AppSec communities, is a contributor to the Open Web Application Security Project (OWASP), and occasionally helps out as an analyst for the SANS Institute.

Jim is also the author of the O'Reilly report, *DevOps for Finance: Reducing Risk through Continuous Delivery*.