1 Securing DevOps

This chapter covers

- Getting to know DevOps and its impact on building cloud services
- Using continuous integration, continuous delivery, and infrastructure as a service
- Evaluating the role and goals of security in a DevOps culture
- Defining the three components of a DevOps security strategy

Connected applications that make little parts of our life easier are the technological revolution of the twenty-first century. From helping us do our taxes, share photos with friends and families, and find a good restaurant in a new neighborhood, to tracking our progress at the gym, applications that allow us to do more in less time are increasingly beneficial. The growth rates of services like Twitter, Facebook, Instagram, and Google show that customers find tremendous value in each application, either on their smartphones' home screen or in a web browser.

Part of this revolution was made possible by improved tooling in creating and operating these applications. Competition is tough on the internet. Ideas don't stay new for long, and organizations must move quickly to collect market shares and lock in users of their products. In the startup world, the speed and cost at which organizations can build an idea into a product is a critical factor for success. DevOps, by industrializing the tools and techniques of the internet world, embodies the revolution that made it possible to run online services at a low cost, and let small startups compete with tech giants.

In the startup gold rush, data security sometimes suffers. Customers have shown their willingness to trust applications with their data in exchange for features, leading many organizations to store enormous amounts of personal information about their users, often before the organization has a security plan to handle the data. A competitive landscape that makes companies take risks, mixed with large amount of sensitive data, is a perfect recipe for disaster. And so, as the number of online services increases, the frequency of data breaches increases as well.

Securing DevOps is about helping organizations operate securely and protect the data their customers entrust them with. I introduce a model I refer to as "continuous security," which focuses on integrating strong security principles into the various components of a DevOps strategy. I explain culture, architectural principles, techniques, and risk management with the goal of going from no security to a mature program. This book is primarily about principles and concepts, but throughout the chapters we'll use specific tools and environments as examples.

DevOps can mean many different things, depending on which part of information technology (IT) it's being applied to. Operating the infrastructure of a nuclear plant is very different from processing credit card payments on websites, yet both equally benefit from DevOps to optimize and strengthen their operations. I couldn't possibly cover all of DevOps and IT in a single book, and decided to focus on cloud services, an area of IT dedicated to the development and operations of web applications. Throughout the book, I invite the reader to develop, operate, secure, and defend a web application hosted in the cloud. The concepts and examples I present best apply to cloud services, in organizations that don't yet have a dedicated security team, yet an open-minded reader could easily transfer them into any DevOps environment.

In this first chapter, we'll explore how DevOps and security can work together, allowing organizations to take risks without compromising the safety of their customers.

1.1 The DevOps approach

DevOps is the process of continuously improving software products through rapid release cycles, global automation of integration and delivery pipelines, and close collaboration between teams. The goal of DevOps is to shorten the time and reduce the cost of transforming an idea into a product that customers use. DevOps makes heavy use of automated processes to speed up development and deployment. Figure 1.1 shows a comparison of a traditional software-building approach at the top, with DevOps at the bottom.

- *In the top section, the time between conceptualization and availability to customers is eight days.* Deploying the infrastructure consumes most of that time, as engineers need to create the components needed to host the software on the internet. Another big time-consumer is the testing-and-reviewing step between deployments.

- *In the bottom section, the time between conceptualization and delivery is reduced to two days*. This is achieved by using automated processes to handle the infrastructure deployment and software test/review.

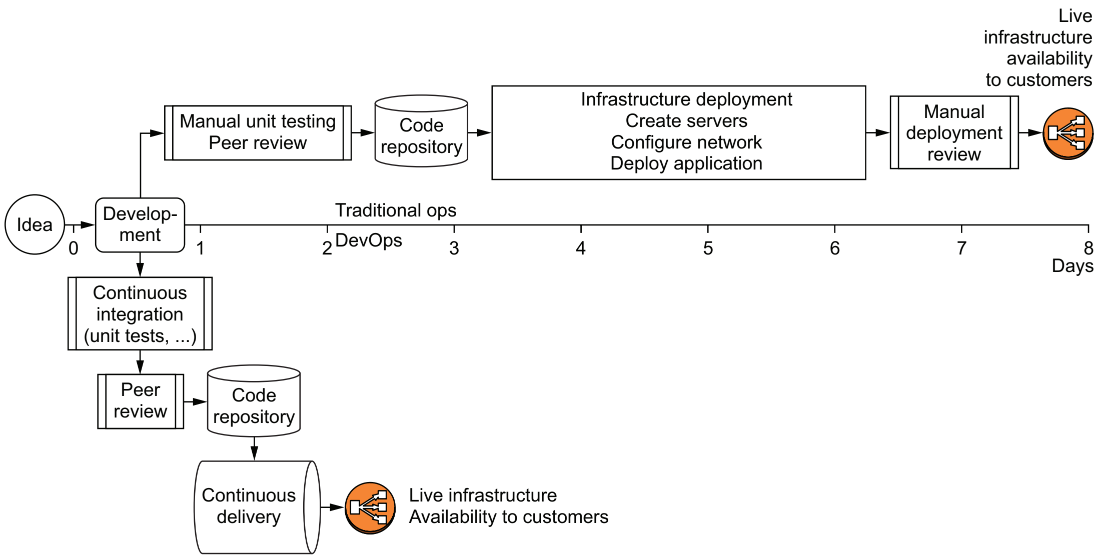
Live infrastructure availability to customers
Manual unit testing Peer review
Code repository
Infrastructure deployment Create servers Configure network Deploy application
Manual deployment review
Idea
Develop-ment
Traditional ops
DevOps
0 1 2 3 4 5 6 7 8
Days
Continuous integration (unit tests, ...)
Peer review
Code repository
Continuous delivery
Live infrastructure Availability to customers

Figure 1.1 DevOps reduces the time between feature conception and its availability to customers.

An organization able to build software four times faster than its competitor has a significant competitive advantage. History shows that customers value innovative products that may be incomplete at first but improve quickly and steadily. Organizations adopt DevOps to reduce the cost and latency of development cycles and answer their customers' demands.

With DevOps, developers can release new versions of their software, test them, and deploy them to customers in as little as a few hours. That doesn't mean versions are always released that quickly, and it can take time to do proper quality assurance (QA), but DevOps provides the ability to move quickly if needed. Figure 1.2 zooms into the bottom section of figure 1.1 to detail how the techniques of continuous integration, continuous delivery, and infrastructure as a service are used together to achieve fast release cycles.

The key component of the pipeline in figure 1.2 is the chaining of automated steps to go from a developer's patch submission to a service deployed in a production environment in a completely automated fashion. Should any of the automated steps fail along the way, the pipeline is stopped, and the code isn't deployed. This mechanism ensures that tests of all kinds pass before a new version of the software can be released into production.

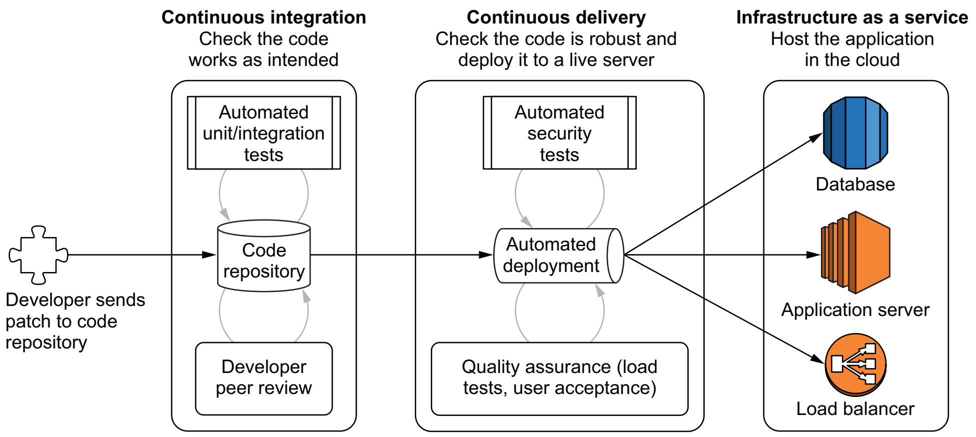
Continuous integration
Check the code works as intended
Continuous delivery
Check the code is robust and deploy it to a live server
Infrastructure as a service
Host the application in the cloud
Automated unit/integration tests
Code repository
Developer peer review
Developer sends patch to code repository
Automated security tests
Automated deployment
Quality assurance (load tests, user acceptance)
Database
Application server
Load balancer

Figure 1.2 Continuous integration (CI), continuous delivery (CD), and infrastructure as a service (IaaS) form an automated pipeline that allows DevOps to speed up the process of testing and deploying software.

1.1.1 Continuous integration

The process of quickly integrating new features into software is called *continuous integration* (CI). CI defines a workflow to implement, test, and merge features into software products. Product managers and developers define sets of small features that are implemented in short cycles. Each feature is added into a branch of the main source code and submitted for review by a peer of the developer who authored it. Automated tests happen at the review stage to verify that the change doesn't introduce any regressions, and that the quality level is maintained. After review, the change is merged into the central source-code repository, ready for deployment. Quick iterations over small features make the process smooth and prevent breakage of functionalities that come with large code changes.

1.1.2 Continuous delivery

The automation of deploying software into services available to customers is called *continuous delivery* (CD). Rather than managing infrastructure components by hand, DevOps recommends that engineers program their infrastructure to handle change rapidly. When developers merge code changes into the software, operators trigger a deployment of the updated software from the CD pipeline, which automatically retrieves the latest version of the source code, packages it, and creates a new infrastructure for it. If the deployment goes smoothly, possibly after the QA team has manually or automatically reviewed it, the environment is promoted as the new staging or production environment. Users are directed to it, and the old environment is destroyed.

The process of managing servers and networks with code alleviates the long delays usually needed to handle deployments.

1.1.3 Infrastructure as a service

Infrastructure as a service (IaaS) is the cloud. It's the notion that the data center, network, servers, and sometimes systems an organization relies on, are entirely operated by a third party, controllable through APIs and code, and exposed to operators as a service. IaaS is a central tool in the DevOps arsenal because it plays an important role in the cost reduction of operating infrastructures. Its programmable nature makes IaaS different from traditional infrastructure and encourages operators to write code that creates and modifies the infrastructure instead of performing those tasks by hand.

Operating in-house

Many organizations prefer to keep their infrastructure operated internally for a variety of reasons (regulation, security, cost, and so on). It's important to note that adopting an IaaS doesn't necessarily mean outsourcing infrastructure management to a third party. An organization can deploy and operate IaaS in-house, using platforms like Kubernetes or OpenStack, to benefit from the flexibility those intermediate management layers bring over directly running applications on hardware.

For the purposes of this book, I use an IaaS system operated by a third party—AWS—popular in many organizations for reducing the complexity of managing infrastructure and allowing them to focus on their core product. Yet, most infrastructure security concepts I present apply to any type of IaaS, whether you control the hardware or let a third party do it for you.

Managing the lower layers of an infrastructure brings a whole new set of problems, like network security and data-center access controls, that you should be taking care of. I don't cover those in this book, as they aren't DevOps-specific, but you shouldn't have trouble finding help in well-established literature.

Amazon Web Services (AWS), which will be used as our example environment throughout the book, is the most emblematic IaaS. Figure 1.3 shows the components of AWS that are managed by the provider, at the bottom, versus the ones managed by the operator, at the top.

CI, CD, and IaaS are fundamental components of a successful DevOps strategy. Organizations that master the CI/CD/IaaS workflow can deploy software to end users rapidly, possibly several times a day, in a fully automated fashion. The automation of all the testing and deployment steps guarantees that minimal human involvement is needed to operate the pipeline, and that the infrastructure is fully recoverable in case of disaster.

Beyond the technical benefits, DevOps also influences the culture of an organization, and in many ways, contributes to making people happier.

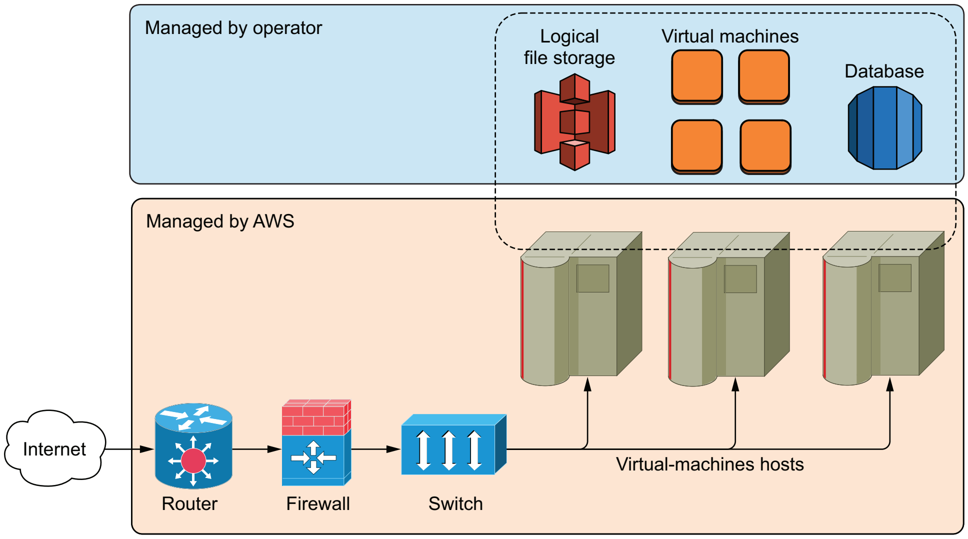

Figure 1.3 AWS is an IaaS that reduces the operational burden by handling the management of core infrastructure components. In this diagram, equipment in the lower box is managed entirely by Amazon, and the operator manages the components in the upper box. In a traditional infrastructure, operators must manage all the components themselves.

1.1.4 Culture and trust

Improved tooling is the first phase of a successful DevOps approach. Culture shifts accompany this change, and organizations that mature the technical aspects of DevOps gain confidence and trust in their ability to bring new products to their users. An interesting side effect of increased trust is the reduced need for management as engineers are empowered to deliver value to the organization with minimal overhead. Some DevOps organizations went as far as experimenting with flat structures that had no managers at all. Although removing management entirely is an extreme that suits few organizations, the overall trend of reduced management is evidently linked to mature DevOps environments.

Organizations that adopt and succeed at DevOps are often better at finding and retaining talent. It's common to hear developers and operators express their frustration with working in environments that are slow and cluttered. Developers feel annoyed waiting for weeks to deploy a patch to a production system. Operators, product managers, and designers all dislike slow iterations. People leave those companies and turnover rates can damage the quality of a product. Companies that bring products to market faster have a competitive advantage, not only because they deliver features to their users faster, but also because they keep their engineers happy by alleviating operational complexity.

DevOps teaches us that shipping products faster makes organizations healthier and more competitive, but increasing the speed of shipping software can make the work of security engineers difficult. Rapid release cycles leave little room for thorough security reviews and require organizations to take on more technological risks than in a slower structure. Integrating security in DevOps comes with a new set of challenges, starting with a fundamental security culture shift.

1.2 Security in DevOps

"A ship is safe in harbor, but that's not what ships are built for."

—John A. Shedd

To succeed in a competitive market, organizations need to move fast, take risks, and operate at a reasonable cost. The role of security teams in those organizations is to be the safety net that protects the company's assets while helping it to succeed. Security teams need to work closely with the engineers and managers who build the company's products. When a company adopts DevOps, security must change its culture to adopt DevOps as well, starting with a focus on the customer.

DevOps and its predecessors—the Agile Manifesto (http://agilemanifesto.org/) and Deming's 14 principles (https://deming.org/explore/fourteen-points)—have one trait in common: a focus on shipping better products to customers faster. Every successful strategy starts with a focus on the customer (http://mng.bz/GN43):

"We're not competitor obsessed, we're customer obsessed. We start with what the customer needs and we work backwards."

—Jeff Bezos, Amazon

In DevOps, everyone in the product pipeline is focused on the customer:

- Product managers measure engagement and retention ratios.
- Developers measure ergonomics and usability.
- Operators measure uptime and response times.

The *customer* is where the company's attention is. The satisfaction of the customer is the metric everyone aligns their goals against.

In contrast, many security teams focus on security-centric goals, such as

- Compliance with a security standard
- Number of security incidents
- Count of unpatched vulnerabilities on production systems

When the company's focus is directed outward to its customers, security teams direct their focus inward to their own environment. One wants to increase the value of the organization, while the other wants to protect its existing value. Both sides are necessary for a healthy ecosystem, but the goal disconnect hurts communication and efficiency.

In organizations that actively measure goals and performance of individual teams to mete out bonuses and allocate rewards, each side is pressured to ignore the others and focus on its own achievements. To meet a goal, developers and operators ignore security recommendations when shipping a product that may be considered risky. Security blocks projects making use of unsafe techniques and recommends unrealistic solutions to avoid incidents that could hurt their bottom line. In situations like these, both sides often hold valid arguments, and are well intended, but fail to understand and adapt to the motivation of the other.

As a security engineer, I've never encountered development or operational teams that didn't care about security, but I have met many frustrated with the interaction and goal disconnects. Security teams that lack the understanding of the product strategy, organize arbitrary security audits that prevent shipping features, or require complex controls that are difficult to implement are all indicators of a security system that's anything but agile. Seen from the other side, product teams that ignore the experience and feedback of their security team are a source of risk that ultimately hurts the organization.

DevOps teaches us that a successful strategy requires bringing the operational side closer to the development side and breaking the communication barrier between various developers and operators. Similarly, securing DevOps must start with a close integration between security teams and their engineer peers. Security needs to serve the customer by being a function of the service, and the internal goals of security teams and DevOps teams need to be aligned.

When security becomes an integral part of DevOps, security engineers can build controls directly into the product rather than bolting them on top of it after the fact. Everyone shares the same goals of making the organization succeed. Goals are aligned, communication is improved, and data safety increases. The core idea behind bringing security into DevOps is for security teams to adopt the techniques of DevOps and switch their focus from defending only the infrastructure to protecting the entire organization by improving it continuously.

Throughout the book, I call this approach *continuous security*. In the following section, you'll see how to implement continuous security gradually, starting with simple and easy-to-implement security controls, and progressively maturing the security strategy to cover the entire organization.

1.3 Continuous security

Continuous security is composed of three areas, outlined in the gray boxes of figure 1.4. Each area focuses on a specific aspect of the DevOps pipeline. As customer feedback spurs organizational growth that drives new features, the same is true of continuous security. This book has three parts; each covers one area of continuous security:

- *Test-driven security (TDS)*—The first step of a security program is to define, implement, and test security controls. TDS covers simple controls like the standard configuration of a Linux server, or the security headers that web applications must implement. A great deal of security can be obtained by consistently implementing basic controls and relentlessly testing those controls for accuracy. In good DevOps, manual testing should be the exception, not the rule. Security testing should be handled the same way all application tests are handled in the CI and CD pipelines: automatically, and all the time. We'll cover TDS by applying layers of security to a simple DevOps pipeline in part 1.

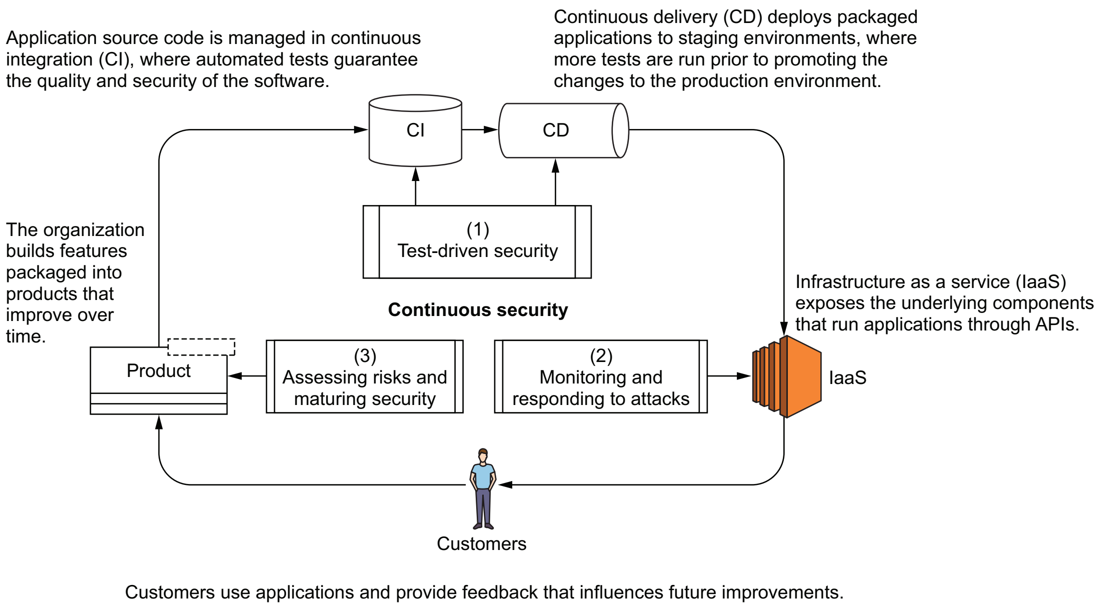
Application source code is managed in continuous integration (CI), where automated tests guarantee the quality and security of the software.
Continuous delivery (CD) deploys packaged applications to staging environments, where more tests are run prior to promoting the changes to the production environment.
CI
CD
(1) Test-driven security
Continuous security
The organization builds features packaged into products that improve over time.
Infrastructure as a service (IaaS) exposes the underlying components that run applications through APIs.
Product
(3) Assessing risks and maturing security
(2) Monitoring and responding to attacks
IaaS
Customers
Customers use applications and provide feedback that influences future improvements.

Figure 1.4 The three phases of continuous security protect the organization's products and customers by constantly improving security through feedback loops.

- *Monitoring and responding to attacks*—It's the fate of online services that they will get broken into eventually. When incidents happen, organizations turn to their security teams for help, and a team must be prepared to react. The second phase of continuous security is to monitor and respond to threats and protect the services and data the organization relies on. In part 2, I talk about techniques like fraud and intrusion detection, digital forensics, and incident response, with the goal of increasing an organization's preparedness for an incident.
- *Assessing risks and maturing security*—I talk about technology a lot in the first two parts of the book, but a successful security strategy can't succeed when solely focused on technical issues. The third phase of continuous security is to go beyond the technology and look at the organization's security posture from a high altitude. In part 3, I explain how risk management and security testing, both internal and external, help organizations refocus their security efforts and invest their resources more efficiently.

Mature organizations trust their security programs and work together with their security teams. Reaching that point requires focus, experience, and a good sense of knowing when to take, or refuse to take, risks. A comprehensive security strategy mixes technology and people to identify areas of improvement and allocate resources appropriately, all in rapid improvement cycles. This book aims to give you the tools you need to reach that level of maturity in your organization.

With a model of continuous security in mind, let's now take a detailed look at each of its three components, and what they mean in terms of product security.

1.3.1 Test-driven security

The myth of attackers breaking through layers of firewalls or decoding encryption with their smartphones makes for great movies, but poor real-world examples. In most cases, attackers go for easy targets: web frameworks with security vulnerabilities, out-of-date systems, administration pages open to the internet with guessable passwords, and security credentials mistakenly leaked in open source code are all popular candidates. Our first goal in implementing a continuous security strategy is to take care of the baseline: apply elementary sets of controls on the application and infrastructure of the organization and test them continuously. For example:

- SSH root login must be disabled on all systems.
- Systems and applications must be patched to the latest available version within 30 days of its release.
- Web applications must use HTTPS, never HTTP.
- Secrets and credentials must not be stored with application code, but handled separately in a vault accessible only to operators.
- Administration interfaces must be protected behind a VPN.

The list of security best practices should be established between the security team and the developers and operators to make sure everyone agrees on their value. A list of baseline requirements can be rapidly assembled by collecting those best practices and adding some common sense. In part 1 of the book, I talk about various steps in securing applications, infrastructure, and CI/CD pipelines.

APPLICATION SECURITY

Modern web applications are exposed to a wide range of attacks. The Open Web Application Security Project (OWASP) ranks the most common attacks in a top-10 list published every three years (http://mng.bz/yXd3): cross-site scripting, SQL injections, cross-site request forgery, brute-force attacks, and so on, seemingly endlessly. Thankfully, each attack vector can be covered using the right security controls in the right places. In chapter 3, which covers application security, we'll take a closer look at the controls a DevOps team should implement to keep web applications safe.

INFRASTRUCTURE SECURITY

Relying on IaaS to run software doesn't exempt a DevOps team from caring about infrastructure security. All systems have entry points that grant elevated privileges, like VPNs, SSH gateways, or administration panels. When an organization grows, special care must be taken to continuously protect the systems and networks while opening new accesses and integrating more pieces together.

Pipeline security

The DevOps way of shipping products through automation is vastly different from traditional operations most security teams are used to. Compromising a CI/CD pipeline can grant an attacker full control over the software that runs in production. Securing the automated steps taken to deliver code to production systems can be done using integrity controls like commit or container signing. I'll explain how to add trust to the CI/CD pipeline and guarantee the integrity of the code that runs in production.

Testing continuously

In each of the three areas I just defined, the security controls implemented remain fairly simple to apply in isolation. The difficulty comes from testing and implementing them everywhere and all the time. This is where test-driven security comes in. TDS is a similar approach to test-driven development (TDD), which recommends developers write tests that represent the desired behavior first, and then write the code that implements the tests. TDS proposes to write security tests first, representing the expected state, and then implement the controls that pass the tests.

In a traditional environment, implementing TDS is difficult because tests must run on systems that live for years. But in DevOps, every change to the software or infrastructure goes through the CI/CD pipeline and is a perfect place to implement TDS, as shown in figure 1.5.

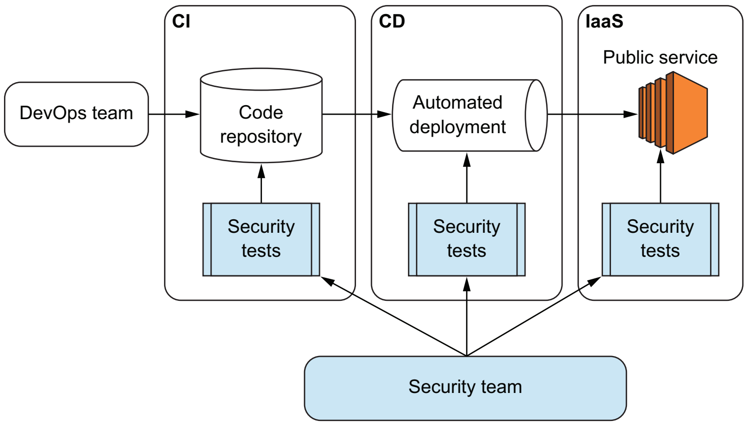

Figure 1.5 Test-driven security integrates into CI/CD to run security tests ahead of deployment in the production infrastructure.

The TDS approach brings several benefits:

- Writing tests forces security engineers to clarify and document expectations. Engineers can build products with the full knowledge of the required controls rather than catching up post-implementation.
- Controls must be small, specific units that are easy to test. Vague requirements such as "encrypt network communication" are avoided; instead, we use the

explicit "enforce HTTPS with ciphers X, Y, and Z on all traffic," which clearly states what's expected.

- Reusability of the tests across products is high, as most products and services share the same base infrastructure. Once a set of baseline tests is written, the security team can focus on more-complex tasks.
- Missing security controls are detected prior to deployment, giving developers and operators an opportunity to fix the issues before putting customers at risk.

Tests in the TDS approach will fail initially. This is expected to verify their correctness once they pass, after the feature is implemented. At first, security teams should help developers and operators implement controls in their software and infrastructure, taking each test one by one and providing guidance on implementation, and eventually transferring ownership of the tests to the DevOps teams. When a test passes, the teams are confident the control is implemented correctly, and the test should never fail again.

An important part of TDS is to treat security as a feature of the product. This is achieved by implementing controls directly into the code or the systems of the product. Security teams that build security outside of the applications and infrastructure will likely instigate a culture of distrust. We should shy away from this approach. Not only does it create tensions between teams, it also provides poor security as controls aren't aware of the exact behavior of the application and miss things. A security strategy that isn't owned by the engineering teams won't survive for long and will slowly degrade over time. It's critical for the security team to define, implement, and test, but it's equally critical to delegate ownership of key components to the right people.

TDS adopts the DevOps principles of automating the pipeline and working closely with teams. It forces security folks to build and test security controls within the environments adopted by developers and operators, instead of building their own separate security infrastructure. Covering the security basics via TDS significantly reduces the risk of a service getting breached but doesn't remove the need for monitoring production environments.

1.3.2 Monitoring and responding to attacks

When security engineers get bored, we like to play games. A popular game we used to play in the mid-2000s was to install a virtual machine with Windows XP completely unpatched, plug it directly into the internet (no firewall, no antivirus, no proxy), and wait. Can you guess how long it took for it to get hacked?

Scanners operated by malware makers would detect the system in no time and send one of the many exploit codes Windows XP was vulnerable to. Within hours, the system was breached and a backdoor was opened to invite more viruses to contaminate the system. It was fun to watch, but more importantly, it helped teach an important lesson: all systems connected to the internet will eventually get attacked—there are no exceptions.

Operating a popular service on the public internet is, in essence, similar to our Windows XP experiment: at some point, a scanner will pick it up and attempt to break in.

The attack might target specific users and try to guess their passwords, it might take the service down and ask for a ransom, or it might exploit a vulnerability in the infrastructure to reach the data layer and extract information.

Modern organizations are complex enough that covering every angle at a reasonable cost is often not possible. Security teams must pick priorities. Our approach to monitoring and responding to attacks focuses on three areas:

- Logging and fraud detection
- Detecting intrusions
- Responding to incidents

Organization that can achieve these three items are prepared to face a security incident. Let's take a high-level view of each of these phases.

Logging and detecting fraud

Generating, storing, and analyzing logs are areas that serve every part of the organization. Developers and operators need logs to track the health of services. Product managers use them to measure the popularity of features or retention of users. With regards to security, we focus on two specific needs:

- Detecting security anomalies
- Providing forensic capabilities when incidents are being investigated

Although ideal, log collection and analysis is rarely possible. The sheer amount of data makes storing them impractical. In part 2 of this book, I talk about how to select logs for security analysis and focus our efforts on specific parts of the DevOps pipeline.

We'll explore the concept of a *logging pipeline* to process and centralize log events from various sources. Logging pipelines are powerful because they provide a single tunnel where anomaly detection can be performed. It's a simpler model than asking each component to perform detection themselves but can be difficult to implement in a large environment. Figure 1.6 shows an overview of the core components of a logging pipeline, which I cover in detail in chapter 7. It has five layers:

- A collection layer to record log events from various components of the infrastructure
- A streaming layer to capture and route the log events
- An analysis layer to inspect the content of logs, detect fraud, and raise alerts
- A storage layer to archive logs
- An access layer to allow operators and developers to access logs

A powerful logging pipeline gives a security team the core functionalities it needs to keep an eye on the infrastructure. In chapter 8, I talk about how to build a solid analysis layer in the logging pipeline and demonstrate various techniques that are useful for monitoring systems and applications. It will set the foundations that we need to work on intrusion detection in chapter 9.

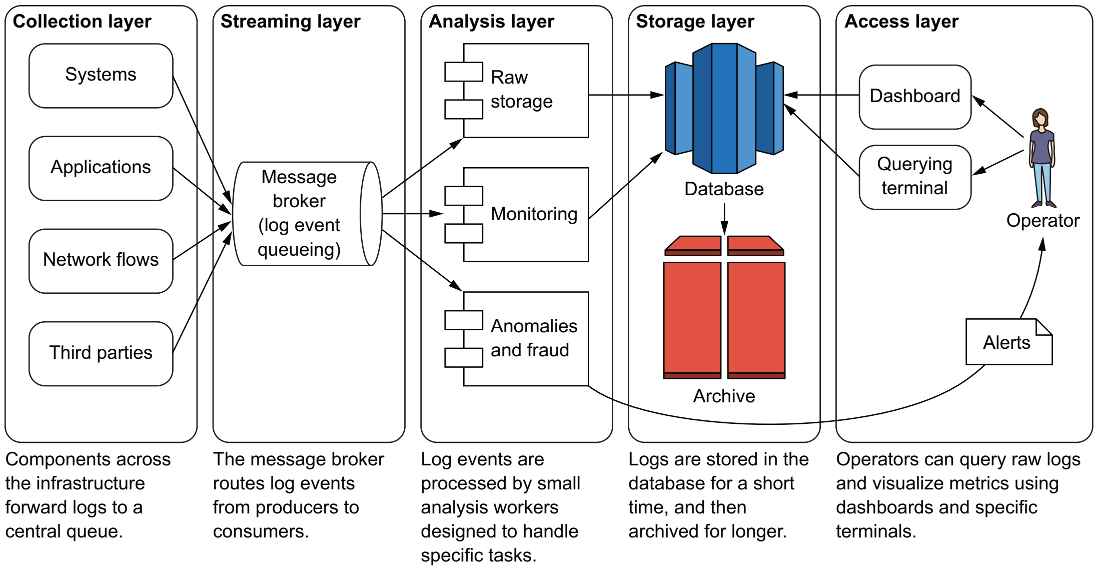
Collection layer
Systems
Applications
Network flows
Third parties
Streaming layer
Message broker (log event queueing)
Analysis layer
Raw storage
Monitoring
Anomalies and fraud
Storage layer
Database
Archive
Access layer
Dashboard
Querying terminal
Operator
Alerts
Components across the infrastructure forward logs to a central queue.
The message broker routes log events from producers to consumers.
Log events are processed by small analysis workers designed to handle specific tasks.
Logs are stored in the database for a short time, and then archived for longer.
Operators can query raw logs and visualize metrics using dashboards and specific terminals.

Figure 1.6 A logging pipeline implements a standard tunnel where events generated by the infrastructure are analyzed and stored.

DETECTING INTRUSIONS

When breaking into an infrastructure, attackers typically follow these four steps:

1. Drop a payload on the target servers. The payload is some kind of backdoor script or malware small enough to be downloaded and executed without attracting attention.
2. Once deployed, the backdoor contacts the mother ship to receive further instructions using a command-and-control (C2) channel. C2 channels can take the form of an outbound IRC connection, HTML pages that contain special keywords hidden in the body of the page, or DNS requests with commands embedded in TXT records.
3. The backdoor applies the instructions and attempts to move laterally inside the network, scanning and breaking into other hosts until it finds a valuable target.
4. When a target is found, its data must be exfiltrated, possibly through a channel parallel to the C2 channel.

In chapter 9, I explain how every single one of these steps can be detected by a vigilant security team. Our focus will be on watching and analyzing network traffic and system events using these security tools:

- *Intrusion detection system (IDS)*—Figure 1.7 shows how an IDS can detect a C2 channel by continuously analyzing a copy of the network traffic and applying complex logic to network connections to detect fraudulent activity. IDSs are great at inspecting gigabytes of network traffic in real time for patterns of fraudulent activity and, as such, have gained the trust of many security teams. We explore how to use them in an IaaS environment.

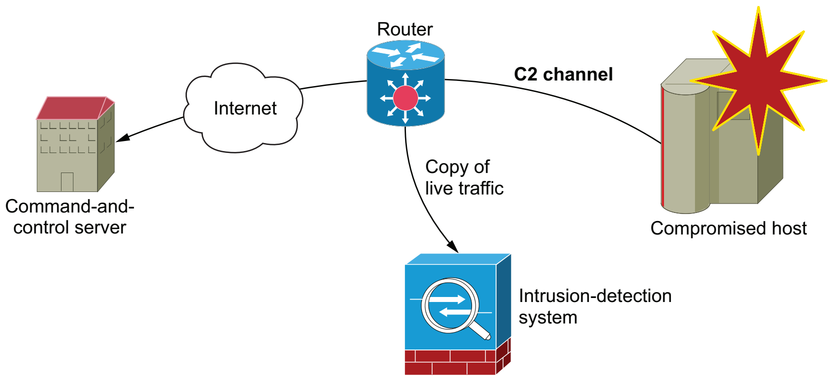

Figure 1.7 Intrusion-detection systems can detect compromised hosts calling home by finding patterns of fraudulent activity and applying statistical analysis to outbound traffic.

- *Connection auditing*—Analyzing the entire network traffic going through an infrastructure isn't always a realistic approach. NetFlow provides an alternative to audit network connections by logging them into the pipeline. NetFlow is a great way to audit the activity of the network layer in an IaaS environment when low-level access isn't available.
- *System auditing*—Auditing the integrity of live systems is an excellent way to keep track of what's happening across the infrastructure. On Linux, the audit subsystem of the kernel can log system calls performed on a system. Attackers often trip on this type of logging when breaching systems, and sending audit events into the logging pipeline can help detect intrusions.

Detecting intrusions is difficult and often requires security and operations teams to work closely together. When done wrong, these systems can consume resources that should be dedicated to operating production services. You'll see how a progressive and conservative approach to intrusion detection helps integrate it into DevOps effectively.

INCIDENT RESPONSE

Perhaps the most stressful situation any organization can find itself in is dealing with a security breach. Security incidents create chaos and bring uncertainty that can severely damage the health of even the most stable companies. As engineering teams scramble to recover the integrity of their systems and applications, leadership must deal with damage control and ensure the business will return to normal operations as quickly as possible.

In chapter 10, I introduce the six-phases playbook organizations should follow when reacting to a security incident. They are as follows:

- *Preparation*—Make sure you have the bare minimum processes to deal with an incident.
- *Identification*—Decide quickly whether an anomaly is a security incident.
- *Containment*—Prevent the breach from going any further.

- *Eradication*—Remove threats from the organization.
- *Recovery*—Bring the organization back to normal operations.
- *Lessons learned*—Revisit the incident after the fact to learn from it.

Every security breach is different, and organizations react to them in specific ways, making it difficult to generalize actionable advice to the reader. In chapter 10, we'll approach incident response as a case study to demonstrate how a typical company goes through this disruptive process, while using DevOps techniques as much as possible.

1.3.3 Assessing risks and maturing security

A complete continuous-security strategy goes beyond the technical aspects of implementing security controls and responding to incidents. Although present throughout the book, the "people" aspect of continuous security is the most critical when approaching risk management.

ASSESSING RISKS

For many engineers and managers, risk management is about making large spreadsheets with colored boxes that pile up in our inbox. This is, unfortunately, too often the case and has led many organizations to shy away from risk management. In part 3 of this book, I talk about how to break away from this pattern and bring lean and efficient risk management to a DevOps organization.

Managing risk is about identifying and prioritizing issues that threaten survival and growth. Colored boxes in spreadsheets can indeed help, but they're not the main point. A good risk-management approach must reach three targets:

- Run in small iterations, often and quickly. Software and infrastructure change constantly, and an organization must be able to discuss risks without involving weeks of procedures.
- Automate! This is DevOps, and doing things by hand should be the exception, not the rule.
- Require everyone in the organization to take part in risk discussions. Making secure products and maintaining security is a team effort.

A risk-management framework that achieves all three of these targets is presented in chapter 11. When implemented properly, it can be a real asset to an organization and become a core component of the product lifecycle that everyone in the organization welcomes and seeks.

SECURITY TESTING

Another core strength of a mature security program is the ability to evaluate how well it's doing on a regular basis through security testing. In chapter 12, we'll examine three important areas of a successful testing strategy that help mature the security of an organization:

- Evaluating the security of applications and infrastructure internally, using security techniques like vulnerability scanning, fuzzing, static code analysis, or configuration auditing. We'll discuss various techniques that can be integrated in a CI/CD pipeline and become part of the software development lifecycle (SDLC) of a DevOps strategy.

- Using external firms to audit the security of core services. When targeted properly, security audits bring a lot of value to an organization and help bring fresh ideas and new perspectives to a security program. We'll discuss how to use external audit and "red teams" efficiently and make the best use of their involvement.
- Establishing a bug bounty program. DevOps organizations often embrace open source and publish large amounts of their source code publicly. These are great resources for independent security researchers that, in exchange for a few thousand dollars, will perform testing of your applications and report security findings to you.

Maturing a continuous security program takes years, but the effort leads security teams to become an integral part of the product strategy of an organization. In chapter 13, we'll end this book with a discussion on how to implement a successful security program over a period of three years. Through close collaboration across teams, good handling of security incidents, and technical guidance, security teams acquire the trust they need from their peers to keep customers safe. At its core, a successful continuous security strategy is about bringing security people, with their tools and knowledge, as close as possible to the rest of DevOps.

Summary

- To truly protect customers, security must be integrated into the product and work closely with developers and operators.
- Test-driven security, monitoring and responding to attacks, and maturing security are the three phases that drive an organization to implement a continuous security strategy.
- Techniques from traditional security, such as vulnerability scanning, intrusion detection, and log monitoring, should be reused and adapted to fit in the DevOps pipeline.