O'REILLY®

Compliments of
Red Hat
Advanced Cluster Security
for Kubernetes

DevSecOps in Kubernetes

Wei Lien Dang &
Ajmal Kohgadai

REPORT

Red Hat
Advanced Cluster
Security
for Kubernetes

Continuous Security for Cloud-Native Applications

Red Hat Advanced Cluster Security for Kubernetes — The first Kubernetes-native security platform that enables organizations to securely build, deploy, and run cloud-native applications anywhere.

Lower Operational Cost

DevOps and Security teams can use a common language and source of truth

Reduce Operational Risk

Ensure alignment between security and infrastructure to reduce application downtime

Increase Developer Productivity

Leverage Kubernetes to seamlessly provide guardrails supporting developer velocity

Secure Supply Chain

Standardize on Kubernetes as your common platform for declarative and continuous security

Secure Infrastructure

Leverage native controls for built-in security that works with Kubernetes, not against

Secure Workloads

Accelerate your pace of innovation while ensuring your applications comply with security policies

Contact us today to get started
redhat.com/products/kubernetes-security

DevSecOps in Kubernetes

Wei Lien Dang and Ajmal Kohgadai

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

DevSecOps in Kubernetes

by Wei Lien Dang and Ajmal Kohgadai

Copyright © 2021 O'Reilly Media. All rights reserved.

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://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor Gary O'Brien
Production Editor: Daniel Elfanbaum
Copyeditor: Penelope Perkins
Proofreader: Christina Edwards
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2021: First Edition

Revision History for the First Edition

2021-07-22: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *DevSecOps in Kubernetes*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Red Hat. See our statement of editorial independence.

978-1-098-10175-6

[LSI]

Table of Contents

Introduction vii

1. Software Development Life Cycles 1
The Value of Processes 1
DevOps 2
Kubernetes and the Software Development Life Cycle 4
Summary 5

2. Architectural Designs 7
Traditional Application Design 8
Service-Oriented Architecture 10
Cloud Native Design 12
Cloud Native Design with Kubernetes 16
Summary 17

3. DevOps and DevSecOps 19
DevOps 20
DevSecOps and Shifting Left 26
DevSecOps with Kubernetes 29
Summary 30

4. Security and Requirements 33
Risk and Threat and Vulnerability 34
Threat Modeling 37
Summary 43

5. **Managing Threats** 45
Open Source Threat Intelligence 46
Attack Phases 48
Using an Attack Matrix 52
Summary 55

6. **Wrapping Up** 57

Introduction

You know that security is important. And whether your system is cloud native, has transitioned into the cloud with a traditional architecture, or is just starting that journey, you know that the shift into the cloud has made security more complex than ever. What's more, security is now everyone's job.

For decades, software development and delivery were slow processes. Early enterprise software was delivered to customers by hand and installed by a trained technician. Cloud providers have changed all that, leveraging economies of scale to offer capabilities that many organizations couldn't achieve on their own and making it cheaper and easier to use virtualization technologies.

Fortunately, as more application hosting is outsourced to cloud providers, the resulting changes have brought design, development, testing, deployment, and management teams closer together. That's important, because while virtualization offers fantastic opportunities and capabilities, it also means there are many more moving pieces than there used to be. To handle that, not only do you need solid management capabilities—you also need automation.

In the cloud, automation is often referred to as orchestration, because there is one piece in the middle that has to keep all the elements in tempo and on key. Using a tool like Kubernetes to orchestrate the development, deployment, and runtime phases of containerized applications can help immensely with automating and scaling application delivery, but it's not magic. You'll still need to bring together all of the groups involved in development, orchestration, and deployment, because all of them will have different and important insights. For example, when you introduce orchestration,

that means development work is needed to create scripts and configurations. Those will then need to be tested and, of course, the team responsible for deployment needs to weigh in on whether they have what they need.

It's complicated, but with cultural change within your organization and a shift in viewpoint, you can bring all the right people, perspectives, and insights together into a single team. That's where DevOps in general, and the DevSecOps model in particular, can help. Don't worry if you're not sure what that means just yet. I'll introduce you to DevOps in Chapter 1, and by Chapter 3 we'll be wading into DevSecOps. Along the way, we will be using Kubernetes as an example of a technology that works well with a DevSecOps culture.

We'll get into some of these cloud native technologies later on, but there are two big aspects of cloud native that Kubernetes can support. The first is automation. Kubernetes natively supports the automation of all of your elements so deployment of applications is easier and more manageable. On top of the automation, Kubernetes brings abstraction. Software development has long tried to abstract away from hardware as much as possible, and more granular levels of virtualization in application development and management take us much further away from the hardware. Kubernetes supports that abstraction, allowing developers to focus on functionality without needing to worry about how all the different components go together.

If you only know one thing about DevSecOps right now, it should be this: security isn't the job of an isolated team. Security needs to be baked into every step of the overall development life cycle and owned by the development and operations teams as much as it is by the security team. It takes work to identify the right tools, processes, and requirements for your application design and implementation. This report will walk you through the basics of how to do that, starting at the beginning with generating requirements and cascading through the rest of the life cycle. Think of it as a first step toward enhancing your security.

If your work touches or manages any part of the software development life cycle (SDLC), this report is for you. In the first half, we'll look at software development methodologies and architectural designs, with a focus on how to integrate security measures into these processes as a whole. Then, in the second half, we'll dive into

DevSecOps and the art and science of embedding security into the SDLC to identify and address threats. By the end of this report, you'll have a better understanding of how adopting the DevSecOps mindset can make your team and organization stronger, more resilient, and more secure, from start to finish. Along the way, we will look at how technologies like Kubernetes can be leveraged to better enable DevSecOps philosophies and practices.

CHAPTER 1

Software Development Life Cycles

The problem with software development, as an intellectual activity, is that you can't manage development projects in the same way you can other projects. With a project like building a house, adding more people to the project (as long as they can operate construction tools) can shorten the overall time it takes to complete the house. In software development, the relationship between people and time to complete is much more complex.

In this chapter, we'll take a quick look at the origins of software development. We will examine the process's benefits and the essential stages that any methodology must include. Then we'll dive a little deeper into the pros and cons of several of the most popular methodologies.

The Value of Processes

Why bother with a methodology? Developers and project managers have learned a lot since the early days of serious software development projects of the 1960s. They've created a variety of methodologies to help get large-scale software projects (and even small ones) done faster, more efficiently, and with higher quality. Sometimes the new methodologies spring from a desire to solve a problem the developer experienced with another methodology.

Different methodologies have different goals and different problems. A methodology that works wonderfully for one organization

might be a bad fit for another. One thing is clear, though: having a methodology is vital.

Repeatability and Consistency

One goal of any process or methodology is repeatability: you know how to do it again. When you are building something to be sold, you want consistency. You want to know that you've done everything as well as it can be done and in the same way as all the other things, without variation. Variability can lead to defects, which will of course make it more difficult to sell your product. In software, defects are bugs, and no one wants to release buggy code. Consistency is one way to help limit the number of bugs that are in your released software.

Process Improvement

With a process, you can evaluate how you're doing something and learn from how well it works. You can constantly improve your process to remove defects. (Some methodologies emphasize reducing defects, such as Six Sigma.)

Focusing on process improvement is one of the hallmarks of a mature organization. In Capability Maturity Model Integration, a maturity model developed at Carnegie Mellon University, the highest level of maturity an organization can reach is Optimizing. This is where the organization has developed a process that it follows regularly and puts effort into learning lessons and improving that process over time.

There may not be as much concern for individual projects or even very small projects getting repeatability, consistency, and process improvement, but commercial organizations should be concerned with these. Without the right methodology to help organizations focus on these factors (or, worse, without any methodology at all), the quality of the product is likely to be poor or inconsistent.

DevOps

Historically, there have been a lot of problems with software development methodologies. Long-standing methodologies like waterfall result in long development times with limited communication. This

can lead to high cost and low quality, which are not good results from a software development process.

Communication problems are nothing new in software development. As companies move toward web applications and web-based deployments, operations teams have become bigger stakeholders than they have in the past. There are several great comics and graphics that highlight how software development used to happen before DevOps became popular. These illustrations usually involve a developer on one side of a wall tossing a software project over to the other side of the wall where the operations person stands. There is no communication and there may not even be much of anything in the way of documentation.

DevOps is not so much a methodology, as it is a culture. The culture of DevOps is meant to more tightly integrate the operations staff with the development staff. The operations staff is meant to be there through the requirements gathering to ensure they can deploy and manage the application when it's developed.

DevOps focuses a lot on automation and testing. The goal of DevOps is to improve the overall quality of software development processes by automating as many aspects of software development, building, and deployment as possible. Many DevOps organizations use something called a development pipeline, which means that when software is checked in to the code repository, indicating the developer has completed a task and they want to return the file to the repository so someone else can use it, an automated process starts to perform tests, build the software, and then perform more testing. An approach like this means you can ensure the testing is completed prior to deployment rather than rushing to deployment while testing is done in parallel.

This is not to say that all tests can be automated. In some cases, the testing team will still need to perform testing while deployment is going on, but with short development cycles, any bugs that are found can be resolved quickly and moved into production quickly.

All of this automation means that processes can be fully tested. When you move to automation, you have another piece of software that can be tested because the automation is a program or script. This removes the human potential for error from the equation. The automation programs or scripts can be fully tested to ensure they do

exactly what they are supposed to do prior to using them. Any changes can similarly be tested prior to moving into production.

Kubernetes and the Software Development Life Cycle

When developing software, whether it's a complete application or just an application programming interface that can be used by either a web-based or mobile frontend, someone needs to be thinking about the whole application life cycle—not just the part where the developers are building software. Kubernetes can help support these efforts. During the requirements gathering phase, a team may find that the best way to support the overall goals of the project is to implement Kubernetes. This will not only resolve some requirements, potentially, but also introduce other requirements. Applications are not usually fully formed out of the gate, but instead a picture of what the application should be arises from an iterative view of requirements. One requirement begets another requirement and so forth until the entire view of the application or project is complete.

When a development project commences, there is a requirements phase during which a lot of decisions about the direction of the project are made. This is where Kubernetes comes in, since it is best to decide to select Kubernetes as the orchestration platform up front. This will direct some of the development efforts since Kubernetes was designed for containers. The developers and operations staff know from the very beginning that they will be working with a virtualized application environment. This also means additional development is necessary, or at least a lot of communication is required here since the operations team will be responsible for deploying the project, which means they will probably also be responsible for developing the orchestration scripts and configurations that will be managed through Kubernetes. This is where DevOps and DevSecOps comes in handy as a practice, which we will get into in more detail in Chapter 3.

One problem, historically, with testing has been that test environments look different from development environments. Developers write software on their systems, which are configured entirely differently from test systems. Testing containerized applications can be made considerably easier in a Kubernetes environment, since

Kubernetes will help make sure every instance of the application space looks the same every time. This means the testing team can be working in exactly the same environments as the development team. When you abstract away from the individual system and virtualize the application, you get the benefit of the application running in an isolated space with only the prerequisites and dependencies necessary for the proper functioning of that application.

Operations teams have spent years trying to make their lives easier, first by writing scripts and creating frameworks to support large numbers of physical systems, and then later doing the same for virtual machines and all the networking and other supporting infrastructure necessary for a complete and complex web-based application. In both deployment and post-deployment, Kubernetes supports operations teams by providing a management framework that takes care of all orchestration that teams had to previously perform manually due to a lack of knowledge or resources, including provisioning of compute resources, storage, and networking, to name a few.

Deciding to use Kubernetes can have a big, positive impact on the overall development of an application from requirements through deployment and operation of the application. It's important to ensure all team members are aware of the benefits and requirements of the use of a tool like Kubernetes, though.

Summary

Software development is a complex task with a lot of people involved in addition to the software developers themselves. Just writing code is a complex endeavor. It helps to have a development methodology to ensure the software that is developed is what the business, the user, or the customer is looking for. As always, identify the problem before identifying the solution.

There have been a lot of different development methodologies over the years. In some cases, a development methodology is a philosophy or a culture as much as anything else. This is especially true when it comes to DevOps. DevOps fits very well with modern software development practices, though it can be a challenge to implement due to the strong focus on communication. For entrenched development teams used to walls or silos constraining the different teams, this can be a challenge that needs to be overcome. Focusing

project management, development, testing, and operations around a single platform like Kubernetes can help guide the necessary communication across teams because everyone is working in the same direction on the same technology, which has not always been the case with development teams.

Additionally, DevOps fits well with Kubernetes, which requires support from all of the team members across the SDLC. Deciding to use a tool like Kubernetes during the requirements/design phase of the life cycle can introduce additional requirements but can also provide a lot of answers to other team members like developers and testers, which may make life easier and contribute to the overall quality of the application.

Beyond this teaser, we will spend the rest of this book looking at other practices within DevOps that will be beneficial for high-quality application development.

CHAPTER 2
Architectural Designs

Software applications are complex animals. There are a lot of components. When it comes to web-based applications, there may be a lot of systems involved in addition to the elements that are software-based. When companies develop web-based applications, they become responsible for a lot more than just software development since they also have to be concerned with the delivery platforms. This is a significant change from developing native applications (meaning applications that run directly on your desktop), where the end user is responsible for the platforms the software runs on.

Traditionally, web-based applications have used something called an n-tier design, which is a common approach to software development in general. Even in native applications, you can see the n-tier design in use. For the last decade and more, though, there has been a push to a different model in application development. With more companies using cloud services to deliver applications, whether specifically for delivery through a web browser or through a mobile application with a web-based backend, there has been a shift to service-oriented architecture. This shift has changed how applications are delivered as well as how they are designed and developed. It has also changed the security posture of web applications and how they are managed and deployed.

In this chapter, we will look at some traditional application designs as well as some more modern approaches to developing larger systems and applications. These more modern approaches are especially common with web applications, which we see taking over

native applications in many areas. Each design has some security considerations and we'll take a look at those too.

Traditional Application Design

An n-tier application design has multiple components that communicate with one another in well-defined ways that you can think of as levels or tiers. One common approach to n-tier application design is the model-view-controller (MVC) design pattern. The model is where data is stored and represented within the application. The view is how that same data is presented to the user. This may be a very different representation than the one used within the model because the model has to make sense from a programmatic perspective, while the view needs to be visual. That is, a user needs to be able to look at it and know what is happening within the application. The controller, finally, is the intermediate tier that takes the data out of the model and transforms it (either coming out of the model or going into the model). This tier is where business logic may exist and it prevents the view from communicating directly with the model. This allows you to create interface-agnostic programs. You may end up with the same model for different types of applications, for instance. The view may change but the model and the controller can remain in place, as long as the interface (the calls to functions, methods, and properties) remains the same between views.

A traditional web application uses something similar to this design pattern. Rather than being called MVC, though, it may be called a three-tier or n-tier design. This is because there may be multiple layers, but they may not align nicely to the way MVC is described. Figure 2-1 shows a traditional web application design, built for resilience. On the left side of the diagram is the user. You might think of this as the presentation layer, where the output from the application is displayed and any input from the user is accepted and sent back to the rest of the application. The diagram shows the user communicating with the application through the internet. This may not be the case in practice, though, since the application could be hosted within an enterprise data center, with the user communicating over the enterprise network. Either way, the user is sending traffic to a network they are not directly connected to (their local network), no matter who happens to own the network.

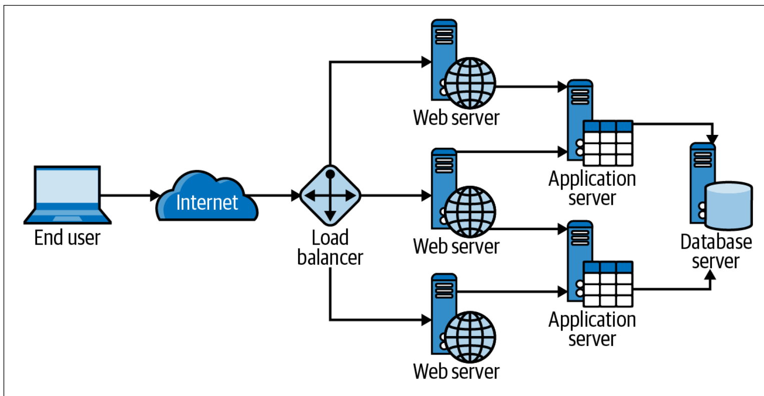

Figure 2-1. Traditional web application

Inside the application there may be multiple networks. A tiered network design is generally preferred because it theoretically makes it harder for an attacker to get deeper into the application where more sensitive data is stored. In the center of Figure 2-1, you'll see a load balancer. This is done not only for resilience, but also to more easily handle a large load. The load balancer's job is simple: it accepts network requests and funnels them to a set of defined servers. There is almost no processing involved at all. The servers, though, have to do some work handling the requests so they can get busy from the perspective of processor utilization. This is less common with much faster processors today than were available even a decade ago, but application design may still follow this pattern.

Behind the web server, which serves up any static content and sends dynamic requests on, is the application server. This is where the business logic lives. The application server may be a .NET server that can run programs written in any of the .NET languages, such as VB.NET or C#. It's also common to use Java-based web application servers like JBoss, Oracle WebLogic, and Apache Tomcat. The application server handles all dynamic requests where programmatic processing is needed. Input from the user is handled by the application, and any response back to the user is generated, primarily, by the application server.

The application server also manages the interface with the database server, which you can think of as the model in the MVC approach to application design. Traditionally, the database server has been a relational database, where data is stored in rows and columns in tables.

The columns define the properties in the table while the rows are collections of the instances of the properties. They are called relational databases because each table may be related to another table because one column in each of these tables may contain the same data. This is how you can have complex data sets without making each table too complex or unwieldy.

In practice, the traditional architecture requires an individual system for each of the tiers. In fact, there may be multiple systems at each tier, as shown in Figure 2-1, where three web servers sit behind the load balancer. At one point, this was done with physical systems, but more recently, it is usually done with virtual machines. A virtual machine is a system that is defined by software: there is a piece of software that intercepts all of the requests from the operating system that would normally go to the hardware. The guest operating system has no idea there is a piece of intermediate software acting as though it were hardware. This allows you to have multiple "systems" running on top of a single operating system, saving you the cost of hardware and other support costs.

All of this additional overhead from the operating system to all the supporting applications needed requires work. You may need people who manage the operating system and all that entails (updates, patches, configuration, etc.). You may need people who manage the applications. Ideally, every system that is deployed, whether virtual or physical, has been hardened, meaning it has been configured in such a way that it is difficult for attackers to compromise from the outside. All of this work and the people it requires is one reason companies are moving away from traditional application design, especially for cloud deployments.

Service-Oriented Architecture

In the traditional design just discussed, applications are composed of multiple services that are working together: you have a web server, an application server, and a database server. All of these servers, whether they are physical or virtual, are really just services that expose an interface to the rest of the network to interact with as needed. If you deconstruct each of these servers to the functions they provide underneath, you end up with a lot of services within the application. A traditional application has a central processing node, like the application server. In a service-oriented architecture,

everything is more exposed. Figure 2-2 shows a simple diagram of what that might look like.

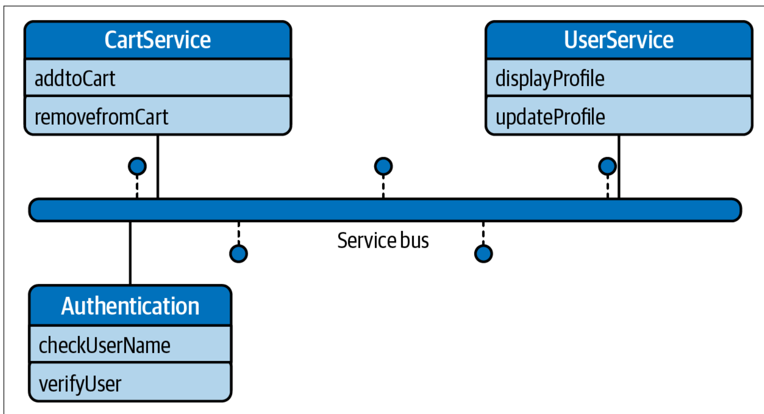

Figure 2-2. Service-oriented design

In a service-oriented design, there is a central means of communication called a service bus. Messages are put on the bus and the bus takes care of getting each message to the right service to be consumed. For example, if you want to call an authentication function from the profile service, you just put the message (basically, the method call including the name and any parameters the method might need) onto the bus and the bus takes care of calling it. The response to the method call also gets put onto the bus and it gets returned to the correct calling party.

You may be wondering why this is a useful way of designing things, since we're just calling functions/methods and getting responses from them. This is just how programming normally works. The difference with this design is that the bus can manage services spread out across multiple systems. You are no longer constrained by having to have a monolithic application design with all the components on one system. You are also not constrained by sometimes needing to use cumbersome remote application services like remote method invocation (RMI) or remote procedure calls (RPCs). The service bus is a different piece of software that does all the communication management.

In theory, you could use a service-oriented architecture inside a traditional application design. Your application server could host the service-oriented architecture behind the scenes, while still exposing

the traditional application server interface to the rest of the world. Calls to the application would get passed to the service bus and handled by the different methods that are connected to the bus.

In practice, the service-oriented architecture becomes an API. The entire application can be managed from outside the application internals by simply calling the different methods available. This opens the door to something called microservices. Without the need to have a monolithic software application, the application can be deconstructed into individual pieces that other components or services may interact with in different ways. This can mean you end up with multiple applications, as functionality is put together in different ways. No longer does a single entity control the flow of the application.

Service-oriented architecture can also help to enforce a programming practice called design by contract, where each method or function within the overall application space has a set of expectations that must be met (the contract) before it can be called. If you have arbitrary entities calling methods within the application space, it may be essential to have those contracts in place. If the contract isn't met, meaning the preconditions aren't in place as defined by the function or method, the call to the function or method will fail. This helps to drive better programming behavior by ensuring only "correct" data is sent into functions. This reduces the burden on the function so it can focus on what it's supposed to do without having to make sure the data passed in falls within acceptable bounds.

Once you start to deconstruct the application, you no longer need so many servers. Nor do you care about traditional design where every service has a physical or virtual server. The only thing you care about is the individual components of the application, meaning the functions or methods that comprise the interface to the application.

Cloud Native Design

Cloud native design builds on top of the move away from the traditional monolithic applications each having their own server approach that service-oriented architecture takes. Cloud native design takes advantage of that decoupling of traditional services and servers. It uses virtualized deployments, but rather than virtualizing an entire system, it virtualizes an application. From a security

perspective, this is a good thing. Before we get there, though, let's take a look at what containers look like.

Figure 2-3 shows a diagram of containers. At the very bottom is all of the infrastructure required to make any computer system run—power, hardware, networking, etc. Above that is the operating system that runs on top of the hardware. The next layer is the container software, which takes care of all of the application virtualization, meaning it manages the interactions between the application and the operating system.

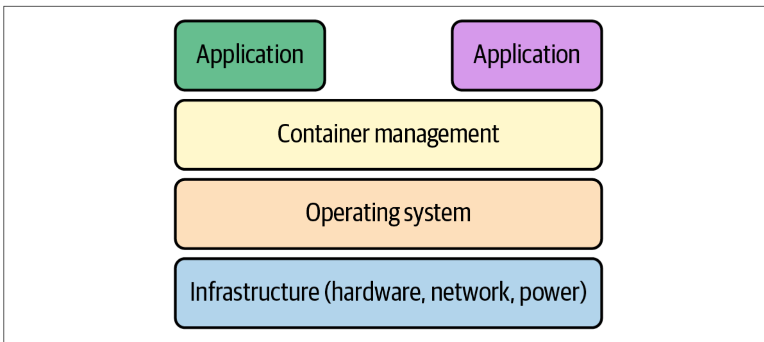

Figure 2-3. Container design

The way this works is by tagging memory in the actual operating system. The containerized application lives in this tagged space and can only read or write to memory that has the same tag. It is up to the operating system to enforce this control. This means the containerized application and anything within that space can't communicate with the operating system memory space. It also can't interact with the memory of any other application. This design has benefits for security: any attacker who manages to compromise an application that has been containerized would get access to just the space of that application. If they managed to get interactive access to the space the application resides in (typically called shell access because the part of an operating environment the user interacts with is called the shell), they could not get to the underlying operating system. They would get a very limited view of files and processes running. This is not to say that containers are a perfect solution; vulnerabilities in the container software could allow more access than theory would suggest.

Another great thing about virtualized applications is that they are very easy and quick to start up. Starting up a containerized application can be so fast that it can be started in response to a user request, exist only so long as the request is being processed, and then go away. If the request was from an attacker who had managed to compromise the virtualized application, the access to that container would go away. Even if the attacker were able to maintain access to the container, it would exist only to serve the attacker—no other requests would go through that container so the attacker would have access to almost nothing.

You can take this virtualization a step further, and some cloud providers have. If you are building your own application, you are going to construct it from a number of functions or methods written by application developers. The application development team doesn't care about anything other than the function they are writing. Why not virtualize the function itself so it only exists as long as it is being called? At that point, there really is nothing underneath the function (in theory) that could be compromised in any way. You get a small chunk of memory that belongs to the function, just like a function would get a stack frame in a native application.

You can develop applications using these techniques using cloud providers. Providers such as Microsoft Azure, Google Cloud Platform (GCP), and Amazon Web Services (AWS) all have serverless functions that can be used to develop and deploy an application. This can make life a lot easier from the perspective of the security professionals as well as the system administration team.

Management Considerations

Once you start decoupling functions from applications and applications from servers, meaning the application no longer lives within a single application server space, you have a lot of moving pieces to manage. This is also true when you are talking about large-scale applications where the different application servers, such as the web server, application server, or even the database server, are run from inside a container. Suddenly, you have a lot of virtualization possibilities and a lot of different pieces that need to be managed.

As mentioned earlier, once you start virtualizing applications and even functions, you have a lot to juggle. Application virtualization speeds up deployment, which means you can easily design your

applications to make use of this speed. You can respond very quickly to load, meaning volume of requests, by standing up individual instances of the virtual components on an as-needed basis.

Alongside this capability, though, is the problem of maintaining the application content and configuration. You need the ability to easily manage the life cycle of each virtualized application. Content within a web server's space, for instance, may change. If the content is stored inside a container, any update to the content requires tearing the container down and replacing it. Similarly, if the application changes, you need to be able to take down all the containers that are running it when you update the application. After all, one advantage of using web-based applications is the control you have over the version that has been deployed. If you can't manage the lifetimes of the containers running your code, you start to have less control over your application and the version being presented to users, which leads to inconsistent behaviors within the application and potentially disgruntled users.

A common container runtime implementation is Docker. Docker provides the interface between the operating system and the application being virtualized. It will run on common operating systems like Linux and Windows. While you can develop your own applications and containerize them within Docker, you might also take advantage of the Docker Hub registry, which stores implementations of a number of common applications. Docker will automatically download the latest for you, initialize it, and then run the application.

While you can definitely make use of Docker, containerd, or CRI-O to manage containers, you may find you need another piece of software to manage the deployment and configuration of the containers. Another software platform, such as Kubernetes, can make this process easier because in addition to providing a container implementation, it also provides a complete orchestration system to manage the deployment and life cycle of the containers. Google initially developed Kubernetes in 2014. It provides extensive capabilities for managing containerized applications across the entire life cycle of the container. This includes the abilities to schedule instances and manage the images being executed.

When you have extensive management capabilities of containers, you get access to just about everything cloud providers offer. You can better manage your application and all of the components.

When managed well, containerization can reduce the foothold an attacker can have within your application. There is a class of attackers called advanced persistent threats and their objective is to get access to your environment, essentially taking up residence in your systems, potentially for years. The incident response company Mandiant releases a report each year called M-Trends with a break out of the percentage of investigations each year where the attacker has been in place for multiple years. Each year there is a small percentage where the duration of residence has been more than five years.

Cloud Native Design with Kubernetes

As mentioned earlier, cloud native design uses virtualization as an important foundation. However, cloud native design is about far more than just virtualization or even using a microservice model for component design. Cloud native design is as much about being responsive to user demand as anything else. This is the sort of problem Kubernetes has been built for. When you move to a design where the application itself is containerized, you increase the responsiveness of the application. The application, or instances of it, can start up very quickly because it doesn't require an entire operating system to start up first. Once this becomes a reality, there are other possibilities.

One of the challenges with any application is upgrades. Whether the upgrade is to software that is underlying the application, like the operating system or any dependencies, or whether it's the application software itself, Kubernetes will take care of ensuring the latest software is in place for any instance of the application or its components. In the case of a microservices model, where there are multiple components that each require their own virtualized space, there are a lot of moving pieces and a lot of dependencies. Kubernetes can take care of replacing instances of each microservice, without any services-consuming users being aware that the replacement is happening. Any instance that is in use can be swapped out as soon as it falls out of use.

Because Kubernetes is capable of turning on instances of an application component or microservice quickly, it makes a great platform to enable rapid scaling. Kubernetes can ensure new instances of application components are in place as they are needed based on user demand. In the case of cloud computing with a service

provider, where you pay as you go, enabling services as they are needed saves a lot of money. Before containerization, massive marketing events like advertising during the Super Bowl for instance, would require a lot of work ahead of time. New physical systems or virtual machines would have to be built and then added to whatever load-balancing strategy was in place. This would need to be done well before the event because of the time required to perform all that work. In today's world, as long as you have created the application with the right components, such as using Kubernetes for the orchestration management, you can scale to demand without any additional work.

Summary

Cloud native design removes a lot of the constraints that were imposed by a traditional n-tier design, opening the door to more resilient and responsive application architectures. Using cloud native design, the same backend application can support both a web interface using a client's browser as well as a mobile application where the interface runs on a phone or tablet, while still using the application programming interface exposed by the backend application to drive the functionality of the mobile application.

Selecting an orchestration management platform like Kubernetes can make using a cloud native application design easier because you are in a better position to scale to demand. Kubernetes can take a lot of the heavy lifting because it's designed to be able to support this application scaling without additional work on the part of the developers to manage resources.

CHAPTER 3

DevOps and DevSecOps

While there are probably a lot of other reasons for the value, importance, and uptake of DevOps and DevSecOps, a significant one is the rise of web applications. When you start to deliver applications using the web, the way you structure your development process starts to change, because deployment can be (and often is) done much faster than the traditional approach, which can take months. Once your deployment cadence starts speeding up, there is a burden on the operations team to support that deployment—after all, it's your operations team that has to handle the deployment now, since the customer is no longer performing the installation.

DevOps has become a much more prevalent set of practices over time, as operations teams continue to get more say in the overall development process. There are advantages for others as well using DevOps. From a business perspective, of course, there is the potential to reduce overall costs. From the perspective of the development team, there is the potential to increase overall quality across the life cycle of the product being developed.

DevSecOps is another set of practices that is gaining a lot of traction, again driven by development shops focused on web application development. It is simplistic, though, to say that DevSecOps is just a question of inserting security into the existing DevOps culture. As security itself is as much a culture as anything else, it's not as simple as just saying "we do security with all the other stuff." DevSecOps is about practices that introduce security throughout the entire development life cycle.

This chapter will cover what DevOps is, as well as how it differs from DevSecOps. It will also cover the cultural changes that may be required to introduce or support these philosophies within an organization.

DevOps

There is an old saying that you can pick only two of the following traits: good, fast, and cheap. You can't have all three. The idea is that if you go the route of fast and good, it will be expensive. If you try good and cheap, it will take time to get it delivered. Fast and cheap will yield poor quality. At least, this has always been the thinking. Without commenting on cheap, because there is so much more involved in determining overall cost, DevOps is focused heavily on being good and fast. This does not necessarily bring cost along with it. One way you get to good and fast is to automate as much as possible. It's easy enough to see how this will get you fast, because automation should be much faster than having a human do the task. It may be harder to see how you get to good. We can investigate that in more detail as we keep discussing the benefits of DevOps.

DevOps cultures may be built around tools or toolchains. As you might expect from the name, the toolchain starts with the development part of the SDLC, rather than the earlier stages of requirements and design. DevOps cultures focus a lot around automation. The development stage, from a tooling perspective, is much less about the actual development (meaning the tooling isn't in the editors or development environments being used) than it is about the source code repository, where the source code is managed between multiple developers. There is intelligence required in managing the files to make sure one developer doesn't overwrite changes made by another developer.

Older source code management or versioning solutions might require a developer to check out a file before making any changes. This would in effect lock the file so no other developer could check it out. There was a version that was stored on a server somewhere and that was the version of record. While this is essentially still true in modern source code management solutions, for reasons you'll see shortly, more control is put into the hands of developers and it becomes more of a distributed process.

For example, if two developers, Jon and Tom, are working on a software project using a commonly used modern source code management solution like Git, they can both be working on the same source code file at the same time. This works because, if the project is well-managed, Jon and Tom will not be working on the same lines of code at the same time. Instead, they will be making changes to different sections of the file. When they are both done with their changes and have fully tested what they have done, they will push their changes up to the master repository. Let's say Jon pushes first. When Tom pushes his changes, the Git software will tell him there is a set of changes already in place that he doesn't have. He will then be able to merge his changes with the changes already in place in the master repository. Figure 3-1 shows a collection of folders and files managed using the public repository GitHub.

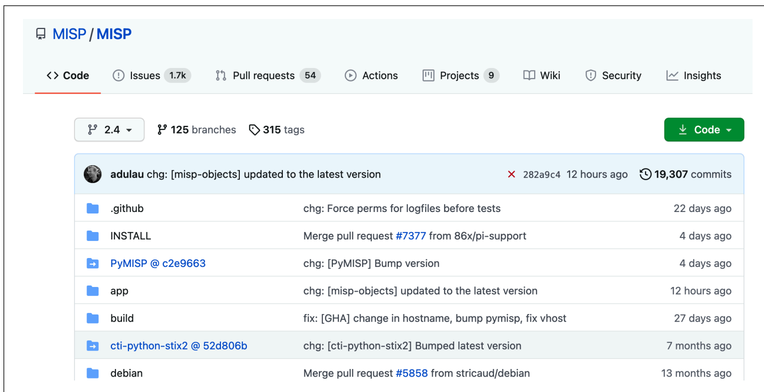

Figure 3-1. GitHub repository

Once the changes to the source code have been committed to the master repository, the software can be built. This is the process of taking the source code and compiling it to a form that is native to a computer's processor. Not all software needs to be compiled. Some will remain in its original, source code form. This is the case for languages that are interpreted, such as Python or JavaScript. In the case of an interpreted language, a build is a run through the source code to ensure there are no failures that result from the recent changes. This is similar to a compilation in the sense of looking for syntax errors, but it does not result in an executable as a compilation process does.

This build or construction step can be done automatically. If developers are trustworthy, the whole check-in process can be automated to the extent that once conflicts are merged the server may be able to push all the changes into the current branch of the software, ready for building and testing. Once the build has been completed and all files and components are in place, testing can also be automated. One way to do this is to ensure test cases are written alongside any functional development. If you are adding functionality to the software, you need to develop a test case that will validate that functionality. This should include a small function that will implement that test case. These test cases can then be executed to validate the software performs as expected.

In addition to these functional tests that are written by developers, test engineers will also write test cases that can be executed in an automated fashion. One advantage of automated testing is that the pass/fail results can be easily charted to show where you are in terms of the health of the software development. Both build failures and test failures can result in alerts to interested parties, as well as be reported on a dashboard that shows green/red health status, giving project managers or business leaders an instant read-out of the state of the software.

Once the software has been built, checking for syntax errors that may cause problems with the user experience later, and then tested, it can be deployed. This may require pulling all the necessary files into a package that can be installed. It could also be as simple as just copying files into place. In many cases, the application may be managed not just as application files but also systems. A deployment plan may include instructions on how to build up components, which may be virtual machines or containers.

This is where a system like Kubernetes can help by managing the overall application build and deployment process. You may have multiple containers that include different aspects of the overall application. Kubernetes can take the build instructions and create containers that house all the application components.

All of this automation is how a company like Etsy does 50 deployments per day. This speed of development and deployment has some obvious advantages—bugs don't live in production very long because rapid development and deployment replaces them very quickly with corrected code. You can also introduce new features

quickly without having to wait for longer development cycles to create a complete package. A new feature can be developed and deployed while other features are being developed alongside. You need solid orchestration throughout your tool set to be successful in a DevOps world, but you reap the benefits by always being able to respond to user demands and market requirements.

Cultural Change

As I've said, DevOps is a culture. It requires everyone on the project team to buy into how the application is being developed. Two important elements of the DevOps culture are testing at all levels and automation. Automation requires that processes be written down. Once you've written a process down so it can be automated, typically in a script of some sort, that process or script can be tested. This requires someone to have at least sketched out the process to the point where some code can be written. This can be difficult for some teams who prefer to operate in an ad hoc fashion. Ad hoc is the enemy of quality, since you're never sure what is going to be done or how it's going to be done. Just getting people on board with having to document processes so they can be automated can be challenging. However, once a process has been automated, that automation can be tested. Does it do what it's supposed to do? At the same time, using automation, you get both speed and quality. The quality comes from the consistency and repeatability of the tests, while the speed comes from not requiring a human to do the task.

Some developers are used to monolithic builds of software. DevOps tends to focus on smaller components that work together, typically using Agile development processes. Again, containerization supports this. One great aspect of approaching development and deployment from this perspective is that changes can be incremental rather than universal. You may just need to redeploy one container to make a change to the overall application rather than having to build and redeploy the entire system, as would be the case with a monolithic approach to software development.

One other aspect of a DevOps approach is the speed. Traditional development practices are much slower. They also don't include the level of communication expected or required from DevOps. The speed of a DevOps environment means coordination is critical. Changes require communication, which means silos need to be broken down. Developers need to talk to operations team members.

Additionally, developers need to talk to testers. Testers need to provide feedback to developers to ensure necessary changes are made to increase the overall quality of the solution. A communication tool like Slack can be helpful. It enables not only direct communication between individuals but also group conversations as well as other capabilities that can enable collaboration.

The lack of silos in DevOps can be one of the biggest challenges to overcome, since traditional software development has been more siloed. Fortunately, teams that have at least started the move toward Agile development may have started to break down some of those silos.

Critical Roles

With an increased focus on speed and quality, automation is essential. This means engineers who have scripting capabilities are essential to the software development team. Automation tasks run across almost all aspects of the development life cycle, including software builds and testing, as mentioned earlier. On top of that, though, there is an increased focus on automating system builds. Each application component may require its own space to operate in, which may be a virtual machine or a virtualized application, housed in a container. In either case, the application will have some requirements that need to be in place to operate correctly. You may start from a base virtual machine with just the operating system or you could have an empty container. In either case, you need some help getting your individual components built up correctly.

Again, this is where selecting tools can help out. However, you also need to make sure you are matching your tools to your people. Kubernetes can help with a lot of your system management automation across the lifetime of the application, but you need to make sure you have people on staff who can correctly manage Kubernetes. Just having a tool in place doesn't help if you don't have the right people to manage it.

Another important element of DevOps is testing because quality is such a focus of the DevOps philosophy and you can get to higher quality with a solid testing strategy. Testing should start at the very beginning of the development process with the developers, which is another of those cultural things since developers are not often trained to write tests or test cases. Skilled test engineers will be

essential. They will either need to be able to automate their own tests or work closely with automation engineers to do the automation.

This is not to say that all testing has to be automated, but speed of deployment relies on that automation. Any testing that has to be done manually may happen post-deployment. This has long been a common approach in software development, especially when it comes to security testing. The advantage to the DevOps model is that issues that have been identified during post-deployment testing can be resolved quickly.

Cattle and Pets

Not all systems or containers are created equal. This is the idea behind the cattle versus pets analogy that you will often hear in DevOps circles. People who talk about this have likely never spoken with farmers or ranchers, but the fundamental principle is that you treat pets differently than you treat cattle or livestock. You expect to have a pet for a long time and so you take good care of that pet. Cattle are considered expendable and you may not care as much about cattle as you do your cat or dog.

Applied to an application deployment model, pets are the essential elements of the application that need to always be available and can't be easily stood up or torn down. You have standing infrastructure that is carefully managed. The idea with a pet, or a standing system, is that you would name the pet, as you would perhaps provide a meaningful name to a permanent system. You don't just tear down an infrastructure system, just as you wouldn't discard a pet to replace it whenever something changed.

Cattle on the other hand don't get named (again, the people who came up with this analogy may have never met farmers or ranchers) in any meaningful way. Cattle are also expendable. If you lose one head of cattle, you replace it with another head of cattle. Applying this to the application deployment space, you may have a lot of virtualized components that are easy to stand up and tear down as needed.

Again, tools are helpful here to be able to manage life cycles in an appropriate manner. You need to be able to distinguish your pets from your cattle in your application setup. Some components of your application may need to be set up as permanent. Others are not permanent and are built up on an as-needed basis. Some DevOps

proponents may argue that your entire application should be developed as though you are working with cattle. Keep in mind that when you are working in an outsourced model like you are when using cloud provider infrastructure, you are paying for every minute a system is up and using resources. If you can develop your application so components are instantiated only when they are needed, you can save yourself some money.

This does require that you have something in place that can quickly instantiate components. It also requires that the components be designed for quick deployment. This means making design decisions upfront so that each element is as streamlined as possible. Remember that any time spent bringing systems online to respond to a user request is time the user is waiting for that request to be fulfilled.

DevSecOps and Shifting Left

Take everything you have learned about DevOps now and add in security. Sounds easy, right? Well, security is a culture too. As much as anything else, security is a culture. Technology does not solve security problems. People solve security problems. If it were all about technology, firewalls everywhere would keep constant data breaches from happening. However, that's not the case. This means that to get to DevSecOps, you need to not just add security people, you need to imbue your entire team with security awareness and practices.

DevSecOps has to start at the very beginning of the software development life cycle, shown in Figure 3-2. Traditionally, security wasn't even an element of software development. Decades ago, systems were protected by the fact that only a limited number of people would ever touch the software. Then, suddenly, everything was connected and systems and applications were reachable by others who didn't have the best of intentions. Protecting people and their personal data, as well as information belonging to a business like intellectual property, is very slowly becoming more of a priority, which means software development has to change to reduce the ability to misuse software.

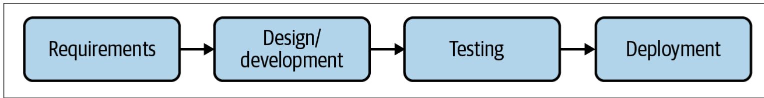

Figure 3-2. Software development life cycle

Shifting left means introducing security as far to the left in the software development process as possible. Look at Figure 3-2 and imagine how you might introduce security into each phase of the software development process, keeping in mind these are the essential elements of software development projects and not tied to any specific methodology. Following are some ways that you could consider introducing security into the development process.

Requirements
: The best place to introduce security is in the requirements phase. There are significant cost increases to fixing problems in later phases. There may be a number of ways of addressing security in the requirements phase but a simple one is threat modeling, and then ensuring that the threats identified have mitigations in place.

Design/development
: The same threats identified in the requirements should be kept in mind during the design and development phase. Designing how the product is going to operate should not introduce new threats, while also limiting the impact of existing threats that couldn't be removed. Additionally, developers should always be following secure programming practices that are specific to the language they are using. This may include using techniques like style guides to ensure all developers are writing code in a similar way, which may reduce vulnerabilities while also making the source code easier to read and fix later on. Additionally, in cases where developers are expected to write test cases, they should be trained to write misuse cases and test against those so their software is more resistant to bad or malformed data being passed through functions they write.

Testing
: All requirements should be addressed during testing, but specifically any threat mitigation identified in the requirements phase should be tested to ensure the mitigation is in place and works. This is definitely a case where misuse and boundaries should be

tested. Data should be introduced that would violate the specifications to ensure any application failure is safe, meaning it doesn't present an attacker with the opportunity to manipulate the program space or introduce and run code.

Deployment

The deployment phase may be one of the most critical, especially in the case of web applications or any application that is hosted in a network environment. All systems and containers that are deployed should be hardened, meaning unnecessary software or services should be removed and all access requirements reviewed and tightly controlled. The goal of deployment should always be to limit the attack surface, meaning the ability of an attacker to see any service from either the internet or within the network space controlled by the deployment team.

While we talk about shifting left, the reality is that in pushing backwards through the development life cycle, security should be introduced into each of those phases. Ideally, you'd start with the requirements phase but in cases where you have well-entrenched development processes and teams, dropping security into the early phases may be challenging. Again, you are making large cultural changes and people will likely resist them. It may be easier to slowly push your way backward from the far right, ensuring that at least security testing is done, then slowly introducing secure programming practices to the developers, and then trying to introduce threat modeling during the requirements phase. It's worth noting that security will typically require specialized staff, and you may not be able to find a single person who understands security from the perspective of all phases, so you'll be adding additional bodies to your development team.

Security professionals may balk at a DevOps/DevSecOps model. They sometimes feel that the speed of development and deployment will sacrifice security. The reality is the speed ends up helping overall security of the application since security bugs can be fixed and updated software deployed much faster. Additionally, moving toward a deployment model where everything is virtualized can also help security. As mentioned previously, in cases where systems are up permanently, attackers can take up permanent residence in them. If you have systems that are more fluid, such as a container model where applications are virtualized and there is very little else exposed to an attacker, even if an attacker gets access to an

application, when that container is shut down—because it's not needed or an update has been deployed so the existing container is removed and a new instance spun up in its place—the attacker has to start all over in compromising the application.

DevSecOps with Kubernetes

As much as anything else, the move to DevOps/DevSecOps is about getting humans out of the middle of processes that are simply better handled by software. Once you get a human out of the equation, you get rid of errors caused by mistakes, typos, no process, jumping ahead of instructions, or any number of other problems that humans are prone to. This is not to say that all errors are caused by humans and that automation removes all errors. After all, if there are errors in the instructions provided to the automation system, you will just keep having the software make the same mistake over and over and over.

Along with automation comes consistency, of course. You develop a plan you want your orchestration system to take care of for you, you test it to make sure it is doing what you expect, and then you let your orchestration system keep taking care of it over and over and over—exactly the same way, every time, until you change what you want done.

Fortunately, this is where Kubernetes excels. There are lots of ways to implement automation including a lot of Infrastructure as Code (IaC) software such as Ansible, Puppet, Chef, and Terraform. Once you start moving to a container-based model, however, the additional complexity presented by those software platforms may not be necessary. IaC is about building up systems and deploying software onto systems where the IaC software is managing the platform. If you are using containers, you can skip system builds because you aren't using systems: you are using containers, which don't necessarily require IaC systems. Rather, the container management system manages the definition of the container, including the application that will reside in the container as well as all the dependencies needed for that application to run successfully.

Reducing complexity is usually a good thing, especially when it comes to security. The fewer software components running, especially from a management perspective, the smaller the attack surface that can be compromised. Additionally, it means there are fewer

things that can go wrong, which means there is less potential for system or application failures. A failure often means the service is unavailable to a user. While you can still use all the IaC software you like if that's the preference of your DevOps engineers, it is not strictly necessary for a Kubernetes implementation if Kubernetes is used extensively for the application development and deployment.

Another job of operations staff is often monitoring the application for performance, load, and health. Kubernetes takes care of this, making life easier for operations staff. Kubernetes will keep an eye on all the containers under its care to ensure they are behaving as expected, replacing containers as needed or adding additional containers to support the overall health of the application.

With Kubernetes taking care of a lot of concerns that operations staff typically have, the entire development team can focus on what's important—making sure the application supports user needs—rather than spending a lot of time addressing the needs of the operations staff to efficiently support the application. As with so many other things, letting another system take care of the low-level tasks in order to focus on what's most important is a good idea.

Summary

Introducing DevOps to an existing development shop can require a significant change in culture. It can take a lot of work to ensure all team members are on board with the differences since the focus of development shifts to speed and quality. The speed change alone can be jarring to some development teams. Getting to speed and quality requires a lot of automation. This is also a significant change for some development teams, who may rely on either ad hoc or strictly manual processes. Moving to automation requires formalization of processes since you can't automate an ad hoc process. But not all developers or engineers like to sit down and clearly define what they do.

Similarly, security is a culture shift. Security is often seen as someone else's problem. The best approach to security is to ensure applications are developed in a way that makes them resistant to attack from outside. This requires introducing security in the requirements phase, where security becomes part of the job of everyone in every phase through the remainder of the software development life cycle.

Tools are essential to automation. You need to make sure you are selecting tool sets that will support development from the source code repository, through the build process, testing, and deployment. You should also consider how you are deploying your applications. Are you using pets—machines, virtual or physical, that you keep and maintain on a persistent basis? Or are you using cattle, where nothing is sacred and everything can be virtualized, meaning you can remove it as needed? Selecting a management and orchestration platform such as Kubernetes can help the process of moving from pets to cattle and guaranteeing consistency across your deployment.

Keeping security in mind throughout the life cycle can be challenging but also beneficial. Security professionals should keep an open mind to the possibilities of a DevOps/DevSecOps model. There are a lot of advantages in being resilient to attack with regular and consistent updates to an application as well as automation of testing and deployment. Finally, any deployment model where no component has much of a lifespan, because everything has been containerized and is constantly being instantiated afresh, will make an attacker's life difficult.

Kubernetes again makes a great complement to a DevOps/DevSecOps practice because it allows the automation that is essential for operations staff as well as helping to ensure the latest updates to software and dependencies are implemented, which is good for the overall security and health of the application. Using a tool like Kubernetes can help to shift the focus from operating the low-level aspects of the application to focusing on the user-oriented aspects of the application to improve the overall user experience.

CHAPTER 4

Security and Requirements

The best place to start introducing security into the systems development process is in the requirements gathering stage. While we've been referring to software development so far, it's really systems development because when it comes to web applications or even backends to mobile applications, we aren't talking about a single software package any longer. We are talking about multiple components that are installed either on virtual machines or in virtual containers. This effectively makes it systems development, even if the purpose of the full system is to deploy and provide access to applications.

When approaching systems development security, it's really easy to panic and be afraid of everything. The best approach is not to try to address every problem that may potentially arise, particularly if it's very unlikely for that situation to happen. The best approach is to follow good practices in hardening deployments and secure programming, but also to think rationally about threats that may remain. Even following the best hardening and secure programming practices will leave an exposure to attack simply because there will always be ways for an attacker to get in. The moment there is a program running, that program can be misused. For this reason, some technology providers, such as Microsoft, espouse the principle of "assume breach," where you're operating under a tacit assumption that there has already been a breach, and your job is to find it and stop it from spreading.

One way to improve the overall quality and security posture of any systems development project is to start with a threat modeling exercise. The purpose of threat modeling is to identify areas that may be misused by an attacker. Once these areas have been identified, you can develop requirements to either remove the potential threat or you can focus on mitigating the threat, meaning you are attempting to minimize the potential impact that could result from the threat being actualized.

This chapter will go deeper into how you can extract security requirements, primarily by looking at threats. Once you have identified threats, you can translate the mitigation of those threats into requirements.

Risk and Threat and Vulnerability

Software development organizations may regularly talk about risk, but often when you hear risk in the context of a software project or a systems project, the risk has to do with the timing of a release, meaning are we going to hit the promised release date or not. In one sense, this misuses the term. The concept of risk is often poorly understood across the information technology community, which leads to the word being used incorrectly.

Let's start with a clear definition of risk. Risk is the exposure to loss, based on the likelihood of an event occurring. In other words, when you say there is a risk of rain today, you are really only talking about half of a risk calculation. You are talking about a high likelihood, perhaps. What you are missing is the loss. What loss are you going to incur if it rains today? If you are hosting an outdoor event where people have paid a lot of money for tickets and you have spent a lot of money on preparation, rain may require you to refund the tickets, while still being out the costs of preparation. There is a loss there. When you factor in the high likelihood of rain, meaning a high likelihood of that loss occurring, you have a high risk. You'd need to know how much money you were going to be out (assuming there was no chance of rescheduling the event) to determine the actual risk.

There are two types of risk assessment. The first, and preferable, type (though it's harder to achieve) is quantitative. This means you assign numeric values to the likelihood (probability) and the loss (typically expressed in a monetary unit like dollars), and you'll get a

number out. Some people find this very hard because they either haven't looked for or can't determine the likelihood value and may find it difficult to assign an actual monetary value to the loss. This quantitative approach is commonly expressed as multiplying the annual rate of occurrence, which is an expression of probability meaning the number of times a year an event can be expected to occur, by the single loss expectancy (a monetary value indicating the amount that would be lost if the event occurred). You end up with an annualized loss expectancy. This is a numeric value you can assign to any given event to be able to assess the risk of that event versus any other event.

Because they find it difficult to assign values to probability and loss, people often follow the second type of risk assessment, a qualitative approach. For example, they may use T-shirt sizing—small, medium, large, and extra large—to scope both the probability and the impact. You then take the T-shirt size for both of those factors and end up with a new T-shirt size—still small, medium, large, or extra large—that you assign to an event. You can then use these T-shirt sizes to compare the risk of one event to another event.

The reason for comparing the risk of one event to that of another is that resources in any situation are limited; you can't address or remediate every risk. This means you should prioritize high-risk events. It also means you need to take a rational approach to evaluating risk. Humans have a tendency to catastrophize. They think an event with a high impact (loss potential) is a high risk, even if the probability is extremely low. Think about flying in an airplane, for instance. Some people will say it's a high-risk event because they can imagine the traumatic experience of the plane coming apart in the air, an event that results in death, and perhaps a very painful and scary death. The reality is that the probability of that event happening is extremely low, but people can't get the scariness out of their heads so it "feels" like a high-risk activity.

This brings us to understanding threats. Risk assessments can be tedious activities if you are trying to evaluate the risk of every potential event, even those that are highly unlikely. As mentioned earlier, businesses have finite resources, and it is more rational to focus on events that are more likely to happen. If you think about a bell curve, such as the one in Figure 4-1, with high probability/low impact events on one tail and high impact/low probability events on the other tail, you want to focus on the events that are going to be in

the middle of the bell curve because that's where you are likely to incur losses.

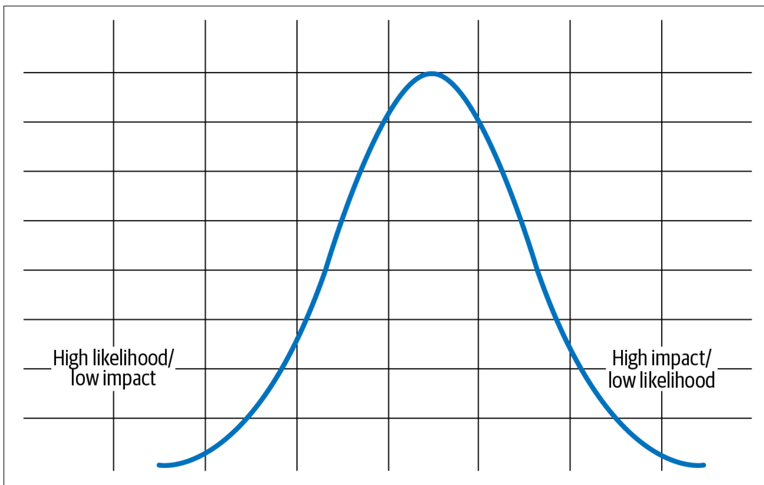

Figure 4-1. Risk bell curve

Finding those events in the middle of the bell curve comes down to identifying threats. Because we are limited by our own imaginations —focusing only on threats we may be immediately aware of since we can't easily imagine events we don't have any exposure to—threat modeling may not be an easy activity. This is where it can be helpful to have a framework in which to identify threats. We'll look at some common threat-modeling frameworks later on, but we should start by clearly defining what a threat is. A threat is a potential negative event that may result from a vulnerability being exploited.

A vulnerability, by extension, is a weakness in a system or piece of software. People can exploit vulnerabilities by triggering them. However, exploiting a vulnerability is not necessarily a malicious action. In fact, it's probably helpful not to think strictly of actions that are malicious in nature. While malicious actions are bad, you can also have serious problems that result from actions that are simply errors or even mistakes. For example, with the right set of factors in place, a mistake can easily lead to a serious and prolonged outage. This, again, is why it can be helpful to have a framework to use for identifying threats.

Threat Modeling

As mentioned previously, there are several threat-modeling frameworks that are used in systems or software development. While these frameworks are not necessarily perfect in determining threats, they can help you to focus on what is most troubling. Of course, once you have identified threats, you still need to know what to do about mitigating or removing them. It takes some practice and knowledge to be able to identify both threats and mitigations to threats. As you will see in the following discussion, some frameworks are going to be better than others depending on how your organization thinks about threats. You can also mix and match the different threat modeling approaches instead of using one exclusively.

It's important to note here that the goal of a threat model is not to eliminate threats. In any system or software application that interacts with users, especially remote users, it's not possible to eliminate threats. The goal of developing a threat model is to better understand the interactions within complex systems to find areas where you can limit either the impact or likelihood of a threat manifesting. Ideally, you reduce the overall risk resulting from these threats to a level the business is comfortable with. We're going to take a look at three commonly used threat-modeling frameworks: STRIDE, DREAD, and PASTA.

NOTE

One important element of risk that isn't included in the previous definition is the need to be informed. Businesses regularly make decisions based on risk assessments. It's not possible to make an informed decision if the risk is not clearly understood or even identified. Pretending a risk doesn't exist, incorrectly identifying likelihood or impact, or simply not assessing the risk at all means the business has not made an informed decision.

STRIDE

STRIDE, a model introduced by Microsoft in 1999, is a commonly used approach to assessing threats, especially within software development processes. It's based on a set of threat categories identified by the developers of this methodology. The following categories,

which form the STRIDE acronym, help to better identify problem areas within a complex system:

Spoofing
: Spoofing is an attempt by one entity to falsify data in order to pretend to be another entity. This may be one user pretending to be another user, or it may be one system, in the form of an IP address for instance, pretending to be another system. Spoofing can impact confidentiality or integrity in any system. One way to protect against spoofing is strong authentication and data verification.

Tampering
: Integrity is one of the three essential security properties—confidentiality, integrity, and availability. Tampering is when data is altered, meaning it has lost its integrity, since it is not in the same state when it is retrieved as it was the last time it was stored. Tampering attacks can be remediated with strong verification using techniques like machine authentication codes.

Repudiation
: Repudiation is any entity being able to say it didn't perform an action. An example is someone writing a check then later saying that they didn't write the check, even though their signature appears on the check. As signatures can be falsified, without witnesses it may be impossible to say with certainty who wrote out and signed the check. Any action that can't be clearly assigned to an entity may violate the concept of non-repudiation.

Information disclosure
: A privacy breach or inadvertent leak of data is an information disclosure violation. The use of encryption can be one way to protect against information disclosure, but it's not a perfect solution since keys can be stolen and used to decrypt information, resulting in a disclosure. Encryption without appropriate key management is not sufficient to protect against information disclosure.

Denial of service
: Anytime an application or service is unavailable to a user when the user expects it to be available is a denial of service. The same is true when a user expects to be able to get to data and that data

is unavailable. These types of attacks can't always be protected against since some of them are simply outside the control of the system developer. However, ensuring applications are resistant to crashing is a good start.

Elevation of privilege
: Attackers who manage to get control of a running process will have the level of permission or privilege assigned to the user that owns that process. Commonly, this is a low level of access, which means the attacker is often going to attempt to obtain elevated or escalated privileges so they can do more on the system they have compromised. Any ability to move from a low level of privilege to a higher level of privilege is privilege escalation, also called elevation of privilege. By always using the principle of least privilege, that is, never giving any user or process more permissions than it needs to perform essential tasks, you can help protect against privilege escalations.

Having an understanding of these categories helps to shape your threat-modeling actions, but sometimes you need some additional support. Microsoft offers a Threat Modeling Tool, which allows you to diagram your application, including defining all interactions between your components and how those interactions may be implemented. You can see an example diagram in Figure 4-2, which is a sample that comes with the Threat Modeling Tool. Once you have diagrammed and defined your solution, the tool will automatically generate a report of threats for you.

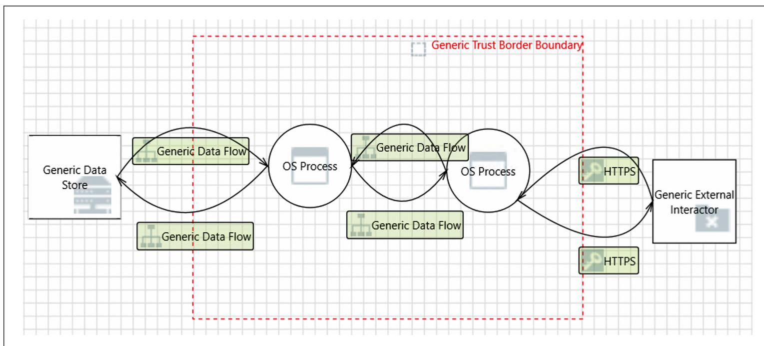

Figure 4-2. Threat Modeling Tool diagram

The tool follows the STRIDE model in developing the list of threats, so you will find threats identified with the categories discussed earlier. Figure 4-2 includes threats like Potential Data Repudiation by OS Process, Potential Process Crash, Weak Access Control for a Resource, and Spoofing the OS Process. One advantage of the Threat Modeling Tool is that it not only identifies potential threats for you, but also provides the means to manage those threats by either redesigning the system and running the tool again or by documenting mitigations that may be implemented to reduce the likelihood or impact of the threat being actualized.

DREAD

DREAD was initially proposed as a threat-modeling methodology, but in fact, it probably works better for risk assessment and can be used in conjunction with STRIDE. Once you have identified your threats, you can run each one through the DREAD model to help clearly identify risks that may result from them. One way of implementing this for quantitative assessment is to give a rating of 1 to 10 for each category. You'll end up with a numeric value to assign to each threat, which can help you better derive risk. One problem with this approach, as is the case with risk assessment in general, is that it is subjective without hard data, such as previous experience. Just as with STRIDE, DREAD is an acronym for the categories laid out in the following:

Damage
: If an event happened, how bad would the damage be?

Reproducibility
: How easy is it for this event to occur, meaning what is the level of effort or level of difficulty involved in making this event occur? Reproducibility may be much higher if there is a widely available proof of concept or exploit available, as it requires nothing but the ability of the attacker to find the exploit.

Exploitability
: Exploitability may seem the same as reproducibility but there are subtle differences between whether an attack can be reproduced and how exploitable it is. Let's say it's easy to reproduce the attack, but in each run through you get a different result. Not all of the attack attempts end up giving the attacker access to the system. Sometimes, the application under attack just ends

up shrugging off the attack. Other times, the attacker gets control of the process space. Exploitability may be low in this case while reproducibility is high.

Affected Users
: How many users are going to be impacted? You may also factor in the type of user who is impacted. Let's say customers can get access to the application but the application can't be managed by the operations team, for instance. You may want to factor in the level of the user and rank users by how important it is for them to get access.

Discoverability
: How easy is it to discover that the exploit is possible? As before, this may be a function of whether details about the vulnerability and exploit are available publicly.

As you can see, there can be a lot of subjectivity in each of these categories. Microsoft used this model for a period of time but it is no longer in use there. Some of the categories here are similar to those used by the Common Vulnerabilities Scoring System (CVSS), which uses a set of factors to generate a severity score for a known vulnerability. This can help you make decisions about whether to remediate the vulnerability quickly or whether the remediation can wait. DREAD can be used in the same way: it's another data point that can be used to determine what to do about a threat that has been identified. You can use the same DREAD model for vulnerability assessment once a vulnerability has been identified.

PASTA

PASTA is the Process for Attack Simulation and Threat Analysis, and rather than being a threat model in the way STRIDE is, it is a process that can be used to identify threats and mitigations for them. PASTA is a seven-step process:

1. Define objectives
: As always, it's better to clearly define the problem before looking for a solution. Rather than trying to tackle everything at once, this step clearly defines what is in scope for this assessment. You may choose to look only at critical assets or critical data sources, for instance. You may also define the tools and testing methods in this step.

2. Define technical scope

Complex systems have a lot of dependencies, so it may be essential to clearly define the technical scope to limit the inquiry. You don't want to be digging into libraries, for instance, that are out of your control. You may want to limit yourself to only one part of the system rather than the entire system.

3. Decomposition and analysis of application

This is where you start drilling into the way the application or system is composed. This may be similar to what was done earlier for the Microsoft Threat Modeling Tool. You define the individual components or elements and also identify trust boundaries. This is where data may move from outside the application to inside the application, for instance. This means you are moving from an untrustworthy zone (where the user lives) to a trustworthy zone (where the application controls the data and developers may assume the data has been sanitized—not a good assumption as a general rule, but an example of why you might say one zone is trustworthy).

4. Threat analysis

Based on intelligence sources, assess the known threats. For example, you may use components that have known vulnerabilities and there may be exploits for those vulnerabilities, or you may be exposed to common known vulnerabilities because of development practices used.

5. Attack/exploit enumeration and modeling

In this stage, you use an attack tree to model what an attack might look like and how it may operate. An attack tree is a way of diagramming a process with decision points or options along the way. You may make different decisions about how you handle an attack based on the path through the model. An example of an attack tree is shown in Figure 4-3.

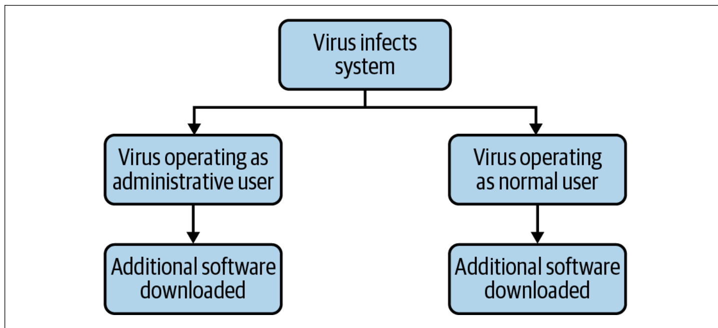

Figure 4-3. Attack tree diagram

6. *Analyze modeling and simulation*
   Once you have created your attack trees, you can start to understand the potential for damage from the attack. This is done by running simulations of the attack and determining its likelihood.

7. *Risk and impact assessment*
   Once you have identified the likelihood, you should also assess the impact based on a successful execution of the attack, and then determine the risk. Based on all the data that has been collected, you should be determining what controls you can put in place to mitigate the risk from the potential attack.

You'll see that PASTA is a very detailed approach to threat assessment and there is nothing here that would necessarily prevent you from folding in other approaches. You could use STRIDE as you are looking for attacks and exploits by looking for places where information disclosure is possible, for instance. You might also use DREAD as you are doing the risk and impact assessment to give you a broader view of the aspects of the attack that may impact the system.

Summary

You always need to have a starting point when you are developing something. You need to define the problem before you start working on the solution. If you don't, how do you know if your solution fits the problem? Without a clear definition, you have a solution in search of a problem, which is not a great way to try to sell or market anything. The same is true when it comes to addressing security for

any system or application. You need to know what it is you are protecting against, since you can't protect against everything. A good place to start is by identifying threats, which will help you better identify potential mitigations to address those threats. Once you know what threats you face, you can start to generate requirements based on the threats identified.

The problem then becomes how to identify threats. There are some methodologies that can be used, including the STRIDE methodology, which identifies six categories of threats: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. As you are assessing your system or software, you should be looking for places where your application may introduce the potential for an incident in one of these categories.

Another framework to help better understand the impact from a threat that has been identified is DREAD. Using DREAD, you look at damage, reproducibility, exploitability, affected users, and discoverability. Assessing the questions associated with these factors will help you get a better understanding of the overall risk associated with a threat because they will give you a deeper insight into the probability and loss that might result from a threat being actualized.

While STRIDE provides a set of categories, PASTA offers a process. PASTA is the Process for Attack Simulation and Threat Analysis. It is a highly structured approach to identifying threats and their impact and likelihood. To implement PASTA, you need to be able to deconstruct the application, identifying trust zones and how data passes through the different components. Once you have run through the PASTA process, you can still identify categories using STRIDE and help understand the risk using DREAD.

CHAPTER 5
Managing Threats

One of the problems that system or application developers contend with is complexity. Any application that does anything of significance is going to be complex because of the sheer number of potential pathways through the application. Once you start moving to a web application deployment model, you may have multiple systems in place, which adds additional complexity. You now have systems with operating systems and applications and services. Even if you are using a cloud native design, you have to deal with virtualized services that may be coming and going. Versioning is a concern because different container instances may have different versions of an application running in them.

All of this is to say that software has to face a lot of potential problems. It all starts with threats—any potential problem or failure that may beset an application, piece of software, or system. This may simply be a result of all the complexity. There is also a possibility of bad data being introduced to a program, causing a failure. That bad data may be a result of a malicious action.

Just having threat information is insufficient. You need to know what you are going to do about these threats. There will always be threats that you won't be able to do anything about, but it's best to clearly understand the threat so you can make an informed decision about possible mitigations. Of course, in the case of systems or application development, all of these mitigations should be fed into the requirements process.

This chapter will discuss ways of gathering more information about threats as well as taxonomies you can use to better understand their impact. Finally, we'll talk about using a threat matrix to map threats to mitigations and priorities.

Open Source Threat Intelligence

Threat intelligence is available from a number of sources. The easiest way to get up-to-date threat intelligence is by purchasing a commercial service. That's not the only way, though. There are places to get threat intelligence for free. One of these is by signing up with an Information Sharing and Analysis Center (ISAC). ISACs are generally organized around industries. For example, there is a financial services ISAC (FS-ISAC) as well as an information technology ISAC (IT-ISAC). The purpose of these organizations is to provide a safe place where people in the same industry can talk about issues.

Threat intelligence can take many forms. Typically, what security professionals are looking for will be indicators of compromise (IoCs). These are pieces of data that will identify when an attacker is trying to gain access to an account or network. This may be something like an internet address, a domain name, or a cryptographic hash that could identify the location of an attacker or a piece of known malware. In the case of software or systems development, those are less likely to be useful since the objective isn't to find an attacker in a network or system. Instead, you want to know how attackers behave. This will help identify how they may try to misuse a piece of software or a system. What we are looking for are tactics, techniques, and procedures (TTPs).

Fortunately, there are a lot of sources of TTPs, which are the actions attackers take. One of the challenges with TTPs is there can be a lot of them. Having a place to store them in order to search through them easily is helpful. In a security operations organization, this would be a security information and event management (SIEM) system. There are several pieces of software that can do the log collection and analysis for you. One of them is the Open Source Threat Intelligence and Sharing Platform, formerly known as the Malware Information Sharing Program, and still called MISP.

MISP will consume threat intelligence feeds in different formats. It then organizes all the information so it can be searched. Figure 5-1 shows the MISP interface. The overlay you see in the figure is a

quick look at the data available about the Babuk ransomware (ransomware is misspelled in the data feed).

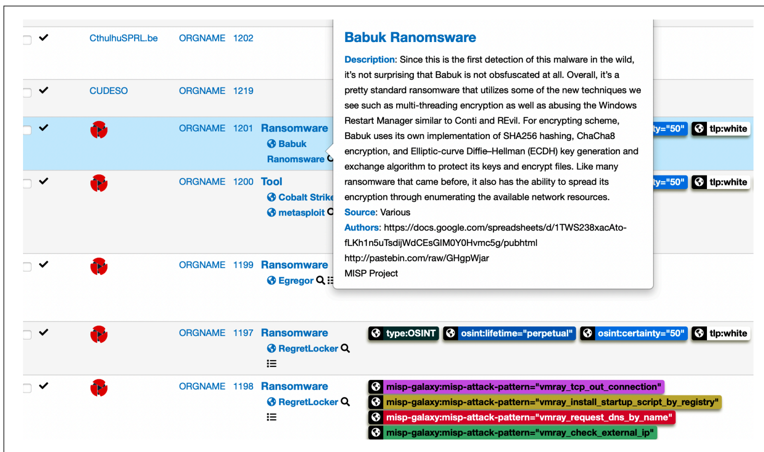

Figure 5-1. MISP output

Once you have added some feeds to MISP, you can start browsing through updated details about how attackers operate. Some of this information won't be useful. You may not care, as a systems or software developer, about ransomware operators, for instance. You'll find a lot of different categories available, including tools and threat actors. You will also find entries for attack patterns. For example, Figure 5-2 shows details on attack patterns for the threat group APT28, sometimes called Fancy Bear. This threat actor uses techniques such as a run key in the registry to achieve persistence, which allows the attacker to keep malicious software running across reboots of the system.

It will take a lot of digging, but you can get a lot of detail about how attackers operate by reading through threat intelligence feeds. There are some benefits to using threat intelligence feeds even beyond what you might use for threat modeling exercises. For a start, these feeds will show you the importance of introducing security into your products as early as possible in the development life cycle simply because of the ways attackers can misuse any potential vulnerability in software. Even if your software isn't remote in any way, meaning there are no network listeners, there is the potential for attackers to get additional permissions by using your software.

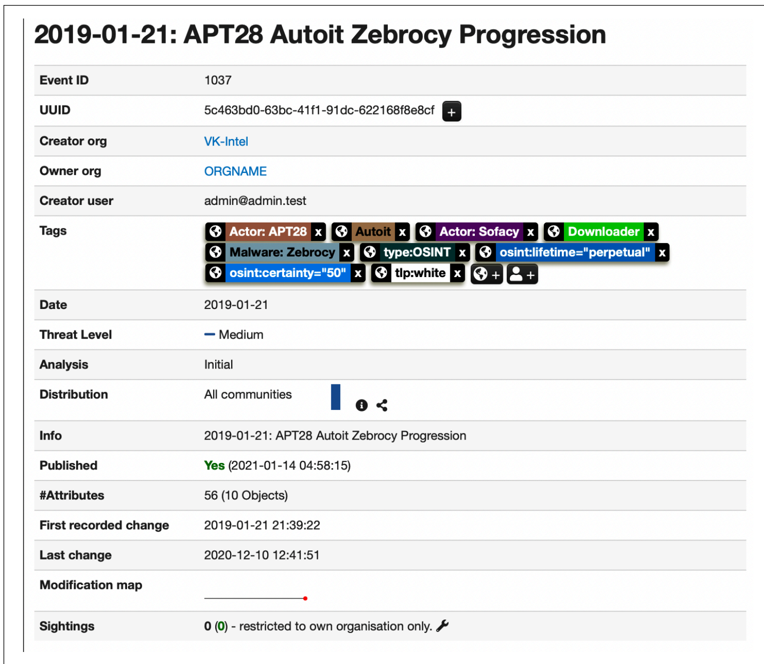

Figure 5-2. APT28 details

Attack Phases

Before we get into determining threats to any system or software, you need to be able to categorize how those attacks may work. There are two good ways of thinking about this. The first, which is the more comprehensive, is the MITRE ATT&CK Framework. The second, used by the security consulting company Mandiant, widely known for its incident response expertise and based on decades of observing how attackers operate, is the attack life cycle.

MITRE ATT&CK Framework

The MITRE Corporation has a history of developing information categorization strategies that benefit the industry at large. In recent years, MITRE has looked to develop a taxonomy of attacks, which is broken down into categories, or phases, of attacker activity. However, this is not simply a list of phases that attackers will go through. The ATT&CK Framework is also a database of known TTPs. The following are the phases of the framework, including some of the TTPs used by attackers:

Reconnaissance

The first thing an attacker is going to do is scout for targets, and then look for information about their target. This may include gathering information about systems, people, or financial data. This may come through using open source information that can happen without the victim ever knowing, or it could come from scanning victim networks to gather details, which is likely to be noisier and more prone to detection.

Resource development

In this phase, the attacker is preparing to attack. They may be gathering account information from available sources. This could include preparing for credential stuffing exercises by gathering known usernames as well as a large collection of passwords that are known to be used commonly. This may also include acquiring systems, either legally or illegally, that will be used to attack from.

Initial access

The initial access phase is where the attacker starts compromising systems. This may come through phishing messages or by introducing malware in websites that are known to be used. It may also come through the use of removable media like USB sticks left around or even delivered to users. In cases of targeted attacks, the social engineering messages sent to users are called spear phishing. This is because these users are identified specifically, rather than the attacker sending out as many messages as possible to get as many people as they can regardless of what company a user works at.

Persistence

Gaining initial access is not enough because at some point the user will log out or the system will be rebooted. The attacker will want to be able to persist their remote access means across reboots and login/logout cycles. This may be done through registry keys on Windows systems or scheduled tasks on any operating system. It could also include malicious software that executes before the operating system boots.

Privilege escalation

If a security organization is doing its job well, the permissions of normal users at a company will be restricted, and an attacker who compromises a user account won't be able to do much.

Instead, the attacker needs to get elevated or escalated permissions to be able to do things like extract passwords from memory. This may involve taking advantage of software vulnerabilities to get additional permissions.

Defense evasion
Most companies will have some sort of protection capability in place, even if it's basic anti-malware software. Attackers have to perform some trickery to get past this detection software. They may obfuscate what they are doing by encrypting data or encoding it in some way. They may also use techniques like alternate data streams in Windows to hide data where some detection software won't look for it.

Credential access
Attackers are going to want to move around within a network, from system to system, to search for information or more access. They may do this by extracting additional usernames and passwords from disks and memory on systems they have already compromised or by performing network attacks against repositories of this information.

Discovery
In the process of looking for information they may want, whether it's intellectual property or personal information that can be monetized, attackers will be looking for additional systems and services that may contain that information. This information may come from searching through the history on a system, including hostnames that are visited by a user.

Lateral movement
Lateral movement is the process of hopping from one system to another within an environment. This may be accomplished by further phishing from a compromised user to an uncompromised user or from session hijacking, where an existing connection to a service is hijacked by the attacker to gain access to the service for themselves.

Collection
This is the process of data collection. This could be data that can be used to further compromise the system, or it may be data that could be useful to the attacker—intellectual property, credit card information, personal information, etc.

Command and control

The attacker needs to be able to manage the compromised system remotely. After all, the attacker isn't sitting on the physical network, and it may not be possible to connect individually to a compromised system. It may be easier for the attacker to configure the compromised system to connect out to a remote system, which will issue commands to the victim system. These management systems are usually referred to as command and control (C2) systems.

Exfiltration

The attacker will need to be able to retrieve the data that they have collected. There are a lot of ways this may happen, depending on what protections are in place. For example, attackers may push data out of a network by embedding it into well-known protocols, since those are the ones that are most likely to be allowed out through firewalls and other forms of protection.

Impact

Attackers aren't only looking for information to steal. Sometimes they also want to manipulate or destroy data—or even wipe the system. This may have been the goal all along, or it may just be a result of getting caught in the act and then attempting to destroy as much as possible on the way out, perhaps in part to obscure their actions.

As before, there is a lot here and not all of it is going to be useful to your situation. However, this framework will add ideas to your arsenal as you start thinking about all the ways your system or software may be attacked.

Attack Life Cycle

Just creating a list of the different TTPs attackers may use is not sufficient. You need to know what phase of an attack the attacker is in. This will influence not only the sense of urgency with which you should act, but knowing the different phases of an attack also will help you map out what action you should take. An attack life cycle is not a structured methodology in the way MITRE's is. Instead, it's a way of thinking about how attackers operate in the real world. The attack life cycle shown in Figure 5-3 is commonly used by the incident response company Mandiant in its approach to investigations. This diagram shows all of the stages an attacker goes through from

reconnaissance to completing the mission. This is based on years of observations of how attackers work. If you go back to well-known infiltrations from the 1970s and 1980s even, including those by Kevin Mitnick and the Chaos Computer Club, you can easily map them into this attack life cycle.

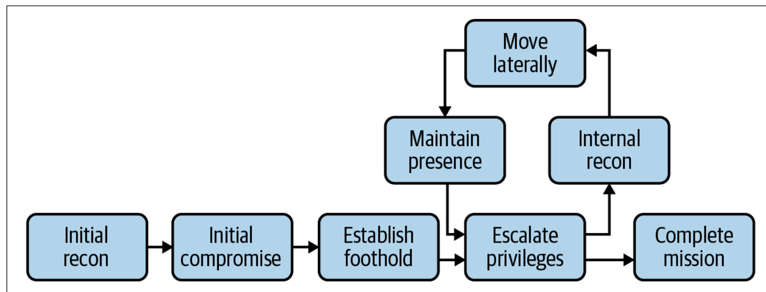

Figure 5-3. Attack life cycle

The difference between this and the MITRE ATT&CK Framework is that the attack life cycle is more targeted. The ATT&CK Framework is a taxonomy, while the attack life cycle is meant to describe how attackers operate in the real world. One advantage to the attack life cycle is that it clearly illustrates the cycle or loop that happens once attackers are in the environment. This cycle is suggested in the ATT&CK Framework, but since that framework is focused on cataloging TTPs, it's not as clear what actually happens. The attack life cycle allows you to visualize it.

Using an Attack Matrix

Now that we know how attackers operate and what they do, what can we do? Once you have been through a threat modeling exercise to identify all the potential threats to your software system, you can start mapping those into the phases of either the ATT&CK Framework or the attack life cycle, depending on which makes more sense to you. The threat modeling exercise, whether it's using STRIDE or another threat modeling methodology, is going to create a list of all of the possible threats to your system or application—or at least a list of all the threats you are able to identify. Remember that threat identification requires some skill, some experience, and some imagination. Each of those threats should be categorized into one of the attacker phases so you can better visualize how the attacks based on those threats will happen and when they would be used.

We can use an example attack matrix developed by Microsoft to describe the potential threats to a Kubernetes installation. A portion of the attack matrix is shown in Figure 5-4. There are other threats that were identified but as this is used for the purposes of explanation, there is no reason to get exhaustive about these threats. We will be able to use the threats here to demonstrate how you would use the threat matrix as part of a DevSecOps environment. The company StackRox (now part of Red Hat), which specializes in Kubernetes security, has provided some guidance on how to better protect your Kubernetes implementation based on this threat matrix. StackRox took the analysis Microsoft did and provided some detail around how you can mitigate the threats that were identified. However, we are going to take a different look at how to use the threat matrix and derive requirements from it.

Initial access	Execution	Persistence	Privilege escalation	Defense evasion	Credential access	Discovery	Lateral movement	Impact
App Vuln	Exploit	Backdoor	Privileged container	Clear container logs	List secrets	Access K8S API	Access cloud resources	Data destruction
Credentials	Shell access	CronJob	hostPath Mount	Delete K8S events	Mount service principle	Access Kubelet API	Container service account	Resource hijacking
Misconfig	Exposed SSH	Writable hostPath	Cluster-admin binding	Pod/container name similarity	Access container service amount	Network mapping	Access dashboard	Denial of service

Figure 5-4. Kubernetes attack matrix

Let's take one of these threats as an example and talk about how you can generate some requirements based on it. An easy one is an exposed dashboard. Keep in mind that not all threats are going to be introduced by an application under development; they can encompass any application, platform, or management software you may be using to support the application or system. In the abbreviated matrix shown in Figure 5-4, an exposed dashboard would fall under misconfiguration. The requirement for the exposed dashboard threat is the Kubernetes dashboard will not be available/exposed to any unknown system or network. This means we need to make sure the Kubernetes dashboard, which is being used to monitor and manage all of the virtualization within the application, is only accessible from known network devices. Additionally, the Kubernetes dashboard should only be accessible to authorized users who have

been authenticated using multifactor authentication and have appropriate roles assigned.

You may notice this set of requirements does not define the implementation. The requirements then cascade through the rest of the systems development life cycle. From a design perspective, the Kubernetes dashboard may be placed in the virtual network in a location that isn't accessible. You may implement network security groups that limit access to known Internet Protocol (IP) address blocks. Additionally, you would need to design the appropriate identity and access management that would allow you to force multifactor authentication on users. You will also need appropriate procedures in place to ensure that provisioned users should have access to the dashboard.

Beyond the design phase, we need to introduce requirements to the testing phase as well. The testing group should be fed this requirement. They should test to ensure that in the completed application, the dashboard is not exposed and cannot be accessed from unknown network segments. They should also test to ensure multifactor authentication is required for all users. There isn't much they can do in the way of testing to ensure that only appropriate users get access. That's up to someone else later in the process.

The deployment phase also has to be factored in. The deployment team needs to ensure the application is deployed based on specifications. In this case, since we are using Kubernetes, there will be automation in place to manage the deployment. Once you start automating, you can validate the script or configuration that dictates the deployment and test the script to ensure it behaves as expected and the results match the requirements.

The National Institute of Standards and Technology (NIST) provides a lot of guidance for federal agencies regarding information security. NIST's Cybersecurity Framework (CSF) breaks out security operations into the following phases: identify, protect, detect, respond, and recover. These are useful to consider when developing applications and deploying systems.

So far, we've been focused on protection, but there's a lot more to security than just expecting that you will be able to prevent all attacks from happening. In many cases, you may find that attack traffic will look like legitimate usage. This is the case where the Kubernetes dashboard is exposed. An attack leveraging a public

exposure of the Kubernetes dashboard will use the same web-based requests that authorized users are making. The difference is they will be coming from networks that don't belong to the organization that owns the application. This is something we can monitor post-deployment.

One problem with deployment is that it can drift. People make changes to running systems that alter configurations and behaviors. Once you get through deployment, you need to manage and maintain the application. This should always require monitoring. You can monitor access to the Kubernetes dashboard, as in the example threat under consideration, and throw up an alert if there is successful access from an unexpected address. Similarly, you can monitor for any successful access that doesn't use multifactor authentication.

While this is a single example using a very simple threat that's easy to manage, you can see how the process would work. It can be time consuming, but by using this process, you can get better at generating requirements that not only cover the extent of the software development life cycle, but also can be used to help detect attacks against the system or application, which will make the application more resilient to attack.

Summary

Attackers continue to be prolific, which helps generate a lot of information about how they operate. This can be extremely useful if you are following a threat-based approach to application security. One problem with using threat modeling is that it requires either a lot of experience or a lot of information about how attackers operate. Identifying threats is not an easy task. They don't just fall out of a book or the existing requirements. You need to know a lot of TTPs so you can inform your idea of what a threat is.

Fortunately, there are open source threat intelligence tools that can help you expand your imagination and develop a list of threats. Once you have a list of threats, you need to categorize them using a taxonomy like the MITRE ATT&CK Framework or the attack life cycle to help you prioritize them. There are problems that are within your control and problems outside of your control. You may find that identified threats in the exfiltration phase may not fall under your control and may not be anything you need to worry about.

Once you have categorized your list of threats, you can start to generate requirements. You should think about requirements across all phases of the software development life cycle. Threats may be mitigated in the design/development phases but those mitigations should be tested to ensure they work. Additionally, some threats may be mitigated in the deployment phase. Threats should always be followed all the way through from design to deployment.

You should also consider requirements that may be necessary beyond deployment. After all, not every attack is going to look abnormal. In fact, you may consider the idea that most attacks are probably going to look normal since it's common to attack applications using the protocols and data they speak and understand. This means you should consider the phases of the NIST CSF: identify, protect, detect, respond, and recover. You aren't going to be able to protect everything. You need to be able to detect when bad things happen that couldn't be protected against.

Knowing TTPs across all phases of an attack life cycle can help you better identify threats to your system or application. This, in turn, can help you better define requirements that are more comprehensive.

CHAPTER 6

Wrapping Up

Moving to DevOps or DevSecOps can be a considerable cultural shift. Moving to the cloud can be an enormous change as well, especially if you are trying to adopt cloud native approaches rather than just a lift and shift, where you outsource your existing infrastructure and systems to a cloud provider. Following are some considerations as you are thinking about migrating to DevOps/DevSecOps:

1. Make sure you have the right team members in place. This is especially true if you are going to a cloud native design using virtualized applications rather than virtualized systems. The deployment is very different.
2. Select the right tools. If you are going to virtualized applications rather than systems, make sure you have a solid orchestration and management platform in place. This may be something like Kubernetes, which can manage the entire life cycle of virtualized applications.
3. Automate as much as possible. Automation is testable, which helps ensure you are using the right process. This can also help you scale your application in case of surges in requests.
4. Verify everything. Attackers will go after the source code and even the toolchains. Automation is great for most things, but you also need humans to validate that the right things are happening (e.g., the right source code has been checked in) and the right tools are in place. The SolarWinds compromise has shown

that tools are just as capable of introducing malicious source code into the build as replacing source code directly.

5. Introduce security as early as possible. Attacker activity has continued to increase in recent years. Ensuring you are thinking about security early will help to prevent a lot of pain later on and also cost significantly less money. Embed security in the developer workflows and provide developers with guardrails to shift security left.
6. Identify threats. Use a threat-modeling practice to identify threats to the overall application or system design.
7. Develop requirements based on the threats. Do not assume that the only place to mitigate threats is in the software development. System design needs to be taken into consideration as well. Deployment should also be following requirements to ensure a hardened, protected application in production.
8. Test requirements. Make sure all of the requirements that are generated end up as test cases and are verified before deployment. Because testing can be time consuming when done manually, this is another case where automation can be beneficial.
9. Consider that not all threats may be preventable. Make sure you are generating requirements that follow the NIST CSF so you are at least considering prevention and detection. Work with the security operations staff to make sure they can respond to threats that have been detected.

About the Authors

Wei Lien Dang is senior director of product and marketing for Red Hat® Advanced Cluster Security for Kubernetes. He was previously a cofounder at StackRox prior to its acquisition by Red Hat. He was also Head of Product at CoreOS and held senior product management roles for security and cloud infrastructure at Amazon Web Services, Splunk, and Bracket Computing. He was also part of the investment team at the venture capital firm Andreessen Horowitz. Wei Lien holds an MBA with high distinction from Harvard Business School and a BS in applied physics with honors from Caltech.

Ajmal Kohgadai is principal product marketing manager for Red Hat Advanced Cluster Security for Kubernetes. Prior to its acquisition by Red Hat, he was the Director of Product Marketing and Growth at StackRox, and previously held senior product marketing roles at McAfee and Skyhigh Networks (acquired by McAfee).