O'REILLY®

Practical Cloud Security

A Guide for Secure Design and Deployment

Free Chapters

Sponsored by

orca security

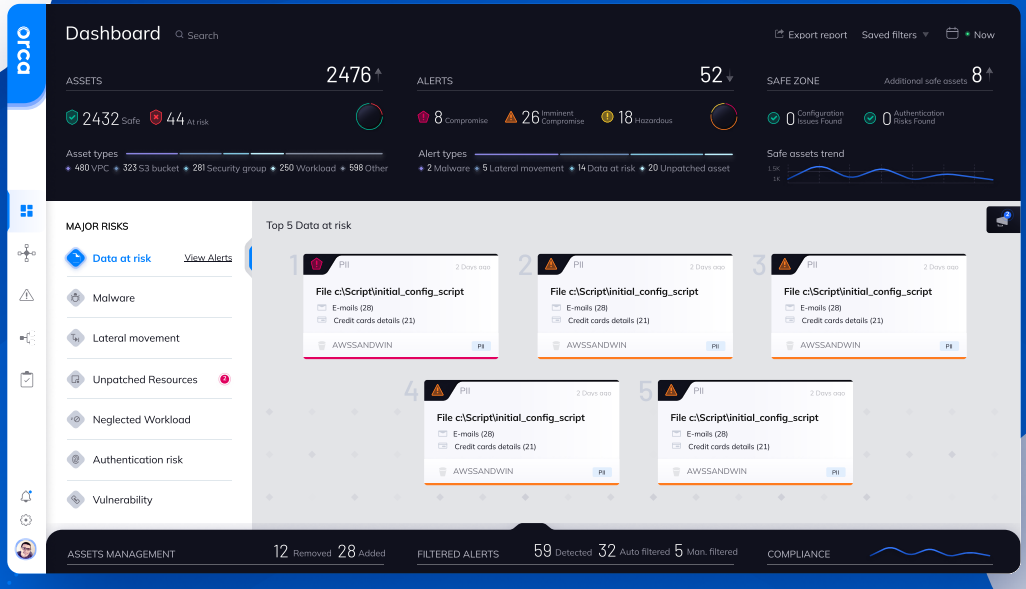
Chris Dotson

orca security

CLOUD SECURITY RISK ASSESSMENT

See for yourself how easy cloud security can be.

Start Free Trial



Practical Cloud Security

A Guide for Secure Design and Deployment

This excerpt contains Chapters 3–5. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Chris Dotson

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

Practical Cloud Security
by Chris Dotson

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: Rachel Roumeliotis
Developmental Editors: Andy Oram and Nikki McDonald
Production Editor: Nan Barber
Copyeditor: Rachel Head
Proofreader: Amanda Kersey
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2019: First Edition

Revision History for the First Edition
2019-03-01: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492037514* for release details.

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

978-1-492-03751-4

[LSI]

Table of Contents

Foreword

Whether driven by the need to support remote workers during a global pandemic or to fuel innovation, cloud adoption continues to accelerate. Analyst firm Gartner predicts that spending on public cloud services will amount to $396 billion in 2021 and reach $482 billion in 2022. They further predict that by 2026, public cloud spending will exceed 45% of all enterprise IT spending—up from less than 17% in 2021.

Seen initially as a way to reduce costs, companies are now leveraging the cloud to accelerate IT service delivery, streamline application development, speed up go-to-market activities, improve business continuity, and provide greater flexibility. However, as cloud environments expand at an unprecedented rate, new security risks arise. Cloud assets are spun up and torn down on demand, making them difficult to track and manage. Are enterprises managing to keep up with the security and visibility of their complex and ever-expanding cloud estates? The answer is: no. According to an IDC survey conducted among 200 CISOs and security decision-makers, 98% of enterprises have contended with a cloud security breach in the last 18 months. Of those, 67% of respondents reported three or more incidents, and 63% of organizations had sensitive data exposed in the cloud.

I've been involved in cybersecurity since I was 13 years old. One thing I've learned along the way is that there can be all kinds of hyped-up new technologies, but if they are not easy to implement, they won't prevent breaches. To be effective, those of us in security have to be pretty well-grounded (even as we reach for the cloud). No matter what technology we use, it still boils down to consistently following best practices. Security teams need to be able to answer the following basic questions: What assets do we have? Who owns them and who has access to them? What are the major risks and vulnerabilities in our cloud environment? Do we have full visibility into all our cloud assets, or are there blind spots? Security starts with awareness of cloud security risks. And after awareness, visibility into your cloud estate is critical. After all, what you cannot see, you cannot protect.

With that goal in mind, we have chosen to share with you three central chapters of *Practical Cloud Security*, written by Chris Dotson and published by O'Reilly Media. I believe these chapters will strengthen your understanding of cloud risks and explain the foundational concepts of securing your most critical assets and data.

Chapter 3, "Cloud Asset Management and Protection," looks at cloud assets and how to inventory and protect them. It lists the major categories of cloud assets—compute assets, storage assets, and network assets—and their functions. It also covers the various assets themselves (e.g., VMs, containers, block and file storage, VPCs, etc.), tips for inventorying each one, their primary vulnerabilities, and ways to manage access to ensure security best practices and reduce risks.

Chapter 4, "Identity and Access Management," examines arguably the most important set of security controls—lost or stolen credentials are usually the culprit in breaches involving web applications. This chapter reviews key concepts such as authentication and authorization, roles, cloud IAM identities, multi-factor authentication, password hygiene, federated identity, single sign-on, secrets management, revalidation, and more.

Finally, Chapter 5, "Vulnerability Management," discusses everything from scanners to patching to configuration management. Attackers can get unauthorized access to your systems through vulnerabilities at many different layers of the application stack. You need to spend some time understanding the different layers, what your vulnerability management responsibility is for each of those layers, and where your biggest risks are likely to be.

I believe that whether you are a CISO, security practitioner, DevOps engineer, compliance officer, or have another IT role, these chapters offer valuable and practical information for securing your cloud estate.

It is my hope and recommendation that you spend some time reading about these industry best practices and consistently apply them so that your organization can reap the full benefits of all the cloud has to offer without having to compromise on security.

— Avi Shua, CEO and cofounder of Orca Security
Los Angeles, California, November 2021

CHAPTER 3

Cloud Asset Management and Protection

At this point, you should have a good idea of what data you have, where it's stored, and how you plan to protect it at rest. Now it's time to look at other cloud assets and how to inventory and protect them.

As mentioned in Chapter 2, cloud providers maintain a list of which assets you have provisioned, because they want to be able to bill you! They also provide APIs to view this list, and sometimes they have specialized applications to help you with inventory and asset management.

In general, your cloud provider will know only about assets you provision via its portal or APIs. For example, if you provision a virtual machine and then manually create containers on it, the cloud provider will have no way of knowing about the containers.

Cloud infrastructure and services are often inexpensive and easy to provision, which can quickly lead to having a huge number of assets strewn all over the world and forgotten. Each of these forgotten assets is like a ticking time bomb, waiting to explode into a security incident.

Differences from Traditional IT

One important difference with cloud asset management and protection is that you generally don't have to worry about physical assets or protection at all for your cloud environments! You can gleefully outsource asset tags, anti-tailgating, slab-to-slab barriers, placement of data center windows, cameras, and other physical security and physical asset tracking controls.

Another important difference lies in the IT group's participation in the process of provisioning cloud assets. In a traditional IT environment, creating an asset such as a server is often difficult and time-consuming. It usually requires going to a centralized IT group, which will follow a detailed provisioning process and maintain a list of assets in a database or a spreadsheet. There is a natural barrier to creating shadow IT (IT resources that are hidden or not officially approved for use), because IT typically requires capital assets. In most organizations, large capital expenditures are carefully controlled.

One important benefit of cloud computing is replacing these large capital expenditures with monthly expenses, and offloading the capacity planning to an IaaS provider. This is great, but it also means that it's more difficult for the IT and finance areas of the business to be effective gatekeepers for IT resources. Anyone in any area of the business can easily provision a huge number of IT resources with only a credit card (and sometimes not even that). This can quickly lead to asset management problems.

Prior to the cloud, most organizations had some amount of shadow IT. In the cloud era, this problem is often far worse—and the assets aren't just servers.

Types of Cloud Assets

Before we can effectively manage cloud assets, we need to understand what they are and their security-relevant characteristics. I find that creating clearly defined categories of assets helps to organize my thinking. For this reason, I have categorized cloud assets as compute, storage, and network assets, but you could choose different categories.

More types of cloud assets are created every day, and it's likely that you will not have all of these types of assets. You also don't need to track all of these assets in a single place. The important thing is to know about all assets that are relevant to your security.

If you are coming into an environment with a large number of existing cloud assets, keep in mind that you don't have to have a 100% solution for asset management immediately. Concentrate on the assets that are the most security-relevant to get immediate value, and then add additional types of assets to your inventory incrementally. For many organizations, the most security-relevant assets will be a few types of data storage and compute assets.

As you read through the types of cloud assets, it may help to jot down notes of the types of assets that you already know about, and put stars next to the ones that are most relevant for security. Although this chapter is primarily about asset management, some of the security properties of these assets may inform the current or future

designs of your cloud environment. In the second part of this chapter, I'll share some ideas on how to inventory the cloud asset types you've identified here.

Many cloud assets are ephemeral, in that they are created and deleted fairly often. This can make asset management more difficult, and it may also make some popular methods of asset tracking, such as tracking by IP address, ineffective.

Compute Assets

Compute assets typically take data, process it, and do something with the results. For example, a very simple compute resource might take data from a database and send it to a web browser on request, or send it to a business partner, or combine it with data in another database.

These cloud asset categories are not completely distinct. Compute resources may also store data, particularly temporary data. With some types of regulated data, it may be necessary to ensure that you're tracking every place that data could be, so don't forget about temporary data storage.

Virtual machines

Virtual machines (VMs) are the most familiar cloud asset type. VMs run operating systems and processes that perform business functions. VMs in cloud environments behave very similarly to their on-premises equivalents in many cases.

Virtual Machine Attacks

VMs in the cloud differ fundamentally from on-premises VMs in one important way: in a cloud environment, you may be sharing the same *physical* system with other cloud customers. These other customers might simply be inconsiderate and cause "noisy neighbor" problems by using up all of the processor time, network bandwidth, or storage bandwidth so that your VM cannot get its work done efficiently. However, these other customers might also be deliberately malicious and attempt to exploit the fact that you're on the same physical hardware to attack the confidentiality, integrity, and availability of your system. These are additional risks to the standard "front-channel" risks for servers, such as the use of stolen credentials or the exploitation of software vulnerabilities on the server.

In general, there are two primary ways that other customers (or even attackers who have gained access to your own VMs) might attack you. The first is via a "hypervisor breakout" or "VM escape," where an attacker on one VM is able to breach the hypervisor and take full control over the physical system. Fortunately, this isn't easy, because hypervisors are designed to accept very little input from the virtual machines. In general, a VM that wants to take over the hypervisor needs to find a vulnerability

in either the paravirtualized storage or network interfaces, which is not a large attack surface. If physical systems are like separate buildings, virtual machines are like separate apartments that can contact the superintendent only via two mail slots labeled "network" and "storage." I call these "back-channel" attacks, because they attack the infrastructure behind the VM.

The other way that attackers may gain information is through "side-channel" attacks, which are based on unintended side effects of running code on a physical system. When running on the same hardware, attackers may be able to deduce important information about your VM, such as passwords or encryption keys, by carefully watching the timing of processor instructions or cache accesses. This is essentially how the famous Spectre and Meltdown vulnerabilities work.

This doesn't mean you shouldn't use VMs; the risks of these types of side-channel and back-channel attacks are acceptable to most organizations. However, it's important to know that there are some potential vulnerabilities from sharing physical hardware. The good news is that, like physical security, mitigating these types of attacks is almost always the responsibility of your cloud provider (although in a some cases you may also need to install operating system fixes on your VMs).

VMs always have an operating system, which includes a kernel as well as other "user-space" programs shipped with the kernel by the operating system vendor. Some servers can perform all of their functions using only the software shipped as part of the operating system. However, most VMs have additional software installed, such as platform/middleware software and custom application code that your organization has written.

Because so many different components can be mixed together to make up a VM, we need to be careful about vulnerability management, access management, and configuration management for each of the different layers of a server. Successful attackers may get access to any data the VM has access to, and can use that VM to attack the rest of your infrastructure or other people.

Here are some example inventory items to track for VMs:

- The operating system name and version. Operating system vendors support versions with security fixes for only a limited amount of time, so it's important to stay reasonably up to date and run a supported version of your OS.
- The names and versions of any platform or middleware software. This may be software such as web servers, database servers, or queue managers. It's important to track this software for vulnerability management purposes (in case security advisories are released for it) as well as license management.
- Any custom application code on the VM that your organization maintains.

- The IP addresses of the VM and what virtual private cloud network it's in, if applicable.
- The users allowed access to the operating system, and to the platform/middleware/application software if different.

Most of these are the same as with on-premises VMs. However, cloud VMs generally only take a minute or two to create, which means that they can be created and deleted as needed. This is great for scaling up and down quickly to meet demand, but can make asset management more difficult. For this reason, you will probably need to use agents installed on your VMs or an inventory system from your cloud provider to collect all of the relevant information automatically.

In addition to tracking the VMs themselves (often called "instances"), you also need to track the "images" or templates that are copied to create new VMs. You don't want new servers to come online with critical vulnerabilities, even if they are patched quickly after starting.

Some cloud providers provide "bare-metal" systems in addition to VMs.[1] These have the same security needs as VMs, but may also have firmware that occasionally needs to be updated.

Many cloud providers also provide "dedicated" VMs. These are created in the same way as regular VMs, except that the provider promises to not schedule any other customer's VMs on the same physical systems with yours.

Bare-metal machines and dedicated VMs are not subject to the risks described in "Virtual Machine Attacks" on page 3, but typically cost more. As with all security decisions, you must weigh the costs and benefits. In general, I do not require bare-metal machines or dedicated VMs for additional security until the more common problems such as vulnerability management and access management are well under control.

Note that many of the following asset types can be seen as a deconstruction of a VM into smaller components provided "as a service."

Containers

Like VMs, containers run processes that perform business functions, such as web servers or custom application code. However, unlike VMs, they do not contain a full

1 There are people who claim that bare metal is not cloud. By the most commonly accepted definition, NIST SP 800-145, the essential characteristics of cloud computing are on-demand self-service, broad network access, resource pooling, rapid elasticity, and managed service. None of these essential characteristics require virtualization technology, although there can be arguments over the definition of "rapid."

operating system. Containers use the kernel of the VM they are hosted on, and might not have any of the other software that comes with the operating system.

Containers can start up in under a second, which means that in many environments they are created and deleted almost constantly.

Container Attacks

Whereas the hypervisors that run VMs have a very small attack surface, the shared kernel used by all of the containers has a much larger attack surface. For example, the Linux kernel contains over 300 system calls, many of which may be used by containers. A vulnerability in any of these system calls may allow code running in one container to gain access to the entire system.

This doesn't mean that containers are inherently insecure, but you should be careful not to use containers as your only trust boundary between components with wildly different security requirements. For example, having containers that allow internet users to run their own code on the same server as containers that process your most sensitive data is probably asking for trouble.

Container isolation will continue to mature over time. Containers may be limited to fewer and fewer system calls using technologies like *seccomp*, reducing the likelihood that one of those system calls has a vulnerability. The kernel may also perform additional checks as another layer of protection against containerized processes "escaping." Hybrid solutions that which combine the greater isolation of VMs or separate physical systems with the ease of deployment offered by containers are possible, too.

If your containers do contain a full copy of the operating system and allow administrators to log in, they are basically miniature VMs. Although containers can be used in this "mini-VM" model, this isn't the best way to use them. Your asset management strategy for containers depends partly upon how you are using them. We will look at two models, the "native" container model and the "mini-VM" model.

Native container model. In the native container model:

- Containers should hold the bare minimum operating system components needed to perform their function.
- Each container should perform only a single function (or "concern" in some documentation).
- Containers are immutable, meaning that they don't change over time. A container may make changes in some other component, such as writing data to a storage service, but that storage is maintained separately from the container itself.

- Immutable containers remain a perfect copy of the code in the image during their lifetimes—they don't update their own code, and nobody logs in to change it. Rather than updating containers, old containers are destroyed and new containers are created with updated code.

Native, immutable containers should not need to have administrators logging into them for routine maintenance, although you probably need some provision for obtaining emergency access occasionally. If container logins are not allowed in general, access management to the containers becomes less of a risk than with servers. Vulnerability and configuration management are still important risks, but the scope for a given container is much narrower than the scope for a server that might perform many different functions.

Native containers are generally created and destroyed much more often than VMs. That means it makes more sense to inventory the container images than the containers themselves, and just keep track of which image a container is copied from. A container image needs to be inventoried primarily in order to track the software and configurations in the image, so that the image may be updated with security fixes and new configurations as vulnerabilities are discovered.

"Mini-VM" container model. In a model where you treat containers like miniature VMs:

- Containers will usually run a full copy of the user-mode components of the operating system.
- Containers perform multiple functions or concerns, such as running two different types of services in the same container.
- Containers allow administrative logins and change over time.

If you're using containers like mini-VMs, you should inventory and protect them just like VMs. This means installing agents to inventory them and tracking users, software, and all the other items mentioned in the preceding section on VMs.

In both models, you should inventory and update the images, because you don't want new containers to be brought up with vulnerabilities.

Container orchestration systems. Containers are great, but what's even better is to have something that takes care of bundling containers together to perform higher-level functions, starting up multiple copies of these bundles, performing load balancing to those copies, and providing other features such as easy ways for the components to talk to one another. This type of system is called a *container orchestration system*.

The most popular implementation of container orchestration as of this writing is Kubernetes with Docker containers. In a Kubernetes deployment, the primary assets

are clusters, which hold pods, which hold Docker containers, which are copied from images. In a Kubernetes environment, consider inventorying the following components:

- Kubernetes clusters, so that access to them can be controlled and the Kubernetes software may be kept up to date. Vulnerabilities in the Kubernetes software could compromise all of the pods running on it.
- Kubernetes pods, which may contain one or more Docker containers. The Kubernetes command line or API may be used to track the pods currently in existence and which containers make up those pods.
- Docker container images.

Application Platform as a Service

Application Platform-as-a-Service (aPaaS) offerings, such as Cloud Foundry or AWS Elastic Beanstalk, allow you to deploy your code without provisioning VMs yourself. These offerings also provide many resources, such as databases, as part of the platform. So, for example, a deployment may consist of the code you've written plus a database provisioned by the aPaaS. The deployment starts running when you create it and stops running when you destroy it, but you never have to actually create a VM or container to hold it; that's done for you by your cloud provider.

Security of an aPaaS is very specific to the aPaaS and to the provider's implementation of that aPaaS. It's important to understand the isolation model that keeps your compute, network, and storage assets separate from those of other cloud customers. For example, with many Cloud Foundry deployments, you will be running on the same VMs as other customers, which provides limited compute isolation. You will often not be able to contact other containers on the network, so you may have good network isolation. Storage isolation will depend upon what level, if any, of encryption is performed by the persistent storage services available from your provider, and may vary from one storage service to another.

When you create an aPaaS deployment, you need to track both the deployment itself and its dependencies (such as build packs or other subcomponents) for the purposes of vulnerability and configuration management. However, you don't need to inventory anything about the underlying compute resources or storage resources, because these are outside of your control.

Serverless

Serverless functions are a way to have your code running only as needed; some examples are AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions.

Serverless offerings differ from aPaaS offers because nothing runs until its service has been requested; there's nothing specific to you that sits around waiting for incoming requests. This means you don't have to track both an "image" and the "instances" that are created from that image, because there are no long-running instances.

For serverless assets, you don't need to inventory any operating system or platform components. You only need to inventory the serverless deployments you have so that you can manage vulnerabilities in your code and control access to the function.

Storage Assets

Storage assets typically "persist" data, and as such tend to be more permanent than the other types of assets mentioned here. Sometimes data is described as "sticky," because moving large amounts of data around can be difficult and time-consuming. You identified your most important data and storage assets in Chapter 2, but there may be other storage assets that you haven't considered. We'll look at some of the possibilities here.

Because I recommend an asset-oriented approach to risk assessment for most organizations, this book places particular emphasis on storage assets. Access management is the most important security consideration for all of the cloud storage assets listed in this section.

Block storage

Block storage is just the cloud version of a hard drive; data is made available in small blocks (say, 16 KB) to a server in the same manner as a spinning disk controller. Some examples are AWS Elastic Block Storage, Azure Virtual Disks, Google Persistent Disks, and IBM Cloud Block Storage.

The primary security concern with block storage is access management, because an attacker who gets direct access to the block storage bypasses any operating system–level controls you may have on the server using that storage.

File storage

File storage is the cloud version of a filesystem, organizing data into directories and files. Some examples are AWS Elastic File System, Azure Files, Google Cloud Storage FUSE, and IBM Cloud File Storage. As with block storage, the primary concern is access management. Although the filesystem itself often provides access control lists (ACLs) for the files, these are enforced by the operating system, not by the file storage. An attacker with access to the file storage can read all files stored there.

Object storage

In storage terms, an *object* is very similar to a flat file, in that it is a stream of bytes with metadata about the object. The primary differences are:

- Files are stored in folders that may be inside other folders. Objects are all thrown together into a "bucket," without any further levels of organization inside the bucket.[2]
- Objects may have custom metadata associated with them. Files are limited to the types of metadata that a filesystem provides, such as creator, creation time, and permissions.
- Objects cannot be changed after creation. To make updates, you replace the object with a new object. With files, you may update only part of a file, or add additional data to it.
- Object storage offers per-object access control that is enforced by the object storage system. File storage typically enforces access control to the whole filesystem, but then depends upon the operating system using the filesystem to enforce per-file controls.

Most object storage offers different layers of access control, such as high-level policies for a bucket and individual ACLs for specific objects. There have been many notable data breaches when object storage bucket policies were set for open access, so it's very important to keep track of your object storage assets and the access control policies for each one.

Some examples of object storage services are Amazon S3, Azure Blob Storage, Google Cloud Storage, and IBM Cloud Object Storage.

Images

Images are chunks of code—including all the underlying system components, such as the operating system—that you use to run VMs, containers, or aPaaS deployments in a cloud environment. You make a copy of an image and start that copy running. The new copy is often called an "instance" and may begin to diverge from the image at that point. VMs, bare-metal systems, containers, and aPaaS environments all copy images to create running systems.

While images are stored on some type of cloud storage, such as block storage or object storage, access to images is often controlled separately from the underlying storage.

2 You can simulate a folder hierarchy in object storage by using object names with slashes in them. However, if you want to display the objects in a "folder" named *A*, the object storage system is really just searching for all object names that begin with *A/*.

Different types of cloud assets and providers manage images in different ways, but often there are many people in the organization who can get access to the contents of the images and create instances from them. For this reason, images shouldn't contain every bit of information needed for an instance to run. For example, images should not contain sensitive information such as passwords or API keys, because not everyone who has access to create or view the image should know these secrets. An image should be configured so that when a copy (instance) of that image is started, the instance gets the secrets from a secure location that very few people have access to. This is discussed further in "Secrets Management" on page 36. Depending on how you build images, you may be able to perform some checks to ensure secrets aren't included in the image.

If your images do contain sensitive information, it's important to control access to them so that an attacker can't look into an image, pull out the credentials, and use them. In addition, all images must be tracked so that they can be kept up to date with security patches for the operating system, middleware/platform, or custom application software. Otherwise, you'll create cloud assets that are vulnerable as soon as they are created. This is discussed further in Chapter 5.

Cloud databases

Entire treatises have been written about the different types of databases, but as an extreme simplification, cloud databases tend to come in relational and nonrelational flavors. A *relational database* will typically have multiple tables with defined ways to link the data in the different tables. A *nonrelational database* will typically just have the data dumped in a single location in a semistructured format.

Database choices can have significant impacts on the security of the overall application. For example, some in-memory databases used for fast performance do not natively offer encryption either over the network or on disk, which may be a risk, depending on the types of data stored.

Most cloud providers offer several different flavors of both relational and nonrelational databases. All cloud databases can provide access control at the database layer, and some databases can provide more fine-grained control of data in the database.

Message queues

Message queues allow components to send small amounts of data (typically less than 256 KB) to one another, usually through a "publisher/subscriber" model. Although this can be convenient, even these small chunks may contain sensitive data such as personally identifiable information, so it's important to protect access to your message queues. In addition, if some of your components take instructions from messages, an attacker with write access to the message queue might be able to make them do something undesirable.

Secrets, such as encryption keys or passwords, should not be sent across a message queue in general, but should use a storage service specifically designed for this type of data, as described in the following subsection and in Chapter 4.

Configuration storage

In many cases, a cloud deployment brings together code and configuration. The same code is usually shared between different instances of the application, and instances are deployed to different areas or regions using different configurations. *Configuration storage* allows you keep this configuration information separate from the code. Some examples are etcd, HashiCorp Consul, and AWS Systems Manager Parameter Store.

Secrets configuration storage

Secrets configuration storage is a subset of configuration storage specifically designed to hold secret data that may be used to access other systems. Just as it's a good practice to separate your code and configurations, it's also a good idea to separate access to your secrets from other configuration data. Many people may need to be able to view your code and your configurations, but very few people should be able to view the secrets! Therefore, it's important to identify any assets that store secrets, make sure they're built to protect those secrets, and carefully control access.

This is discussed in more detail in Chapter 4. Some examples of secret storage solutions are HashiCorp Vault, Keywhiz, Kubernetes Secrets, and AWS Secrets Manager.

Encryption key storage

Encryption keys are a specific type of secret that are used for encrypting and decrypting data. As with secrets configuration, there are many benefits to using a special-purpose service for this type of data, such as being able to perform wrap and unwrap operations without exposing the master key. You need to identify any assets that store encryption keys and carefully control access to these, in addition to controlling access to the encrypted data.

These types of systems were discussed in detail in Chapter 2. The main types of encryption key storage are dedicated hardware security modules and multitenant key management systems.

Certificate storage

Another specialization of secret storage, *certificate storage* systems can safely store your X.509 private keys, which are used to cryptographically prove that you own the certificate. In addition, these systems can alert you when one of your certificates is due to expire.

Source code repositories and deployment pipelines

Many organizations carefully track other types of assets, but allow their source code to be distributed all over the place and built using many different pipelines.

In many cases, source code doesn't need to be kept secret if good practices such as separating out configuration and secrets are followed. However, ensuring that an attacker doesn't modify your source code or any artifacts during the deployment path is very important, so these assets need to be tracked to protect integrity.

In addition, you need to have a good inventory of your source code repositories in order to effectively check for vulnerabilities. There are tools available to check for bugs in code you've written as well as known vulnerabilities in code you have incorporated from other sources. These tools cannot operate on code that they are not aware of! This will be covered in more depth in Chapter 5.

Network Assets

Network assets are the cloud equivalent of on-premises switches, routers, virtual LANs (VLANs), subnets, load balancer appliances, and similar assets. They enable communication between other assets and to the outside world, and they often perform some security functions.

Virtual private clouds and subnets

Virtual private clouds (VPCs) and *subnets* are high-level ways to draw boundaries around what's allowed to talk to what. It's important to have a good inventory of these; as mentioned earlier, many other controls, such as network scanners, depend on having good inputs for what to scan to be effective. Subnets and VPCs are discussed further in Chapter 6.

Content delivery networks

Content delivery networks (CDNs) can distribute content globally for low-latency access. While the information in a CDN may not be sensitive in most cases, an attacker with access to the CDN can poison the content with malware, bitcoin miners, or distributed denial-of-service (DDoS) code.

DNS records

You need to track your Domain Name System (DNS) records and the registrars you use to register them. Although Transport Layer Security (TLS) connections offer protection against spoofing, as of this writing some browsers do not default to TLS. Spoofing DNS records can lead someone to go to an attacker's site instead of yours,

and then the attacker can steal their credentials, read all of the data going through to your site, and even change data in transit.

In addition to security concerns, if you don't track one of your DNS domains and forget to renew it, you'll have a service outage!

TLS certificates

TLS certificates--often still called SSL certificates, and more properly X.509 certificates—rely on cryptographic principles. They are the best line of defense against an attacker spoofing your website. You need to track your TLS certificates for the following reasons:

- There are cases where an entire class of certificates needs to be reissued, such as when a particular cryptographic algorithm is found to be weak or when a certificate authority has a security issue.
- You must track who has access to the private keys, because these individuals have the ability to impersonate your site.
- Like with DNS domains, if you forget to renew a certificate, you will often have a service outage because connections will fail when a certificate has expired.

If you have a large number of certificates, consider using a certificate storage service, discussed earlier, to track them.

Load balancers, reverse proxies, and web application firewalls

DNS records usually point to one of these network assets for processing and traffic direction. It's important to have a good inventory of these assets for proper access control, because they can usually see and modify all of the network traffic to your applications. These are covered in more detail in Chapter 6.

Asset Management Pipeline

So, now that you know what types of assets to look for, what can you do to track them? In most organizations, there are natural control points on the way to provision services and infrastructure. These will vary between organizations, but you must find the control points and tighten them up to ensure you know about all of your cloud assets and manage the risks appropriately.

I like to explain this using a plumbing analogy. Imagine you have a pipeline containing your various cloud assets, flowing from your cloud providers and leading to your different security systems. You must try to prevent all of the "leaks" that could allow assets to get left out of important security efforts. This is true whether you're running

your entire company's IT, or whether you're only responsible for a single application. Conceptually this looks like Figure 3-1. We'll look now at each piece of the plumbing.

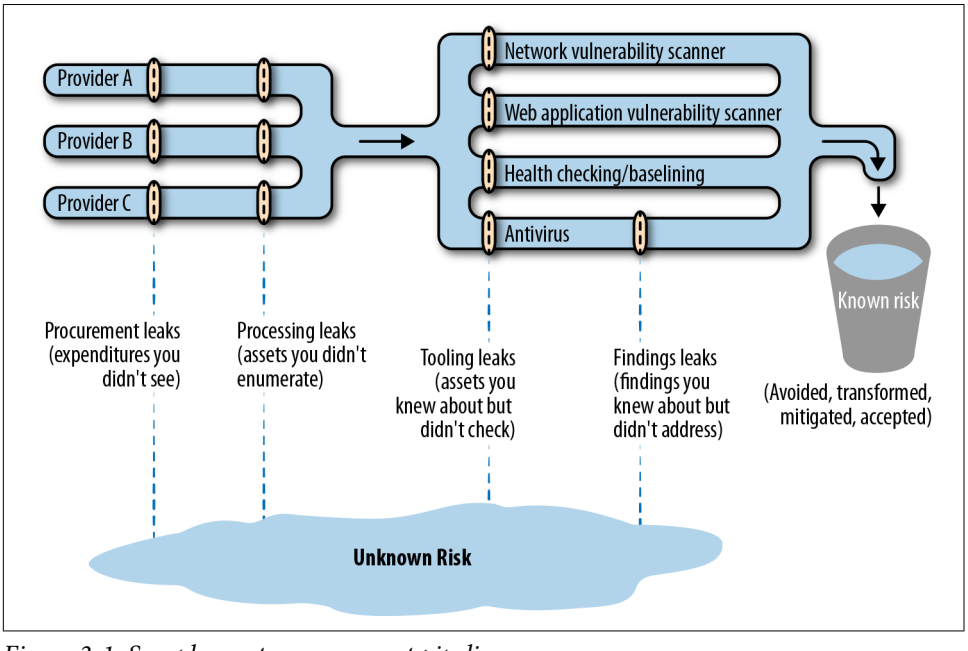


Figure 3-1. Sample asset management pipeline

Procurement Leaks

At the source, you have multiple ways for assets to be created. You may have multiple cloud providers with different delivery models (IaaS, PaaS, SaaS) provisioning many different types of assets. In most cases, you'll be charged for these assets. That often means that a good first step is with the procurement process.

Some cloud providers have built-in asset management systems that already integrate with the other services they provide, and may even have ways to bring in assets from your on-premises environments or other cloud providers. This is a growing field, so look into what your providers offer before building something custom-made.

This isn't foolproof—some cloud resources can be provisioned without spending any money, and in larger organizations people may be able to categorize their cloud expenses in different ways. However, it's a good start.

Look through your IT charges. For each cloud expense, you need to go to the individual responsible for incurring the charges and get some limited auditing credentials.[3] This will allow you to automatically pull inventory information. A "leak" here usually means that you've missed an entire cloud provider, either because you didn't see the expense or because it's a free service.[4]

Processing Leaks

The second step is to use those audit credentials to find out exactly what the cloud providers are doing for you. That means you need to use their portals, APIs, or inventory systems to pull a list of assets. Note that you may have assets inside of other assets. For example, you may have a web server inside a container inside a VM.

Every cloud provider has a portal, API, or set of command-line utilities that can be used to retrieve information about assets. Almost always, automation using the API or command-line tools is preferable because manual inventories are difficult to keep up to date. However, a manual inventory is better than nothing, and might even be sufficient if changes are very infrequent.

In addition to portals and APIs, some cloud providers and third parties have inventory or security tracking systems. As of this writing this is an immature area, but these offer considerable promise, so investigate whether there is a system that meets your requirements before creating something custom-made. Some systems allow you to track down to the level of what's installed on different virtual machines, feed directly into other security services available (such as scanners), and import assets from other providers or on-premises infrastructure. Table 3-1 lists some current services.

Table 3-1. Options for auditing cloud activity

Infrastructure	Ways to audit usage
Amazon Web Services	API, portal, command line, AWS Systems Manager Inventory
Microsoft Azure	API, portal, command line, Azure Automation Inventory
Google Compute Platform	API, portal, command line, Cloud Security Command Center Asset Inventory
IBM Cloud	API, portal, command line, IBM Cloud Security Advisor
Kubernetes	API, dashboard

3 Make sure to follow the least privilege principle, and ensure that credentials for inventory automation don't provide more power to your inventory system than absolutely necessary! An inventory system should not need to read anything but metadata or modify anything other than tags.

4 Note that free services are often not entirely "free"; the provider may get to use your data or get certain rights to your data, so you should inspect the terms of service!

Make sure you delve into each asset type to find additional assets that could be important from a security perspective. A "leak" here means that you queried the cloud provider for assets, but you didn't inventory some cloud assets for that provider. For example, you may have inventoried all of the virtual machines, but missed the object storage buckets that your team provisioned. If you don't inventory those object storage buckets, your downstream tools and processes cannot check the buckets to make sure that access to them is controlled properly, or that they've been assigned the proper tags.

Tooling Leaks

The third step is to ensure that each tool that helps check the security of your assets is tied into this asset inventory and can obtain the information it needs to do its job. Here are some examples:

- Your network vulnerability scanner should be able to obtain the IP addresses in use from the VM information or VPC subnet information.
- Your web application vulnerability scanner should be able to obtain the URLs of each of your web applications.
- Your health checking or baselining system needs to know about the different VMs so that it can check the configurations of each.
- If your organization uses Windows systems, your antivirus solution will need a list of all Windows systems in order to effectively track alerts and ensure antivirus signatures are up to date.

A "leak" in this area means that you knew about some assets but didn't have your tools or processes check those assets for security issues. More information on these tools and protective measures will be given in Chapter 5, but there's really no way for the tools to find security issues in assets that they don't know about.

Findings Leaks

The final step is to ensure you're actually addressing any findings from your tooling systems. This may seem simple, but in practice these findings are often ignored, particularly with "noisy" scanning systems that create a lot of false positives.

It's perfectly acceptable to decide to accept a finding (risk) without fixing it, but ignoring the findings without any sort of review is a "leak."

Tagging Cloud Assets

It makes sense to categorize and organize your assets when creating them, so that you know what they contain and what they are used for. Tags can make automation and access control much easier. Just as you tagged your data assets with the types of data on them in Chapter 2, you also need to tag other types of assets to indicate both the types of data processed by them and why the assets are needed.

It's important to use the same data tags from Chapter 2 to indicate the types of data processed on compute assets, so that you have a consistent view of where your data is stored and processed. However, while it's relatively simple to come up with a set of data classification levels or a list of compliance requirements, there are almost endless possibilities for other operational tags.

Here are some examples of the types of tags that may be useful:

- Function of the asset
- Environment type for the asset, such as development, test, or production
- Application or project that the asset is used for
- Department that is responsible for the asset
- Version number
- Automation tags, which can indicate whether the asset should be selected for action by scripts, scanners, or other automation

With many cloud providers, tags are case sensitive, so *ApplicationA* and *applicationA* won't match.

Looking at our sample application from Chapter 1, we can add some tags to the servers as seen in Figure 3-2.

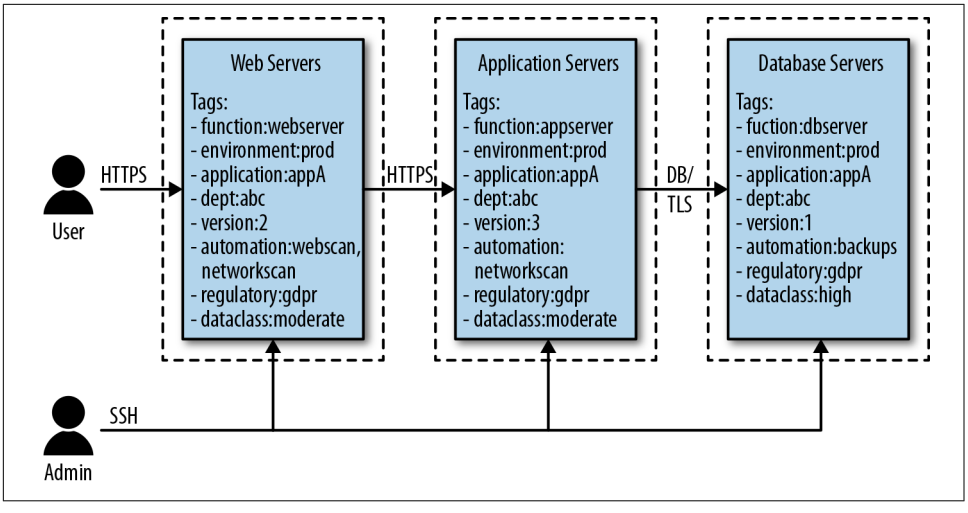


Figure 3-2. Sample application diagram with tags

Proper tagging can enable automated security checks. For example, perhaps you have a very sensible policy that sensitive data must not be stored or accessed on development and test systems. To help enforce this policy, you could:

1. Have automation that searches VMs and tags them with *dataclass:sensitive-data* if the automation detects either certain types of data (such as credit card numbers) or credentials to access sensitive data (such as the production database).
2. Have automation in your build processes to automatically tag VMs as *environment:development*, *environment:test*, or *environment:production* as they're created.
3. Create a report of any assets that have a *dataclass:sensitive-data* tag along with either an *environment:development* or *environment:test* tag.

For tags to be effective, you must maintain a consistent set of tag names and allowed values, which means having a tagging policy and sticking to it. In most smaller organizations, the tagging policy should be organization-wide. A larger organization will need to agree on some organization-wide tags as well as allowing tags specific to business units. In either case, there should be a clear owner of the tagging policy who adds additional tags to the official list as needed.

You may want to develop automation to collect all of the tags currently in use and report on any that are not specified in the tagging policy for your organization or business unit.

Summary

There are so many different as-a-Service offerings available today that it can be difficult to understand and track all of them.

You need to get the biggest bang for the buck for your tracking efforts. This means prioritizing the tracking of providers and assets where losing track of an asset is most likely to cause a large impact, such as assets that store or process sensitive data or that have administrative control over other assets. For example, you may choose not to worry about tracking all of your virtual machine images until you have tight tracking of all of your databases where customer data is stored, your existing virtual machines that have access to those databases, and your source code (and dependent libraries) that process customer data.

Use a pipeline approach that tracks cloud providers, assets created by those providers, what your security tooling does with those assets, and what you do with the findings from those security tools. If you have on-premises resources, treat those the same way as resources at a third-party cloud provider, although you may not have tagging or an API for automation.

Asset management can also have important benefits besides security. For example, you may discover that you have assets that are no longer needed, and deleting these can cut costs in addition to reducing security risks. If you're having difficulty getting support for an asset management solution based solely on security requirements, try pitching it also as a cost-control measure.

CHAPTER 4
Identity and Access Management

Identity and access management (IAM) is perhaps the most important set of security controls. In breaches involving web applications, lost or stolen credentials have been attackers' most-used tool for several years running.[1] If an attacker has valid credentials to log into your system, all of the patches and firewalls in the world won't keep them out!

Identity and access management are often discussed together, but it's important to understand that they are two distinct concepts:

- Each entity (such as a user, administrator, or system) needs an identity. The process of verifying that identity is called *authentication* (often abbreviated as "authn").
- Access management is about ensuring that entities can perform only the tasks they need to perform. The process of checking what access an entity should have is called *authorization* (abbreviated as "authz").

Authentication is proving your identity—that you are who you say you are. In the physical world, this might take the form of presenting an ID card issued by a trusted authority that has your picture on it. Anyone can inspect that credential, look at you, and decide whether to believe that you are who you say you are. As an example, if you drive up to a military base and present your driver's license, you're attempting to authenticate yourself with the guard. The guard may choose to believe you, or may decide you've provided someone else's driver's license, or that it's been forged, or may tell you that the base only accepts military IDs and not driver's licenses.

1 See, for example, the Verizon Data Breach Investigations Report.

Authorization refers to the ability to perform a certain action, and generally depends first on authentication (knowing who someone is). For example, the guard at the base may say, "Yes, I believe you are who you say you are, but you're not allowed to enter this base." Or you may be allowed in, but may not be allowed access to most buildings once inside.

In IT security, we often muddle these two concepts. For example, we may create an identity for someone (with associated credentials such as a password) and then implicitly allow that anyone with a valid identity is authorized to access all data on the system. Or we may revoke someone's access by deleting the person's identity. While these solutions may be appropriate in some cases, it's important to understand the distinctions. Is it really appropriate to authorize every user for full access to the system? What if you have to give someone outside the organization an identity in order to allow them to access some other area of the system—will that user also automatically gain access to internal resources?

Note that the concepts (and analogies) can get complicated very quickly. For example, imagine a system where instead of showing your license everywhere, you check out an access badge which you show to others, and a refresh badge which you need to show only to the badge issuer. The access badge authenticates you to everyone else, but works for only one day, after which you have to go to the badge office and show the refresh badge to get a new access badge. Each site where you present your access badge verifies the signature on it to make sure it's valid, and then calls a central authority to ask whether you're on the list for access to that resource. This is similar to the way some IT access systems work, although fortunately your browser and the systems providing service to you take care of these details for you!

An important idea here, as well as in other areas of security, is to minimize the number of organizations and people whom you have to trust. For example, except for cases involving zero-knowledge encryption,[2] you're going to have to trust your cloud provider. You have to accept the risk that if your provider is compromised, your data is compromised.[3] However, since you've already decided to trust the cloud provider, you want to avoid trusting any other people or organizations if you can instead leverage that existing trust. Think of it like paying an admission fee; once you've already paid the "fee" of trusting a particular organization, you should use it for all it's worth to avoid introducing additional risk into the system.

2 Zero-knowledge encryption means that your provider has no technical way of decrypting the data, usually because you only send encrypted data without the keys. This sharply limits what the provider can do, and is most suitable for backup services where the provider just needs to hold a lot of data without any processing.

3 I like to jokingly refer to this as the "principle of already screwed." It is good to have a way to monitor your provider's actions, though, to detect a potential compromise.

Differences from Traditional IT

In traditional IT environments, access management is often performed in part by physical access controls (who can enter the building) or network access controls (who has VPN [virtual private network] access to the network). As an example, you may be able to count on a perimeter firewall as a second layer of protection if you fire an admin and forget to revoke their access to one of the servers.

It's important to note that this is often a very weak level of security—are you confident that the access controls for all of your Ethernet ports, wireless access points, and VPN endpoints will stand up to even casual attack? In most organizations, someone could ask to use the bathroom and plug a $5 remote access device into an Ethernet port in seconds, or steal wireless or VPN credentials to get in without even stepping foot on the premises. The chance of any given individual having their credentials stolen might be small, but the overall odds increase quickly as you add more and more people to the environment.

As mentioned previously, access control is sometimes performed simply by revoking a user's entire identity, so that they can no longer log in at all. It's important to note that in cloud environments this often won't take care of the entire problem! Many services provide long-lived authentication tokens that will continue to work even without the ability to "log in." Unless you're careful to integrate an "offboarding" feed that notifies applications when someone leaves so that you can revoke all access, people may retain access to things you didn't intend. As an example, when was the last time you typed in your Gmail password? Changing your Gmail password or preventing you from using the login page wouldn't do any good if Gmail didn't also revoke the access tokens stored in your browser cookies during a password change operation.

There are many examples of data breaches caused by leaving Amazon Web Services S3 buckets with public access. If these were file shares left open to the enterprise behind a corporate firewall, they might not have been found by an attacker or researcher on the internet. (In any organization of a reasonable size, there are almost certainly bad actors on the organization's internal network who could have stolen that information, perhaps without detection.)

Many organizations find that they've lived with lax identity and access management controls on-premises, and need to improve them significantly for the cloud. Fortunately, there are services available to make this easier.

Life Cycle for Identity and Access

Many people make the mistake of thinking of IAM as only authentication and authorization, and we jumped directly into authentication and authorization in the introduction. Those are both very important, but there are other parts of the identity life cycle that happen before and after. In the example taken earlier from an imaginary real-life situation, we assumed that the requester already had an identity (the driver's license)—but how did they get that? And who put the requester's name on the list of people who were allowed on the base?

Many organizations handle this poorly. Requesting an identity might be done by calling or messaging an administrator, who approves and creates the identity without keeping any record of it. This might work fine for really small organizations, but many times you need a system to record when someone requests access, how the requester was authenticated, and who approved the new identity or the access.

Even more important is the backend of the life cycle. You need a system that will automatically check every so often if a user's identity and access are still needed. Perhaps the person has left the company, or moved to a different department, and should no longer have access. (Or worse, imagine having the unpleasant task of firing someone, and realizing a month later that due to human error the person still has access to an important system!)

There are many different versions of IAM life cycle diagrams with varying amounts of detail in the steps. The one in Figure 4-1 shows the minimum number of steps, and addresses both creation and deletion of identities along with creation and deletion of access rules for those identities. Identity and access may be handled by different systems or the same system, but the steps are similar.

Note that you don't necessarily need a fancy automated system to implement every one of these steps. In an environment with few requesters and few approvers, a mostly manual process can work fine as long as it's consistently implemented and there are checks to prevent a single human error from causing problems. As of this writing, most automated systems to manage the entire life cycle (often called identity governance systems) are geared toward larger enterprises; they are usually expensive and difficult to implement. However, there is a growing trend to provide these governance solutions in the cloud like other services. These are often included as part of other identity and access services, so even smaller organizations will be able to benefit from them.

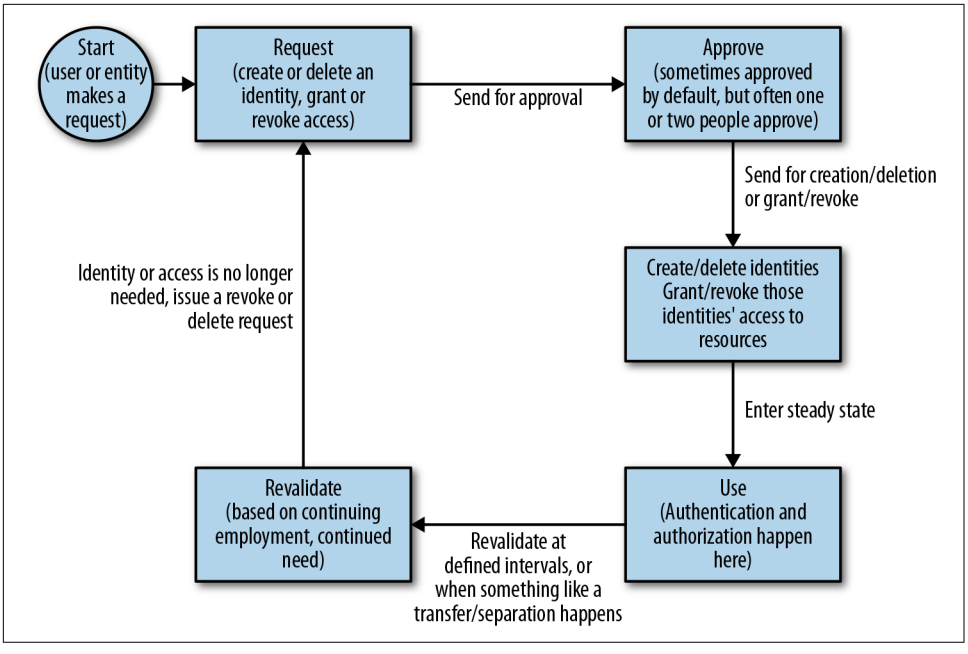


Figure 4-1. IAM life cycle

Also, note that the processes and services used might differ considerably, depending on who the entities are. The types of identity and access management used to give your employees access to your cloud provider and your internal applications differ considerably from those used to grant your customers end-user access to your applications. I'll distinguish between these two general cases in the following discussion.

Don't forget about identities for non-human things in the system, such as applications. These need to be managed too, just like human identities. Many teams do a great job of controlling access for people, but have very lax controls on what automation is authorized to do.

Let's go through each of these steps. The process starts when someone or something puts in a request. This might be the manager of a newly hired employee, or some automation such as your HR system.

Request

An entity makes an identity or access management request. This entity should usually be authenticated in some fashion. Inside your organization, you don't want any

anonymous requests for access, although in some cases the authentication may be as simple as someone visually recognizing the person.

When providing access to the general public, such as access to your web application, you often want to link to some other identity such as an existing email address or a mobile phone number.

The common requests are:

- Create an identity (and often implicitly grant that identity at least a base level of access).
- Delete an identity, if the entity no longer needs to authenticate anywhere.
- Grant access to an existing identity, such as access to a new system.
- Revoke access from an existing identity.

In cloud environments, the request process often happens "out of band," using a request process inside your organization that doesn't involve the cloud IAM system yet.

Approve

In some cases, it's acceptable to implicitly approve access. For example, when granting access to a publicly available web application, anyone who requests access is often approved automatically, provided that they meet certain requirements. These requirements might be anti-fraud in nature, such as providing a valid mobile number or email address, providing a valid credit card number, completing a CAPTCHA or "I am not a robot" form, or not originating from an anonymizing location such as an end-user VPN provider or a known Tor exit node.

However, inside an organization, most access requests should be explicitly approved. In many cases, two approvals are reasonable—for example, the user's immediate supervisor, as well as the owner of the system to which access is being requested. The important thing is that the approver or approvers are in a position to know whether the requested access is reasonable and necessary. This is also an internal process for your team that usually happens with no interaction with your cloud providers.

Create, Delete, Grant, or Revoke

After approval, the actual action to create an identity, delete an identity, grant access, or revoke access may happen automatically. For example, the request/approve system may use cloud provider APIs to create the identity or grant the access.

In other cases, this may generate a ticket, email, or other notification for a person to take manual action. For example, another admin may log into the cloud portal to create the new identity and grant it a certain level of access.

Authentication

So far, much of what has been discussed is not really different from access management in on-premises environments—before an identity exists, you have to request it and have a process to create it. However, authentication is where cloud environments begin to differ because of the many identity services available.

It's important to distinguish between the identity *store*, which is the database that holds all of the identities, and the *protocol* used to authenticate users and verify their identities, which can be OpenID, SAML, LDAP, or others.

It's also important to distinguish *whom* you are authenticating. There are often different systems available for:

- Authenticating your organization's employees with your cloud providers (generically business-to-business, and often called something like "Cloud IAM" by cloud providers)
- Authenticating your organization's customers with your own applications (business-to-consumer)
- Authenticating your organization's employees with your own applications (business-to-employee)

Cloud IAM Identities

Many cloud providers offer IAM services at no additional charge for accessing their cloud services. These systems allow you to have one central location to manage identities of cloud administrators in your organization, along with the access that you have granted those identities to all of the services that cloud provider offers.

This can be a big help. If you are using dozens or hundreds of services from a cloud provider, it can be difficult to get a good picture of what level of access a given person has. It can also be difficult to make sure you've deleted all of their identities when that person leaves your organization. As previously mentioned, removing access is especially important, given that many of these services may be used directly from the internet!

Table 4-1 lists some examples of identity services to authenticate your cloud administrators with cloud provider services.

Table 4-1. Cloud provider identity services

Provider	Cloud identity system
Amazon Web Services	Amazon IAM
Microsoft Azure	Azure Active Directory B2C
Google Compute Cloud	Cloud Identity
IBM Cloud	Cloud IAM

Business-to-Consumer and Business-to-Employee

In addition to the identities your organization uses for accessing cloud provider services, you may also need to manage identities for your end users, whether they are external customers or your own employees.

Although you can do customer identity management yourself by simply creating rows in a database with passwords, this is often not an ideal experience for your end users, who will have to juggle yet another login and password. In addition, there are significant security pitfalls to avoid when verifying passwords, as described in "Passwords and API Keys" on page 31. There are two better options:

- Use an existing identity service. This may be an internal identity service for your employees or your customer's employees. For end customers, it may also be an external service such as Facebook, Google, or LinkedIn. This requires you to trust that identity service to properly authenticate users for you. It also makes your association with the identity service obvious to your end users when they log in, which may not always be desirable.
- Use customer identities specific to your application, and use a cloud service to manage these customer identities.

The names of these Identity-as-a-Service (IDaaS) offerings do not always make it clear what they do. Table 4-2 lists some examples from major cloud infrastructure providers as well as third-party providers. There are many third-party providers in this space and they change often, so this isn't an endorsement of any particular providers. For business-to-employee cases, most of these IDaaS services can also use your employee information store, such as your internal directory.

Table 4-2. ID management systems

Provider	Customer identity management system
Amazon Web Services	Amazon Cognito
Microsoft Azure	Azure Active Directory B2C
Google Compute Cloud	Firebase
IBM Cloud	Cloud Identity
Auth0	Customer Identity Management

Provider	Customer identity management system
Ping	Customer Identity and Access Management
Okta	Customer Identity Management
Oracle	Oracle Identity Cloud Service

Note that whether you're creating identities yourself or using a cloud service, any personally identifiable information you collect may be subject to regulatory requirements such as the EU's GDPR.

Multi-Factor Authentication

Multi-factor authentication is one of the best ways to guard against weak or stolen credentials, and if implemented properly will only be a small additional burden on users. Most of the identity services shown in Table 4-2 support multi-factor authentication.

As background, the different authentication factors are commonly defined as:

1. Something you know. Passwords are the most common examples.
2. Something you have. For example, an access badge or your mobile phone. Note that this is typically defined as a physical item that's difficult to replicate, rather than a piece of data that's easily copied.
3. Something you are. For example, your fingerprint or retinal pattern.

As the name implies, multi-factor authentication is using more than one of these factors for authentication. Using two of the same factor, like two different passwords, does not help much! The most common implementation is *two-factor access* (2FA), which uses something you know (like a password) and something you have (like your mobile phone).

2FA should really be the default for most access; if implemented correctly, it requires very little extra effort for most users. You should absolutely use 2FA any place where the impact of lost or stolen credentials would be high, such as for any privileged access, access to read or modify sensitive data, or access to systems such as email that can be leveraged to reset other passwords. For example, if you're running a banking site, you may decide that the impact is low if someone is able to read a user's bank balance, but high (with 2FA required) if someone is attempting to transfer money.

If you're managing a cloud environment, unauthorized administrative access to the cloud portal or APIs is a very high risk to you, because an attacker with that access can usually leverage it to compromise all of your data. You should turn on two-factor authentication for this type of access; most cloud providers natively support this.

Alternatively, if you're using single sign-on (SSO), as discussed in "Single Sign-On" on page 33, your SSO provider may already perform 2FA for you.

Many services offer multiple 2FA methods. The most common methods for verifying "something you have" are:

- Text messages to a mobile device (SMS). This method is quickly falling out of favor because of the ease of stealing someone's phone number (via SIM cloning or number porting) or intercepting the message, so new implementations should not use SMS, and existing implementations should move to another method. This does require network access to receive the text messages.
- Time-based one-time passwords (TOTPs). This method requires providing a mobile device with an initial "secret" (usually transferred by a 2D barcode). The secret is a formula for computing a one-time password every minute or so. The one-time password needs to be kept safe for only a minute or two, but the initial secret can allow any device to generate valid passwords and so should be forgotten or put in a physically safe place after use. After the initial secret is transferred, network access is not required for the mobile device, only a synchronized clock.
- Push notifications. With this method, an already-authenticated client application on a mobile device makes a connection to a server, which "pushes" back a one-time-use code as needed. This is secure as long as the authentication for the already-authenticated client application is secure, but does require network access for the mobile device.
- A hardware device, such as one complying with the FIDO U2F standard, which can provide a one-time password when needed. Devices like this will likely become ubiquitous in the near future, integrated with smartphones or wearable technologies such as watches and rings, and will probably be the only form of authentication required for lower-risk transactions (such as transactions below a certain dollar amount or access to many web sites).

Note that all of these methods to verify "something you have" are vulnerable to social attacks, such as calling the user under false pretenses and asking for the one-time password! In addition to rolling out multi-factor authentication, you must provide some minimal training to users so that they don't accidentally negate the protection provided by the second factor.

All major cloud providers offer ways to implement multi-factor authentication, although Google uses the friendlier term "2-Step Verification."

Passwords and API Keys

If you're using multi-factor authentication, passwords are no longer your only line of defense. That said, and despite the cries of "passwords are dead," as of this writing it's still important to choose good passwords. This is often even more true in cloud environments, because in many cases an attacker can guess passwords directly over the internet from anywhere in the world.

While there is lots of advice and debate about good passwords, my recommendations for choosing passwords are simple:

1. Never reuse passwords unless you genuinely don't care about an unauthorized user getting access to the resources protected by that password. When you type a password into a site, you should assume that the site's administrators are malicious and will use the password you have provided to break into other sites. For example, you might use the same password on a dozen forum systems because you don't really care if someone posts as you on any or all of those forums. (Even then, though, there is still some risk that the user can somehow leverage that access to reset other passwords, so it's best not to reuse passwords at all.)
2. Not reusing passwords means you'll end up with a lot of passwords, so use a reputable password manager to keep track of them. Store copies of any master passwords or recovery keys in a physically secured location, such as a good safe or a bank safe deposit box.
3. For passwords that you do not need to remember (for example, that you can copy and paste from your password manager), use a secure random generator. Twenty characters is a good target, although you may find some systems that won't accept that many characters; for those, use as varied a character set as possible.[4]
4. For passwords you do need to remember, such as the password for your password manager, create a six-word Diceware[5] password and put the same non-alphabetic character, such as a dollar sign, equals sign, or comma, between each word. Feel free to regenerate the password a few times until you find one that you can construct some sort of silly story to help you remember. This will be easy to memorize quickly and nearly impossible for an attacker to guess. The only draw-

4 Password strength is usually measured in "bits of entropy." A *very* oversimplified explanation is that if you give an attacker all of the information you can about how a password is constructed but not the actual password, such as "it's 20 uppercase alphabetic characters," the number of bits of entropy is about $\log_2$(number of possible passwords).

5 Diceware is based on the idea that it's far easier for humans to remember phrases than characters, and that almost everyone can find some six-sided dice. There are wordlists you can download, and you can then roll dice to randomly pick five or six words off the list. The result is an extremely secure password that's easy to remember.

back is that it takes a while to type, so you don't want to have to type it constantly!

API keys are very similar to passwords but are designed for use by automation, not people. For that reason, you cannot use multi-factor authentication with API keys, and they should be long random strings, as noted in item 3 in the preceding list. Unlike most user identities where you have a public user ID and a private password, you have only a private API key that provides both identity and authentication.

Verifying Passwords

You may also be tasked with verifying users' passwords, which can be much more complicated than it seems. Avoid this task if possible!

The simplest way to verify passwords is to store a list of the users and passwords and then check to see whether the password entered matches what's on the list. This is a very bad idea, however, because if someone gets access to your list, they have everything they need to impersonate every user on the list!

A much better method is to not store the passwords themselves, but to store something that can be used to verify the passwords. This is implemented using a *one-way hash*, which is something you can derive by a function if you have the password, but which cannot be used to go backwards to get the password. However, the devil is in the details—if you use the wrong function or the wrong parameters for the function, the passwords can be easily obtained ("cracked") through a brute-force attack, by guessing a lot of possible passwords. Perfectly good hash algorithms such as SHA-256 are terrible for password hashes because they're fast to compute, by design.

As of this writing, password hashes should be stored using scrypt, bcrypt, or PBKDF2 functions with reasonable parameters. The recommendations for functions and parameters change over time as cracking hardware gets more sophisticated and weaknesses are found in hashing algorithms, so you must reevaluate your choices at least annually. When you change algorithms or parameters, all new passwords will use the new methods, but by design there's no way to convert the old hashes to new hashes. If there's an urgent need to change (such as evidence of a breach that might have gained access to password hashes), you must reset all user passwords immediately.

Even if you store hashes securely, you should have a testing mechanism in place to prevent users from using really easy-to-guess passwords like *abc123* or *Fall2018*. Attackers are increasingly using techniques such as "password spraying," where they try an easy password on hundreds or thousands of IDs at once. This often doesn't trigger any alarms because it shows up as only a single failed login for each ID.

For cloud services and applications, use a federated identity from another provider, or a consumer/employee IAM cloud service where possible. For system-level access, use key-based authentication or centralized authentication with password strength test-

ing. Don't store and verify password hashes yourself unless there is no good alternative.

Shared IDs

Shared IDs are identities for which more than one person has the password or other credentials, such as the built-in *root* or *administrator* accounts on a system. These can be difficult to handle well in cloud environments, just as they are on-premises.

In general, users should use personal IDs rather than shared IDs. They may assume a role or use a separate higher-privileged ID for some activities. When you do need to use shared IDs, you should be able to tell exactly which individual was using the ID for any access. In practice, this usually means that you have some sort of check-in/check-out process.

Federated Identity

Federated identity is a concept, not a specific technology. It means that you may have identities on two different systems, and the administrators of those systems both agree to use technologies that link those identities together so that you don't have to manually create separate accounts on each system. From your perspective as a user, you have only a single identity.

In practice what this usually means is that Company A and Company B both use your corporate email address, *user@company-a.com*, as your identity, and Company B defers to Company A to actually verify your identity. Company A will then pass an assertion or token back with its seal of approval: "Yes, this is indeed *user@company-a.com*; I have verified them, here is my signature to prove that it's me, and you've already agreed that you'll trust me to verify identities that end in *@company-a.com*."

Single Sign-On

Single sign-on (SSO) is a set of technology implementations that rely upon the concept of federated identity.

In the bad old days, every website had a separate login and password (admittedly, this is still the dominant model today). That's a lot of passwords for users to keep track of! The predictable result is that users often reuse the same password across multiple sites, meaning that the user's password is only as well protected as the weakest site.

Enter SSO. The idea is that instead of a website asking for a user's ID and password, the website instead redirects the user to a centralized identity provider (IdP) that it trusts. (Note that the identity provider may not even be part of the same organization—the only requirement is that the website trusts it.) The IdP will do the work of authenticating the user, via means such as a username and password, and hopefully

an additional authorization factor such as possession of a phone or hardware key. It will then send the user back to the original website with proof that it has verified the user. In some cases, the IdP will also send information (such as group membership) that the website can use to make authorization decisions, such as whether the user should be allowed in as a regular user, as an administrator, or not at all.

For the most part, SSO works only for websites and mobile applications. You need a different protocol for performing authentication on non-web assets such as network devices or operating systems, like LDAP, Kerberos, TACACS+, or RADIUS.

Rarely do you find something that's both easier for users *and* provides better security! Users only have to remember one set of credentials, and because these credentials are only ever seen by the identity provider (and not any of the individual sites), a compromise of those sites won't compromise the user's credentials. The only drawback is that this is slightly more difficult for the website to implement than poor authentication mechanisms, such as comparing against a plaintext password or an insecurely hashed password in a database.

SAML and OIDC

As of this writing, SAML (Security Assertion Markup Language—the abbreviation rhymes with "camel") and OIDC (OpenID Connect) are the most common SSO technologies. While the end results are similar, the mechanisms are somewhat different.

The current SAML version is 2.0, and it has been around since 2005. It is one of the most common SSO technologies, particularly for large enterprise applications. While there are many in-depth explanations of how SAML works, here is a very simplified version:

1. You point your web browser at a web page you want to access (called a *service provider* or SP).
2. The SP web page says, "Hey, you don't have a SAML cookie, so I don't know who you are. Go over here to this identity provider web page and get one," and redirects you.
3. You go to the IdP and log in using your username, password, and possibly a second factor.
4. When the IdP is satisfied it's really you, it gives your browser a cookie with a cryptographically signed XML "assertion" that says, "I'm the identity provider, and this user is authenticated," and then redirects you back.
5. Your web browser hands that cookie back to the first web page (SP). The SP verifies the cryptographic signature and says, "You managed to convince the IdP of your identity, so that's good enough for me. Come on in."

After you've logged in once, this all happens automatically for a while until those assertion documents expire, at which point you have to log into the IdP again.

One important thing to note is that there was never any direct communication between the initial web page and the identity provider—your browser did all of the hard work to get the information from one place to another. That can be important in some cases where network communications are restricted.

Also note that SAML provides only identity information, by design. Whether or not you're authorized to log in or take other actions is a different question, although some SAML implementations pass additional information along with the assertion (such as group membership) that can be used to make authorization decisions.

OpenID Connect is a much newer authentication layer, finalized in 2014, on top of OAuth 2.0. It uses JSON Web Tokens (JWTs, pronounced "jots") instead of XML, and uses somewhat different terminology ("relying party" is usually used in OIDC versus "service provider" in SAML, for example).

OIDC offers both *Authorization Code Flows* (for traditional web applications) and *Implicit Flows* (for applications implemented using JavaScript on the client side). While there are numerous differences from SAML, the end results are similar in that the application you're authenticating with never sees your actual password, and you don't have to reauthenticate for every application.

Note that some services can take requests from OIDC-enabled applications and "translate" these to requests to a SAML IdP. In larger organizations, it's very common to have both standards in use.

SSO with legacy applications

What if you want to provide single sign-on to a legacy application that doesn't support it? In this case you can put something in front of the application that handles the SSO requests and then tells the legacy application who the user is.

The legacy application will trust this frontend service (often a reverse proxy) to perform authentication, and it must not accept connections from anything else. Techniques like this are often needed when moving an existing application to the cloud. Many of the Identity-as-a-Service providers listed earlier also offer ways to SSO-enable legacy applications.

Instance Metadata and Identity Documents

As mentioned earlier in this chapter, we often assume that automation, such as a program running on a system, has already been assigned an identity and a way to prove that identity. For example, if I start up a new system, I can create a username and

password for that system and supply that information as part of creating the system. However, in many cloud environments, there are easier ways.

A process running on a particular system can contact a well-known endpoint that will tell it all about the system it's running on, and the process will also provide a cryptographically signed way to prove that system's identity. The exact details differ from provider to provider, but conceptually it looks like Figure 4-2.

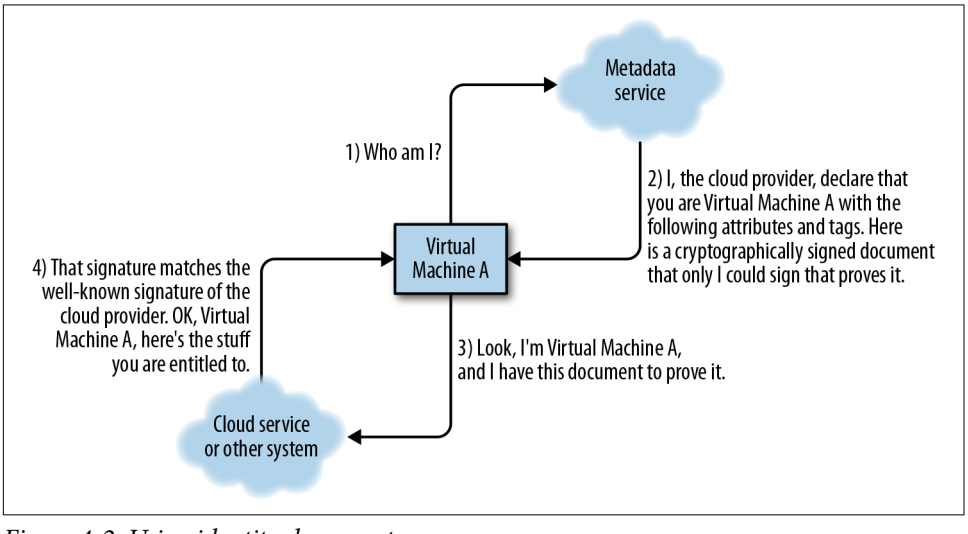


Figure 4-2. Using identity documents

This is not foolproof, however, in that any process on the system can request this metadata, regardless of its privilege level on the system. This means you either need to put only processes of the same trust level, or take actions to block lower-privileged processes from assuming the identity of the entire system. This can be a particular concern in container environments, where any container on a host system could request the metadata and then pretend to be that host system. In cases like this, you need to block the containers from reaching the metadata service.

Secrets Management

We talked about passwords earlier primarily in the context of a person authenticating with a system. Administrative users and end users have had *secrets management* techniques for as long as there have been secrets, ranging from good (password managers and physical safes) to really bad (the ubiquitous Post-it note on the monitor or under the keyboard).

In many cases you also need one system, such as an application server, to automatically authenticate with another system, such as a database server. Clearly multi-factor

authentication can't be used here; the application doesn't have a mobile phone! This means you need to be very careful with the authentication credentials.

These authentication credentials may involve a password, API key, cryptographic token, or public/private key pair. All these solutions have something that needs to be kept secret. In addition, you may have items unrelated to authentication that need to be kept secret, such as encryption keys. We refer to all of these things simply as *secrets*, and secrets management is about making them available to the entity that needs them—and nobody else.

Secrets are dangerous things that should be handled carefully. Here are some principles for managing secrets:

- Secrets should be easy to change at regular intervals and whenever there's any reason to think they may have leaked out. If changing the secret means that you have to take the application down and manually change it in many places, that's a problem.
- Secrets should always be encrypted at rest and in motion, and they should be distributed to systems only after proper authentication and authorization.
- If possible, no human should know the secrets—not the developers who write the code, not the operators who can look at the running system, nobody. This often is not possible, but we should at least strive to minimize number of the people who know secrets!
- The system storing and handing out the secrets should be well protected. If you put all the secrets in a vault and then hand out keys to the vault to dozens of people, that's a problem.
- Secrets should be as useless to an attacker as possible while allowing the system to function. This is again an instance of least privilege; try not to keep secrets around that offer the keys to the kingdom, such as providing root access to all systems, but instead have limited secrets, such as a secret that allows read-only access to a specific database.
- All accesses and changes to secrets should be logged.

Even organizations that do a great job with authentication and authorization often overlook secrets management. For example, you may do a great job keeping track of which people have personal IDs with access to a database, but how many people know the password that the application server uses to talk to the database? Does it get changed when someone leaves the organization? In the worst case, this password is stated directly in the application server code and checked into some public repository, such as GitHub.

In 2016, Uber had a data breach involving 57 million of its drivers and customers because some secrets (AWS credentials, in this case) were in its source code. The code

needed the AWS credentials to function, but putting secrets directly into the source code (or into the source code repository as part of a configuration file) is a bad idea, for two reasons:

- The source code repository is probably not designed primarily for keeping information secret. Its primary function is protecting the *integrity* of the source code—preventing unauthorized modification to insert a backdoor, for example. In many cases the source code repository may show the source code to everyone by default as part of social coding initiatives.[6]
- Even if the source code repository is perfectly safe, it's very unlikely that everyone who has access to the source code should also be authorized to see the secrets used in the production environment.

The most obvious solution is to take the secrets out of the source code and place them somewhere else, such as in a safe place in your deployment tooling or on a dedicated secrets server.

In most cases, a deployment of an application will consist of three pieces that come together:

- The application code
- The configuration for this particular deployment
- The secrets needed for this particular deployment

Storing all three of these things together is a really bad idea, as previously discussed. Having configuration and secrets together is also often a bad idea, because systems designed to hold configuration data may not be properly designed for keeping that data secret.

Let's take a look at four reasonable approaches to secrets management, ranging from minimally secure to highly secure.

The first approach is to use existing configuration management systems and deployment systems for storing secrets. Many popular systems now have some ability to hold secrets in addition to normal configuration data—for example, Ansible Vault and Chef encrypted data bags. This can be a reasonable approach if the deployment tooling is careful with the secrets, and more importantly if access to the deployment system and encryption keys is tightly controlled. However, there are often too many people who can read the secrets. In addition, changing secrets usually requires redeploying the system, which may be more difficult in some environments.

6 There is actually a common term for secrets found in public GitHub repositories: "GitHub dorks."

The second approach is to use a secrets server. With a separate secrets server, you need only a reference to the secret in the configuration data and the ability to talk to the secrets server. At that point, either the deployment software or the application can get the secret by authenticating with the secrets server using a secrets server password…but you see the problem, right? Now you have another secret (the password to the secrets server) to worry about.

Although imperfect, there's still considerable value to this approach to secrets management:

- The secrets server requests can be logged, so you may be able to detect and prevent an unauthorized user or deployment from accessing the secrets. This is discussed more in Chapter 7.
- Access to the secrets server may use other authentication methods than just the password, such as the IP address range requesting the secret. As discussed in Chapter 6, IP whitelisting usually isn't sufficient by itself, but it is a useful secondary control.
- You can easily update the secrets later, and all of your systems that retrieve the secrets will get the new ones automatically.

The third approach has all of the benefits of a secrets server, but uses a secure introduction method to reduce the likelihood that an attacker can get the credentials to access the secrets server:

1. Your deployment tooling communicates with the secrets server to get a one-time-use secret, which it passes along to the application.
2. The application then trades that in for the real secret to the secrets server, and it uses that to obtain all the other secrets it needs and hold them in memory. If someone has already used the one-time secret, this step will fail, and the application can send an alert that something is wrong.

Your deployment tooling still needs one set of static credentials to your secrets server, but this allows it only to obtain one-time keys and not to view secrets directly. (If your deployment tooling is completely compromised, then an attacker could deploy a fake copy of an application to read secrets, but that's more difficult than reading the secrets directly and is more likely to be detected.)

Operations personnel cannot view the secrets, or the credentials to the secrets server, without more complicated memory-scraping techniques. For example, instead of simply reading the secret out of a configuration file, a rogue operator would have to dump the system memory out and search through it for the secret, or attach a debugger to a process to find the secret.

The fourth approach, if available, is to leverage some offerings built into your cloud platform by its provider to avoid the "turtles all the way down" problem:

1. Some cloud providers offer instance metadata or identity documents to systems provisioned in the cloud. Your application can retrieve this identity document, which will say something like, "I am server ABC. The cloud provider cryptographically signed this document for me, which proves my identity."
2. The secrets server then knows the identity of the server, as well as metadata such as tags about the server. It can use this information to authenticate and authorize an application running on the server and provide it the rest of the secrets it needs to function.

Let's summarize the four reasonable approaches to secrets management:

- The first approach stores secrets only in the deployment system, using features designed to hold secrets, and tightly controls access to the deployment system. Nobody sees the secrets by default, and only authorized individuals have the technical ability to view or change them in the deployment system.
- The second approach is to use a secrets server to hold secrets. Either the deployment server or the deployed application contacts the secrets server to get the necessary secrets and use them. In many cases the secrets are still visible in the configuration files of the running application after deployment, so operations personnel may be able to easily view the secrets or the credentials to the secrets server.
- The third approach has the deployment server only able to get a one-time token and pass it to the application, which then retrieves the secrets and holds them in memory. This protects you from having the credentials to the secrets server or the secrets themselves intercepted.
- The fourth approach leverages the cloud provider itself as the root of trust. The cloud provider provides trusted identity documents and metadata that the secrets server can use to decide which secrets to provide to each application.

Although this is still a relatively new market as of this writing, several products and services are available to help you manage secrets. HashiCorp Vault and Keywhiz are standalone products that may be implemented on-premises or in the cloud, and AWS Secrets Manager is available through an as-a-Service model.

Authorization

Once you've completed the authentication phase and you know who your users are, it's time to make sure they are limited to performing only the actions they are supposed to perform. Some examples of authorization may be permission to access an

application at all, to access an application with write access, to access a portion of the network, or to access the cloud console.

End-user applications often handle authorization themselves. For example, there may be a database row or document for each user listing the access level that user has. This makes some sense, because each application may have specific functions to authorize, but it means that you have to visit every application to see all of the access a user has.

The most important concepts to remember for authorization are *least privilege* and *separation of duties*. As a reminder, least privilege means that your users, systems, or tools should be able to access only what they need to do their jobs, and no more. In practice, this usually means that you have a "deny by default" policy in place, so that unless you specifically authorize something, it's not allowed.

Separation (or segregation) of duties actually comes from the world of financial controls, where two signatures may be needed for checks over a certain amount. In the world of cloud security, this usually translates more generally into making sure that no one person can completely undermine the security of the entire environment. For example, someone with the ability to make changes on systems should not also have the ability to alter the logs from those systems, or the responsibility for reviewing the logs from those systems.

For cloud services and internal applications, *centralized authorization* is becoming more popular.

Centralized Authorization

The old, ad hoc practice of scattering identities all over the place has been solved through federated identities and single sign-on. However, you may still have authorization records scattered all over the place—every application may be keeping its own record of who's allowed to do what in that application.

You can deauthorize someone completely by deleting their identity (assuming persistent access tokens don't keep them authorized for a while), but what about revoking only some access? The ability to remove someone's identity is important, but it's a pretty heavy-handed way to perform access management. You often need more fine-grained ways to manage access. Centralized authorization can let you see and control what your users have access to in a single place.

In a traditional application, all of the authorization work was performed internally in the application. In the world of centralized authorization, the responsibilities typically get divided up between the application and the centralized authorization system. There are more details in some systems, but here are the basic components:

Policy Enforcement Point (PEP)
: This point is implemented in the application, where the application controls access. If you don't have the specified access in the policy, the service or application won't let you perform that function. The application checks for access by asking the Policy Decision Point for a decision.

Policy Decision Point (PDP)
: This point is implemented in the centralized authorization system. The PDP takes the information provided by the application (such as identity and requested function), consults its policy, and gives the application its decision on whether access is granted for that particular function.

Policy Administration Point (PAP)
: This point is also implemented in the centralized authorization system. This is usually a web user interface and associated API where you can tell the centralized authorization system who's allowed to do what.

Most cloud providers have a centralized access management solution that their services will consult for access decisions, rather than making the decisions on their own. You should use these mechanisms where available, so that you can see all of the access granted to a particular administrator in one place.

Roles

Many cloud providers offer *roles*, which are similar to shared IDs in that you assume a role, perform actions that role allows, and drop the role. This is slightly different from the traditional implementation of a role, which is a set of permissions permanently granted to a user or group.

The primary difference between shared IDs and roles is that a shared ID is a standalone identity with fixed credentials. A cloud provider role is not a full identity; it is a special status taken on by another identity that is authorized to access a role, and is then assigned temporary credentials to access that role.

Role-based access can add an additional layer of security by requiring users or services to explicitly assume a separate role for more privileged operations, following the principle of least privilege. Most of the time the user can't perform those privileged activities unless they explicitly put on the role "hat" and take it off when they're done. The system can also log each request to take on a role, so administrators can later determine who had that role at a particular time and compare that information to actions on the system that have security consequences.

People aren't the only entities who can assume roles. Some components (such as virtual machines) can assume a role when created and perform actions using the privileges assigned to that role.

Roles Versus Groups

At some point many people ask, "What's the difference between a role and a group?" In their purest forms, these are the differences:

- A *group* is a collection of entities, such as users, without any information about what authorizations are granted to the entities in that group. The group *VMAdminGroup* might contain Chris and Barbara, but you don't know what they're allowed to do.
- A *role* is a collection of permissions that may be granted to users, groups, or other entities such as VMs. However, a "pure" role doesn't inherently contain any information about who those permissions are granted to. A role named *VMAdminRole* might grant you the permission to create and delete virtual machines, but the role definition doesn't tell you who actually gets those permissions. In some cases a role is permanently assigned to certain users or groups, and in some cases a user may be authorized to explicitly "assume" a role and drop that role when no longer needed.

In practice, many roles also specify the users (or groups) that they apply to, and in many cases group membership provides the group members with a single permanent set of permissions (a single role). The terms often tend to be used interchangeably, but with some cloud providers the distinction is important (such as with AWS IAM *Groups and Roles*).

Revalidate

At this point, your users and automation should have identities and be authorized to do only what they need to do. You need to make sure that this withstands the test of time.

As previously mentioned, the revalidation step is very important in both traditional and cloud environments, but in cloud environments you may not have any additional controls (such as physical building access or network controls) to save you if you forget to revoke access. You need to periodically check each authorization to ensure that it still needs to be there.

The first type of revalidation is automated revalidation based on certain parameters. For example, you should have a system that automatically puts in a request to revoke all access when someone leaves the organization. Note that simply deleting the user's identity may not be sufficient, because the user may have cached credentials such as access tokens that can be used even without the ability to log in. In situations like this, you need an "offboarding feed," which is a list of entities whose access should be

revoked. Any system that hands out longer-lived credentials such as access tokens must process this offboarding feed at least daily and revoke all access.

The second type of revalidation requires human judgment to determine whether a particular entity still needs access. There are generally two types of judgment-based revalidation:

Positive confirmation
: This is stronger—it means that access is lost unless someone explicitly says, "This access is still needed."

Negative confirmation
: This is weaker—it means that access is retained unless someone says, "This access is no longer needed."

Negative confirmation is appropriate for lower-impact authorization levels, but for types of access with high impact to the business, you should use positive confirmation. The drawbacks to positive confirmation are that it's more work, and access may be accidentally revoked if the request isn't processed in time (which may cause operational issues).

The largest risk addressed by revalidation is that someone who has left the organization (perhaps under contentious circumstances) retains access to systems. In addition to this, though, access tends to accumulate over time, like junk in the kitchen junk drawer (you know the one). Revalidation clears out this junk.

However, note that if it's difficult to get access, your users will often claim they still need access, even if they no longer do. Your revalidation efforts will be much more effective at pruning unnecessary access if you also have a fast, easy process for granting access when needed. If that's not possible, then it may be more effective to automatically revoke access if not used for a certain period of time instead of asking if it's still needed. This also has risks, because you may find nobody available has access when needed!

Cloud Identity-as-a-Service offerings are increasingly offering management of the entire identity life cycle in addition to authentication and authorization services. In other words, providers are recognizing the importance of the relationship's ending as well as the relationship's beginning, and they are helping to streamline and formalize endings.

Putting It All Together in the Sample Application

Remember our simple web application? Let's add identity and access management information to the diagram, which now looks like Figure 4-3. I've removed the whole application trust boundary to simplify the diagram.

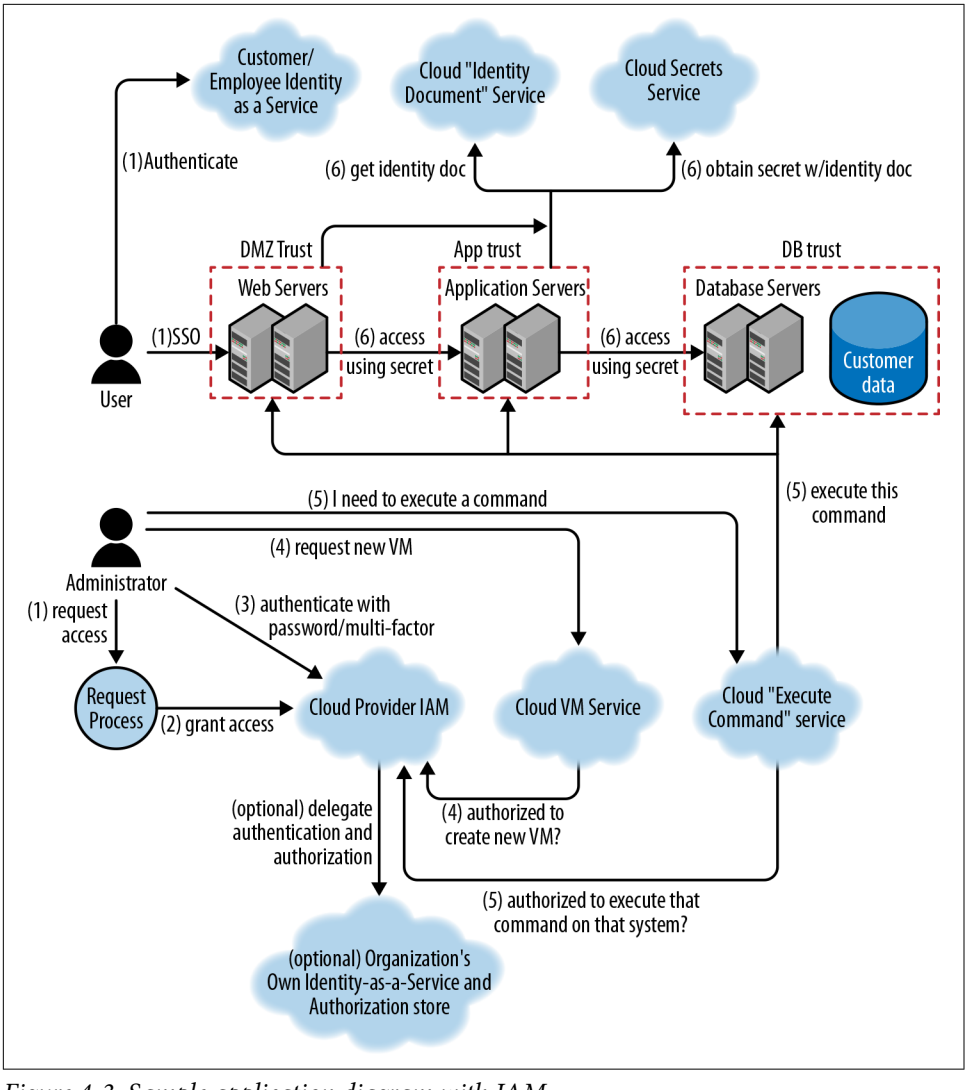


Figure 4-3. Sample application diagram with IAM

Unfortunately, that complicated the diagram quite a bit! Let's look at some of the new interactions in detail:

1. The end user attempts to access the application and is automatically approved for access by virtue of having a valid identity and optionally passing some anti-fraud tests. The end user logs in with SSO, so the application identity is federated with the user's external identity provider, and the application doesn't have to validate passwords. From the user's perspective, they're using the same identity as they do at their company or on their favorite social media site.

2. The administrator requests access to administer the application, which is approved. The administrator is then authorized in a centralized authorization system. The authorization may take place within the cloud's IAM system, or the cloud's IAM system may be configured to ask the organization's own internal authorization system to perform the authorization.
3. The administrator authenticates with the cloud IAM service using a strong password and multi-factor authentication and gets an access token to give to any other services. Again, optionally, the cloud IAM service may be configured to send the user to the organization's internal authentication system.
4. The administrator makes requests to cloud provider services, such as to create a new virtual machine or container. (Behind the scenes, the cloud VM service asks the cloud IAM service whether the administrator is authorized.)
5. The administrator uses a cloud provider service to execute commands on the virtual machines or containers as needed. (Behind the scenes, the cloud "execute command" service asks the cloud's IAM service whether the administrator is authorized to execute that command on that virtual machine or container.) If this feature isn't available from a particular cloud provider, the administrator might use a more traditional method, such as SSH, with the virtual machine using the LDAP protocol to authenticate and authorize administrators against an identity store. Note that in a container environment, executing commands may not even be needed for normal maintenance and upgrades, because the administrator can deploy a new container and delete the old one rather than making changes to the existing container.
6. A secrets service is used to hold the password or API key for the application server to access the database system. Figure 4-3 shows the application server getting an identity document from the cloud provider, accessing the secrets server directly to get the secret, and accessing the database. This is the "best" approach discussed earlier, but the secret might also be pushed in as part of the deployment process in a "good enough" approach. The same process could happen for the authentication between the web server and the application server, but only one secrets service interaction is shown for simplicity. The secrets service may be run by the organization, or may be an as-a-Service offering from a cloud provider.

Note that every time one of our application's trust boundaries is crossed, the entity crossing the trust boundary must be authenticated and authorized in order to perform an action. There are other trust boundaries outside the application that are not pictured, such as the trust boundaries around the cloud and organization systems.

Summary

You might have been somewhat lax about identity and access management in on-premises environments due to other mitigating factors, such as physical security and network controls, but IAM is supremely important in cloud environments. Although the concepts are similar in both cloud and on-premises deployments, there are new technologies and cloud services that improve security and make the job easier.

In the whole identity and access life cycle, it is easy to forget about the request, approval, and revalidation steps. Although they can be performed manually, many as-a-Service offerings that initially handled only the authentication and authorization steps now provide workflows for the approval steps as well, and this trend will likely accelerate.

Centralized authentication systems give administrators and end users a single identity to be used across many different applications and services. While these have been around in different forms for a long time, they are even more necessary in cloud environments, where they are available by default. Given the proliferation of cloud systems and services, managing identities individually for each system and service can quickly become a nightmare in all but the smallest deployments. Old, forgotten identities may be used by their former owners or by attackers looking for an easy way in. Even with centralized authentication, you must still use good passwords and multi-factor authentication. Cloud administrators and end users often authenticate via different systems.

As with the authentication systems, centralized authorization systems allow you to see and modify everything an entity is authorized to do in one place. This can make granting and revalidating access easier, and make separation of duties conflicts more obvious. Make sure you follow the principles of least privilege and separation of duties when authorizing both people and automation for tasks, and avoid having super-powered identities and credentials.

Secrets management is a quickly maturing field, where secrets used for system-to-system access are maintained separately from other configuration data and handled according to strict principles of confidentiality and auditing. Secrets management capabilities are available in existing configuration management products, standalone secrets server products, and as-a-Service cloud offerings.

CHAPTER 5
Vulnerability Management

In Greek mythology, Achilles was killed by an arrow to his only weak spot—his heel. Achilles clearly needed a better vulnerability management plan![1] Unlike Achilles, who had only one vulnerable area, your cloud environments will have many different areas where vulnerabilities can appear. After locking down access control, setting up a continuous process for managing potential vulnerabilities is usually the best investment in focus, time, and money that you can make to improve security.

There is considerable overlap between vulnerability management and patch management. For many organizations, the most important reason to install patches is to fix vulnerabilities rather than to fix functional bugs or add features. There is also considerable overlap between vulnerability management and configuration management, since incorrect configurations can often lead to vulnerabilities; even if you've dutifully installed all security patches. There are sometimes different tools and processes for managing vulnerabilities, configuration, and patches, but in the interests of practicality, we'll cover them all together in this chapter.

Unfortunately, vulnerability management is rarely as easy as turning on automatic patching and walking away. In cloud environments, vulnerabilities may be found in many different layers, including the physical facilities, the compute hardware, the operating system, code you've written, and libraries you've included. The cloud shared responsibility model described in Chapter 1 can help you understand where your cloud provider is responsible for vulnerabilities, and the contents of this chapter will help you manage your responsibilities. In most cases, you'll need several different tools and processes to deal with different types of vulnerabilities.

1 Perhaps one that included wearing boots.

Vulnerability Versus Patch Management

The terms "vulnerability management" and "patch management" are often used interchangeably, but they are different. Software patches often fix functional issues in addition to security vulnerabilities, and not all vulnerabilities are fixed by applying patches. For example, your vulnerability management process might identify insecure configurations that are fixed without patching, or it might mitigate a vulnerability by turning off a feature rather than applying a patch.

Differences from Traditional IT

The rate of change is often much higher in cloud environments compared to on-premises, and these constant changes can leave traditional vulnerability management processes in the dust. As discussed in Chapter 3, you must use inventory from cloud APIs to feed each system into your vulnerability management tools as it is created, to avoid missing new systems as they come online.

In addition to the rate of change, popular contemporary hosting models such as containers and serverless hosts change the way that you do vulnerability management, because existing tools either aren't applicable or aren't efficient. You cannot put a heavyweight vulnerability management tool that uses a few percent of your CPU in every container, like you would in virtual machines. You'd likely end up running hundreds of copies of the agent on the system and have no CPU time left for the real work!

Plus, even though continuous integration (CI), continuous delivery (CD), and microservice architectures are separate from cloud computing, they often happen along with cloud adoption. Adoption of these techniques can also radically change vulnerability management.

For example, a traditional vulnerability management process might look something like this:

1. *Discover* that security updates or configuration changes are available.
2. *Prioritize* which updates need to be implemented based on the risk of security incidents.
3. *Test* that the updates work, in a test environment.
4. *Schedule* the updates for a production environment.
5. *Deploy* the updates to production.
6. *Verify* that production still works.

This type of process is reasonably designed to balance the risk of a security incident against the risk of an availability incident in production environments. As I often like to tell people, security is easy—just turn everything off and bury it in concrete. Securing environments while keeping them running and usable is much more difficult.

However, in our brave new world of cloud computing, infrastructure as code, CI/CD, and microservice architectures, we have options for reducing the risk of an availability incident and changing the balance:

- Cloud offerings and infrastructure as code allow the definition of the environment to be part of the code. This allows a new environment and new code to be tested together, rather than combining the environment and the code at the end when you install on an existing machine. In addition, because you can create a new production environment for each deployment and switch back to (or recreate) the old one easily if needed, you can reduce the risk of getting into a state where you cannot roll back quickly. This is similar to "blue/green" deployments in traditional environments, but with the cloud you don't need to pay for the "green" environment all the time, so infrastructure as code can be used even for smaller, lower-budget applications.
- Continuous integration and continuous delivery allow smaller changes to be deployed to production on each iteration. Smaller changes reduce the risk of catastrophic failures and make troubleshooting easier for problems that do arise.
- Microservice architectures can decouple services, so that changes in one microservice are less likely to have undesired side effects in other microservices. This is especially true in container-based microservice environments, because each container is isolated from the others.
- Microservice architectures also tend to scale horizontally, where the application is deployed across more machines and containers as needed to handle the load. This also means that changes can be rolled out in phases across the environment, and potentially disruptive scans[2] will take down only some of the capacity of the application.

Each of these items swings the balance toward higher availability, which means that security updates can be more proactive without lowering the overall availability of the system. This in turn reduces your overall risk. The new vulnerability management process looks like this:

2 One of the barriers to vulnerability scanning is that if you actually find a vulnerability, sometimes the scan will crash the affected component. Sure, you found a problem, but at the cost of incurring downtime! The risk of an outage is much lower if the scan can only crash one of the instances of the application at a time.

1. *Automatically pull* available security updates as part of normal development efforts. For example, this might include updated code libraries or updated operating system components.
2. *Test* the updates as part of the normal application test flow for a deployment. Only if you find a problem at this stage do you need to step back to evaluate whether the updates need to be included.
3. *Deploy* the new version, which automatically creates a new production environment that includes code changes, security updates, and potentially updates to the configuration. This deployment could be to a subset of systems in production, if you are not confident that it won't disrupt operation.
4. *Discover* and address any additional vulnerabilities in test or production environments that aren't covered as part of the normal delivery process, add them as bugs in the development backlog, and address them in the next iteration (or as a special release if urgent).

You still have some manual vulnerability management work to do in step 4, but far less than in the standard process. As we'll see in this chapter, there are many types of vulnerabilities, but this high-level process will work for most.

Vulnerable Areas

What types of vulnerabilities do you have to worry about? Imagine that your application is part of a stack of components, with the application on top and physical computers and facilities at the bottom. We'll start at the top of the stack and work downwards. There are many different ways to categorize the items in the stack, but we'll use the shared responsibility model diagram from Chapter 1 (see Figure 5-1).

Let's look at each layer of this diagram in more detail from the perspective of vulnerability management, starting at the top.

Data Access

Deciding how to grant access to the data in the application or service is almost always the customer's responsibility in a cloud environment. Vulnerabilities at the data access layer almost always boil down to access management problems, such as leaving resources open to the public, leaving access intact for individuals who no longer need it, or using poor credentials. These issues were discussed in detail in Chapter 4.

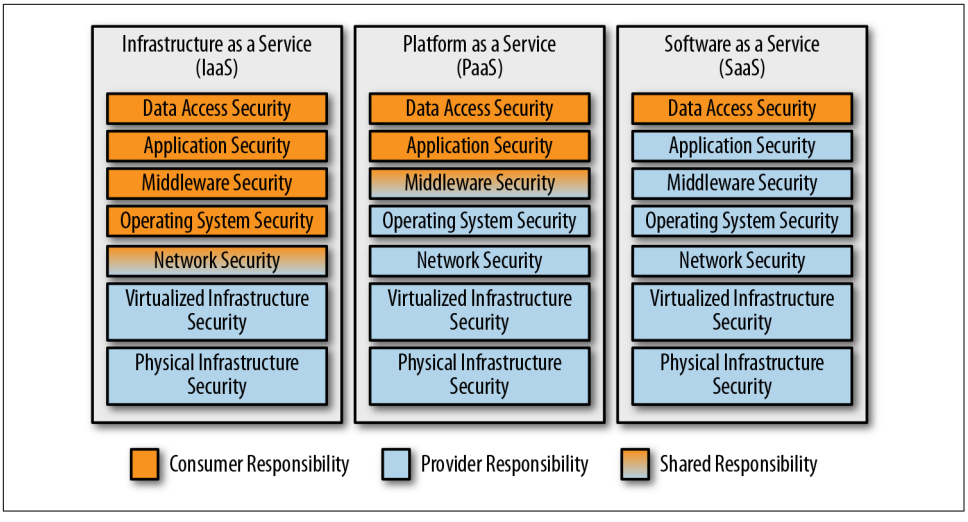


Figure 5-1. Cloud shared responsibility model

Application

If you're using SaaS, the security of the application code will be your provider's responsibility, but there may be security-relevant configuration items that you're responsible for as a customer. For example, if you're using a web email system, it will be up to you to determine and set reasonable configurations such as two-factor authentication or malware scanning. You also need to track and correct these configurations if they drift from your requirements.

If you're not using SaaS, you are probably writing some sort of application code, whether it's hosted on virtual machines, an aPaaS, or a serverless offering. No matter how good your team is, your code is almost certainly going to have some bugs, and at least some of those bugs are likely to impact security. In addition to your own code, you're often going to be using frameworks, libraries, and other code provided by third parties that may contain vulnerabilities. Vulnerabilities in this inherited code are often more likely to be exploited by attackers, because the same basic attack will work across many applications.

Vulnerabilities in popular open source components, such as Apache Struts and OpenSSL, have led to vulnerabilities in many applications that use those components. Exploiting these vulnerabilities is much easier for attackers than researching specific application code, so they tend to be an even higher risk than vulnerabilities in code you've written!

The classic example of an application vulnerability is a buffer overflow. However, many applications are now written in languages that make buffer overflows difficult, so while these attacks still happen, they don't make the top of the list any more. Following are a few examples of application vulnerabilities from the OWASP Top 10 list for 2017. In each of these examples, access controls, firewalls, and other security measures are largely ineffective in protecting the system if these vulnerabilities are present in the application code:

Injection attacks
: Your application gets a piece of untrusted data from a malicious user and sends it to some sort of interpreter. A classic example is SQL injection, where the attacker sends information that causes the query to return everything in the table instead of what was intended.

XML external entity attacks
: An attacker sends XML data that one of your vulnerable libraries processes and that performs undesirable actions.

Cross-site scripting attacks
: An attacker fools your application into sending malicious JavaScript to a user.

Deserialization attacks
: An attacker sends "packed" objects to your application that cause undesirable side effects when unpacked.

Note that all of these application-level attacks are possible regardless of how your application is deployed—on a virtual machine, on an aPaaS, or on a serverless platform. Some tools discussed in Chapter 6, such as web application firewalls, may be able to act as a safety net if there is a vulnerability in application code. However, make no mistake—detecting and fixing vulnerable code and dependencies is your first and most important line of defense.

Although frameworks can be a source of vulnerabilities you have to manage, they can also help you avoid vulnerabilities in your own code. Many frameworks have built-in protections against cross-site scripting (XSS), cross-site request forgery (CSRF), SQL injection (SQLi), clickjacking, and other types of attacks. Understanding the protections offered by your framework and using them can easily enable you to avoid some of these issues.

Middleware

In many cases, your application code uses middleware or platform components, such as databases, application servers, or message queues. Just as with dependent frameworks or libraries, vulnerabilities here can cause you big problems because they're

attractive to attackers—the attacker can exploit that same vulnerability across many different applications, often without having to understand the applications at all.

If you're running these components yourself, you'll need to watch for updates, test them, and apply them. These components might be running directly on your virtual machines, or might be inside containers you've deployed. Note that tools that work for inventorying what's installed on virtual machines will usually not find items installed in containers.

If these components are provided as a service by your cloud provider, your provider will usually have the responsibility for patching. However, there's a catch! In some cases, the updates won't be pushed to you automatically, because they could cause an outage. In those cases, you may still be responsible for testing and then pushing the button to deploy the updates at a convenient time.

In addition to applying patches, you also need to worry about how middleware is configured, even in a PaaS environment. Here are some real-world examples of middleware/platform configuration issues that can lead to a security incident or breach:

- A web server is accidentally configured to allow viewing of the password file.
- A database is not configured for the correct type of authentication, allowing anyone to act as a database manager.
- A Java application server is configured to provide debug output, which reveals a password when a bug is encountered.

For each component you use, you need to examine the configuration settings available and make a list of security-relevant settings and what the correct values are. These should be enforced when the component is initially brought into service and then checked regularly afterward to make sure they're all still set correctly and prevent "configuration drift." This kind of manual monitoring is often called *benchmarking*, *health checking*, or simply *configuration management*.

While you can certainly write benchmarks or configuration specifications from scratch, I recommend starting with a common set of best practices, such as the Center for Internet Security's CIS Benchmarks. You can tailor these for your organization and deployments, and even contribute a change if you find a problem or want to suggest an enhancement. Because the benchmarks are a community-based effort, you're more likely to benefit from up-to-date configuration checks that take into account new threats and new versions of platform products and operating systems. Several popular products can perform the CIS Benchmarks checks out of the box.

Operating System

Operating system patches are what many people think of when they think of vulnerability management. It's Patch Tuesday, time to test the patches and roll them out! But while operating system patches are an important part of vulnerability management, they're not the only consideration.

Just as with the middleware/platform layer of the stack, you must perform proper benchmarking when deploying the operating system instance and then regularly afterward. In addition, operating systems tend to ship with a lot of different components that are not needed in your environment. Leaving these components in a running instance can be a big source of vulnerabilities, either from bugs or misconfiguration, so it's important to turn off anything that's not needed. This is often referred to as *hardening*.

Many cloud providers have a catalog of virtual machine images that are automatically kept up to date, so that you should get a reasonably up-to-date system when deploying. However, if the cloud provider doesn't automatically apply patches upon deployment, you should do so as part of your deployment process.

An operating system typically consists of a *kernel*, which runs all other programs, along with many different userspace programs. Many containers also contain the userspace portions of the operating system, and so operating system vulnerability management and configuration management also factor into container security.

In most cases, the cloud provider is responsible for the hypervisors. However, if you're responsible for any hypervisors, they're also included in this category because they're essentially special-purpose operating systems designed to hold other operating systems. Hypervisors are typically already hardened, but do still require regular patching and have configuration settings that need to be set correctly for your environment.

Network

Vulnerability management at the network layer involves two main tasks: managing the network components themselves and managing which network communications are allowed.

The network components themselves, such as routers, firewalls, and switches, typically require patch management and security configuration management similar to operating systems, but often through different tools.

Managing the security of the network flows implemented by those devices is discussed in detail in Chapter 6.

Virtualized Infrastructure

In an Infrastructure-as-a-Service environment, the virtualized infrastructure (virtual network, virtual machines, storage) will be the responsibility of your cloud provider. However, in a container-based environment, you may have security responsibility for the virtualized infrastructure or platform on top of the one offered by the cloud provider. For example, vulnerabilities may be caused by misconfiguration or missing patches of the container runtime, such as Docker, or the orchestration layer, such as Kubernetes.

Physical Infrastructure

In most cases, physical infrastructure will be the responsibility of your cloud provider.

There are a few cases where you may be responsible for configuration or vulnerability management at the physical level, however. If you are running a private cloud, or if you get bare-metal systems provisioned as a service, you may have some physical infrastructure responsibilities. For example, vulnerabilities can be caused by missing BIOS/microcode updates or poor security configuration of the baseboard management controller that allows remote management of the physical system.

Finding and Fixing Vulnerabilities

Now that you're armed with an understanding of all of the places vulnerabilities might be hiding, you need to prioritize which types of vulnerabilities are most likely to be a problem in your environment. As I've repeated several times in this book, go for the biggest bang for the buck first—pick the most important area for your organization, and get value from it before moving on to other areas. A common pitfall is having four or five different sets of tools and processes in order to check off a box on a list of best practices somewhere, none of which are actually providing a lot of value in finding and fixing vulnerabilities.

If you recall the asset management pipeline discussed in Chapter 3, this is the part where we put our fancy tools into the pipeline (Figure 5-2) to make sure we know about and deal appropriately with our risks. In Chapter 3, we were concerned with the left half of the diagram—watching procurement to find out about shadow IT and making sure we inventoried the assets from all the different cloud providers.

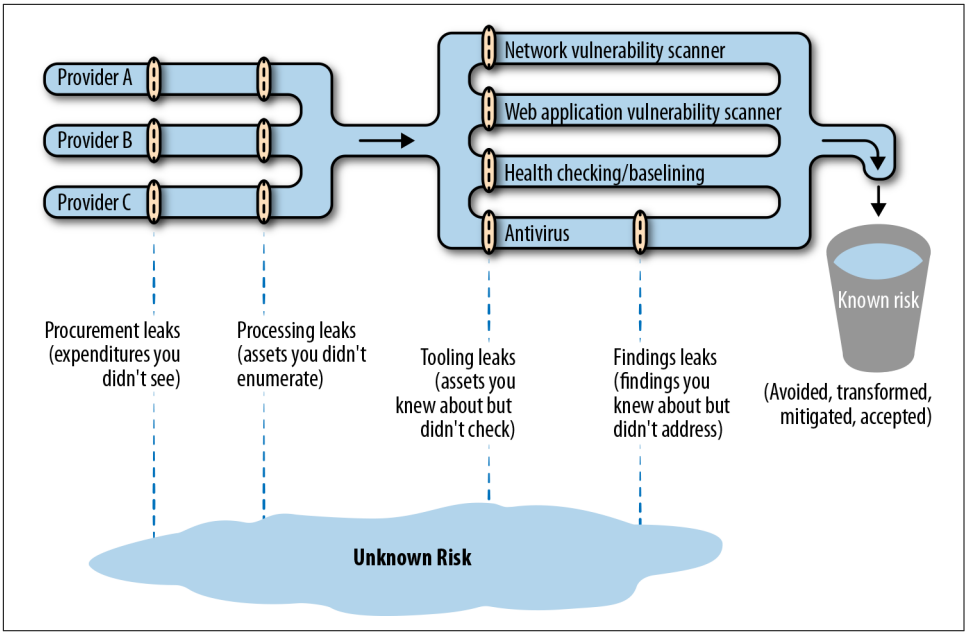


Figure 5-2. Sample asset management pipeline

Here, the goal is to plug the leaks shown on the right half of the diagram. For example, here's where we can minimize our "tooling" leaks (which result from not protecting known assets) as well as our "findings" leaks (which result from not properly dealing with known findings).

First, look at the tooling leaks area of the figure. Imagine the sizes of the pipes in your environment as being determined by a combination of how many problems you might find in these areas, as well as how critical to the business those problems might be. I've found that when I imagine this, I sometimes realize that there is a lot of water gushing out in a particular area, either because there's no tool in that area or because the tool doesn't have visibility to a lot of assets. This can lead to a lot of unknown risk!

For example, if your environment contains a lot of Windows systems with critical data, fixing leaks in your antivirus pipeline might be near the top of your list. On the other hand, if you have mostly web applications running on Linux, aPaaS, or serverless, you probably want to focus on making sure you find and remediate web application vulnerabilities first before worrying too much about a small number of Windows systems that have less critical data.

Next, look at the findings leaks area of the figure. Imagine that the size of this pipe is determined by the number of findings coming out of your tool and how critical those findings might be. You may realize that you've got tools that you're ignoring a lot of important output from, and you're therefore creating a lot of unknown risk.

There are many, many different types of tools, which overlap a lot in the vulnerabilities they search for. Some of the tools have been used in traditional environments for years, and others are newly introduced by cloud environments. Explanations follow of the different categories of vulnerability and configuration management tools, but note that many products will address more than one of these categories.

Network Vulnerability Scanners

In addition to operating system patches, *network vulnerability scans* are the other best-known piece of vulnerability management. This is for a good reason—they're very good at finding some types of vulnerabilities—but it's important to understand their limitations.

Network vulnerability scanners don't look at software components. They simply make network requests, try to figure out what's listening, and check for vulnerable versions of server applications or vulnerable configurations. As an example, a network vulnerability scanner can determine that one of the services on the system is allowing insecure connections, which would make the system vulnerable to a POODLE attack, based on the information in an SSL/TLS handshake. The scanner can't know, however, about the different web applications or REST APIs served up on that network address, nor can it see components such as library versions inside the system.

Obviously, network vulnerability scanners cannot scan the entire internet, or your entire cloud provider, and magically know which systems are your responsibility. You have to provide these tools with lists of network addresses to scan, and if you've missed any addresses, you're going to have vulnerabilities you don't know about. This is where the automated inventory management discussed in Chapter 3 is vital. Because many cloud components are open to the internet, and because attackers can exploit vulnerabilities that they discover in common components very quickly, your cycle time for inventorying internet-facing components, scanning them, and fixing any findings needs to be as fast as possible.

In addition, don't make the mistake of thinking network vulnerability scans are unnecessary just because you have isolated components, which will be described in Chapter 6. There is often a debate between network teams and vulnerability scanner teams on whether to poke holes in the firewall to allow the vulnerability scanner into a restricted area. I maintain that the risk of having an unknown risk is much higher than the risk that an attacker will leverage those specific firewall rules to get into the restricted area, so vulnerability scanners should be allowed to scan every component, even if it means weakening the perimeter network controls slightly. I have seen many incidents where the attacker got behind the perimeter and exploited a vulnerable system there. In contrast, although it has probably happened somewhere, I have not per-

sonally seen or heard of any incidents where the attacker took over the scanner and used its network access to attack systems.

Network vulnerabilities found on a segment of a protected virtual private cloud network have a lower priority than vulnerabilities on a component directly exposed to the internet, but you should still discover them and fix them. Attackers have a very inconvenient habit of ending up in parts of the network where they're not supposed to be.

Depending on how your deployment pipeline works, you should incorporate a network vulnerability scan of the test environment into the deployment process where possible. Any findings in the test environment should feed into a bug tracker, and if not marked as a false positives, they should ideally block the deployment.

There are several cloud-based network vulnerability scanners that you can purchase and run as a service, without purchasing any infrastructure. However, you may need to create relay systems or containers inside your network for scanning areas that are not open to the internet.

Network-based tools can find vulnerabilities without knowing what processes they're talking to; they just see what answers on different TCP/UDP ports on a given IP address. They're very useful because they see the same things an external attacker will see. However, this can also generate false positives, because the tool will often use the reported version of a component, which may not be correct or may not indicate that security patches have been installed. You must have a well-documented, effective process for masking false positives, or you run the risk of teams ignoring all of the scan results because some of them are incorrect.

Agentless Scanners and Configuration Management

If network vulnerability scans bang on the doors and windows of the house, *agentless scanners* and *configuration management systems* come inside the house and poke around. Agentless scanners also connect over the network, but use credentials to get into the systems being tested. In some cases, the same tools may perform both network scans and agentless scans. (The term "agentless" distinguishes these scanners from the ones described in the next section, which require an "agent" to run on each target system.)

Agentless scanners can find vulnerabilities that network vulnerability scanners can't. For example, if you have a local privilege escalation vulnerability, which allows a normal user to take over the entire system, a network vulnerability scanner doesn't have "normal user" privileges in order to see it, but an agentless scanner does.

Agentless scanners often perform both missing patch detection and security configuration management, as the following examples show:

- The agentless scanner may run package manager commands to check that installed software is up to date and has important security fixes. For instance, some versions of the Linux kernel or C libraries have problems that allow someone without *root* privileges to become *root*; these problems can be detected by up-to-date scanners.
- The agentless scanner may check that security configurations are correct and meet policy requirements. For example, the system may be configured to allow Telnet connections (which could allow someone snooping on the network to see passwords, and therefore should be prohibited by policy); the scanner should detect that Telnet is enabled and flag an alert.

In some cases, these tools can actually fix misconfigurations or vulnerable packages in addition to just detecting the problems. But as mentioned earlier, such automated fixes can disrupt availability if they introduce new problems or don't match your environment's requirements. Where possible, it's preferable to roll out an entirely new system that doesn't have the vulnerability rather than trying to fix it in place.

With all of this capability, why would you need both an agentless scanner and a network vulnerability scanner? Although there's a lot of overlap, agentless scanners fundamentally have to understand the system they're looking at, which means that they don't function well on operating system versions, software, or other items they don't recognize. The fact that network vulnerability scans are "dumber" and only bang on network addresses is actually a strength in some cases, because they can find issues with anything on the network—even devices that allow no logins, such as network appliances, IoT devices, or containers.

Agent-Based Scanners and Configuration Management

Agent-based scanners and configuration management systems generally perform the same types of checks as agentless scanners. However, rather than having a central "pull" model, where a controller system reaches out to each system to be scanned and pulls the results in, agent-based scanners install a small component on each system—the agent—that "pushes" results to the controller.

There are both benefits and drawbacks to this approach, described in the following subsections.

Credentials

Agent-based scanners eliminate one source of risk inherent to agentless scanners. The agentless scanner consoles must have credentials to all systems—and usually privi-

leged credentials—in order to perform their scans. Although the risk of granting those credentials is generally much less than the risk of unknown vulnerabilities in your environment, it does make the agentless scanner console a really attractive target for attackers. In contrast, agent-based scanners require privileges to deploy initially, but the scanner console just receives reports from the agents and has only whatever privileges the agent permits the console to use (which may still be full privileged access).

Deployment

Agents have to be deployed and kept up to date, and a vulnerability in the agent can put your entire infrastructure at risk. However, a well-designed agent in a "read-only" mode may be able to mitigate much of the risk of an attacker taking over the scanning console; the attacker will get a wealth of vulnerability information but may not get privileged access on all systems.

Agentless scanners don't require you to deploy any code, but you often have to configure the target systems in order to provide access to the scanner. For example, you may need to create a userID and provide that userID with a certain level of *sudo* access.

Network

Agentless scanners must have inbound network access in order to work. As previously mentioned, allowing this network access can increase the risk to your environment. Most tools also have the option of deploying a relay system inside your network that makes an outbound connection and allows control via that connection, but the relay system is another system that requires management.

Agent-based systems can make only outbound connections, without allowing any inbound connections.

Some tools can perform checks using either an agent model or an agentless model. Ultimately, there's no right answer for all deployments, but it's important to understand the benefits and drawbacks of each when making a decision. I typically favor an agent-based model, but there are good arguments for both sides, and the most important thing is that you address configuration and vulnerability management.

Several cloud providers offer agent-based scanners in their support for your cloud environment. These can be simpler to automatically deploy, and you don't have to manually pull a list of assets from your cloud provider and feed them into the scanner.

Cloud Provider Security Management Tools

Tools in this category are typically specific to a particular cloud provider. They usually either gather configuration and vulnerability management information via agents or agentless methods, or pull in that information from a third-party tool. They're typically marketed as a "one-stop dashboard" for multiple security functions on the provider, including access management, configuration management, and vulnerability management.

These tools may also offer the ability to manage infrastructure or applications not hosted by the cloud provider—either on-premises or hosted by a different cloud provider—as an incentive to use the tool for your entire infrastructure.

Container Scanners

Traditional agent and agentless scans work well for virtual machines, but often don't work well in container environments. Containers are intended to be very lightweight processes, and deploying an agent designed for a virtual machine environment with each container can lead to crippling performance and scalability issues. Also, if used correctly, containers usually don't allow a traditional network login, meaning that agentless scanners designed for virtual machines will also fail.

This is still a relatively new area, but two approaches are popular as of this writing. The first approach is to use scanners that pull apart the container images and look through them for vulnerabilities. If an image is rated as vulnerable, you know to avoid deploying new containers based on it and to replace any existing containers deployed from it. This has the benefit of not requiring any access to the production systems, but the drawback is that once you identify a vulnerable image, you must have good enough inventory information about all of your running containers to ensure you replace all of the vulnerable ones.

In addition, if your containers are mutable (change over time), additional vulnerabilities may have been introduced that scanning the source image won't reveal. For this reason and others, I recommend the use of immutable containers that are replaced by a new container whenever any change is needed. Regularly replacing containers can also help keep threat actors from persisting in your network, because even if they compromise a container, it will be wiped out in a week or so—and the new container will hopefully have a fix for the issue that led to the compromise.

The second approach is to concentrate on the running containers, using an agent on each container host that scans the containers on that system and reports which containers are vulnerable so that they may be fixed (or preferably, replaced). The benefit is that, if the agent is deployed everywhere, you cannot end up with "forgotten" containers that are still running a vulnerable image after you have created a new image with the fix. The primary downside, of course, is that you have to have an agent on

each host. This can potentially be a performance concern, and may not be supported by your provider if you're using a Container-as-a-Service offering.

These approaches are not mutually exclusive, and some tools use both. If you're using containers, or planning to use containers soon, make sure you have a way to scan for vulnerabilities in the images and/or running containers and feed the results into an issue tracking system.

Dynamic Application Scanners (DAST)

Network vulnerability scanners run against network addresses, but *dynamic web application vulnerability scanners* run against specific URLs of running web applications or REST APIs. Dynamic application security testing (DAST) tools can find issues such as cross-site scripting or SQL injection vulnerabilities by using the application or API like a user would. These scanners often require application credentials.

Some of the vulnerabilities found by dynamic scanners can also be blocked by web application firewalls (WAFs), as discussed in Chapter 6. That may allow you to put a lower priority on fixing the issues, but you should fix them fairly quickly anyway to offer security in depth. If the application systems aren't configured properly, an attacker might bypass the WAF and attack the application directly.

Dynamic scanners can generally be invoked automatically on a schedule and when changes are made to the application, and they feed their results into an issue tracking system.

Static Application Scanners (SAST)

Where dynamic application scanners look at the running application, *static application security testing* (SAST) tools look directly at the code you've written. For this reason, they're a good candidate for running as part of the deployment pipeline as soon as new code is committed, to provide immediate feedback. They can spot security-relevant errors such as memory leaks or off-by-one errors that can be very difficult for humans to see. Because they're analyzing the source code, you must use a scanner designed for the language that you're using. Luckily, scanners have been developed for a wide range of popular languages, and can be run as a service. One example is the SWAMP project, supported by the US Department of Homeland Security.

The biggest problem with static scanners is that they tend to have a high false positive rate, which can lead to "security fatigue" in developers. If you deploy static code scanning as part of your deployment pipeline, make sure that it will work with the languages you're using and that you can quickly and easily mask false positives.

Software Composition Analysis Scanners (SCA)

Arguably an extension of static code scanners, *software composition analysis* (SCA) tools look primarily at the open source dependencies that you use rather than the code you've written. Most applications today make heavy use of open source components such as frameworks and libraries, and vulnerabilities in those can cause big problems. SCA tools automatically identify the open source components and versions you are using, then cross-reference against known vulnerabilities for those versions. Some can automatically propose code changes that use newer versions. Also, in addition to vulnerability management, some products can look at the licenses the open source components are using to ensure that you don't use components with unfavorable licensing.

SCA tools have helped mitigate some of the higher-impact vulnerabilities in the past few years, such as those found in Apache Struts and the Spring Development Framework.

Interactive Application Scanners (IAST)

Interactive application security testing (IAST) tools do a little bit of both static scanning and dynamic scanning. They see what the code looks like and watch it from the inside while it runs. This is done by loading the IAST code alongside the application code to watch while the application is exercised by functional tests, a dynamic scanner, or real users. IAST solutions can often be more effective at finding problems and eliminating false positives than either SAST or DAST solutions.

Just like with static code scanners, the specific language and runtime you're using must be supported by the tool. Because this is running along with the application, it can decrease performance in production environments, although with modern application architectures this can usually be mitigated easily with horizontal scaling.

Runtime Application Self-Protection Scanners (RASP)

Although *runtime application self-protection* (RASP) sounds similar to the scanners described previously, it is not a scanning technology. RASP works similarly to IAST in that it is an agent deployed alongside your application code, but RASP tools are designed to block attacks rather than just detect vulnerabilities (several products do both—detect vulnerabilities and block attacks—making them both RASP and IAST products). Just as with IAST products, RASP products can degrade performance in some cases because more code is running in the production environment.

RASP solutions offer some of the same protection as a distributed WAF, because both block attacks in production environments. For this reason, RASP and WAF solutions are discussed in Chapter 6.

Manual Code Reviews

Manual code reviews can be expensive and time-consuming, but they can be better than application testing tools for finding many types of vulnerabilities. In addition, having another person explain why a particular piece of code has a vulnerability can be a more effective way to learn than trying to understand the results from automated tools.

Code reviews are standard practice in many high-security environments. In many other environments, they may be used only for sections of code with special significance to security, such as sections implementing encryption or access control.

Penetration Tests

A *penetration test* (pentest) is performed by someone you've engaged to try to get unauthorized access to your systems and tell you where the vulnerabilities are. It's important to note that automated scans of the types discussed earlier are *not* penetration tests, although those scans may be used as a starting point for a pentester. Larger organizations may have pentesters on staff, but many organizations contract with an external supplier.

Penetration tests by an independent third party are required by PCI DSS and FedRAMP moderate/high standards, and they may be required for other attestations or certifications.

There are some disagreements on terminology, but typically, in *white box pentesting* you provide the pentester with information about the design of the system, but not usually any secret information such as passwords or API keys. In some cases you may also provide more initial access than an outside attacker would start with, either for testing the system's strength against a malicious insider or for seeing what would happen if an attacker found vulnerabilities in the outer defenses. In *black box pentesting*, you point the pentester at the application without any other information. An intermediate approach is _gray box pentesting), where limited information is available.

White box pentesting and gray box pentesting is often more effective and a better use of time than black box pentesting, because the pentesters spend less time on reconnaissance and more time on finding actual vulnerabilities. Remember that the real attackers will usually have more time than your pentesters do!

It's important to note that a pentester will typically find one or two ways into the system, but not *all* the ways. A pentest with negative or minimal findings gives you some confidence in the security of your environment. However, if you have a major finding and you fix that particular vulnerability, you need to keep retesting until you come

back with acceptable results. Pentesting is typically an expensive way to find vulnerabilities, so if the pentesters are coming back with results that an automated scan could have found, you're probably wasting money. Pentesting is often done near the end of the release cycle, which means that problems found during pentesting are more likely to make a release late.

Automated testing often finds potential vulnerabilities, but penetration testing (when done correctly) shows actual, successful exploitation of vulnerabilities in the system. Because of this, you usually want to prioritize fixing pentest results above other findings.

Most cloud service providers require you to get approval prior to conducting penetration tests of applications hosted on their infrastructure or platform. Failure to get approval can be a violation of the provider's terms of service and may cause an outage, depending on the provider's response to the intrusion.

User Reports

In a perfect world, all bugs and vulnerabilities would be discovered and fixed before users see them. Now that you've stopped laughing, you need to consider that you may get reports of security vulnerabilities from your users or through bug bounty programs.

You need to have a well-defined process to quickly verify whether the reported vulnerability is real or not, roll out the fix, and communicate to the users. In the case of a bug bounty program, you may have a limited amount of time before the vulnerability is made public, after which the risk of a successful attack increases sharply.

User reports overlap somewhat with incident management processes. If your security leaders are not comfortable dealing with end users, public relations, or legal issues, you may also need to have someone who specializes in communications and/or a lawyer to assist the security team in avoiding a public relations or legal nightmare. Often, a poor response to a reported vulnerability or breach can be much more damaging to an organization's reputation than the initial problem!

Example Tools for Vulnerability and Configuration Management

Most of the tools listed in the previous sections can be integrated into cloud environments, and most cloud providers have partnerships with vendors or their own proprietary vulnerability management tools.

Because so many tools address more than one area, it doesn't make sense to categorize them into the areas listed earlier. I've put together a list of some representative solutions in the cloud vulnerability and configuration management space, with a very

brief explanation of each. Some of these tools also overlap with detection and response (Chapter 7), access management (Chapter 4), inventory and asset management (Chapter 3), or data asset management (Chapter 2).

I'm not endorsing any of these tools by including them, or snubbing other tools by excluding them; these are just some examples so that when you get past the initial marketing blitz by the vendor, you can realize, "Oh, this tool claims to cover areas x, y, and z." I've included some tools that fit neatly into a single category, some tools that cover many different categories, and some tools that are specific to popular cloud providers. This is a quickly changing space, and different projects and vendors are constantly popping up or adding new capabilities.

Here's the list of tools, in alphabetical order:

- Amazon Inspector is an agent-based scanner that can scan for missing patches and poor configurations on Linux and Windows systems.
- Ansible is an agentless automation engine that can be used for almost any task, including configuration management.
- AWS Config checks the detailed configurations of your AWS resources and keeps historical records of those configurations. For example, you can check that all of your security groups restrict SSH access, that all of your Electric Block Store (EBS) volumes are encrypted, and that all of your Relational Database Service (RDS) instances are encrypted.
- AWS Systems Manager (SSM) is a security management tool that covers many areas, including inventory, configuration management, and patch management. The State Manager component can be used to enforce configurations, and the Patch Manager component can be used to install patches; both of these functions are executed by an SSM agent installed on your instances.
- AWS Trusted Advisor performs checks on several areas such as cost, performance, fault tolerance, and security. In the area of configuration management for AWS resources, Trusted Advisor can perform some high-level checks, such as whether a proper IAM password policy is in place or CloudTrail logging is enabled.
- Azure Security Center is a security management tool that can integrate with partners such as Qualys and Rapid7 to pull in vulnerability information from those agents and consoles.
- Azure Update Management is agent-based and primarily aimed at managing operating system security patches, but it can also perform software inventory and configuration management functions.
- Burp Suite is a dynamic web application scanning suite.

- Chef is an agent-based automation tool that can be used for configuration management, and the InSpec project specifically targets configuration related to security and compliance.
- Contrast provides IAST and RASP solutions.
- Google Cloud Security Command Center is a security management tool that can pull in information from the Google Cloud Security Scanner and other third-party tools, and also provide inventory management functions and network anomaly detection.
- Google Cloud Security Scanner is a DAST tool for applications hosted on Google App Engine.
- IBM Application Security on Cloud is a SaaS solution that uses several IBM and partner products and provides IAST, SAST, DAST, and SCA.
- IBM BigFix is an agent-based automation tool that can be used for configuration and patch management.
- IBM Security Advisor is a security management tool that can pull in vulnerabilities from IBM Vulnerability Advisor as well as network anomaly information.
- IBM Vulnerability Advisor scans container images and running instances.
- Puppet is an agent-based automation tool that can be used for configuration management.
- Qualys has products that cover many of the categories we've discussed, including network vulnerability scanning, dynamic web application scanning, and others.
- Tenable has a range of products including the Nessus network scanner, agent-based and agentless Nessus patch and configuration management scanners, and a container scanner.
- Twistlock can perform configuration and vulnerability management on container images, running containers, and the hosts where the containers run.
- WhiteSource is an SCA solution.

Statistically speaking, people are terrible at statistics. When you evaluate marketing claims, it's important to use tools that have both reasonable false positive *and* false negative rates. As an extreme example, if a tool flags everything as a problem, it will catch every one of the real problems (100% true positive), but the false positive rate will be so high that it's useless. Similarly, if the tool flags nothing as a problem, its false positive rate is perfect (0%), but it has missed everything. Beware of marketing claims that focus on only one side of the equation!

Risk Management Processes

At this point in the process you should understand where the most vulnerable areas are in your environment and which tools and processes you can use to find and fix vulnerabilities. Now you need a system to prioritize any vulnerabilities that can't be fixed quickly, where "quickly" is usually defined in relation to time periods in your security policy.

This is where a risk management program comes in, near the end of the pipeline shown in Figure 3-2. Each vulnerability you find that can't be addressed within your accepted guidelines needs to be evaluated as a risk, so that you consciously understand the likelihood of something bad happening and the impact if it does. In many cases, you might accept the risk as a cost of doing business. However, the risk evaluation might lead to mitigation strategies, such as putting in some extra detection or prevention tools or processes. Risk evaluation might also lead to avoidance, such as turning off the system entirely in some cases.

A leak in the pipeline here means you found the vulnerabilities but couldn't fix them right away, and you also failed to actually understand how bad they could be for your business. Using an existing framework for evaluating risk, such as NIST 800-30 or ISO 31000, can be much easier than starting from scratch.

You don't need a really complicated risk management program to get a lot of value; a simple risk register with an agreed-upon process for assigning severity to the risks goes a long way. However, you're not finished with vulnerability management until you've made a conscious decision about what to do with each unresolved vulnerability. These decisions need to be reevaluated periodically—say, quarterly—in case circumstances have changed.

Vulnerability Management Metrics

If you can't measure how you're doing with your vulnerability management program, you generally can't justify its usefulness or know whether you need to make changes. Metrics are useful but dangerous things; they help drive continuous improvement and reveal problems, but they can also lead to silly decisions. Make sure that part of your process of reviewing metrics and results includes a sanity check on whether there are reasonable extenuating factors to a metric going the wrong direction, or whether the metrics are being manipulated in some way.

There are many different metrics available for vulnerability management, and many tools can automatically calculate metrics for you. Metrics can generally be reported by separate teams or business units. Sometimes a little friendly competition helps motivate teams, but remember that some teams will naturally have a harder job to keep up with vulnerability management than others!

Every organization will be different, but here are some metrics that I've found useful in the past.

Tool Coverage

For each tool, what percentage of the in-scope systems is it able to cover? For example, for a dynamic application scanner, what percentage of your web applications does it test? For a network scanner, what percentage of your cloud IP addresses does it scan? These metrics can help you spot leaks in your asset and vulnerability management pipeline. These metrics should approach 100% over time if the system scope is defined properly for each tool.

If you have tools with a really low coverage rate on systems or applications that should be in scope for them, you're not getting much out of them. In many cases, you should either kick off a project to get the coverage percentage up, or retire the tool.

Mean Time to Remediate

It's often useful to break this metric down by different severities and different environments. For instance, you may track by severity (where you want to see faster fixes for "critical" items than for "low-severity" items) and break those out by types of systems (internal or internet-facing). You can then decide whether these time frames represent an acceptable risk, given your threat model.

Remember that remediation doesn't always mean installing a patch; it could also be turning off a feature so that a vulnerability isn't exploitable. Mitigation through other means than patch installation should be counted correctly.

Note that this metric can be heavily influenced by external factors. For example, when the Spectre/Meltdown vulnerabilities hit, patch availability was delayed for many systems, which caused mean time to remediate (MTTR) metrics to go up. In that particular case, the delays didn't indicate a problem with the organization's vulnerability management program; it meant that the general computing environment had been hit by a severe vulnerability.

Systems/Applications with Open Vulnerabilities

This is usually expressed as a percentage, since the absolute number will tend to go up as additional items are tracked. This metric is often broken down by different system/application classifications, such as internal or internet-facing, as well as the severity of the vulnerability and whether it's due to a missing patch or an incorrect configuration.

Note that the patch management component of this metric will naturally be cyclical, because it will balloon as vulnerabilities are announced and shrink as they're addressed via normal patch management processes. Similarly, changes to the bench-

mark may cause the configuration management component of this metric to temporarily balloon until the systems have been configured to match the new benchmark.

Some organizations measure the absolute number of vulnerabilities, rather than systems or applications that have at least one vulnerability. In most cases, measuring systems or applications is more useful than measuring the absolute number of vulnerabilities. A system that has one critical vulnerability poses about the same risk as a system with five critical vulnerabilities—either can be compromised quickly. In addition, the absolute number of vulnerabilities often isn't much of an indication of the effort required to resolve all issues, which would be useful for prioritization. You might resolve hundreds of vulnerabilities in a few minutes on a Linux system with a command like `yum -y update; shutdown -r now`.

This metric can also be used to derive higher-level metrics around overall risk.

Percentage of False Positives

This metric can help you understand how well your tools are doing, and how much administrative burden is being placed on your teams due to issues with tooling. As mentioned earlier, with some types of tooling, false positives are a fact of life. However, a tool with too many false positives may not be useful.

Percentage of False Negatives

It may be useful to track how many vulnerabilities should have been detected by a given tool or process but were instead found by some other means. A tool or process with too many false negatives can lead to a false sense of security.

Vulnerability Recurrence Rate

If you're seeing vulnerabilities come back after they've been remediated, that can indicate a serious problem with tools or processes.

A Note on Vulnerability Scoring

The first question almost everyone asks about a given vulnerability is, "How bad is it?" The most commonly accepted standard for "badness" is the Common Vulnerability Scoring System (CVSS). CVSS has been around for over a decade, and two major versions are in heavy use (v2 and v3). Both versions have their proponents and critics, but most security professionals agree that the base number you get from either CVSSv2 or CVSSv3 doesn't tell the whole story for your environment and your organization. It's important to have some method to adjust CVSS scores for the threat landscape and your specific environment, either by using CVSS temporal and environmental scores or some other method.

However, this can quickly turn into a game of changing the classification of items to avoid going overdue. While metrics are useful, it's important that you don't lose track of the real goal, which is to prevent security incidents.

In many cases, we don't need to think too hard about how bad the vulnerability is. *The default action in cloud environments should be to automatically apply security patches and run automated tests to see whether they have caused issues.* Only if a security patch or configuration change isn't available, causes problems, or can't be executed for other reasons should you go through the trouble of manually evaluating how big of a risk it is to your environment.

Change Management

Many organizations have some sort of *change management* function. In its simplest form, change management should ensure that changes are made only after they're approved, and that there has been some evaluation of the risk of making a change.

Change management can assist with vulnerability management by making sure that proposed changes don't introduce new security vulnerabilities into the system. If done poorly, change management can also hinder vulnerability management and increase overall risk by slowing down the changes needed to resolve vulnerabilities.

As discussed earlier in the chapter, some of the new technologies in cloud environments may reduce the risk of an overall outage, so that less manual change management is needed to achieve the same level of operational risk. Part of an overall cloud vulnerability management program may be modifying change management processes.

For example, pushing new code along with security fixes to production may be a business-as-usual activity that's automatically approved by a change control board, provided that there's a demonstrated process for quickly getting back to a good state. That might be accomplished by pushing another update, rolling back to a previous

version, or turning off application traffic to the new version while the issue is being worked out. However, larger changes, such as changes to the design of the application, may still need to go through a manual change management process.

Ideally, there should be at least one security practitioner involved with the change control process, either as a change control board member or as an advisor.

A documented change management process is required for several industry and regulatory certifications, including SOC 2, ISO 27001, and PCI DSS.

Putting It All Together in the Sample Application

Remember the really simple three-tier sample application from Chapter 1? It looked like Figure 5-3.

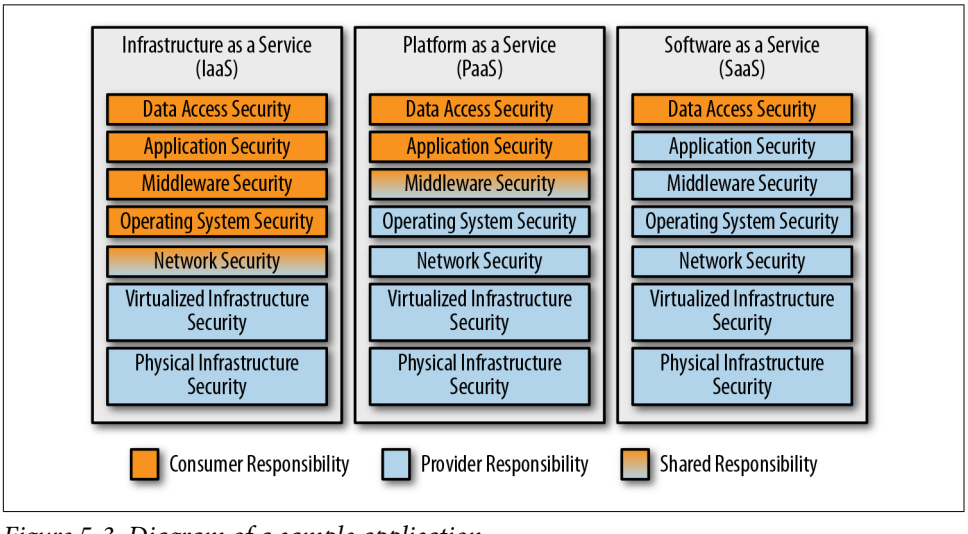


Figure 5-3. Diagram of a sample application

If you're in an orchestrated, container-based microservice environment, with test and production Kubernetes clusters, your sample application may look a bit different. However, you can still spot the same three main tiers in the middle of the diagram (Figure 5-4).

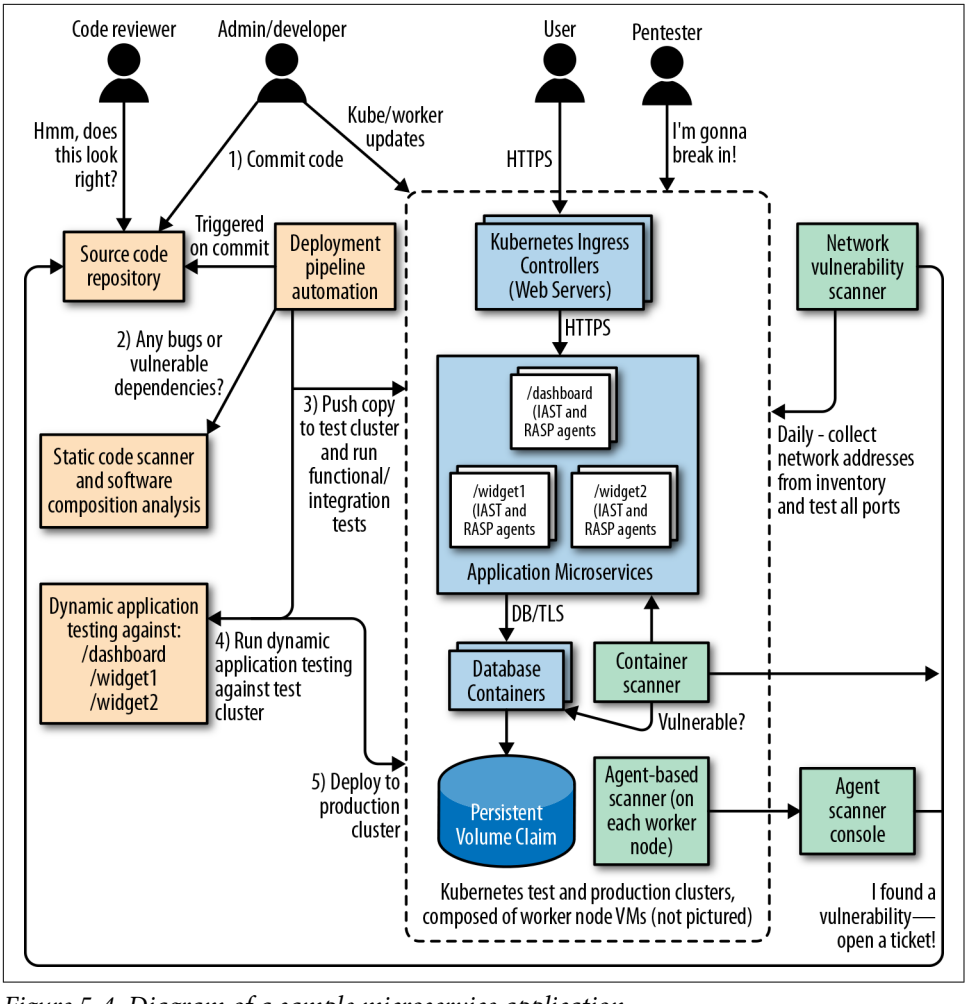


Figure 5-4. Diagram of a sample microservice application

For simplicity, the worker nodes that actually run the containers aren't shown in the diagram, and only one cluster is pictured rather than separate test and production clusters. Let's look at how we might design a vulnerability management process in this environment. First, consider the roles shown on the left:

1. Before deployment, a penetration tester (pentester) tries to break into the system, just like a real attacker would. This test might be run by an external team that's contracted to test this specific system at a given time, an internal *red team* that roams around doing unannounced testing of systems, or both.
2. The user will use the application, just as in the previous examples. In some cases, end users may report security vulnerabilities in addition to functional bugs.

3. The admin/developer is a role with both development and operations/administration responsibilities. In your organization, these responsibilities might lie with a single person or multiple teams, but the people and teams filling this role must do the following:
    a. Ensure that the infrastructure and platform components, such as the Kubernetes master and the worker nodes, are up to date.
    b. Make code updates. Note that these code updates might also represent changes to the infrastructure, such as new microservices or modifications to the "firewall" for each microservice to allow different connections.
    c. Push to production and/or switch traffic to the new version of the application. The process and decision of when to do this will be organization-specific but should usually include business stakeholders in addition to IT staff.
4. The code reviewer may be part of a separate team but is often simply another developer in the organization. Not every organization uses manual code reviews, but they can be a good way to spot security vulnerabilities in critical areas of code.

Second, let's look at the pipeline to deploy, at the bottom of the figure:

1. An admin/developer will commit a change to the codebase, which will trigger the deployment pipeline automation.
2. A static code scanner will flag problems in your proprietary code, such as accepting input without validation. A software composition analysis tool will also look at any open source dependencies to see if there are known vulnerabilities in them. Ideally, the developer will get almost immediate feedback if an issue is found, and issues that are severe enough will block deployment of the new code unless overridden.
3. The automation will then start up a copy of the new code in a test environment and run test cases to see that the code functions.
4. The automation will invoke a dynamic application tester to find any problems. Again, ideally the developer will be notified of any issues here, and severe issues will stop the process.
5. If all tests pass, the code will be deployed as a new instance to production, where the administrator can choose to direct some or all of the production traffic to the new instance. If everything works fine, all traffic can be sent to the new instance and the old instances can be deleted.

Third, let's look at the periodic scanning tools at the top of the figure. For each of these, if a problem is found, a ticket will automatically be entered as an issue in a

tracking repository (shown here as part of the source code repository), and issues will go through the risk management process if they stay around for too long:

1. The network vulnerability scanner will test all of the TCP and UDP[3] ports on the IP addresses of the worker nodes that make up the cluster. In a well-configured cluster, the scanner should only see the HTTPS (tcp/443) ports open, but it may find problems with those (such as a vulnerable version of a web server or a configuration allowing weak TLS ciphers). It may also spot NodePorts opened accidentally that allow traffic in to some other service besides the frontend web server. For example, perhaps someone accidentally left the database open to the internet instead of only to the application microservices!
2. The container scanner will look for problems in each running container. Perhaps the operating system components used by the containers have known vulnerabilities, such as binary libraries that can't be detected by the SCA tools.
3. The agent installed on each worker node (virtual machine) in the cluster will watch to make sure that the operating system components are kept up to date and that the CIS Benchmarks for that operating system pass.
4. Finally, the IAST agent that's part of each microservice will notify its console (not pictured) of problems found while the code was executing, and the RASP agent will attempt to block attacks.

There's a lot going on! Don't panic, though. This is for educational purposes, and many smaller environments won't need all the tools pictured here. Also, many products perform multiple functions: for example, a single tool might perform static scanning, dynamic scanning, and IAST/RASP. The important thing is to understand what the different types of tools do so that you can select tools that address your biggest threats.

Just buying a tool and installing it often doesn't do much good—you need to actually do something with what the tool is telling you. Concentrate on getting a good feedback loop back to your developers and administrators, that you can measure with some useful metrics, before adding another tool into the mix.

Penetration Testing and Red/Blue Teaming

A penetration test is typically scoped to a specific target, such as a new application or service, and is scheduled to occur at a specific time, such as prior to production deployment. A pentester will often start by using various scanning tools to find potential vulnerabilities and then will attempt to exploit those vulnerabilities.

3 UDP scanning, like any other UDP communications, is somewhat unreliable by design.

A *red team* will often use many of the same tools as a pentester but is more loosely engaged to roam around the entire network or organization looking for vulnerabilities. A *blue team* is a defensive team and will attempt to detect the red team (as well as real attackers!). Some organizations also form *purple teams*, where the red and blue teams collaborate on fixing issues after they're found and on creating more effective defenses.

Summary

Vulnerability management, patch management, configuration management, and change management are separate disciplines in their own right, with separate tooling and processes. In this chapter, I've combined them together to quickly cover the most important aspects of each, but there are entire books written on each subject.

Vulnerability management in cloud environments is similar in many ways to on-premises vulnerability management. However, with cloud computing often comes a heightened business focus on rapid deployment of new features. This leads to a need for vulnerability management processes that can keep up with quickly changing infrastructure.

In addition, the philosophies of immutable infrastructure and continuous delivery are often adopted along with the cloud, and these can considerably reduce the risk of an outage due to a change. This alters the balance between operational and security risk. Because applying security fixes is a change, and you can make changes more safely, you can afford to roll out security fixes more aggressively without risking bringing the system down. This means that you should usually adopt different vulnerability management, patch management, and change management processes in cloud environments. In addition, there are both cloud-aware and provider-specific tools that can make vulnerability management easier than it is on-premises.

After access management, vulnerability management is the most critical process to get right for most cloud environments. Attackers can get unauthorized access to your systems through vulnerabilities at many different layers of your application stack. You need to spend some time understanding the different layers, what your vulnerability management responsibility is for each of those layers, and where the biggest risks to your environment are likely to be. You then need to understand the different types of vulnerability management tools available and which ones address the areas that are highest risk for you.

Every vendor will try to convince you that their tool will do everything for you. That's rarely the case; you'll usually need at least a few different tools to cover vulnerability management and configuration management across your cloud environment. Focus on getting value from each tool before throwing more into the mix. For each tool, you should be able to explain clearly what types of vulnerabilities it will find. You should

also be able to sketch out a pipeline of how the tool gets valid inputs, how it finds and/or fixes vulnerabilities, how it communicates vulnerabilities back to the teams who are responsible for fixing them, and how you track the vulnerabilities that can't be fixed right away as risks.

About the Author

Chris Dotson is an IBM Senior Technical Staff Member and an executive security architect in the IBM Cloud and Watson Platform organization. He has 11 professional certifications, including the Open Group Distinguished IT Architect certification, and over 20 years of experience in the IT industry. Chris has been featured as a cloud innovator on the *http://www.ibm.com* home page several times; his focus areas include cloud infrastructure and security, networking infrastructure and security, servers, storage, and bad puns.

Colophon

The image on the cover of *Practical Cloud Security* is the red kite (*Milvus milvus*). Related to eagles, buzzards, and harriers, this bird of prey inhabits Western Europe and parts of Scandinavia. It is seen as far east as the Ural mountains and migrates as far south as Israel and Egypt.

Its plumage is orange-red (rufous) on much of the body and the upper layers of the wing feathers (coverts). It averages 24 to 28 inches long (60 to 70 centimeters) with a 68 to 70 inch wingspan (175 to 179 centimeters). Thanks to its large wingspan and light weight (about as much as a mallard duck), it soars gracefully in search of prey. It can be identified in flight by its forked tail. Like an eagle, it has a hooked beak ideal for tearing meat. It feeds on small animals such as mice, voles, shrews, and rabbits as well as carrion.

Red kites are monogamous birds, and the male and female work together to build their nest and feed their chicks. They may return to the same nest year after year, and the next generation tends to nest within a few miles of where it was hatched.

During the middle ages, the red kite was valued for keeping villages free from rotting food and vermin. In the UK, it was considered a pest and was hunted almost into extinction by the early 20th century. It was reintroduced in the late 20th and early 21st centuries, and is now on the UK's green list, regarded as among the least threatened species.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Lydekker's Royal Natural History*. The cover fonts are Gilroy and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.