Checkmarx

The world runs on code. We secure it.

The Many Risks of Modern Application Development

A Comprehensive Guide for Leaders and Practitioners – Part 2

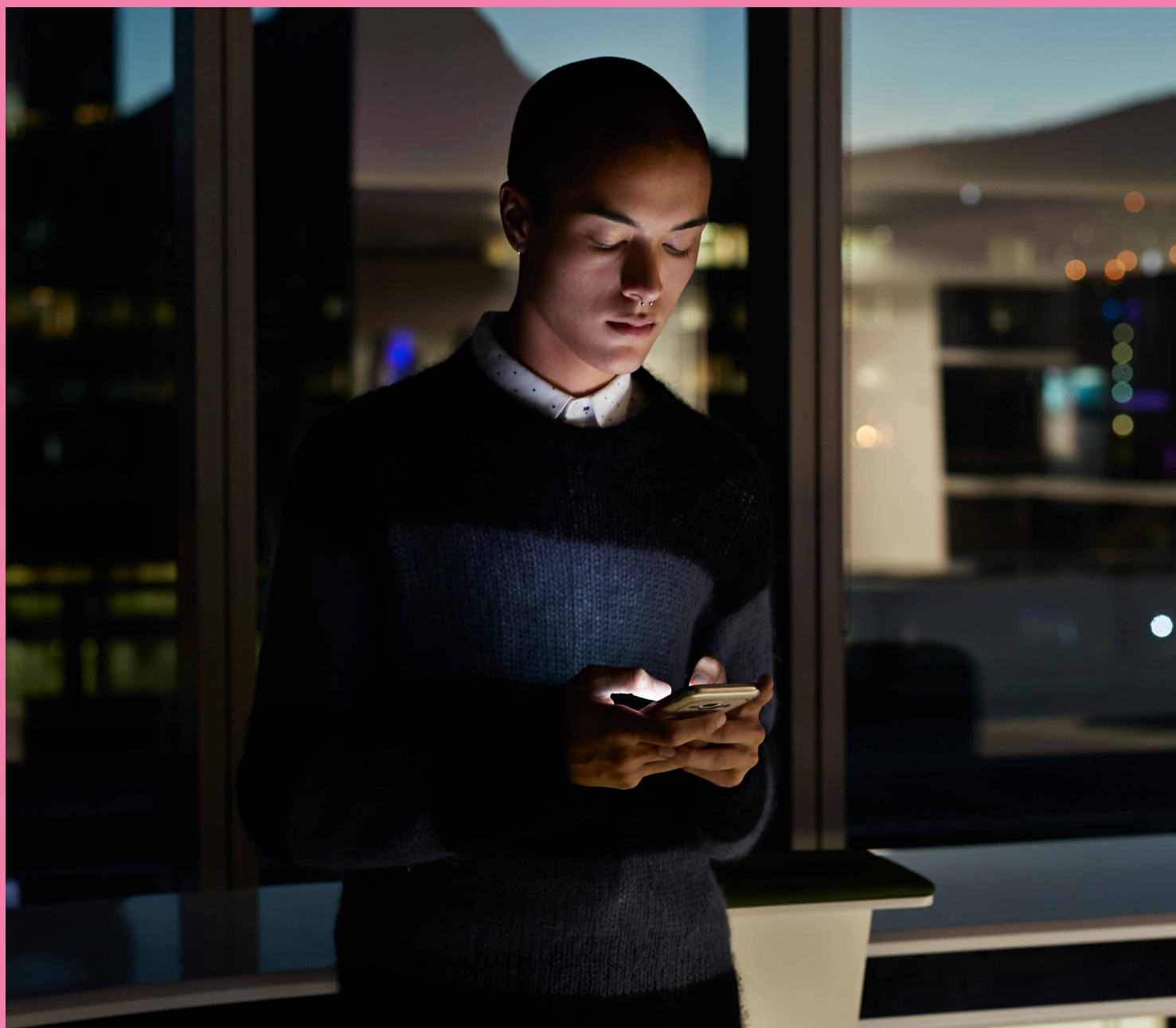


Table of Contents

Introduction

In Part 1 of this e-book, we looked at the many facets of modern application development (MAD), including benefits and challenges to be expected as organizations adopt this new development and delivery approach.

In Part 2, we'll dive much deeper into the security risks that MAD brings to the surface. As of this writing, no other publication contains a more comprehensive list of these risks. Developers and security pros can use what we'll be looking at here as a starting point for further discussion, risk analysis, and risk management.

New risks become apparent in MAD primarily due to the "legolized" methodology, which introduces numerous variables into the risk vs. reward equation. Although the promised benefits of MAD are measurable and well-supported across the development industry, expanded risks must still be acknowledged and addressed. Organizations cannot effectively manage MAD risks if they are not aware of being at risk to begin with.

MAD Security Risks—Deeper Dive

Today, organizations must acknowledge the lists of well-known software risks provided by OWASP, SANS, and others, but an entirely new set of risks also emerge in MAD initiatives. Let's expand on each risk we listed in Part 1 of this e-book.

Open Source Code Risks

It's easy to understand why open source is so pervasive. By importing open source libraries, extensions, and other resources into applications, developers can reuse code others have already written, freeing up more time to write innovative features that don't yet exist.

Open source has its downsides, however. To leverage open source responsibly and avoid the security and compliance challenges that often accompany it, developers need full visibility into the open source software they use as well as the risks associated with it. Here are the top three.

Risk 1: Inconsistent Security Standards

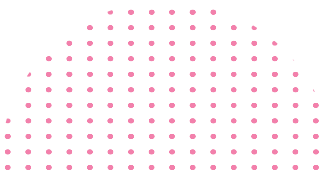
When it comes to security, open source code varies widely. Some projects, like the Linux kernel, maintain very high security standards (but still sometimes let vulnerabilities slip by). On the other hand, there are any number of tools on GitHub that don't set the bar so high. Open source software can be quite secure—but sometimes, it's not very secure at all.

This inconsistency presents a challenge for developers because they need to vet the security of third-party open source code case by case. Although open source advocates like to claim that "many eyeballs make all bugs shallow" and the community is likely to quickly discover and fix vulnerabilities, the number of eyes on a given open source codebase can vary significantly. The less attention an open source project gets and the less experienced its developers, the more likely it is to have low security standards.

Risk 2: Unknown Source Code Origins

It can be difficult to vet the security of open source code when its origin is unclear. For example, you might find a source code tarball on a website that offers little information about who wrote the code. Perhaps a README inside the tarball attributes the code to someone, but you have no way of verifying its authenticity. Perhaps you clone code from a Git repository and assume the code's author maintains the repo, when in fact the repo could contain code that originated somewhere else and was simply copied. Here again, any claims of authorship are difficult to verify.

It's easier to trust third-party code when you know experienced, well-intentioned developers wrote it. Obviously, you should scan the code for vulnerabilities either way, but if you're confident in where it came from, you can make better decisions about whether to use it in the first place.

Risk 3: Licensing Noncompliance

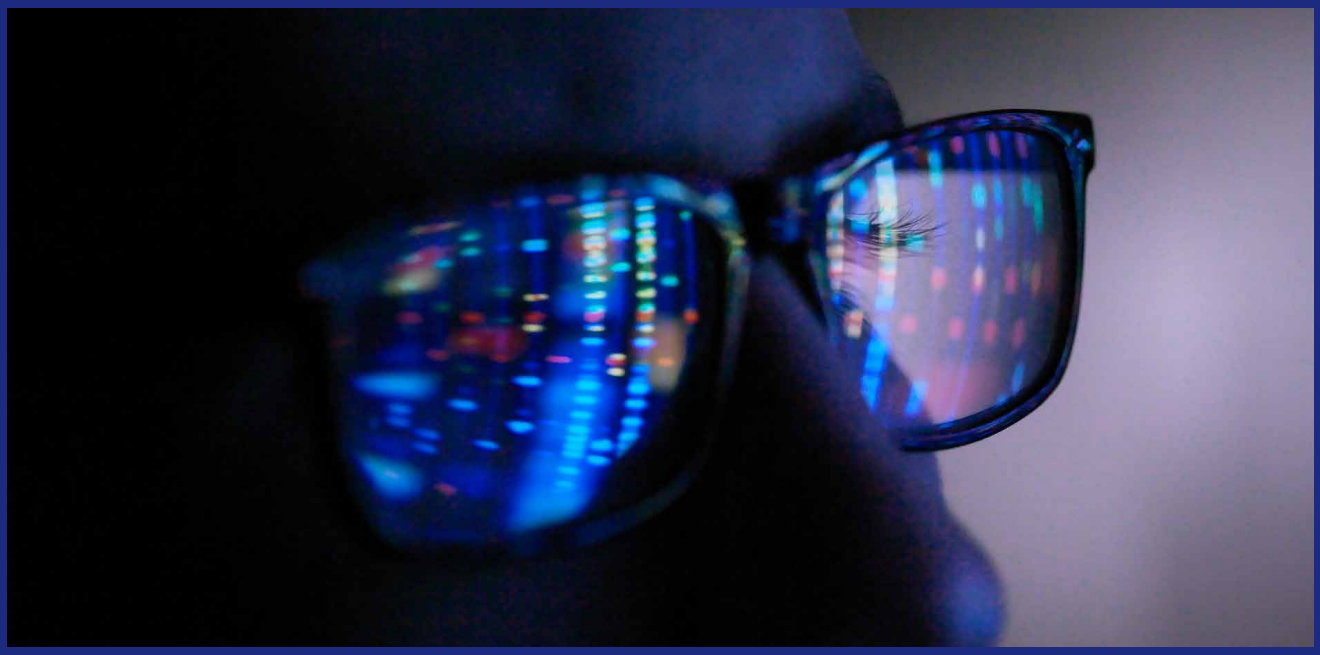
Developers tend to think they're experts in open source licenses, but most aren't. They often misunderstand licensing terms[1] and hold false beliefs, such as "the GPL says you can't sell your code for money" or "under the MIT license, you can do whatever you want as long as you attribute the original developers."

Misconceptions like these create risk in the form of noncompliance with open source licensing terms. If developers don't understand the specifics of the licenses that govern each of the various open source components they use, they may violate licensing agreements, which can put their organization at risk of possible litigation.

Complicating matters further, open source projects occasionally change their licensing terms—as Elastic famously did in 2021, for example—meaning it's not enough to determine the licensing requirements of open source code only the first time you use it. You need to reevaluate every time the code changes versions.

That covers the top three risks associated with open source usage. Next, let's look at the risks associated with microservices.

[1] "Common misconceptions in licensing," Free Software Foundation, February 23, 2015, https://www.fsf.org/bulletin/2014/fall/common-gpl-misconceptions.



Microservices Risks

If you're a developer today, it's hard not to love microservices. By adding agility and resilience to applications, microservices architectures make it easier to build high-performance software. A microservices strategy only pays off, however, if you effectively manage its attendant risks.

In certain ways, security for microservices is fundamentally more challenging than it is for less complex monolithic architectures. If you don't manage the security risks of microservices, you may find yourself with an application that performs poorly because it has been compromised—a problem no amount of agility will solve. Here are the five most common security risks developers should think about when writing microservices-based apps.

Risk 1: Expanding Complexity

If you've ever written or managed a microservices app, you know microservices architectures bring complexity to a whole new level. They make writing the app more complex because developers need to ensure each microservice can find and communicate with other microservices efficiently and reliably. They also make management harder because administrators need to contend with service discovery, distributed log data, instances that constantly spin up and down, and so on.

These challenges translate to security risks because when it's hard to keep track of everything in an environment, it's also more difficult to detect vulnerabilities. To conquer this complexity, developers and security teams need stronger tools for managing source code and monitoring runtime environments than they would when dealing with monolithic apps.

Risk 2: Limited Environment Control

Depending on how you deploy microservices, you may have limited control over the runtime environment. For example, if you use serverless functions to host microservices, you will have little or no access to the host operating system (OS). You only get the monitoring, access control, and other tooling the serverless function platform provides.

This makes security significantly more challenging because you can't rely on OS-level tools to harden your microservices, isolate them from one another, or collect data that might reveal security issues. You have to handle all the risks within the microservice itself. That can certainly be done, but again, it requires a degree of coordination and effort that isn't routine for developers of monolithic apps.

Risk 3: Inappropriately Securing Data

In a monolithic application, data is usually stored in a simple and straightforward way, either on the local file system of the server that hosts the data, or possibly in network-connected storage mapped to the server's local storage. This data is easy to encrypt and lock down with access controls.

Microservices typically use an entirely different storage architecture: Because microservices are usually distributed across a cluster of servers, you can't rely on local storage and OS-level access controls. Instead, you most often use a scale-out storage system that abstracts data from underlying storage media.

These storage systems can still usually be locked down with access controls, but the controls are often more complex than dealing with permissions at the file system level, which means it's easier for developers to make mistakes that invite security breaches.

On top of this, the complexity of ensuring that each microservice has the right level of access to the storage can lead some developers to do the easy but irresponsible thing: fail to configure granular storage policies, allowing all microservices to access all data.

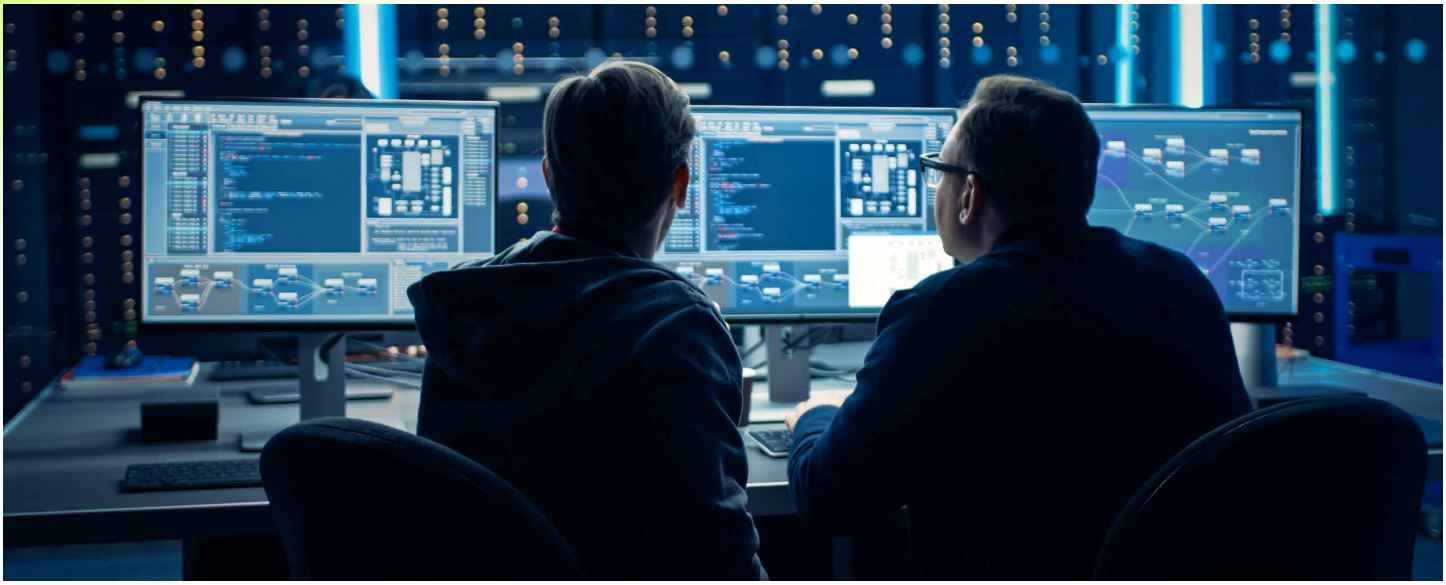
In any case, you end up with storage less secure than that of a conventional monolithic app. To avoid this, you need to take full advantage of granular access control within storage systems and consistently scan for access misconfigurations.

Risk 4: Inappropriately Securing the Network

Securing the network is critical for any application that connects to it—virtually every application today. When it comes to microservices, however, network security is altogether more complex. Microservices don't just communicate with end users or third-party resources over the internet like a monolith would. They also usually rely on a web of overlay networks to share information among themselves.

More networks mean more opportunities for attackers to find and exploit vulnerabilities. They can intercept sensitive data microservices exchange with each other, use internal networks to escalate breaches from one microservice to others, and so on.

Now that we've looked at these common microservices-related risks, let's look at container risks.



Container Risks

Containers are more agile and consume fewer resources than virtual machines (VMs). They provide more flexibility and security than running applications directly on the OS. They are easy to orchestrate at massive scale using platforms like Kubernetes.

At the same time, they present some serious challenges, not least in the realm of security. Although the benefits mostly outweigh the security risks, it's important to assess the security problems containers can introduce to your software stack and take steps to remediate them. Here are the seven most common security risks developers should think about when using containers.

Risk 1: Running Containers from Insecure Sources

Containers have become so popular in part because an administrator can pull a container image from a public registry and deploy it with just a few commands. That's great for agility and speed, but it can pose problems if the container image contains malware.

This risk is not theoretical. Hackers have uploaded malicious container images to Docker Hub (the most widely used public container registry) and given them names intended to trick developers into believing they come from a trusted source. A 2020 study by Prevasio found that 51% of container images on Docker Hub contain at least one vulnerability.[2]

The lesson here is that it's vital to double-check the origins of container images you pull, especially when dealing with public registries.

[2] "Operation Red Kangaroo," Prevasio, December 2020, https://knowledge-base.prevasio.io/pdf.html?file=Red_Kangaroo.pdf

Risk 2: Unknown Source Code Origins

Risks associated with container registries can run in the opposite direction, too: you might upload data to a private registry you assume is secure only to discover that it—and the sensitive data you stored in it—is accessible to the world at large.

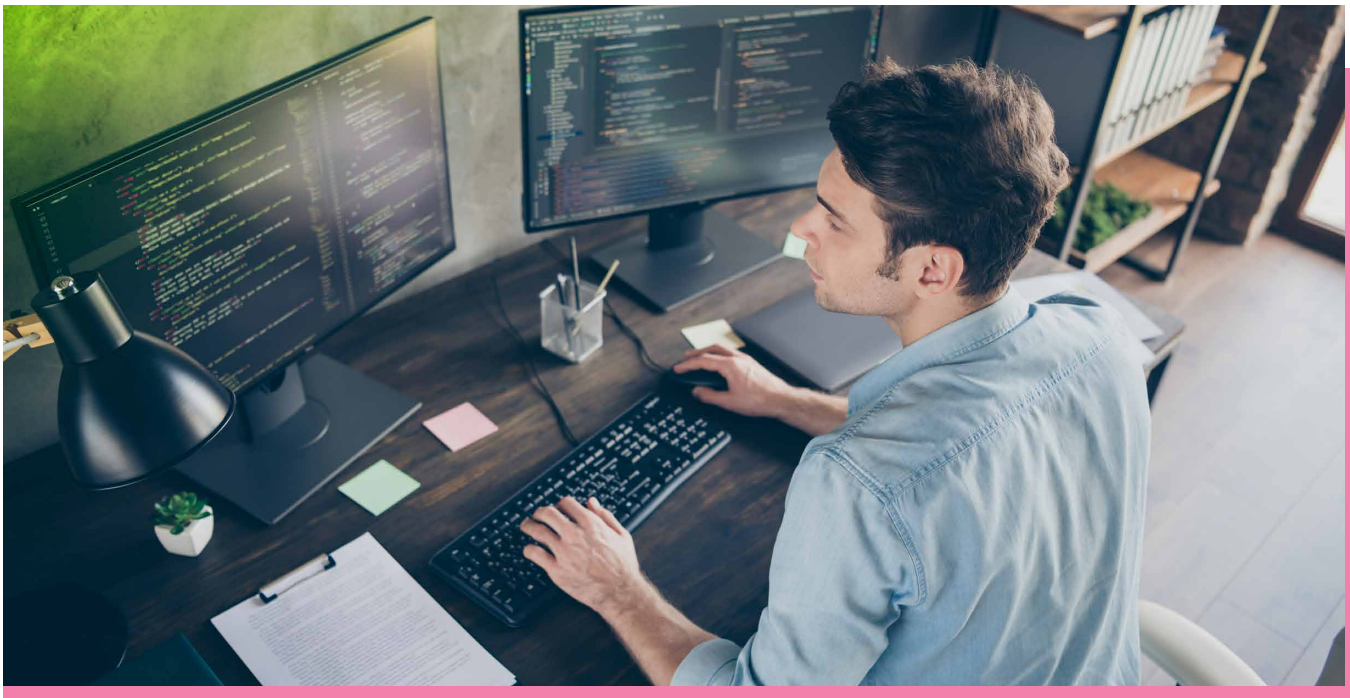
That's what happened to Vine in 2016.[3] The company uploaded a container image that included source code for its entire platform into an insecure registry. The registry's URL hadn't been publicly shared, but anyone who could guess it had unfettered access to the images in it.

Mistakes like this are easy to make. When juggling dozens or hundreds of container images, you can accidentally place a sensitive image in an unsecured registry or even forget that an image contains sensitive data in the first place.

Risk 3: Too Much Faith in Image Scanning

Image scanners are vital tools that can automatically determine whether containers contain known vulnerabilities, but they're hardly a complete guarantee against all types of risks. Because they work by matching the contents of container images against lists of known vulnerabilities, they won't discover security flaws that haven't been publicly disclosed. Scanners may also overlook vulnerabilities if container images are structured in unusual ways or their contents are not labeled in a way the scanner expects.

[3] Hacker Downloaded Vine's Entire Source Code. Here's How...," The Hacker News, July 23, 2016, https://thehackernews.com/2016/07/vine-source-code.html



Risk 4: Broader Attack Surface

Running containers requires more tools and software layers than running a conventional application. In this respect, containers create a broader attack surface.

When you deploy a container, you have to worry about the security of not only the application and the OS that hosts it, but also the container runtime, the orchestrator, and possibly various plugins the orchestrator uses to manage things like networking and storage. If you run "sidecar" containers to help with tasks like logging, those become security risks, too.

This can all be managed, but it requires greater investment in security—and a broader set of security tooling—than a traditional, non-containerized application stack.

Risk 5: Bloated Base Images

Developers use container base images as foundations for creating custom images. Typically, a base image is some kind of OS, along with any common libraries or other resources required to run the types of applications you are deploying.

It can be tempting to pack more than the bare minimum into base images. You never know what you may need to run your applications in the future, so you may decide to include libraries that aren't strictly necessary for your applications today.

The more you include in your base images, though, the greater the risk of vulnerabilities that allow your containers or applications to be compromised. It's best to build base images that are as minimal as possible, even if that means updating them periodically or maintaining different base images for different applications.

Risk 6: Lack of Rigid Isolation

Containers should isolate applications at the process level, but they don't always do that perfectly. Because containers share the same kernel, a bug in the runtime or a misconfiguration in the environment could allow a process running in one container to access resources that live in other containers or even gain root access to the host.

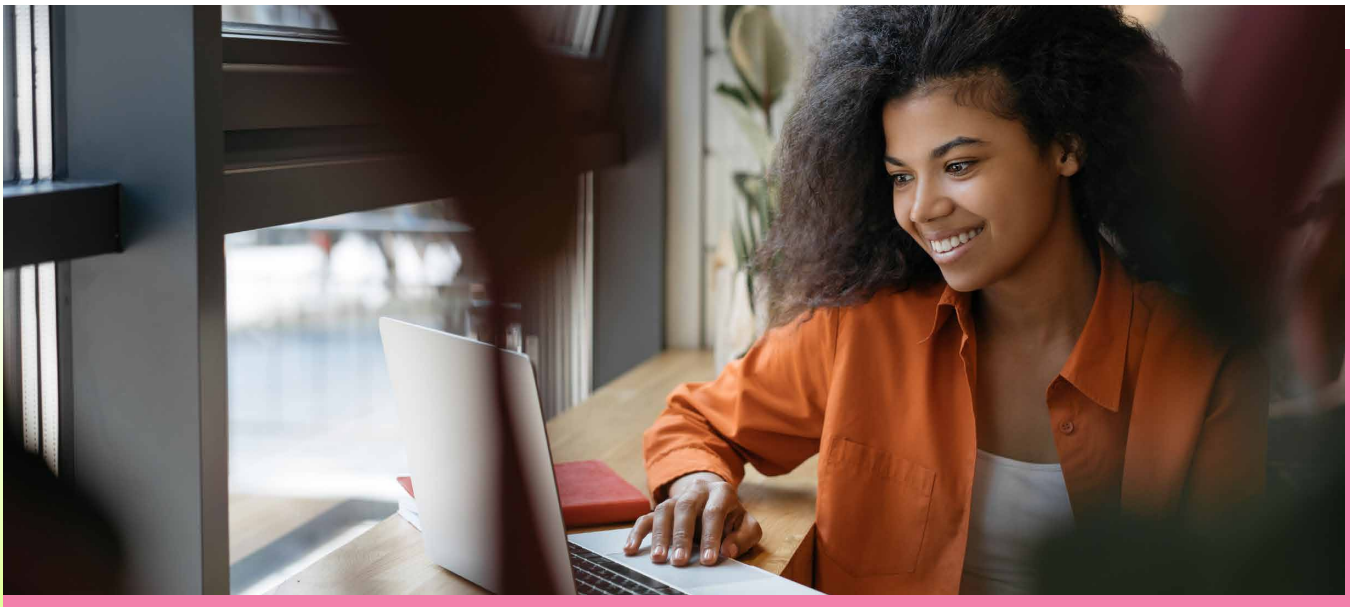
This is why it's extra important to vet your configurations for security in addition to monitoring runtime environments for malicious activity. There is simply a greater risk of privilege escalation and similar issues with containers than there is with VMs.

Risk 7: Less Visibility

The harder it is to observe and monitor an environment, the harder it is to secure it, and when it comes to containers, observability and monitoring are especially difficult. It's not that the data you need to track containers doesn't exist. That data is spread across multiple locations—inside containers, on Kubernetes worker and master nodes—and not always persistent (e.g., logs inside containers will disappear forever when the container instance shuts down unless you move them first).

These challenges are manageable, but they require a more sophisticated strategy for keeping track of what's happening inside your environment than a simpler type of application stack typically would.

Now that we've covered some of the common risks associated with containers, let's look at infrastructure as code (IaC).



Infrastructure as Code Risks

IaC tools are exemplary software solutions that developers and DevOps teams use to describe common infrastructure components like servers, virtual private clouds, IP addresses, and VMs in a configuration language made up of lines of code. Once ready to deploy, this configuration serves as a blueprint to provision actual infrastructure services on demand.

The benefits of IaC are better control of the change process alongside greater efficiency and consistency when deploying changes. However, the devil is in the details, and trying to implement those tools in real life often comes with new risks. Let's look deeper into the common risks when using IaC.

Risk 1: Steep Learning Curve

Using IaC tools without taking the time to learn some of their quirks and limitations can become not just an inconvenience but a serious problem. Many of the most prevalent IaC tools, like Terraform and AWS CloudFormation, are incredibly simple to start using, but they become complicated as requirements change over time. For example, trying to use custom resources in CloudFormation requires some insight into how the tool works under the hood. Failure to delve into the inner details will put your organization at increased risk of configuration drift, pushing the wrong infrastructure components, or obtaining a system in an incomplete state.

Terraform Imports is another example that requires deep insight into how Terraform works. The official page warns: "If you import the same object multiple times, Terraform may exhibit unwanted behavior."[4] To understand the nature of this "unwanted behavior," however, you'll need to look closely at the details:

- **What does it really mean?**
- **Does it break the local TFSTATE file?**
- **Does it create configuration drift?**
- **How can you prevent importing the same object multiple times?**

All these questions take time and dedication to answer. If you're a developer, you should reserve that time as part of the process of learning the tool while using it for real projects.

[4] "Import Usage," Terraform, accessed September 16, 2021, https://www.terraform.io/docs/cli/import/usage.html.

Risk 2: Human Error

Some IaC tools have a set of phases or steps that need to be performed in a specific order. For example, in Terraform, you have:

- **terraform plan,** which creates an execution plan for what Terraform will perform against the infrastructure.
- **terraform apply,** which applies the Terraform plan.

Failure to perform a plan review before using apply can result in destructive changes. A visual inspection before applying gives you one more chance to review the changes first. The caveat here is that, by default, there is no requirement to follow this order—but directly applying the changes, skipping the first step, could lead to unfortunate mistakes.

The repercussions of missing a destructive change can be excruciating. We recommended establishing automated protections to apply changes to infrastructure. For example, instead of manually reviewing Terraform plan outputs (which can be lengthy), you should invest in pull request tools like Atlantis to help discover any unintentional destructive changes early.

Risk 3: Configuration Drift

Terraform and other tools can help maintain infrastructure components and associated attributes as long as they're the only tools managing those things. For example, Terraform can't detect any drift of resources if you've applied changes with other tools, like Chef or Puppet. This can be a major problem for organizations still working in a variety of IaC tools that are unaware of each other or overlap their body of work.

Scanning for drift helps you figure out deviations between your IaC tools and the real infrastructure. Because each case is different, though, you'll find virtually no public or private services that do it accurately. The best way to manage configuration drift for your organization is to make sure:

- There is no overlap between your IaC tools
- Any infrastructure changes are monitored by only one IaC tool
- Reporting and health checking are integrated (external checks can sometimes help verify this)

Risk 4: Exposing Sensitive Data and Ports

Although Terraform and AWS CloudFormation can provision infrastructure components, you need to answer some important questions when using these tools. For example:

- How would you know during the process if it leaked sensitive data?
- How would you know if the S3 bucket it manages contains the security profiles you specified?
- Are you using a provider that leaks sensitive data to stdout?
- Did you unexpectedly open a port to the internet?

This is only a subset of the questions you need to answer when delegating your infrastructure management to IaC tools. It's easier to accidentally expose resources to the public internet than the opposite, so there's a lot at stake here.

Next, let's look at some newer API risks in comparison to the OWASP API Security Top 10 list.

API Risks

The Open Web Application Security Project (OWASP) periodically publishes lists of security risks for web applications and APIs. Obviously, there is overlap between these categories—many web applications use APIs—but OWASP assesses the risks separately because not all web applications use APIs and not all APIs are used in web applications. Here is a summary of the latest OWASP version of the API risk list, which we also included in Part 1 of this e-book.

Although the OWASP API Security Top 10 list is a vital starting point for establishing best practices developers should follow when working with APIs, many security-minded developers and AppSec pros believe it has some weaknesses, which we'll look at in more detail. Your organization should consider these weaknesses when addressing API risks beyond the OWASP list.

New Risk 1: Third-Party APIs

The OWASP API Security Top 10 list draws no distinction between in-house and third-party APIs. Although both may work in the same way in a technical sense, third-party APIs pose additional risks because developers' ability to monitor them is limited, as is their ability to manage authentication and authorization in many cases.

If developers don't build the APIs themselves, they can only interact with the APIs in whatever way the APIs support. In this context, developers should be extra careful when using third-party APIs—not outright avoiding them, but at a minimum understanding their origins and limits.

New Risk 2: Redundant API Considerations

Like many "Top 10" lists, OWASP's API risk list is a bit redundant. That makes it harder to work with and arguably gives developers a false impression of priorities. For instance, as items 2 and 5 both involve authentication or authorization, they could probably be consolidated into a single point about ensuring that APIs properly authenticate and authorize all resources.

Some argue that improper user authentication is a greater risk than broken function-level authorization, but some also argue the opposite. Either way, some believe the difference is significant enough to merit two separate items. The main takeaway is that reliable, granular authentication and authorization are essential for every API interaction, and the OWASP list doesn't state that as clearly as it could.

New Risk 3: Insufficient Emphasis on API Monitoring

Only the last two items on OWASP's list deal with monitoring API resources and behavior. Due to the extremely heavy reliance on APIs in MAD, comprehensive monitoring and logging should be near the top of the list. Simply put, you can't manage API security risks effectively if you lack visibility into them. Granted, logging and monitoring tend to be jobs for IT engineers more than for developers. Perhaps that's why OWASP gives them relatively little weight on this list, which is geared toward developers.

Still, we live in a DevOps-centric world where the line between developers and IT teams is blurred. Plus, you can't track API resources and behavior if the APIs don't provide proper facilities for logging and monitoring, which developers have to implement.

New Risk 4: API Training in MAD Culture

Discussions about "building a security culture" or "training developers in security" often sound fluffy, especially without specific advice about how to achieve these goals. Nonetheless, the OWASP list includes no warnings about lack of shared visibility or collaboration between developers and security teams, which can also introduce increased risk, especially in MAD.

This topic could be included in a specific, concrete way. For example, the list could mention the importance of making API monitoring and log data available to both security teams and developers, which is one small step toward building a modern security culture.

The OWASP API Security Top 10 list is a must-read for any developer who cares about securely designing and using APIs, but it ultimately represents only one set of perspectives about API security. It's crucial to think through the guidance yourself and adapt its recommendations to your own applications and environment.

MAD Next Steps

We've taken a closer look at anticipated risks in the context of the common components found in MAD. Although each of the risks we've highlighted are important, this list isn't exhaustive. As modern development practices continue to evolve, this list of risks will likely expand.

To summarize: We began by highlighting the risks associated with open source and moved on to microservices, two categories heavily utilized in MAD. Next, we moved on to risks associated with containers, followed by infrastructure as code, and finally APIs to keep them top of mind.

Checkmarx solutions are built by developers, for developers, and we acknowledge that risks exist in all software, no matter how, when, or where it was built. We believe developers and security pros should be acutely aware of the risks we've compiled here. Broader awareness here will likely improve software security practices industry-wide.

In Part 3 of this e-book, we'll look at AppSec considerations in the context of MAD and highlight what will likely work best to secure modern applications.

About Checkmarx

Checkmarx is constantly pushing the boundaries of Application Security Testing to make security seamless and simple for the world's developers while giving CISOs the confidence and control they need. As the AppSec testing leader, we provide the industry's most comprehensive solutions, giving development and security teams unparalleled accuracy, coverage, visibility, and guidance to reduce risk across all components of modern software – including proprietary code, open source, APIs, and Infrastructure as Code. Over 1,600 customers, including half of the Fortune 50, trust our security technology, expert research, and global services to securely optimize development, at both speed and scale. For more information, visit our website, check out our blog, or follow us on LinkedIn, Twitter, YouTube, and Facebook.

Checkmarx

Checkmarx at a Glance

1,675+
Customers in 70 countries

750
Employees in 25 countries

45%
of the Fortune 50 are customers

30+
Languages & frameworks

500k+
KICS downloads in 2021

The world runs on code. We secure it.