Security

# Top 10 security traps to avoid when migrating from a monolith to microservices

Eric

August 20, 2019 · 7 minute read · One comment


Your team is thinking about migrating from a monolithic architecture to microservices. You're intrigued. The promises of additional scalability and more predictable deployments sound nice. You've also been down this road before, and you know that those promises don't always equal reality. You also know that migrations to a microservice approach don't always go as planned.

What's more, you know and understand your system now. You have a good grasp on things like time to repair and application security. That familiarity means that you're

comfortable with the risks your team takes, and what they mean for the business. An entirely new architectural approach carries risks that you can't quantify. Migrating to microservices isn't a new trend, though.

There's an opportunity to learn from people who've gone before, and understand their mistakes so you don't repeat them.

The goal of this article is to run down the most common security pitfalls teams fall into when moving to a microservices architecture from a monolithic architecture.

These are the ten most common mistakes that I've run into.

# 1. Rolling your own authentication scheme

In case you're wondering, not having an authentication scheme falls under this mistake. Authentication between microservices isn't something that teams think about when they're coming from a monolith. You don't need to authenticate yourself when calling a function in code between libraries on a server. Microservices are a totally different situation. It's critical to make sure that each service is able to authenticate not just the server requesting data, but also the context of the logged-in user. In a monolith, this is information that's just stored in memory. On a microservice, that's information that has to be passed with each request.

Most teams think that they can handle adding their own authentication for microservices. Those teams are largely incorrect about that. Much like it's a bad idea to build your own in-house authentication system for a typical web application, it's also a bad idea to roll your own authentication for microservices. Instead, use an open standard for authentication like OAuth.

# 2. Failing to ensure transaction reliability

To me, this is one of the toughest problems to solve with microservices. It's common for a

single request to an API to kick off changes in multiple parts of an application. In a monolithic application, this is no big deal. For a microservice, transactions can pose a big problem. One service might complete part of the work, only for it to fail on a second service. In a monolithic system, the failure in one part of the application means that the entire process is rolled back to the previous good state.

In a microservice, this is much harder. It's necessary for your team to program in the option to revert any changes, so that you don't wind up in a situation where your data is inaccurate. Data integrity is an underrated part of application security, and moving your data between multiple services increases the complexity by quite a bit.

# 3. Putting services on the public network

Microservices are often much simpler than the systems that they replace. This doesn't mean that they're inherently more secure. Those services are still subject to the same security vulnerabilities and hardening precautions that more full-featured services are. Fortunately, you can take a major step toward defense-in-depth by moving many of your microservices off of public networks. Servers running microservices are often less robust than monolithic servers, which can make them more susceptible to DDoS attacks. Alternatively, if you're using an auto-scaling architecture, public services are susceptible to malicious actors over-using resources, driving up your costs.

# 4. Passing data between services in plain text

If your team was smart and moved your services off the public network, you might think that you don't need to encrypt data passing between services. And hey, the truth is that you might be OK.  However, if you're passing data between services in plain text, anyone who can set up a listener on your service network can view all the information passing between services. Often times, that simply means that they can read all the information present in your system, even very sensitive information. It's better to be proactive and add another layer of defense here rather than reacting after a breach.

Adding TLS to a microservice is cheap and easy. It's not worth the increased risk of leaving it off.

## 5. Not monitoring your services

In my experience, service monitoring on microservices is skipped for two reasons. The first is that it's tedious. You need to add that monitoring to each service, and that takes time. Secondly, it doesn't feel like that monitoring is needed—the services are so simple in structure that an engineer will "just know" if something's going wrong.

The truth is that most of the time, we don't "just know" that something's wrong. We might figure it out eventually, but that takes time. The time that passes between something going wrong and finding out that something is wrong increases your risk profile significantly. Instead, adopt an Application Security Management solution like Sqreen to give you a heads up when something's going wrong with a service.

## 6. Not hardening the host OS

Many microservice teams build on top of tools like Docker. These kinds of container platforms use both a host OS and a service OS for a server. The service OS provides the context for the running service, and lots of teams make sure that this context is secure. Fewer teams think about also hardening the host operating system. This operating system is just as vulnerable as the service OS, but exploiting it can be far more lucrative. This is because an attacker who gains access to the host OS gains access to every container running inside it. Instead of having to try to infiltrate each of your services at once, they gain access to half a dozen at a time.

## 7. Relying on one external firewall

If you've done the right thing, and moved your services off the public network, you might think that you're fine. It doesn't feel like you have anything more to worry about. The truth is that you should still practice good network segmentation practices, even with

microservices. Some of your services will perform more or less critical functions. You should be thinking about how to segment those critical services from the rest of the network. Critical services shouldn't be accessible by services which don't rely on them. "Defense-in-depth" comes into play here again. You want to protect the perimeter, the application logic layer, and the driver layer.

## 8. Not encrypting data at rest

However you store your data, it's essential for it to be encrypted most of the time. Whether that's your database or your file storage, it shouldn't be possible for anyone who can access the hard drive to read your critical data without also having your security keys. Encrypting data at rest is a best practice for any security context. Microservices don't change that at all, but many teams neglect to do it.

## 9. Inconsistent logging

Because microservices are self-contained, most teams think that each service should be responsible for its own slice of the stack. This approach can come back to bite a team when there's an error in a critical part of the application. A developer investigates and starts digging into the logs, only to find that there aren't any. The team responsible for setting up logging for that service never got around to it.

Logging is just as important, both for application stability and security, on a microservice as it is on a monolith. If your team is beginning to adopt microservices, it's important to agree on how you'll approach considerations like logging. Then, stick to that approach across every service – no exceptions.

## 10. Providing individual services too much access

Another common pitfall when migrating to microservices is to treat all of them the same.

We've touched on this previously, but the truth is that all microservices aren't equal. Some need far more access to critical infrastructure—like databases—than others. Instead of simply allowing all services the same level of access to all architecture, intelligently limit their access. Not every service needs to be able to talk to your database or your file persistence layer.

By limiting what services can access, you simplify the attack surface if one service were to be compromised.

## Defense-in-depth is key

Ultimately, migrating to microservices shouldn't radically alter your security posture. You still want to intelligently segment your networks, and make sure that your services have multiple layers of security. You want to ensure the accuracy of data in your system. As you think about how to do that, you might find that microservices aren't actually a good fit for your use case. That's OK, too. Not every approach is right for every situation.

If you're smart about how you approach your security, migrating to microservices doesn't need to lead to increased risk for your team. Many of the risks you already understand; they just look a little different. For the ones that are different, the best approach is to spend time doing research so that you understand how they differ, then build systems designed to be secure from the ground up.

---

*This post was written by Eric Boersma. Eric is a software developer and development manager who's done everything from IT security in pharmaceuticals to writing intelligence software for the US government to building international development teams for non-profits. He loves to talk about the things he's learned along the way, and he enjoys listening to and learning from others as well.*