ECE459: Programming for Performance Winter 2023

Lecture 35 — DevOps: Operations

Patrick Lam & Jeff Zarnett *2022-10-29*

DevOps, but Operations

Monitoring and Pager Duty

Crypto Mining Issues

Logs with too much data / GDPR

https://sysdig.com/blog/massive-cryptomining-operation-github-actions/

Security, report to the bridge. Having an always-available service accessible over the internet makes security a very big concern. You can run some program with tons of security vulnerabilities offline and feel that the security problems can be managed, but when it's online the risk is enormous. All kinds of vulnerabilities are a problem, but I'll call out two of them as being especially bad: code execution/injection and data leakage (information exposure).

Code execution is exactly what it sounds like: attackers run their code using your platform. They can mess with your data, send spam or harrassing e-mails to your customers, take services without paying for them, mine bitcoin at your expense, and many other things. Sometimes these are company-ending events.

Information exposure is not only terrible for your company's reputation, but also against privacy laws and data protection regulations. A breach of the EU GDPR can get very expensive. See `https://www.enforcementtracker.com/` to find out which companies have recently gotten their wrists slapped or a huge fine. At the time of writing, the record holder is Marriott International Inc of the UK with a fine of 110,390,200 EUR which is roughly 171,898,556.93 CAD (using exchange rates from October 2020). Not exactly pocket change.

There are some companies out there that will check your code for libraries with versions having known security vulnerabilities. This is transitive, so if your app depends on some library that depends on a parsing library with a vulnerability, the vulnerability is noticed and reported. Each vulnerability is usually assigned a severity and the service may even suggest that updating from version X to Y will solve the problem. They can say this because they observe that a vulnerability reported in version X of a library is reported as fixed in version Y.

Sadly, when there is an updated version of a library, there may be breaking changes in it, so an upgrade might be more painful than just changing a version number and rebuilding. It may also take time for libraries to correct and release a fixed version of the library. In the meantime, you may need to implement some mitigation of your own, or just keep an eye open for when the patch is released. To support your wait for upstream, the vulnerability checks give you the ability to ignore or snooze the alert. Tempting as it is to do that and get on with other work, it's leaving the vulnerability in your code. Good practice would be for reviewers to discourage ignoring if it can be avoided.

As with the other parts of managing your application, like build and deployment, checking for vulnerabilities should be an automatic process as part of your build and release procedures.

Monitoring. Monitoring is surprisingly difficult. There are a lot of recommendations about what to monitor and what to do about it. We care about performance so here are a few things to think about:

- CPU Load

- Memory Utilization
- Disk Space
- Disk I/O
- Network Traffic
- Clock Skew
- Application Response Times

With multiple systems, you will want some sort of dashboard that gives an overview of all the multiple systems in a summary. The summary needs to be sufficiently detailed that you can detect if anything is wrong, but not an overwhelming wall of data. Then you do not necessarily want to pay someone to stare at the dashboard and press the "Red Alert!" button if anything goes out of some preset range of what is okay. No, for that we need some automatic monitoring.

Here's one way to think about it.

- **Alerts**: a human must take action now;
- **Tickets**: a human must take action soon (hours or days);
- **Logging**: no need to look at this except for forensic/diagnostic purposes.

A common bad situation is logs-as-tickets: you should never be in the situation where you routinely have to look through logs to find errors. Write code to scan logs.

It is very important to be judicious about the use of alerts. If your alerts are too common, they get ignored. When you hear the fire alarm in a building, chances are your thought is not "the building is on fire; I should leave it immediately in an orderly fashion.". More likely your reaction is "great, some jerk[1] has pulled the fire alarm for a stupid prank or to get out of failing a midterm." This is because we have been trained by far too many false alarms to think that any alarm is a false one. It's a good heuristic; you'll be correct most of the time. But if there is an actual fire, you will not only be wrong, you might also be dead.

Still, alerts and tickets are a great way to make user pain into developer pain. Being woken up in the middle of the night (... day? A lot of programmers are nocturnal, now that I think of it) because of some SUPER CRITICAL ticket OMG KITTENS ARE ENDANGERED is an excellent way to learn the lesson that production code needs to be written carefully, reviewed, QA'd, and perhaps run by a customer or two before it gets deployed to everyone. Developers, being human (... grant me some leeway here), will probably take steps to avoid their pain[2]. and they will take steps that keep these things from happening in the future: good processes and monitoring and all that goes with it.

References

[1] This is the PG-13 version of what I actually think.

[2] There is a great quotation to this effect by Frédéric Bastiat about how men will avoid pain and work is pain.