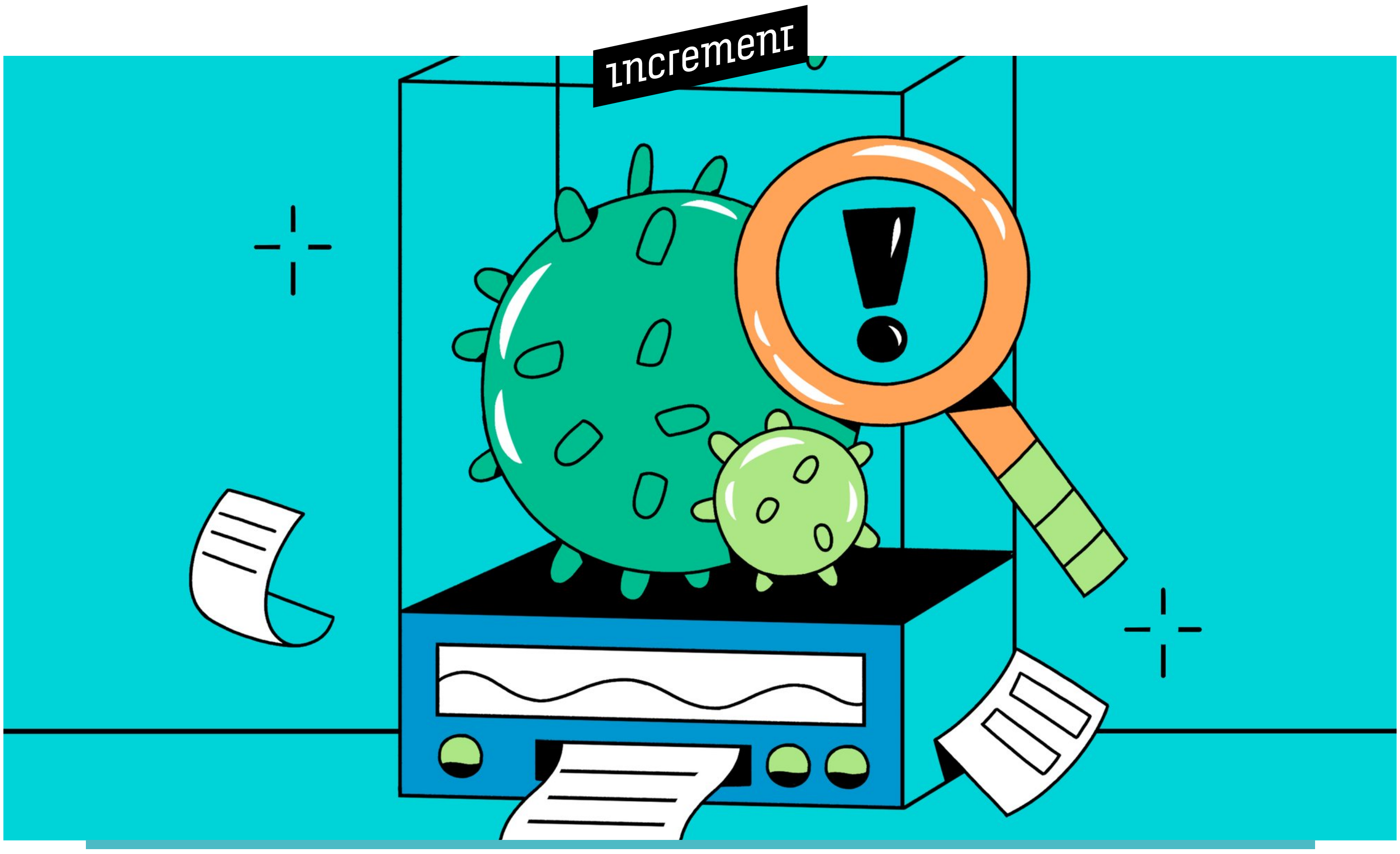


KEVIN RIGGLE

An introduction to approachable threat modeling

In this tale of two threat models, we explore how pairing our existing knowledge and experience with a few simple questions can help us build better systems and keep them safe.

PART OF
ISSUE 7
OCT 2018
Security

Threat modeling is one of the most important parts of the everyday practice of security, at companies large and small. It's also one of the most commonly misunderstood. Whole books have been written about threat modeling, and there are many different methodologies for doing it, but I've seen few of them used in practice. They are usually slow, time-consuming, and require a lot of expertise.

This complexity obscures a simple truth: Threat modeling is just the process of answering a few straightforward questions about any system you're trying to build or extend.

> What is the system, and who cares about it?
>
> What does it need to do?
>
> What bad things can happen to it through bad luck, or be done to it by bad people?
>
> What must be true about the system so that it will still accomplish what it needs to accomplish, *safely*, even if those bad things happen to it?

For the sake of brevity, I'll refer to these questions as Principals, Goals, Adversities, and Invariants. (And, in fact, that's the name of the rubric I'm about to present.)

A good threat model also includes a system diagram, but we leave that out of the rubric name—it's long enough already.

When we make a practice of asking these questions, try to answer them at least somewhat rigorously, and write down our answers somewhere other people can find them, threat modeling is truly revolutionary.

None of this requires specialized training or knowledge, nor does it require you to be a "security person." All that's required is curiosity and an interest in learning what kinds of bad luck and bad people have happened to other systems.

It also doesn't take all that long or all that many people to answer these questions. I'll usually give it an hour to be thorough, but even a 15-minute conversation one-on-one can produce something actionable.

At Akamai, where Brian Sniffen and Michael Stone initially developed this rubric (and many others, including myself, extended it), we used it every day to collect knowledge and communicate with each other, the rest of the engineering org, and the broader company, so that we could build better and safer products for the benefit of the company, our users, and ultimately the world.

Example one: The Dunning Incident

If you want to truly understand a system, study how it fails.

Let's say you're my hypothetical friend Alícia, who just got hired as an engineer for a small software as a service (SaaS) company, Kumquat.

Shortly after she joined, the company's app stopped sending out email verification and password reset emails. By noon, there were a number of frustrated user comments on the company and the CEO's social media feeds. The CEO flagged the issue to the engineering team, and Alícia, along with Shruti, an engineer of longer tenure on the team, sat down to investigate.

First, they checked the company's third-party email gateway service. Fortunately (or perhaps unfortunately), it appeared to be operating normally, and there were no known service interruptions listed on its status page. When they logged into its dashboard, they saw that the send queue was empty, its connection to the Kumquat backend was fine, and mail was being processed normally. But they were sending hundreds upon hundreds of messages with the subject line "ACT NOW: Reactivate your Kumquat account."

Acting on a hunch, Alícia IMed her friend Jayla on the business team, and quickly discovered that the account reactivation emails were part of a dunning campaign Jayla had organized.

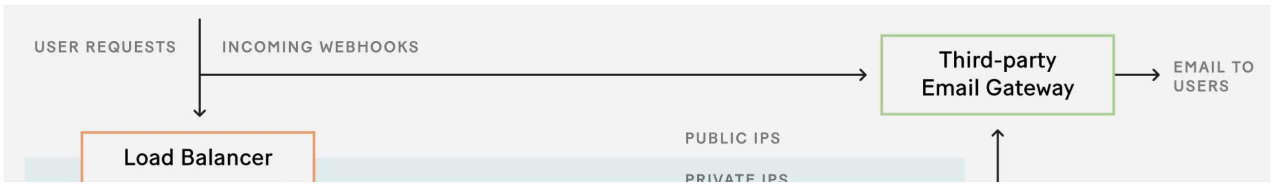
Previously, the company had only tracked active user growth and revenue in aggregate. Jayla, the company's first dedicated business analyst, had noticed that while active user growth kept increasing, revenue wasn't tracking linearly.

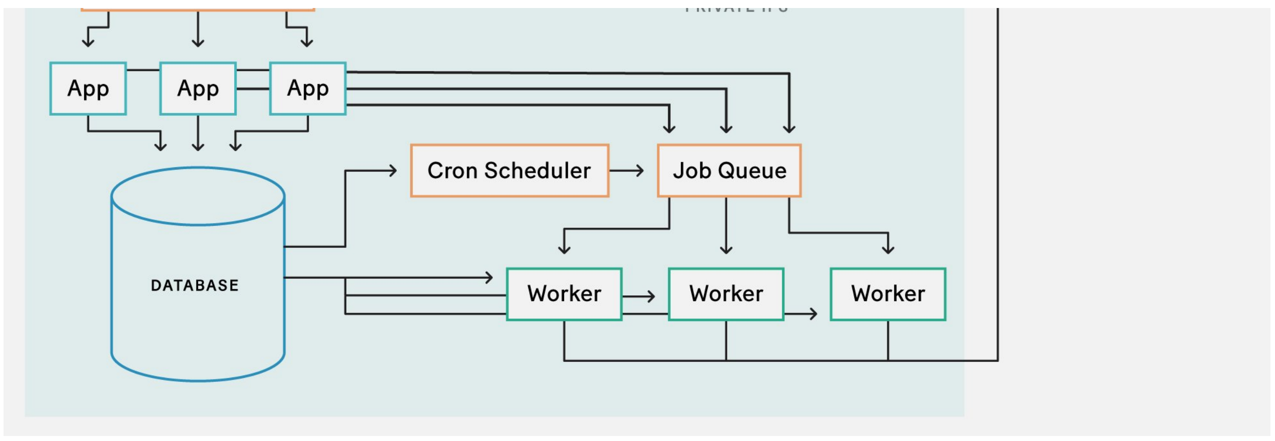
Analyzing the company's user data, she discovered that a number of customers' subscriptions had lapsed, most commonly because their credit cards couldn't be charged, but they were still being allowed to use the service. She had conceived and gotten buy-in to run a dunning campaign to encourage these users to update their payment information.

The emails were overloading something, but Shruti and Alícia weren't sure where they were coming from or what was being overloaded. Jayla said an engineer named Hana had done the work on the backend. They soon learned that Hana had written a job for the cron service to send the emails and scheduled it to run at 6 a.m.

Alícia realized that she didn't understand how the cron service fit into the system as a whole. When she joined the company, Shruti had given a talk on the overall architecture, and Alícia remembered seeing a diagram, but she hadn't retained much of it.

The engineers got together in a room with a whiteboard so they could talk through what was happening. Shruti drew the diagram:



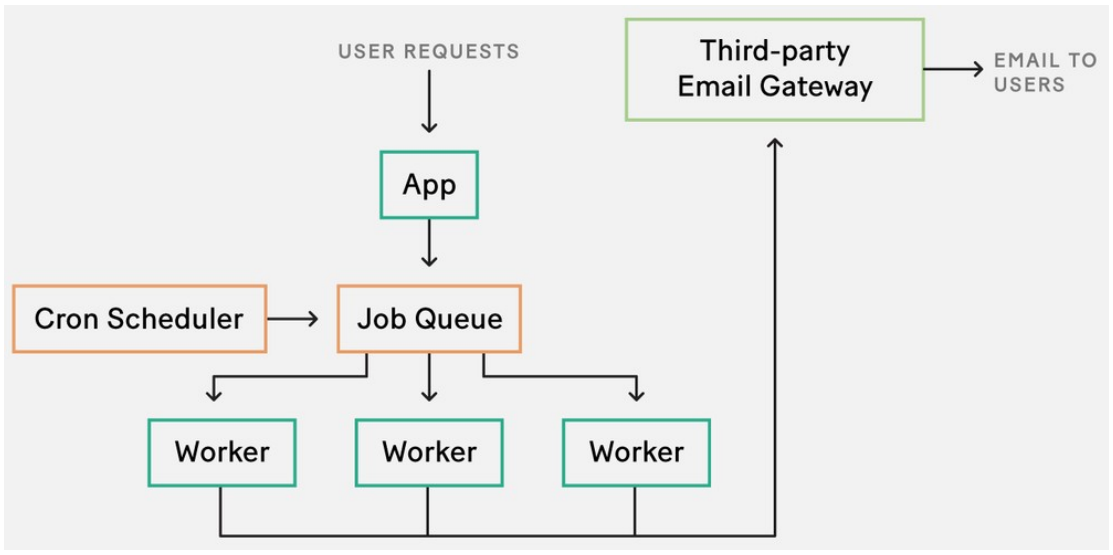


Then they began to talk through it.

"The job running on the cron service is pulling a list from the database of the users whose cards haven't been charged for at least two months. Then it's creating an email job for each one," Hana explained.

"And the cron service sends the email jobs to the job queue service," Shruti said. "The job queue service hands them out to workers, which use the email gateway provider's API to send the email."

Alícia added a simplified sketch of the bit of the system they cared about next to Shruti's diagram. "What have we been using the job queue service for until now?" she asked.



"Just sending emails," Shruti replied. "Email verifications and password resets."

Alícia wrote "**Principals**" on the whiteboard and underlined it. "So this is the system." She gestured at the simplified diagram. "And its users are verifying their email addresses, resetting their passwords, or updating their credit card information."

Principals

Users who want to…

– Verify their email address

– Reset their password

– Update credit card information

“And the **goal** of the system is to…”

“To deliver email reliably to them.”

Goals

– Deliver email reliably to users

“How reliably is reliably? Like timewise?” asked Alícia.

“Within five minutes,” said Shruti.

Alícia added that to the sentence.

Goals

– Deliver email reliably to users *within five minutes*

Alícia wrote down **“Adversities.”** “What are some bad things that can happen, either by happenstance or because someone made them happen?” she asked.

“Email can not get delivered,” Hana said. “More specifically, either the job queue or the email provider can get overloaded.”

Adversities

– Overload the job queue

– Hit email provider API rate limits

“The email provider could have an outage,” Hana added.

“That’s out of our control, though,” said Shruti.

“Let’s leave it out for now,” Alícia agreed. “We can always revisit it later.”

"Oh! Workers can hang and not get restarted," Hana said. "We used this job queue technology at my last job, and we occasionally had trouble with that. It only ever affected a few workers, though. It never caused an outage this bad. We'd just need to periodically go through and clean out stuck workers."

Adversities
– Overload the job queue
– Hit email provider API rate limits
– Silently hang workers/infinite loop

"How many emails have we been sending before today?" Alícia asked.

"Hundreds, but not thousands a day," Shruti estimated. "Sometimes we get a big spike in signups, but that's the steady state."

"And how many users were we attempting to contact with this dunning campaign?"

"About two thousand at once," said Hana. "Out of about eight thousand paying customers total."

"A quarter of our 'paying customers' aren't actually paying?" Alícia asked, incredulous.

"I know!" said Hana. "We thought we should fix that."

"We know we aren't exceeding the email provider rate limits — mail is going through and not getting queued," Shruti said. "Could the job queue have gotten overloaded?"

"These don't feel like huge numbers for a job queueing system in general, but it's possible," said Alícia. "It is a lot more jobs than you say the job queue usually handles."

"Believe me, the job queue is not great software," said Hana, with an attitude that suggested she had *seen things*.

"How many workers are we running?" asked Alícia.

"I haven't touched the job queue system recently. Let me check."

They were silent while Hana tried to pull up the job queue system's configuration file in source control.

"Oh," she said. "We didn't modify the default configuration for the job queue. Which means we only allocate one worker."

"And it's been fine this whole time?"

"We really didn't ask much of it until now," said Shruti.

"And..." Hana said, loading the job queue system's management page, "that one worker is stuck."

Alícia wrote "**Invariants**" on the whiteboard.

> **Invariants**
> – At least one unstuck worker?

"Is there a way for the job queue system to check whether workers are stuck and restart them?" she asked.

"The devs have been promising better worker health management in the next release for two years, but the release keeps slipping," said Hana.

"How many workers did you run at your last job?" asked Shruti.

"Maybe 20? But we also ran a lot more jobs."

"Let's up the number of workers to five and see if the problem reoccurs."

Alícia wasn't very happy with that. "Is there really no way for us to monitor the health of the workers, even if we have to restart them ourselves?"

"I'm sure there is," said Shruti. "But first we need to get mail working again."

"Oh, I already restarted the stuck worker and it looks like mail is sending again," said Hana. "It should take about twenty minutes to clear the backlog."

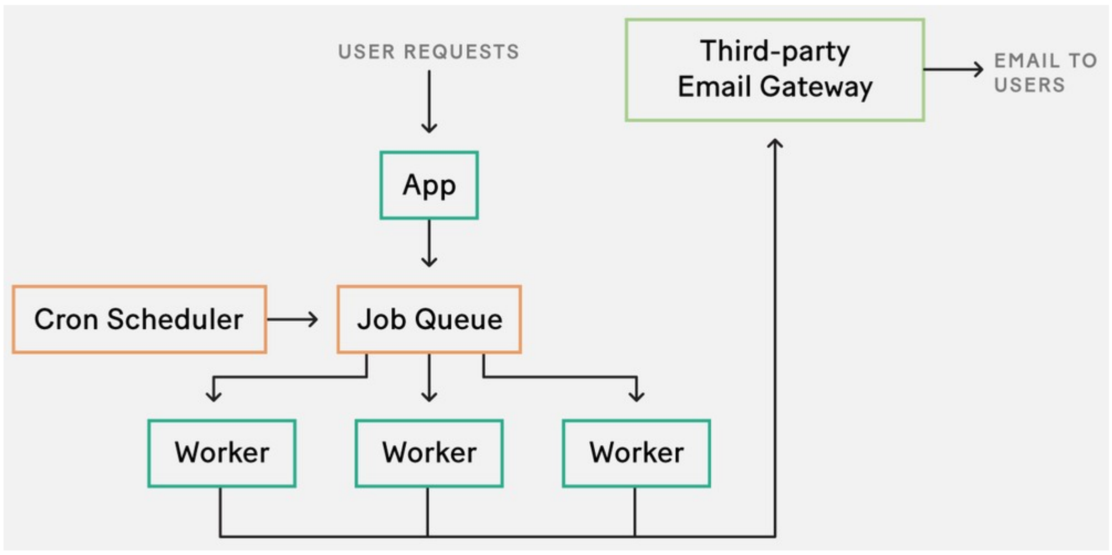
"Good. And then let's increase the number of workers?"

"On it."

"I'm sympathetic to your point of view, Alícia," said Shruti, "but we don't have time to do surgery on someone else's code, especially if upstream is struggling with it, too. Let's write this down and talk it over in the incident post-mortem meeting. For now, let's focus on what we need to do to get product out the door."

Alícia grudgingly agreed.

The final threat model:



Principals

Users who want to…

– Verify their email address

– Reset their password

– Update credit card information

Goals

– Deliver email reliably to users within five minutes

Adversities

– Overload the job queue

– Hit email provider API rate limits

– Silently hang workers/infinite loop

Invariants

– ~~At least one unstuck worker?~~

– More than one worker

If you're an eagle-eyed reader, you may have noticed that although this is threat modeling, there's no active adversary here (except perhaps Murphy, of Murphy's law: "Anything that can go wrong, will"). This is because anything an active adversary can do can also occur by happenstance. An active adversary can cause *very unlikely* happenstances to occur, and particularly sophisticated active adversaries can cause multiple very unlikely

happenstances to occur at once, but, generally speaking, a system that isn't resilient against happenstance can't possibly be resilient against active attack.

Here, it doesn't really matter if happenstance or adversary action caused the worker to hang. In either case, the system would still need at least one unstuck worker to safely continue to achieve its goals.

However, we now have a good understanding of the system Alícia is working with. Here's a more traditional threat modeling example where there is a potential active adversary.

Example two: Outgoing webhooks

Although Alícia wasn't formally a security engineer, she developed a reputation inside Kumquat as someone who knew and cared about security. A couple of months after the Dunning Incident, Yasmin, a product engineer, came to Alícia with a feature she was working on. They sat down in a meeting room with a whiteboard, and Yasmin laid out the project.

"Our app has a pretty traditional model-view-controller architecture. Until now, we've been fine doing all computation synchronously in the model. However, we've recently onboarded some new users who are much larger than our existing users, and this is starting to break, so we need to move long-running bulk actions to run asynchronously.

"We're going to use our existing job queue to run these actions outside of the app. But users' apps need a way to find out when these bulk actions are finished, so we want to get a webhook callback URL from the user and have the job post to it when the work is done. Hana mentioned that there might be some issues, and said we should talk to you."

"Did she say what she was concerned about?"

"How users can specify the URLs. But that's the whole point."

"I'm glad you came to talk to me," said Alícia. "She's right, it's potentially a problem that the user can specify these URLs. But there's another question here—whether our existing job queue system will meet your needs."

She explained what had happened during the Dunning Incident and showed her the threat model that the team had developed.

"Delivering webhooks within five minutes should be fine, since these jobs will be running on the job queue anyway," said Yasmin.

Alícia added that under **Principals**.

Principals

Users who want to…

– Verify their email address

– Reset their password

– Reset their password
– Update credit card information
– Run bulk actions and receive webhook notifications

“How long do these bulk actions take to run right now?” Alícia asked. “Will the webhook notifications be sent as part of the bulk action job?”

“Some of the bulk actions can take up to five minutes for our new large users. That’s the cause of the recent increase in request timeouts. Those pages have simply been unavailable to them,” said Yasmin. “I was planning to just post to the webhook URL from within the bulk action job, but I want to guarantee that the webhook gets sent within five minutes of the completion of the bulk action job. I don’t care how long the bulk action takes—that’s very dependent on how much data the customer has.”

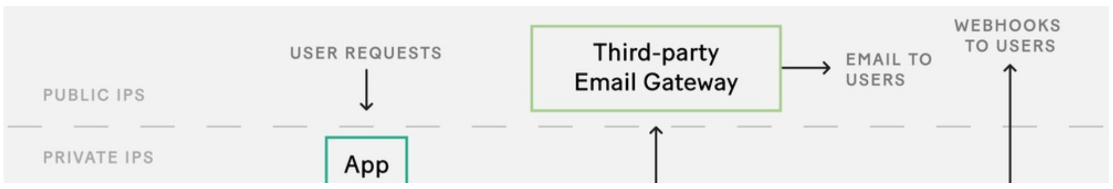
“Does the job queue system have the ability to queue multiple jobs that depend on each other?”

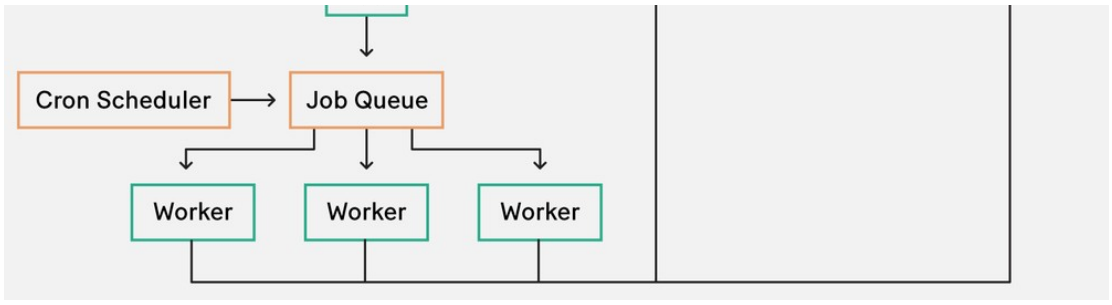
“It does.”

Alícia expanded the **Goals** list.

Goals
– Deliver email to users within five minutes
– Perform asynchronous bulk actions
– Deliver webhooks to users within five minutes of the completion of a bulk action

“Okay, good,” said Alícia. “It sounds like the existing job queue software will meet your needs. Now, on to the question of user-supplied URLs. It sounds like this is what you’re proposing.” She switched markers and sketched a couple additions to the system diagram.





"Adding outgoing webhooks to this system adds a new adversary power."

Adversities/Adversary Powers
– Overload the job queue
– Hit email provider API rate limits
– Silently hang workers/infinite loop
– *Specify outgoing webhook URLs*

"An adversary who can specify outgoing webhook URLs can specify, for example, a Kumquat internal IP address, the localhost address, or the API endpoint of our third-party email provider."

"But how would they know what those IP addresses are?"

"The localhost address is well known. They could also have some knowledge of our internal systems, if, say, they're an ex-employee. They could guess and get lucky. Or they could automate trying a bunch of webhook URLs until they find one that works."

"How do we stop that from happening? Could we try to filter the webhook URLs and reject any that are internal IP addresses?"

"I think it might be productive to approach it from a slightly different direction. I would express what we want this way," Alícia said, adding to the **Invariants** list.

Invariants
– More than one worker
– *Workers can only make outbound connections to external IPs*

"Concretely, in order for this system to be safe, it must be true that workers can only make outbound connections to external IPs."

Yasmin looked unconvinced. "What's better about phrasing the problem that way?"

"Well, we're not phrasing it as a problem. We're phrasing it as an invariant. We're not phrasing it as something that's wrong and therefore bad things happen, but as something that must be true in order for bad things *not* to happen. It's much easier to maintain invariants than to prevent problems. Did you ever play Whack-a-Mole as a kid?"

"That arcade game with the mallets and the plastic moles that you have to hit when they pop out of their holes?"

"Yeah. Sometimes I feel like I'm playing Whack-a-Mole when I'm considering problems rather than systems and invariants. If the invariant I need to uphold is that all the moles must stay in their holes, then I'm considering the system rather than the moles individually, and that opens up other solutions to the problem."

"You could just unplug the machine."

"Exactly!" They shared a laugh. "So much easier! Or, if that's not an option, I could get a bunch of friends with mallets and assign one to each mole. Or add some kind of mechanism on top of the holes to keep the moles from popping up. The fun of the game comes from constraining the problem sufficiently so that you have no choice but to chase the moles around. But I don't want to do that for work, and the system wouldn't be safe if I did."

"So, since we can't just unplug the job queue system here, how do we keep the moles from popping up — I mean, ensure that workers can only make outbound connections to external IPs? How can we constrain what connections workers make? Firewall rules?"

"Yes. Or a proxy for web requests that resolves the domain name to an IP address and filters them."

"Can we do that?"

"Yeah, I know some software. I can help you."

That's all the Principals–Goals–Adversities–Invariants rubric is: a few simple questions that we can use, alongside our existing knowledge and experience, to build better systems and keep them safe.

Getting into the habit of asking these questions and writing down the answers is the best way I know of to understand and communicate the systems we work with, the threats facing them, the tradeoffs we've chosen, and, ultimately, what must be true about them in order for them to be safe. It reduces security teams' frustration when communicating with engineering teams, it lets engineering and ops teams sleep soundly (uninterrupted by middle-of-the-night pages), and it lets business teams meet their goals and build better and safer products—for the benefit of our companies, our users, and ultimately the world.

Now, go forth and threat model.

Thanks to Aviv Ovadya, Cat Okita, Kep Petersen, and Nelson Elhage for their feedback on an early draft of this article.

ABOUT THE AUTHOR

Kevin Riggle lives in San Francisco and has worked on the security teams at Akamai and Stripe. When he's not trying to keep people safe on the internet, he enjoys hiking and gluten-free baking.

@kevinriggle

ARTWORK BY

Valeria Alvarez

behance.net/valerialvarez

TOPICS

Guides & Best Practices

CONTINUE READING