What caused this?

Ryan Eberhardt
March 9, 2022

Hello! 👋

- I’m Ryan! 🥭
- Graduated this past spring. Did undergrad systems track, coterm computer + network security track
- Now working as a Site Reliability Engineer at Coda
- Fun facts:
  - Went to community college before transferring to Stanford
  - I have two cats
  - I love doing pottery and photography
  - Not a fan of french fries
  - Once failed an NSA polygraph (oops)

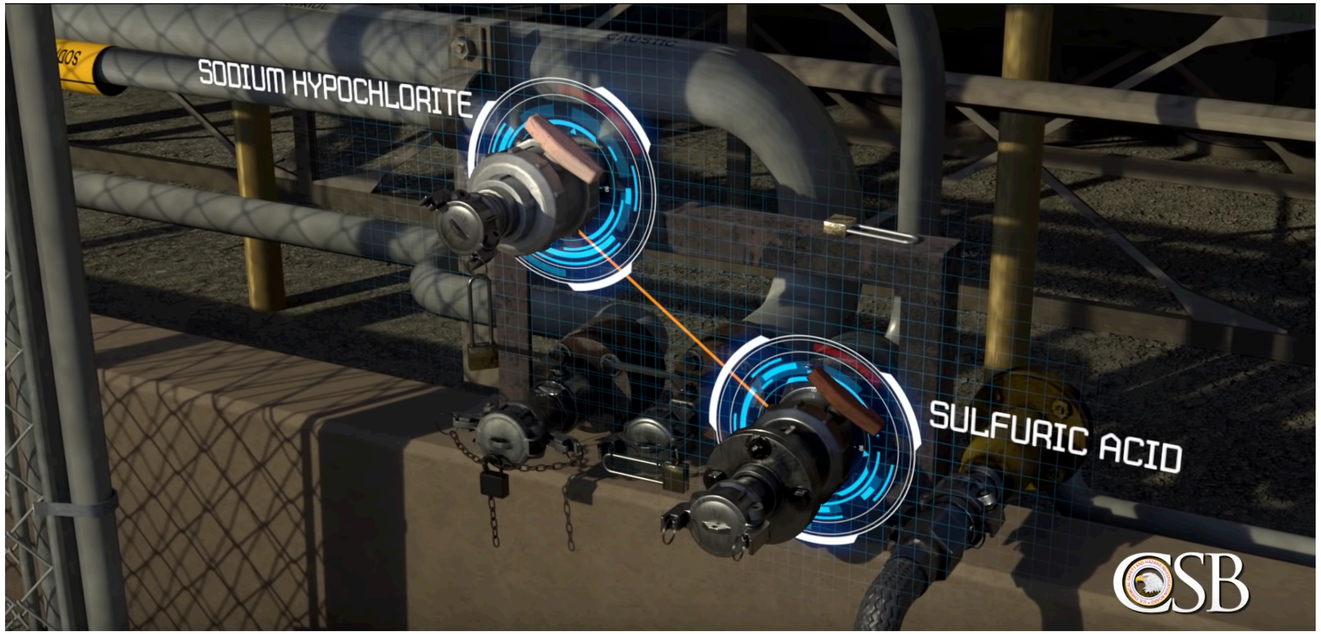
What happened here?

Colorado I-70 semi truck crash

- On April 25, 2019, a commercial truck driver lost control while going down a downhill mountainous stretch
  - The truck brakes failed
  - The driver failed to take a runaway truck ramp
- The resulting crash involved 3 other trucks and 12 cars, killing 4 people and injuring 10
- Who caused this?

MGPI 2016 chemical safety incident

- On October 21, 2016, a driver delivering a truckload of sulfuric acid connected his fill line to the sodium hypochlorite receptacle
- The resulting chlorine gas cloud spread over Atchison, Kansos, leading to the shelter-in-place of thousands of residents. At least 120 employees and members of the public sought medical attention
- Who caused this?



Source: CSB

solarwinds123

- In February 2021, SolarWinds infrastructure was compromised, allowing attackers to generate a software update that led to infiltration of many federal agencies and large corporations
- I don’t think a root cause has been publicly established, but it may have been related to an important password that was set to “solarwinds123” by an intern in 2017
  - SolarWinds CEO: "That related to a mistake that an intern made, and they violated our password policies and they posted that password on their own private GitHub account," Thompson said. "As soon as it was identified and brought to the attention of my security team, they took that down.”
- Who caused this?

Source: ZDNet

Who caused this incident?

~~Who~~ caused this incident?

What caused this incident?

solarwinds123

- What caused this incident?
- Simplistic interpretation: an intern set a bad password, then shared it on GitHub.
- But what else caused this? Why did this happen?
  - An intern was tasked with choosing a critical password
  - There was lacking oversight from more experienced people
  - Shared credentials weren’t evaluated and rotated
  - There were no systems in place to detect weak or leaked passwords
  - 2FA was not used

Source: ZDNet

MGPI 2016 chemical safety incident

- What caused this incident?
- Simplistic interpretation: the truck driver messed up
- But what else caused this? Why did this happen?
    - The two fill lines looked and functioned identically. Same type of connections, less than a meter apart.
    - The fill lines were not labeled where the hose connected.
    - There was no automatic emergency shutoff to limit the impact.
    - Manual shutoff valves were located in the areas affected by toxic chlorine release.

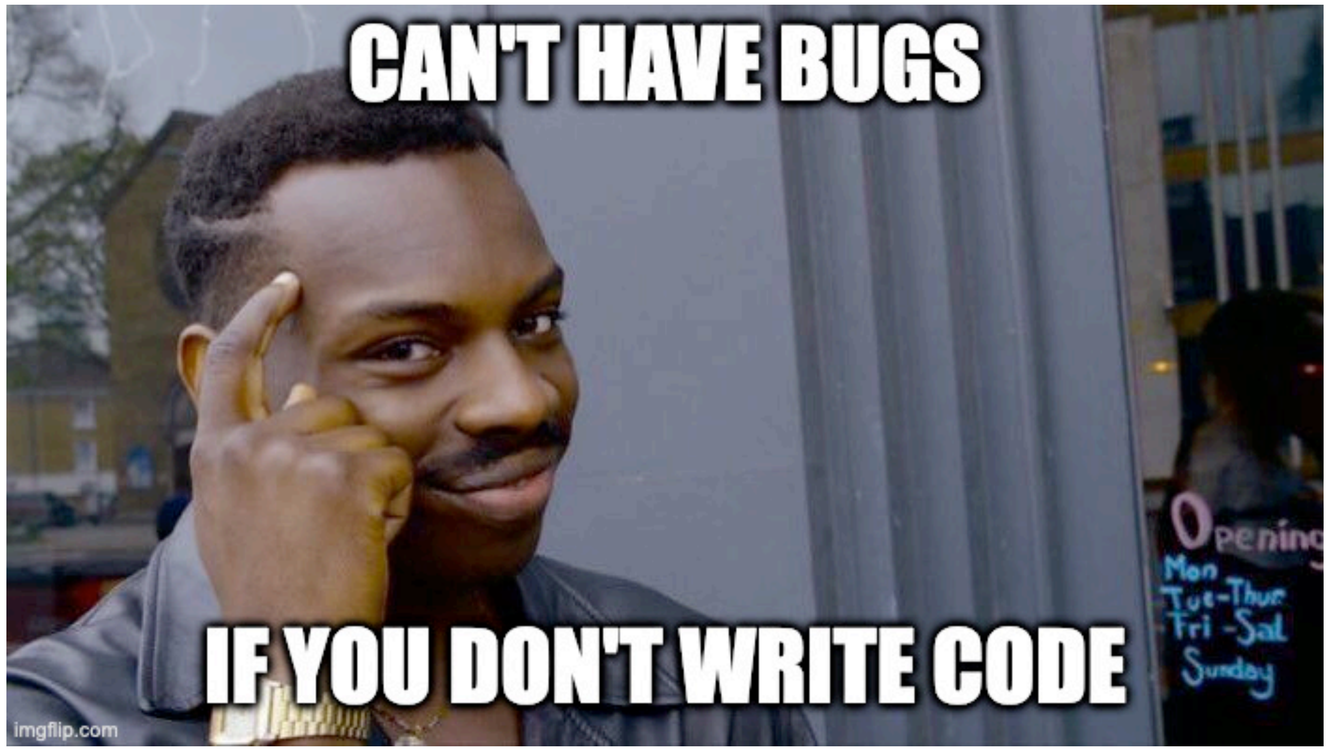
Source: CSB

Colorado I-70 semi truck crash

- What caused this incident?
- Simplistic interpretation: the truck driver messed up
- But what else caused this? Why did this happen?
- I don’t know much about this crash, and I’m not going to dispute the cause. But it’s helpful to think about contributing factors:
  - Driver education
  - CDL licensing processes
  - Truck maintenance
  - In-truck diagnostics and warnings

Systems thinking: what allowed this to happen?

- In incident investigation, it's critical to look at *all* contributing factors, and to be slow to assign blame
- Many will identify the primary cause. Case closed!
  - But this turns a blind eye to the circumstances that created the accident
  - Pinning the blame on an individual is an easy way to avoid collective responsibility
- Identifying and fixing the primary cause is not hard. But how can we ensure that this primary cause is never the primary cause of another incident?


CAN'T HAVE BUGS
IF YOU DON'T WRITE CODE
imgflip.com
Opening
Mon
Tue-Thur
Fri -Sat
Sunday

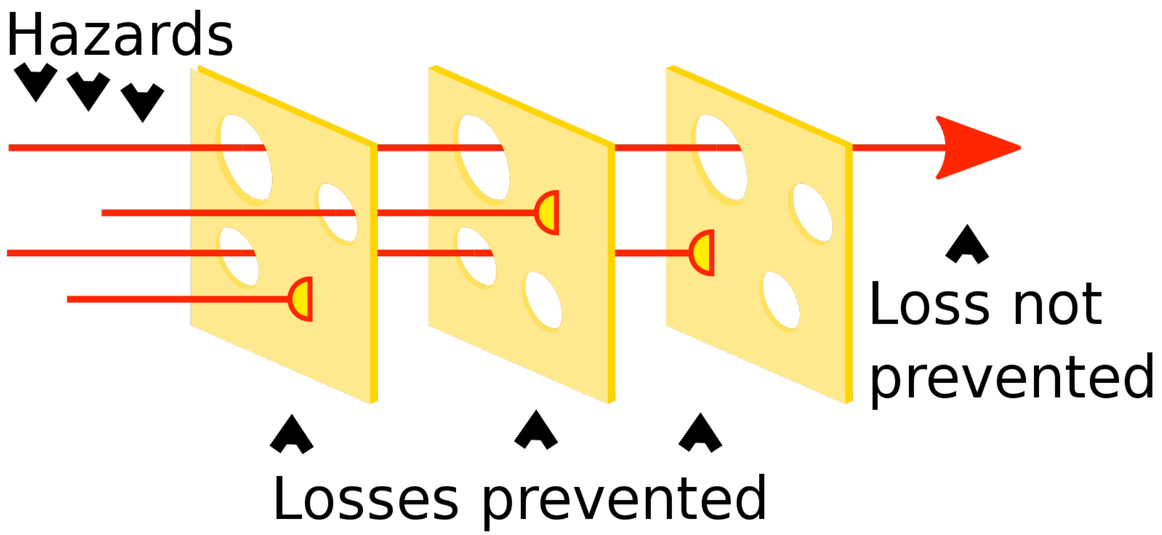
Effective process improvements

- How can we ensure that this primary cause is never the primary cause of another incident?
  - Often no good answer
  - If your root cause is “someone wrote a bug,” you aren’t going to find anything practical that ensures no one ever writes bugs again
- More an art than a science
  - Need to come up with solutions that make it hard to mess up *and also* easy to do what’s right
  - Requires a lot of creativity and good design
  - Also, you don’t have to be perfect! (That’s impossible.)

Swiss cheese model

Swiss cheese model

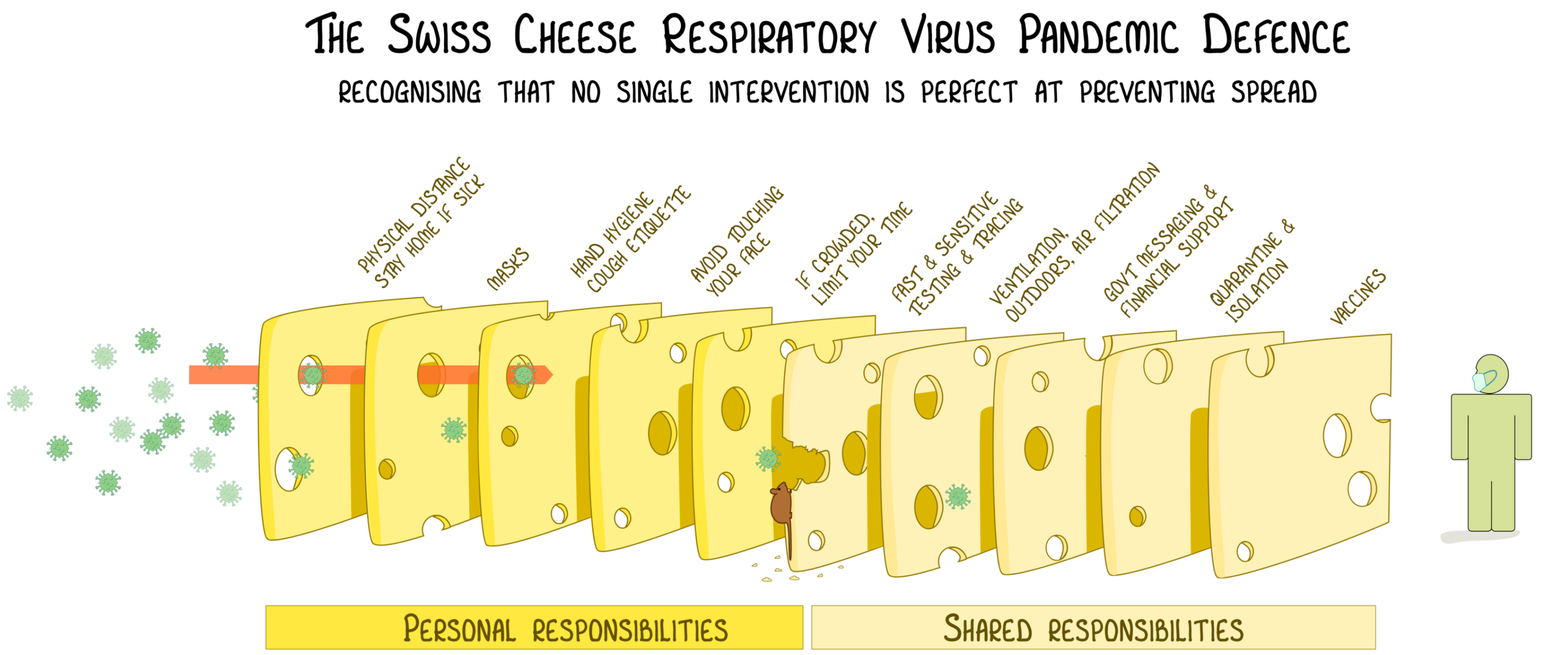
- If you have a thin slice of Swiss cheese, there will be holes in it
- But if your cheese is thicccc, the holes in one layer will be covered by cheese in a different layer



from wikipedia

Swiss cheese model

- If you have a thin slice of Swiss cheese, there will be holes in it
- But if your cheese is thicccc, the holes in one layer will be covered by cheese in a different layer



[from wikipedia]

Swiss cheese model

- If your only slice of cheese is “employees do things right and don’t make mistakes,” then I’m sorry, you’re screwed
- But we don’t need to have any one perfect layer! As long as incidents don’t make it all the way through the cheese
- When they do make it all the way through and bad things happen, we need to evaluate what went wrong, how we can close certain gaps, or add more layers

More interesting incidents

Mozilla NSS signature verification

- Cryptographic signatures:
  - Thea has a private key (which only she knows) and a public key (which everyone knows)
  - Given a message, she can use her private key to generate a signature:
    S(*private key*, *message*) ➡ *signature*
  - Anyone else with her public key can verify that the message was signed with her private key
    V(*public key*, *message*, *signature*) ➡ *accept* or *reject*

Mozilla NSS signature verification

- NSS verification
  - First create a VFYContext struct storing the public key, signature, algorithm, any other necessary info

```
struct VFYContextStr {
    SECOidTag hashAlg; /* the hash algorithm */
    SECKEYPublicKey *key;
    union {
        unsigned char buffer[1];
        unsigned char dsasig[DSA_MAX_SIGNATURE_LEN];
        unsigned char ecdsasig[2 * MAX_ECKEY_LEN];
        unsigned char rsasig[(RSA_MAX_MODULUS_BITS + 7) / 8];
    } u;
    unsigned int pkcs1RSADigestInfoLen;
    unsigned char *pkcs1RSADigestInfo;
    void *wincx;
    void *hashcx;
    const SECHashObject *hashobj;
    SECOidTag encAlg;    /* enc alg */
    PRBool hasSignature;
    SECItem *params;
};
```

★ union can store different types of signatures

Mozilla NSS signature verification

- NSS verification
  - First create a VFYContext struct storing the public key, signature, algorithm, any other necessary info
  - The max signature length that can be stored is 16384 bits. Way more than necessary. Should be good
  - What happens if you make a signature that's bigger than that?

Implementation of vfy_CreateContext:

```
sigLen = SECKEY_SignatureLen(key);
if (sigLen == 0) {
    /* error set by SECKEY_SignatureLen */
    rv = SECFailure;
    break;
}

if (sig->len != sigLen) {
    PORT_SetError(SEC_ERROR_BAD_SIGNATURE);
    rv = SECFailure;
    break;
}

PORT_Memcpy(cx->u.buffer, sig->data, sigLen);
break;
```

⭐ Ensure there is a signature to verify

⭐ Ensure the sig length is consistent (but not that it's within bounds)

💣 Copy the signature into the struct!

```
struct VFYContextStr {
    SECOidTag hashAlg; /* the hash algorithm */
    SECKEYPublicKey *key;
    union {
        unsigned char buffer[1];
        unsigned char dsasig[DSA_MAX_SIGNATURE_LEN];
        unsigned char ecdsasig[2 * MAX_ECKEY_LEN];
        unsigned char rsasig[(RSA_MAX_MODULUS_BITS + 7) / 8];
    } u;
    unsigned int pkcs1RSADigestInfoLen;
    unsigned char *pkcs1RSADigestInfo;
    void *wincx;
    void *hashcx;
    const SECHashObject *hashobj;
    SECOidTag encAlg;    /* enc alg */
    PRBool hasSignature;
    SECItem *params;
};
```

overflow

💣💥 Contains function pointers

Mozilla NSS signature verification

- Impact: RCE on almost anything using NSS for signature verification, including Thunderbird, LibreOffice, Evolution and Evince
- Classic buffer overflow
- How can we prevent this?
  - Code audit / review
    - Bug bounty programs to incentivize public review
  - Static analysis: identifying memcpying untrusted input without bounds checks
  - Dynamic analysis: Testing with ASan to identify memory corruption
  - Fuzzing to generate pathological test cases

Mozilla NSS signature verification

- It turns out that Mozilla already had all of these in place
  - Code audit / review
    - All code is open source, and Mozilla has a great bug bounty program
  - Static analysis
    - Coverity has been monitoring since 2008
  - Dynamic analysis
    - NSS has an extensive test suite, which is run with ASAN enabled
  - Fuzzing to generate pathological test cases
    - NSS uses libFuzzer extensively, and is included in oss-fuzz

Mozilla NSS signature verification

- This postmortem explores the limits of commonly embraced solutions
  - I.e. it identifies holes in the layers of cheese we usually recommend as a fix
  - Read the full postmortem here
  - Key questions: how did this still happen, despite everything in place?
- NSS is fuzzed extensively, which should have caught the problem in theory
- Immediate issues:
  - There was an arbitrary limit of 10,000 bytes on fuzzed input. This prevented fuzzing of vfy_CreateContext from uncovering the problem
  - Missing end-to-end testing: There was another function (QuickDER) that could have generated a huge signature given a small fuzzer input, but as these components were fuzzed independently, that problem was never observed
  - Misleading metrics: Coverage metrics are reported as the combined coverage for the separate fuzzing campaigns. Does not give a good picture of coverage of the system as a whole
- More questions:
  - Why are we still using C?
  - How can we make developers more aware of these pitfalls? (Impetus for starting this class!)

Coda: deployment of bad config

- At Coda, we have a process for deploying code, and a separate process for deploying dynamic configuration
  - Configuration is a JSON file used to quickly enable/disable features, gate access to features, control rate limits, etc.
- In January, we made a change to a configuration setting controlling our external integrations
  - This contained an invalid setting that caused runtime errors in the majority of external integration requests
  - The change deployed successfully. Within a few minutes, our alarms started firing indicating that requests were failing
  - The cause of the outage was not immediately obvious to SRE — no recent code deploys, and no clear errors in our main logging source pointing to an invalid config. Took 25 minutes to identify and revert
- What went wrong?

Coda: deployment of bad config

- Before deploy
  - We committed config change without realizing it was invalid
    - In response, we made config validation more robust, so that invalid configurations are rejected early on instead of causing runtime errors later on
- During deploy
  - Deployment progressed from staging servers to production, even though problems were immediately evident in staging
    - In response, we added additional tests to the deployment process to ensure critical functionality is unaffected by config change
  - After it was deployed to staging, we approved rollout to prod without testing thoroughly
    - Educational component: warn engineers that config changes are only lightly tested, and thorough manual testing is needed
- After deploy
  - The root cause was not immediately obvious, so remediation took much longer than it should have
    - Improve logging so that errors go to a central location, and error messages are helpful

Cloudflare 2019 catastrophic backtracking outage

- One of my favorite postmortems of all time
- Among other things, Cloudflare acts as a reverse proxy, sitting in front of customers' web servers and filtering out malicious traffic
- WAF (web application firewall) product: uses regexes to identify and block malicious requests
- On July 2nd, 2019, Cloudflare had a 27-minute global outage
  - All customers' systems also went down as a result

Catastrophic backtracking

- (live demo)

Cloudflare 2019 catastrophic backtracking outage

- 1:42pm, engineer deploys a new regex to detect malicious traffic
- Three minutes later, widespread alarms started going off
- At 2pm, performance team was able to pull live CPU data showing that the WAF was responsible. Another team saw error logs confirming this
- Cloudflare moved to turn off the WAF, but couldn’t reach the internal control panel to do so due to the outage
- Also couldn’t reach other internal services such as Jira or the build system
    - There was a bypass mechanism to reach these without Cloudflare, but people didn’t know or remember how to use it
- At 2:07pm someone managed to get in to turn off WAF
- By 2:52pm, team felt confident that the problem was understood and fixed, and re-enabled WAF

Cloudflare 2019 catastrophic backtracking outage

- What went wrong?
- This *awesome* postmortem lists 11 different factors:
  - An engineer wrote a regular expression that could easily backtrack enormously.
  - A protection that would have helped prevent excessive CPU use by a regular expression was removed by mistake during a refactoring of the WAF weeks prior—a refactoring that was part of making the WAF use less CPU.
  - The regular expression engine being used didn't have complexity guarantees.
  - The test suite didn't have a way of identifying excessive CPU consumption.
  - The SOP allowed a non-emergency rule change to go globally into production without a staged rollout.
  - The rollback plan required running the complete WAF build twice taking too long.
  - The first alert for the global traffic drop took too long to fire.
  - We didn't update our status page quickly enough.
  - We had difficulty accessing our own systems because of the outage and the bypass procedure wasn't well trained on.
  - SREs had lost access to some systems because their credentials had been timed out for security reasons.
  - Our customers were unable to access the Cloudflare Dashboard or API because they pass through the Cloudflare edge.

Cloudflare 2019 catastrophic backtracking outage

- Follow-up:
  - Re-introduce the excessive CPU usage protection that got removed. (Done)
  - Manually inspecting all 3,868 rules in the WAF Managed Rules to find and correct any other instances of possible excessive backtracking. (Inspection complete)
  - Introduce performance profiling for all rules to the test suite. (ETA: July 19)
  - Switching to either the re2 or Rust regex engine which both have run-time guarantees. (ETA: July 31)
  - Changing the SOP to do staged rollouts of rules in the same manner used for other software at Cloudflare while retaining the ability to do emergency global deployment for active attacks.
  - Putting in place an emergency ability to take the Cloudflare Dashboard and API off Cloudflare's edge.
  - Automating update of the Cloudflare Status page.

Matrix April 2019 security incident

- Matrix is an open standard and communication protocol for secure, decentralized, real-time communication
  - Used for p2p chat amongst other applications
- On April 11th, Matrix discovered an attacker had gained root access to much of their production network
- Attacker had first compromised the Jenkins build server using an old known RCE vulnerability
- On that server, attacker set a trap that would record any SSH keys forwarded by other users. (For convenience, people will sometimes forward all their SSH keys to trusted hosts so that they can easily log into other hosts from there.) Attacker managed to get SSH keys of more team members
- From there, attacker spread across the network, gaining access to at least 19 more hosts including the production database, exfiltrating data and planting backdoors for later use

Matrix April 2019 security incident

- What went wrong?
- Huge list of remediations in the official postmortem. A sampling here:
  - SSH practices
    - SSH agent forwarding should be disabled
    - SSH should not be exposed to the public internet
    - SSH keys should grant minimal access (principle of least privilege)
    - Use two-factor authentication
    - Monitoring of SSH keys
  - Network architecture
  - Keeping patched: instituting a formal regular process for deploying security updates
  - Intrusion detection: have automated systems to detect unusual behavior
  - Avoiding temporary measures which last forever: This was “temporary” infrastructure from 2017 that they didn’t prioritize closing out
  - *So* much more. Read the postmortem!!

Open problem: Coda outages due to database overload

- Coda has had a few minor outages / degradation due to “perfect storms” that overwhelm the database
- One problem that happens sometimes:
  - Engineer writes a database query for a new feature, ensure it performs well, ships it
  - Some time later, more data is accumulated or schemas change, and query performance qualitatively changes (e.g. Postgres switches to a query plan that doesn’t use indexes). No one notices
  - Something else generates a load spike, and that combined with the degraded queries cause things to fall over
- How do we prevent this from happening?

The blame game

The blame game

- Was it an accident? (Was there malicious intent?)
- Are you on the same team? (Are your interests aligned?)
- If yes, then what's the use in pointing fingers?
  - If it was an accident, then it could have happened to anyone, yourself included
- As a team, need to build a strong safety net so that there is room to make honest mistakes without catastrophic effects
  - If everyone is always terrified of making changes that might break things, no one will ever do anything
- Also need to build a supportive culture so that people are not afraid to raise concerns
  - If you don't identify and remedy potential holes in the cheese, your cheese will remain holey

Parting thoughts

Parting thoughts

- Reframe: “who caused this?” ➡ “what caused this?”
  - What could have prevented this?
- Look for opportunities to add layers to your cheese. Generally good things:
  - Code review
  - Static analysis (linters, catching problematic code patterns)
  - Comprehensive testing, dynamic analysis
  - Fuzzing
  - Failure injection and chaos testing
  - Logging and metrics, having good error messages
- Focus on the systems and processes, not the people. Everyone makes mistakes