Principles for Developing More Secure Systems

Eleanor Saitta @Dymaxion

GoTo Chicago 2019

Wait, you want me to start with what?

Security is neither checklists nor infinite unrelated problems

Problems happen when you don't:

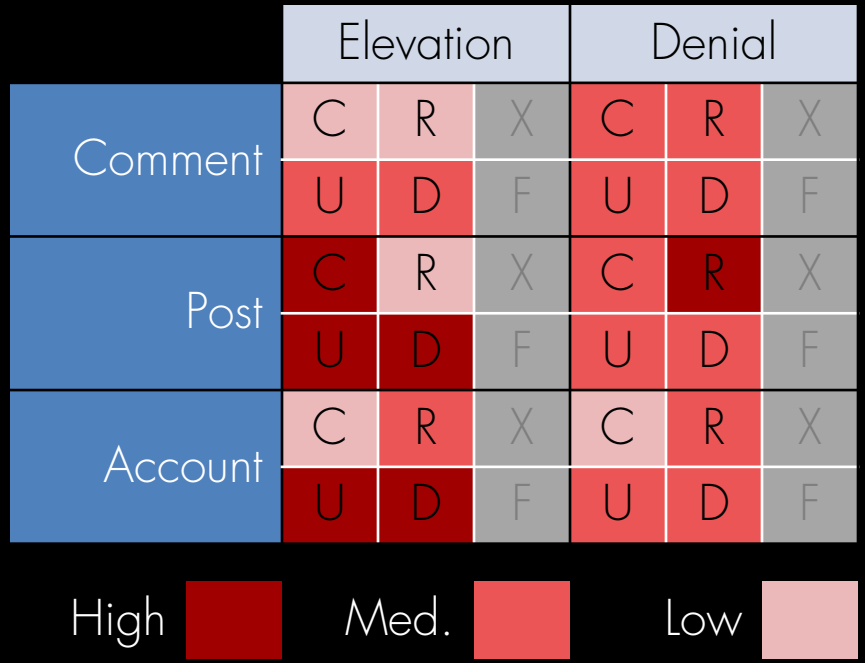
- Understand your users' goals and context
- Understand the traps your tech stack is setting
- Combine your stack's primitives to meet security objectives
- Use processes that catch errors in the previous items
- Make decisions at the edge where there's enough context

Principles give us ways reason about the structure of our work, even when we don't yet understand the problem

Security is the set of activities that reduce the likelihood of a set of adversaries successfully frustrating the goals of a set of users.

Threats:

A threat is a negative outcome in the human process a system is built to support. They come in positive (*elevation of privilege*) and negative (*denial of service*) version.



Now that we know what can go wrong with our system, we can create a list of **security objectives**:

When [person] attempts to [make a bad thing
happen], the system will respond by [doing something sensible].

We draw the set of system responses from another controlled vocabulary:

prevent the attacker from launching the attack
detect and log the attack
detect and alert some actor about the attack
rate limit the attack, as in DDoS reponse
thwart the attack from being successful

To build a secure system:

- Understand your users' goals and context and the strategies they will adopt in using your tool to frustrate their adversaries
- Understand the invariants your system must maintain and behaviors you must enable to let their strategy succeed
- Understand lower-level behavior in your system that you must act to preserve or to eliminate vulnerability from
- Deploy available primitives in a way that satisfies the above
- Test your system to find the places you've failed or where contexts, strategies, or underlying systems have changed
- Iterate

Invariants

Availability

Confidentiality

Integrity

Invariants

Accuracy
Adaptability
Assurance
Availability
Capacity
Concealment
Compartmentalization
Confidentiality
Continuity
Control
Completeness
Cooperation
Correctness
Deployability
Deniability
Depth
Deterrence
Efficacy
Efficiency
Integration
Integrity
Interoperability
Goodwill
Mobility
Nonrepudiation
Observability
Precision
Predictability
Redundancy
Resilience
Responsiveness
Scalability
Simplicity
Simultaneity
Survivability
Synchronization
Trust
Unlinkability
Velocity

Invariants

Accuracy
Adaptability
Assurance
Availability
Capacity
Concealment
Compartmentalization
Confidentiality
Continuity
Control
Completeness
Cooperation
Correctness
Deployability
Deniability
Depth
Deterrence
Efficacy
Efficiency
Integration
Integrity
Interoperability
Goodwill
Mobility
Nonrepudiation
Observability
Precision
Predictability
Redundancy
Resilience
Responsiveness
Scalability
Simplicity
Simultaneity
Survivability
Synchronization
Trust
Unlinkability
Velocity

Efficacy

Product Design

- If you build products and you have a design team, they must be in the security loop
- Design determines the security outcomes your product might help users reach
- Engineering determines if they actually reach them
- If your team is big enough, treat it as its own discipline
- If not, bring consulting security designers in

Design for Human Error

Design for Human Error

- Your staff click on things for a living
- Getting them to not click on one other thing is nigh-impossible if they're going to work at pace

- Do not try to yell^Wcajole^Wtrain them out of this

- Instead, making clicking on things safe

- If you can't, consider goat farming
- The goats will still screw up, but it's funnier

Correctness

- Poorly-tested software always has more bugs that may have security impact
- Undocumented workflows and business rules can't be tested for correctness

- If you don't have functional test suites, docs, and good QA practices, get started on them at the same time as you start on security

Mitigations Always Fail

Kill Bug Classes

- Security engineering changes that don't involve killing bug classes are emergency response work

- …unless those changes kill traversal instead

Confidentiality

You know this one, right?

Encrypt all the things!
Authenticate all the things!
Authorize all the things!

…eliminate side channels for all the things?!

Integrity

This one is familiar too, right?

Don’t let randos manipulate your data!
HMAC all the things!
Don’t use fucking Mongo for data that needs integrity!

Limits to Invariants

Availability

So, how big *is* your AWS bill?

You can mostly solve this by giving various people lots of money
and massively complicating your architecture,
but have you actually tested it?

At least know where you single points of failure are
Make distributed denial of service someone else's problem

Nonrepudiation

Strong authentication means U2F/WebAuthN
If your vendor doesn't offer that, start yelling

No, there are literally no substitutes

Also, did you log enough data to reconstruct user actions?
Can you strongly link them to a user principle?
Yes, staff ssh access to prod hosts counts

Unlinkability

You don’t want anonymity — you don’t know what it means

X cannot be linked to Y
May be a statistical probability; making a risk call for users

Metadata tells most of the story

Don’t lie to yourself (or your users)

Trust

Know what you're trusting and don't lie to yourself

- What's your service dependency graph?
- Which 3rd-party services are root equivalent for you?
- Third-party library auto-pulls in deployment are terrible

Actively manage trust chaining:

- Signed commits and code reviews verified during build
- Binary transparency and signature validation on execution
- TPM attestation to firmware hashes to receive local disk key
- Continuous attestation to system integrity to be a deployment target

You don't control the user's computer, and cannot assert what it will do

Observability

Can you tell if you've been owned?

Centralize all logs. Really, all of them. Make anything that doesn't log log.
Then when you get the bill, figure out what you can live without

You care about *events*, not *lines*. Structure is your friend.
If you cannot correlate it, it does not exist
Time is a lie

If you do not look at it, it does not exist
Logging and alerting workflow is hard
Good indicator of compromise queries are hard, but do something
You cannot go back and log stuff last month when you got owned
Machine learning is mostly bullshit

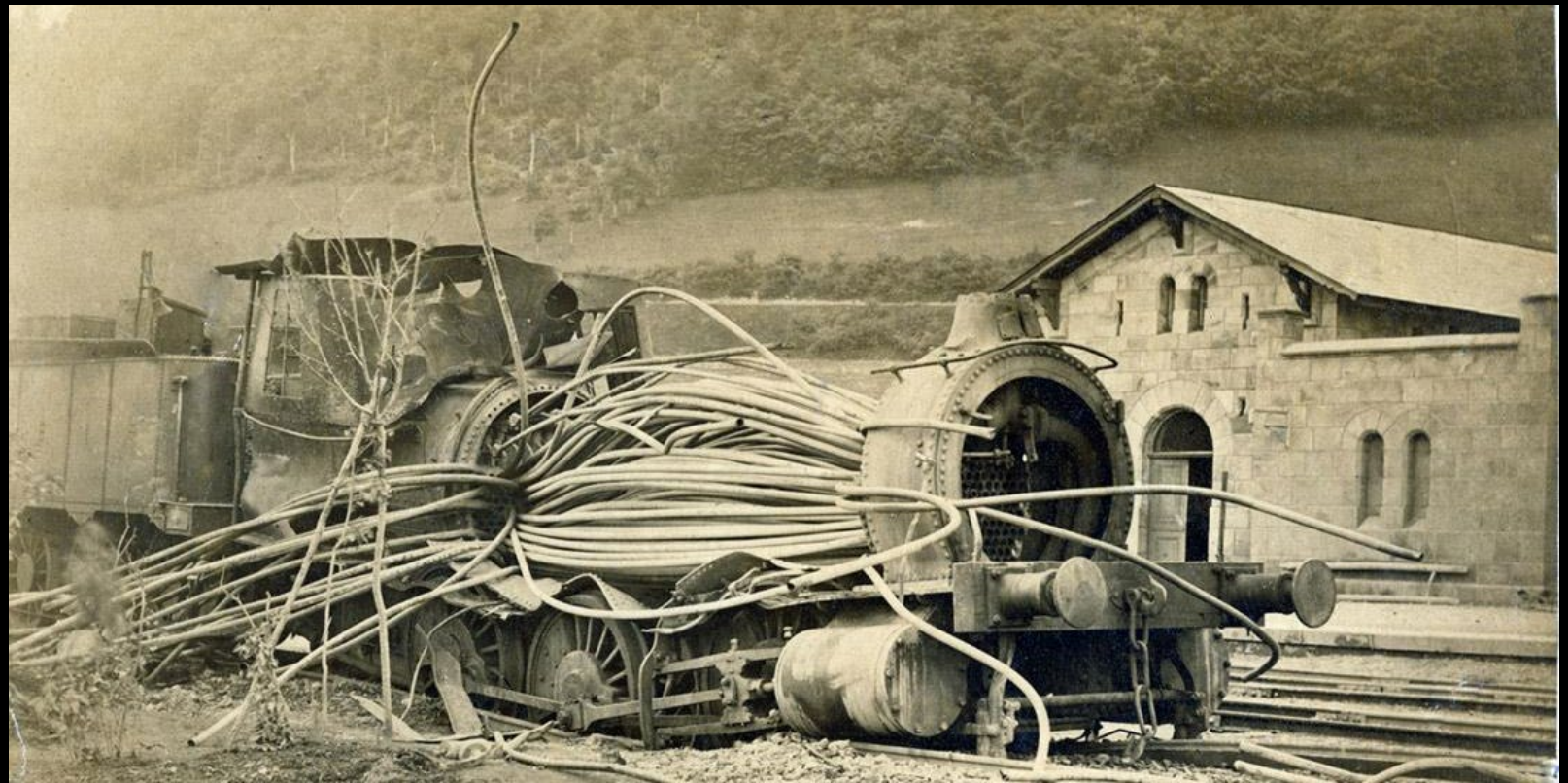
Scalability

Does a 10x scale jump break your security guarantees?

Limits

- Autoscaling is great for fast data exfiltration
- Build infrastructure-level rate limits when load is predictable
- Internal high-trust human interfaces are a key place to put in limits
- Humans notice slow systems and complain — free alerting!
- Key expansion functions can be used to slow index lookups

Predictability



Programmatic Infrastructure

- Repetitive tasks lead to errors
- So does managing state simultaneously via different interfaces/formats

- Keep state in one place
- Ensure you can move cleanly between configurations
- Orchestrate deployment to individual management systems

- Done right, allows:
  - More aggressive segmentation/response
  - Automated cross-checks for rule enforcement
  - Audit trails/two-person-rule infrastructure control (peer-opsing)
  - Faster disaster recovery

Don’t Patch

- Patching deployed systems leads to drift
- Treat disk images as immutable and redeploy
- If redeploying is too slow, fix that
- Get good at rebooting the world — win/win
- Run all prod hosts with read-only system disks

Are you fighting security problems that shouldn't exist?

Try **Langsec** today!

- Do your parsers accept things they shouldn't?
- Are your random numbers not random?
- Do your buffers overflow?

Dozens* of people like you have found a solution!

Compartmentalization

- Horizontally between systems/instances
- Vertically within a system
- Temporally across execution lifetimes

- Service-discovery-driven granular firewalling and per-instance credentials
- Process namespaces, syscall ACLs, limited DB roles
- Fast-roll on VMs not doing slow batch work and dropping privileges in-process

Separation of Concerns

- Engineers are human; the more they have to think about at once, the more mistakes they make
- Split your system into layers
- Let each layer protect engineers against decisions
- At a minimum, physical, infrastructure, application environment, application

Compartmentalize redux

- Beyondcorp is cool! Also you're not Google
- Yes, build stuff like it's Internet-facing
- But maybe don't actually put it on the Internet

- Users have different risk levels, and their machines shouldn't all be hardened identically
- Many small company users expect local admin rights
- Easier to sell hardening only some boxes
- Compensate those folks with better IT service
- Give folks second machines for very high-exposure tasks

- For the love of god harden your office infrastructure

Simplicity

- Capability is a liability
- Delete data, delete code, shred servers, eliminate complexity
- Combinatorial complexity will eat your children
- Never bring a Turing machine to a parser fight
- Adversaries love your excess capacity
- Every system you don't have is one that can't break something you actually care about

Resilience

All of your systems are made of humans

All resilience comes from humans

You cannot automate fixes to problems you do not already see

Design your systems to enable the humans

Adaptive capacity means you need slack in your teams

Resilience

The application environment shall not contain footguns
e.g. SQLi vulnerable query methods, untyped variables, unmanaged memory, etc.

If you have footguns, you have to document every single one of them, or you're just entrapping your engineers and lying to yourself

Probably easier to just fix them

Orchestration

- Not automation — keep a human in the loop or give your adversary shiny new tools
- Especially around security response:
  - Image and burn hosts and their creds
  - Roll any given credential
  - Archive and deactivate users and purge infrastructure of any processes they started
  - Move entire systems to read-only mode
  - Reboot the world without downtime

Documentation

Do you know what your code is supposed to do?

Will your incident response team be able to?

Documentation:

- Drives more rigorous development practices
- Makes teams flatter and more accessible
- Prevents bus errors and makes ramp-up faster

- Orienting documentation first
- Then business rules and security objective enforcement points
- Then maybe other stuff

#blameless

Blameless Engineering

- If you want to make things better you need to know what happened
- If you fire people for fucking up they will never, ever tell you what happened
- Assume everyone who works for you is smart and shows up at work to do a good job
- Systematic underperformance can be handled separately
- Malice is an incident response issue

Thank you!

http

ella@dymaxion.org

twitter

Like what you've heard?
Hire me to help your team:
https://structures.systems