
SBA
Research

Secure Software Development Lifecycle Fundamentals

App Sec Training @ NÖ Landesregierung 2025-12-09

SBA Research

Federal Ministry Innovation, Mobility and Infrastructure Republic of Austria
Federal Ministry Economy, Energy and Tourism Republic of Austria
FFG Promoting Innovation.
vienna business agency
For the City of Vienna
European Commission
FWF Austrian Science Fund
netidee FÖRDERUNGEN

Andreas Boll

- IT Security Consultant at SBA Research
  - Penetration testing
  - Source code audits
  - SSDLC
  - DevSecOps
  - CISSP, CSSLP, GWAPT, GCPN

aboll@sba-research.org

Training “Rules”

- Feel free to **interrupt**! Please **ask questions** any time. We have lots of content to cover but also time for discussions and to answer questions

- If you have made a **different experience** or have another opinion, **please share it**. Let’s discuss it – Security is rarely entirely “black or white”

- If you **know more** about a specific topic or want to correct a statement, please do not hesitate!

Secure Development Lifecycle Fundamentals

- The process, techniques and tools that enable you to create secure software
- We will not talk about *Secure Coding*

- Focusing on breadth, not depth
- Giving you many ideas and starting points for implementing Your SDLC

Agenda

1. Introduction
2. Governance
3. Requirements
4. Design
5. Implementation
6. Verification
7. Operations

Introduction

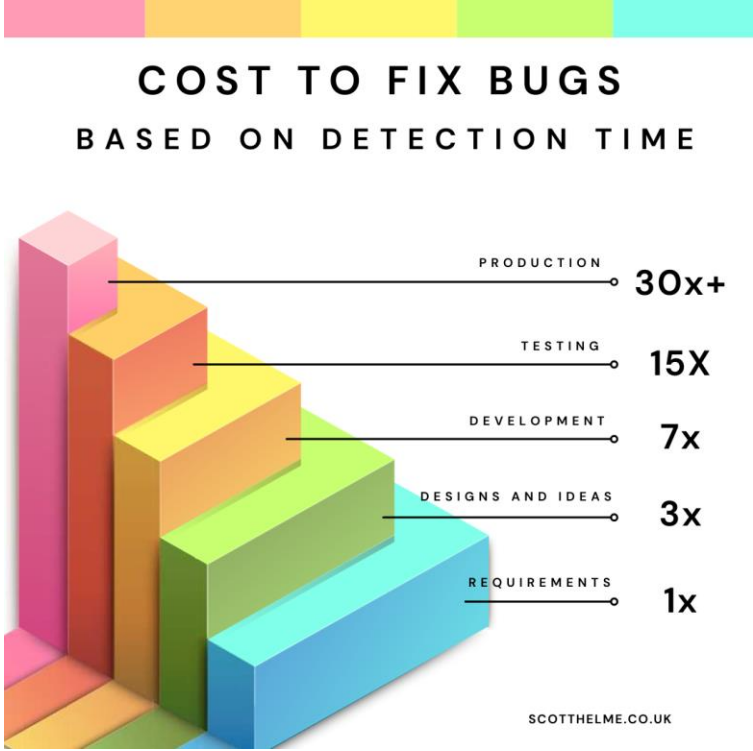
Why are we here?

Why is Security Strategically Important?

- Customer trust is invaluable, but fragile
- Digitalization increases the threat level
- The threat landscape evolves quickly
- Threats aren’t just IT-related

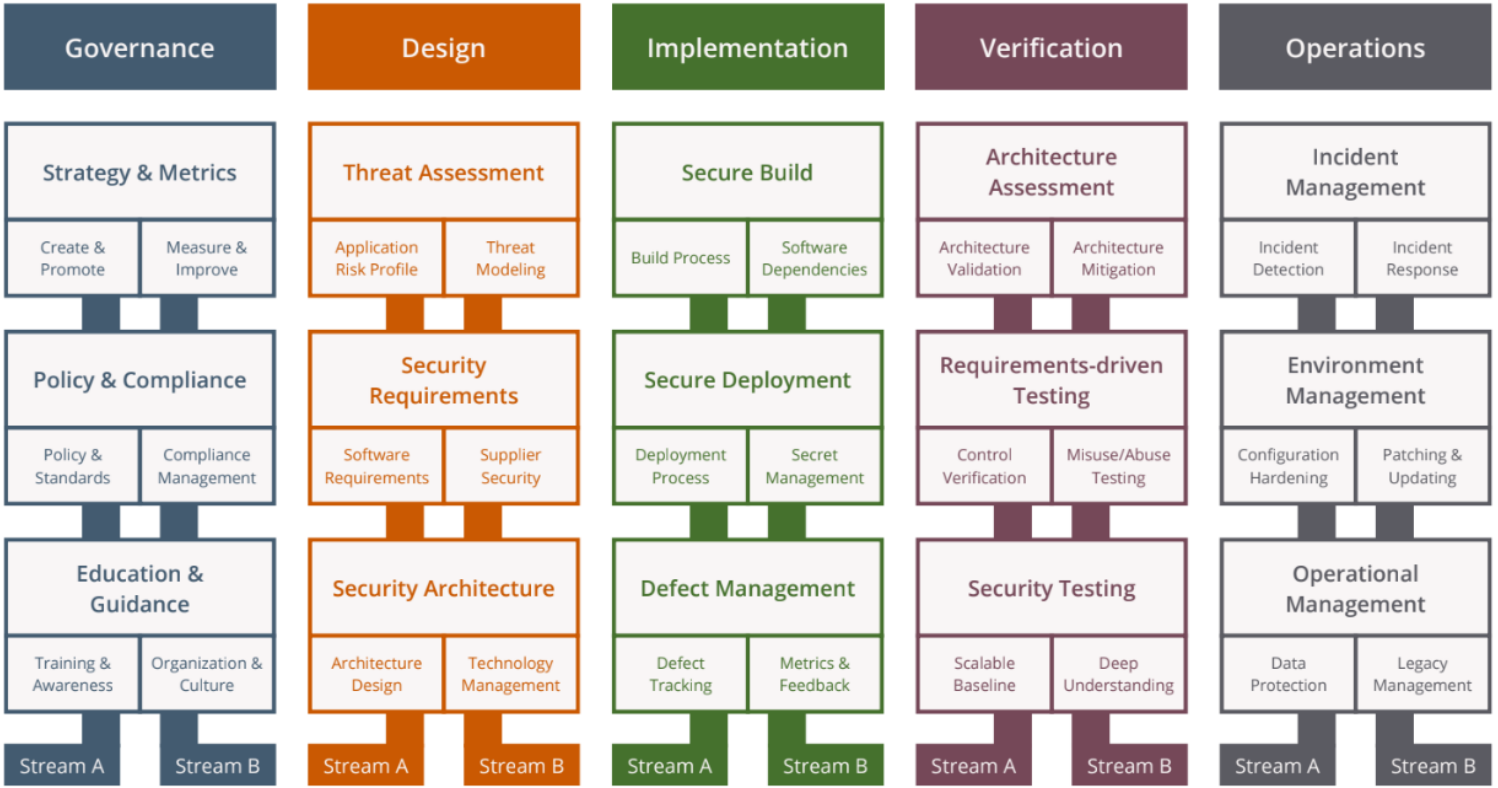
Security and Quality

- Secure software is typically high-quality software
- Security as a usual quality requirement, not something “on top”
- Most cost-effective in the long term when considered from the start



https://scotthelme.co.uk/google-announce-new-minimum-viable-secure-product/

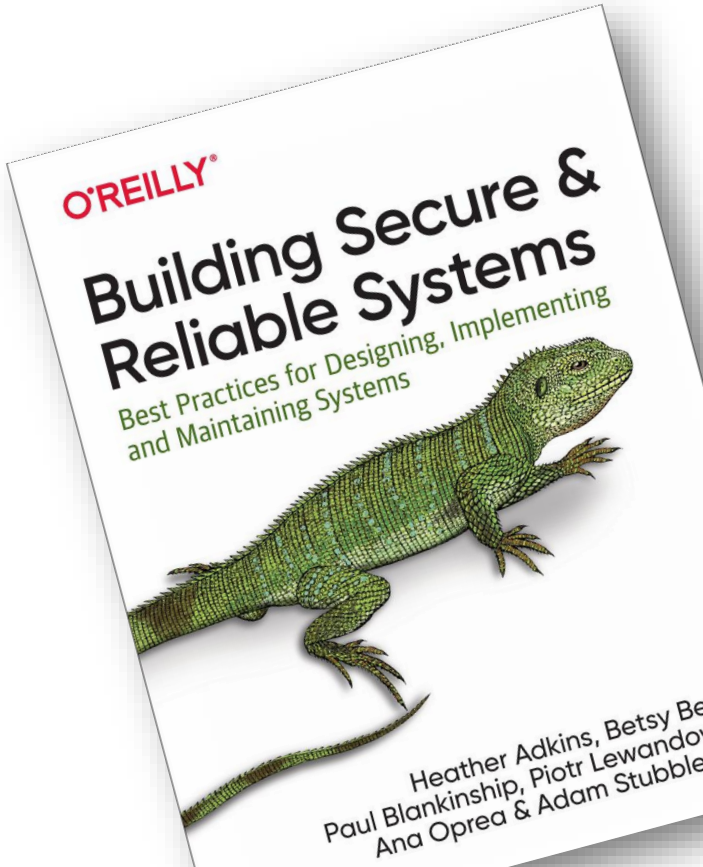
OWASP SAMM


Governance
Design
Implementation
Verification
Operations
Strategy & Metrics
Create & Promote
Measure & Improve
Policy & Compliance
Policy & Standards
Compliance Management
Education & Guidance
Training & Awareness
Organization & Culture
Threat Assessment
Application Risk Profile
Threat Modeling
Security Requirements
Software Requirements
Supplier Security
Security Architecture
Architecture Design
Technology Management
Secure Build
Build Process
Software Dependencies
Secure Deployment
Deployment Process
Secret Management
Defect Management
Defect Tracking
Metrics & Feedback
Architecture Assessment
Architecture Validation
Architecture Mitigation
Requirements-driven Testing
Control Verification
Misuse/Abuse Testing
Security Testing
Scalable Baseline
Deep Understanding
Incident Management
Incident Detection
Incident Response
Environment Management
Configuration Hardening
Patching & Updating
Operational Management
Data Protection
Legacy Management
Stream A
Stream B

Initial Velocity vs. Sustained Velocity

- *„We'll add security later"* –
  You probably won't.
- You hope to gain initial velocity
- But you'll lose sustained velocity

Book recommendation: "Building Secure and Reliable Systems" by Heather Adkins et. al.



Sustained Velocity By Example

Governance

Making software security manageable

Let Me Ask You A Question

How do you ensure security in your software?

Try to answer that in 60 seconds.

What Is Security Governance?

Having overview and ability to control.

- Responsibilities
- Risk, threats, vulnerabilities
- Strategy
- Skills and awareness
- Guidelines
- Budget
- Technical debt
- Metrics
- Continuous and economic improvement

Governance vs. Management

Governance

- Goals, direction
- Responsibilities
- Accountability frameworks

Management

- Allocation of resources
- Overseeing day-to-day operations
- Aligned with governance

Security Strategy

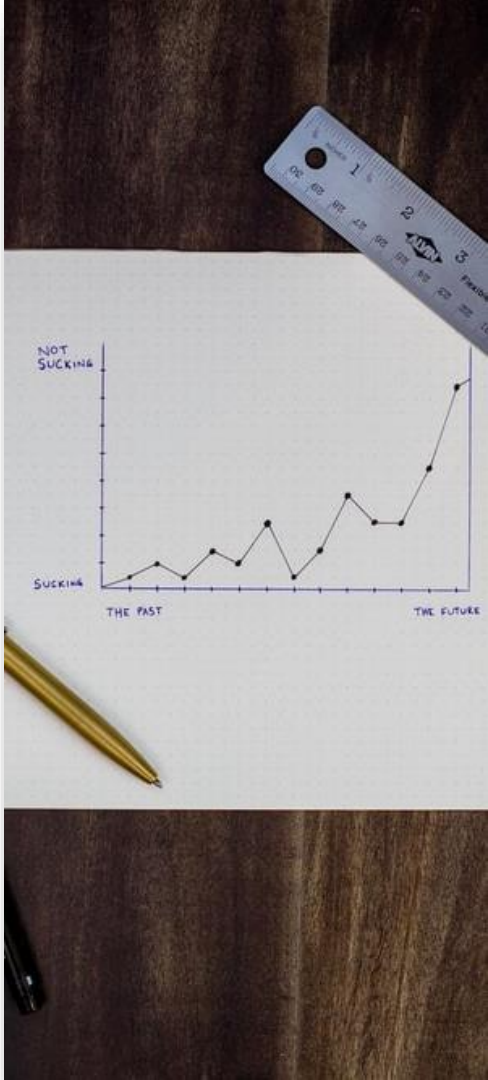
Do you have a strategic plan and use it to make decisions?

- Organization’s priorities
- Milestones and budget
- Roadmap
- Stakeholder’s approval

Metrics

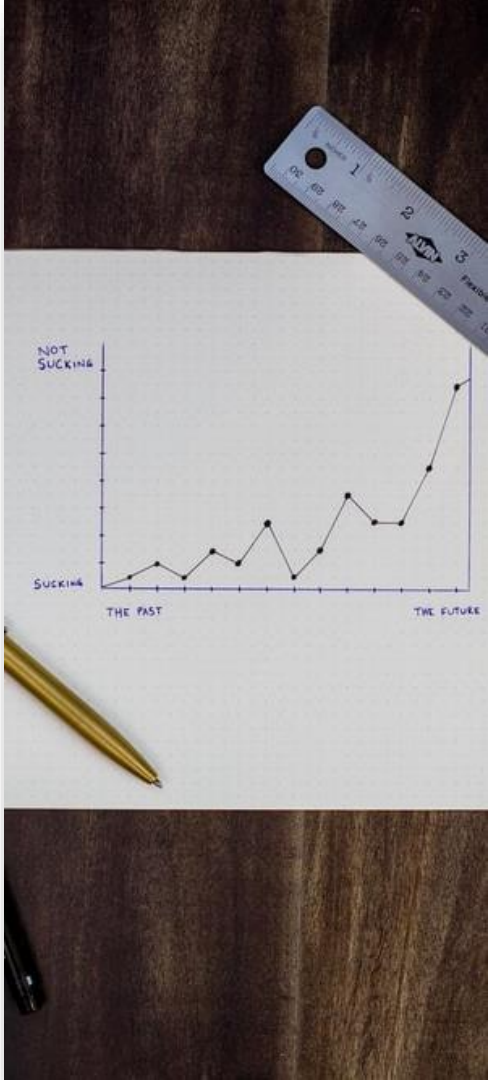
Can you tell *with evidence* that security improved over time?

- How can you tell?
- Do you have a complete picture?
- Do activities show effect?



Metrics

Bad metrics

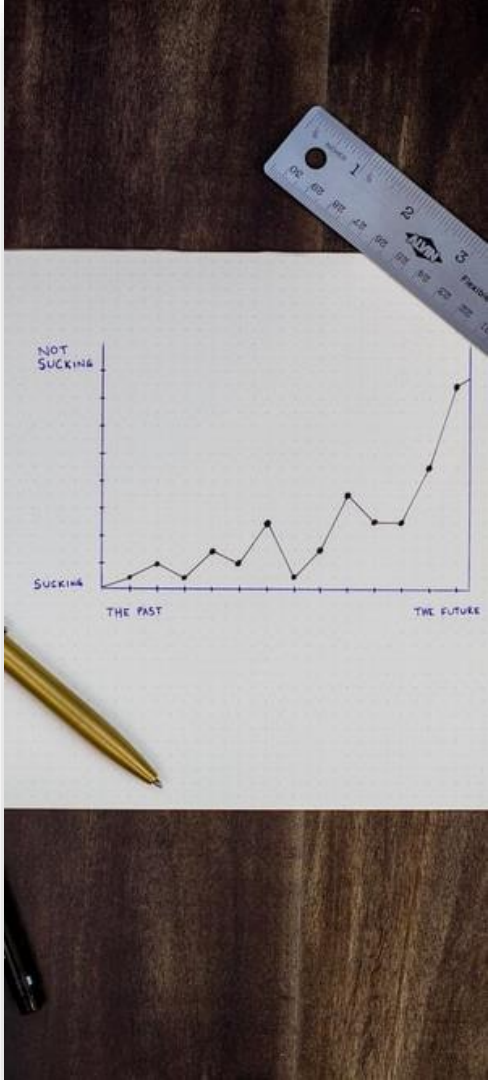


Metrics

Some metrics that matter

- Time to detection
- Time to remediation
- Average number of vulns per LoC

After targeting vulns, do they decline?



Further Reading

- Keynote from Sec4Dev ‘21 “Security Metrics That Matter” by Tanya Janca

Tanya Janca

What To Do With Metrics?

- Educate specifically on problematic areas
- Reduce error-proneness for common bug types
- **Feed insights back to other activities**
  - Secure coding guidelines
  - Security design patterns
  - Automation
  - …

Compliance

Can you tell which internal or external obligations you have?

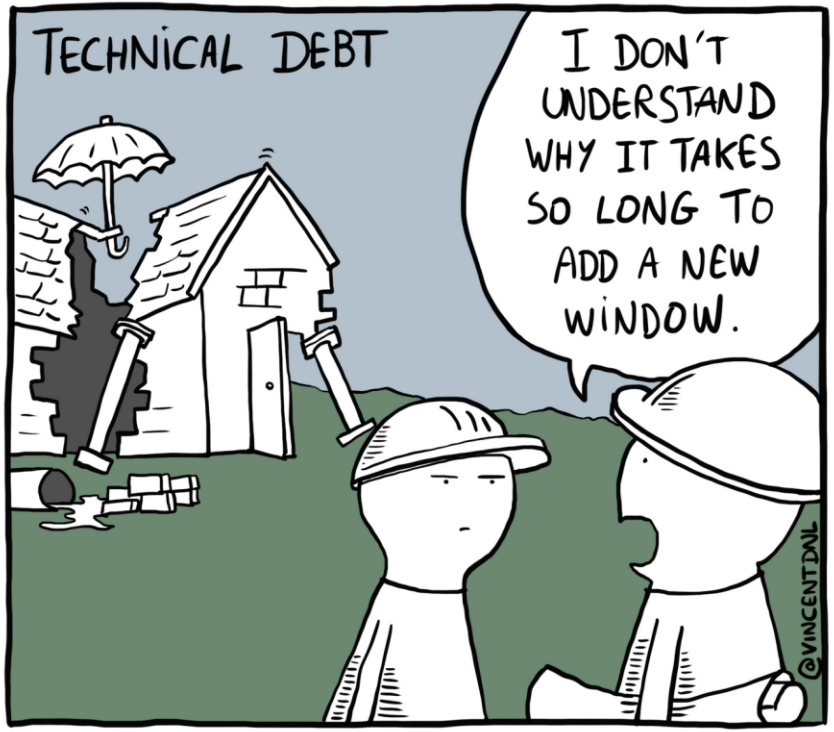
- Laws and regulations
- Contracts
- Internal policies

Compliance: Runbooks

It's hard to tell for developers what's relevant and how to implement compliance requirements.

- Create runbooks for software types
- Filter what's relevant
- Give advice on how to implement the requirements

Technical Debt



Technical Debt: Lehman's Laws

Continuing Change

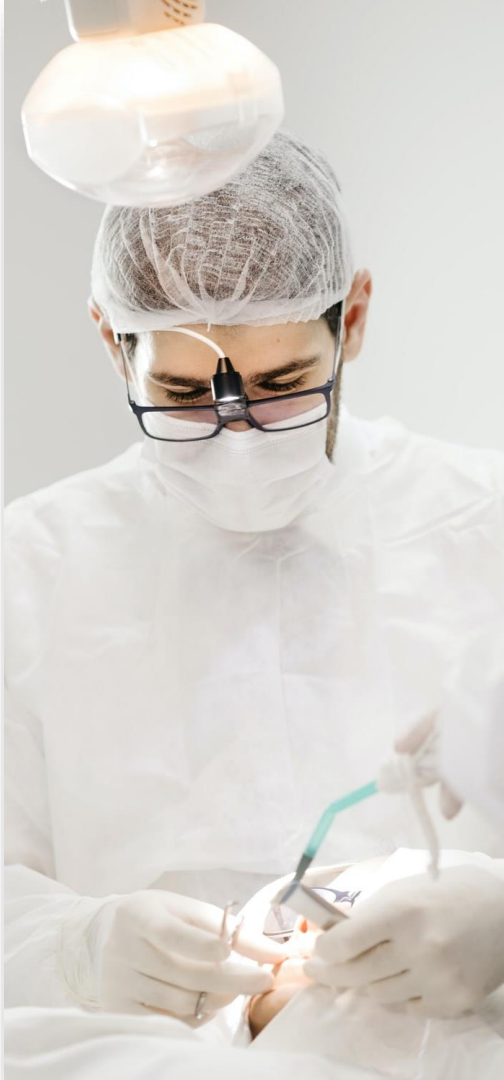
"A system must be continually adapted, or it becomes progressively less satisfactory."

Increasing Complexity

"As a system evolves, its complexity increases unless work is done to maintain or reduce it."

Symptoms of Technical Debt

- Long lead times
- Lack of predictability
- Bugs, vulnerabilities



Technical Debt

We improve on what we're measured.

- Make sure to understand what behavior you reinforce by measurements.
- We need to find a trade-off between improving existing code and adding new features.
- Strive for locality of change.

Recommended Watching

Prioritizing Technical Debt as if Time and Money Matters

Adam Tornhill

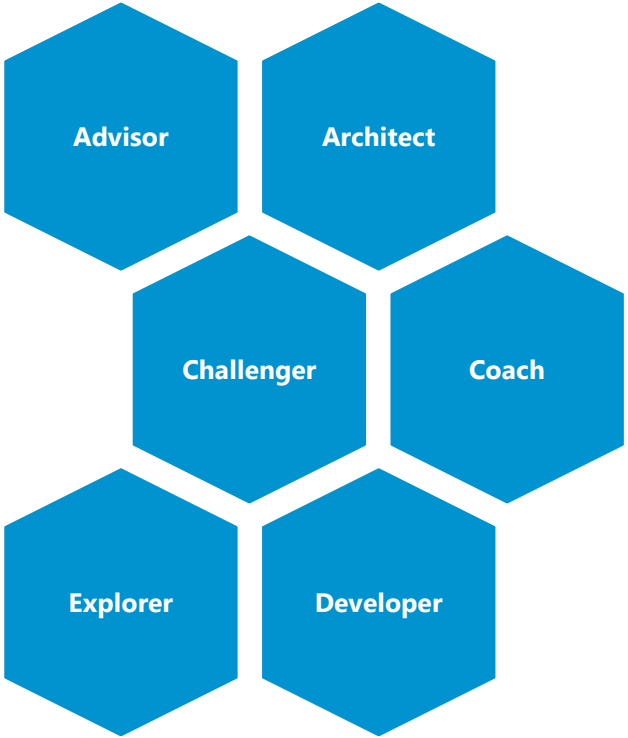
https://www.youtube.com/watch?v=fl4aZ2KXBsQ

Technical Debt: Quick Wins

When you have implemented a feature and test are green, take some time to refactor and *keep* tests green.

- This is the most efficient way to pay down technical debt.
- It's paid down before you start paying interest on it.
- Make it a cultural habit.

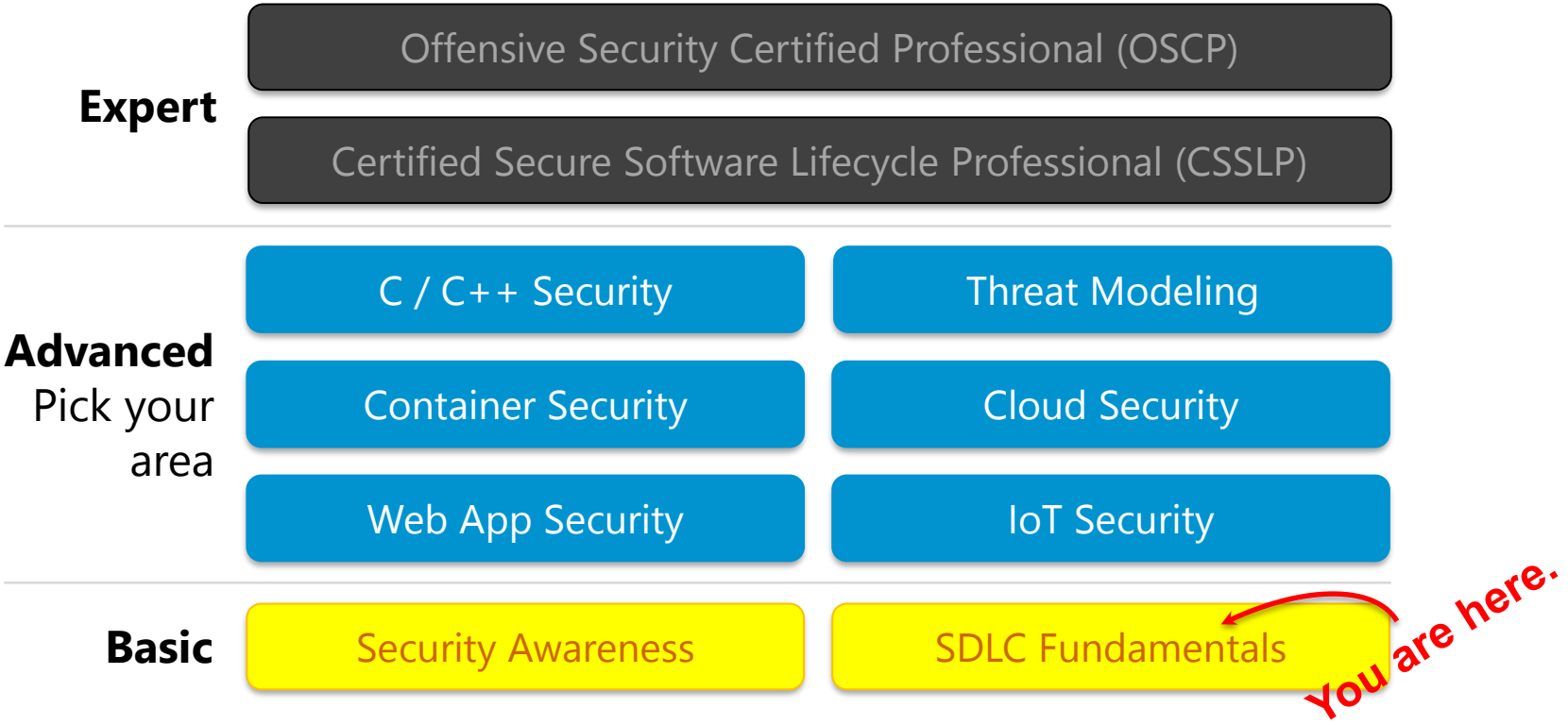
Build Security Into Your Team: Security Champions



Further Reading

Software Security Takes a Champion: A Short Guide on Building and Sustaining a Successful Security Champions Program

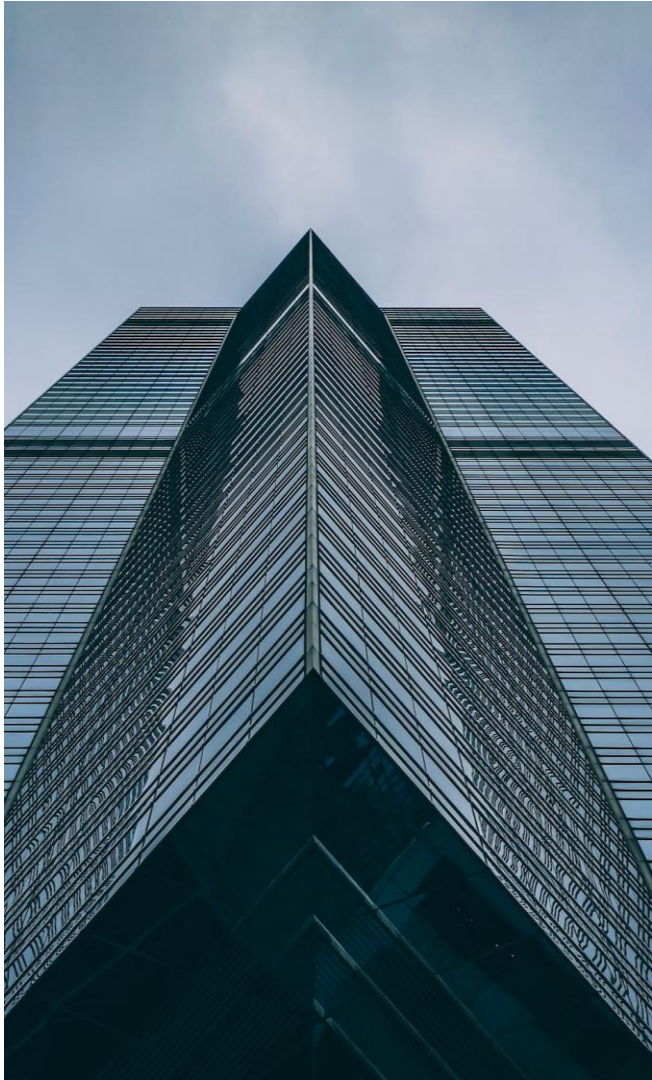
Best ROI: Education



Centralization

A *Secure Coding Center of Excellence* can help you centralize information and tasks

- Documentation
- Technology evaluation
- Education guidance
- Coding guidelines
- Reference architectures
- Security services

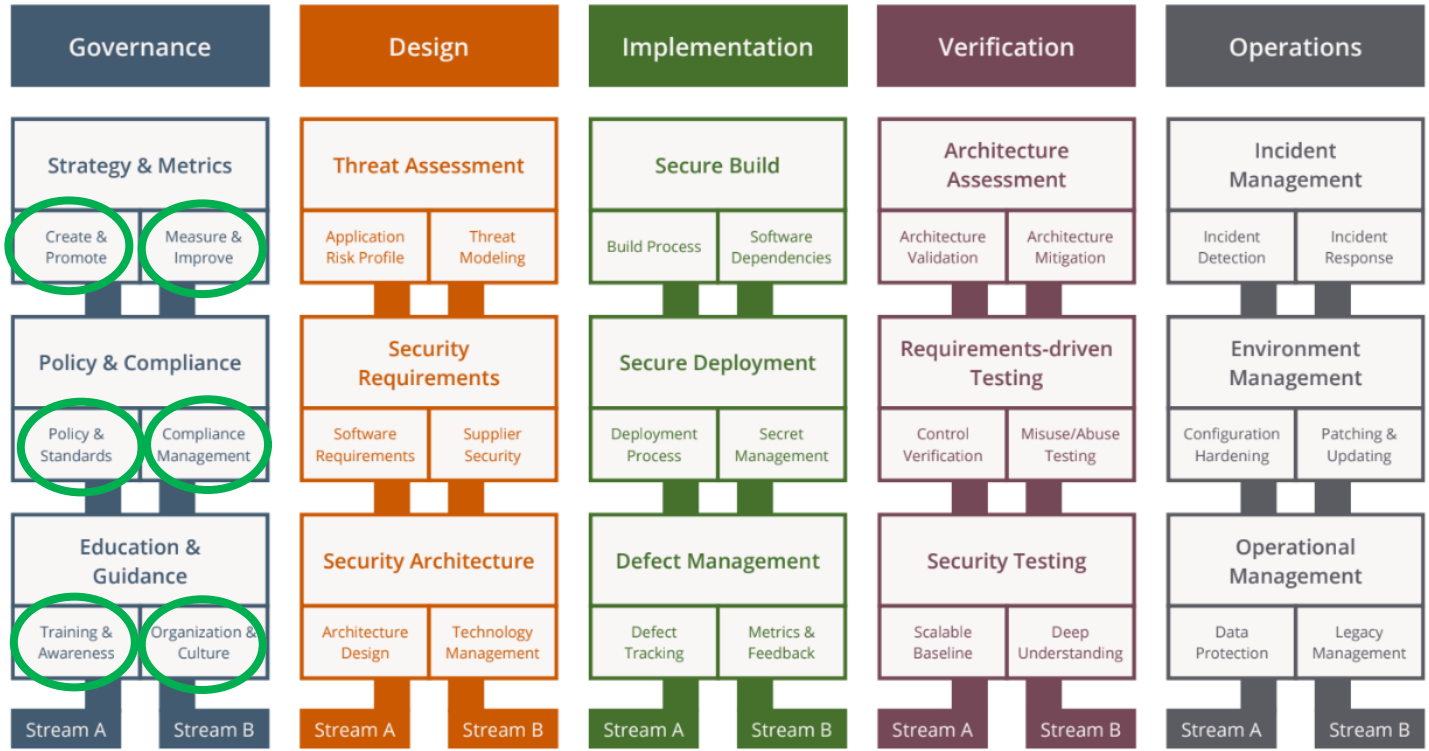


Governance Requires Documentation

There is nothing worse than documentation collecting dust on a hidden share.

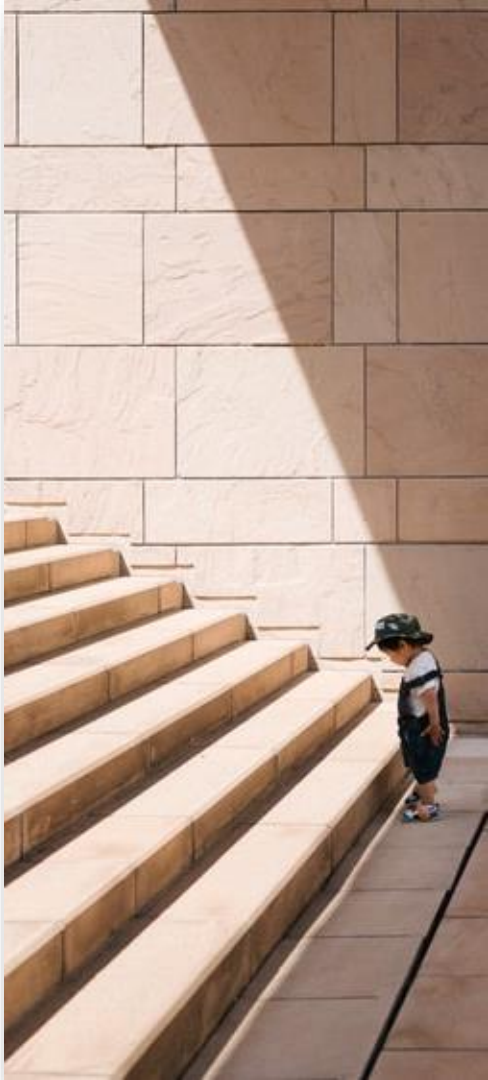
- Have it at an accessible place
- Force yourself to review it regularly
- Keep it slim

Where Are We In OWASP SAMM?



Essentials: Software Security Governance

- ❑ Have a strategy that all stakeholders agree upon, including developers.
- ❑ Collect metrics that prove that you're improving.
- ❑ Be fully aware of your compliance requirements.
- ❑ Create compliance runbooks to make their implementation easier.
- ❑ Track and pay down technical debt.
- ❑ Educate your team.
- ❑ Have documentation on an accessible place and force yourself to review it regularly.



Quiz!

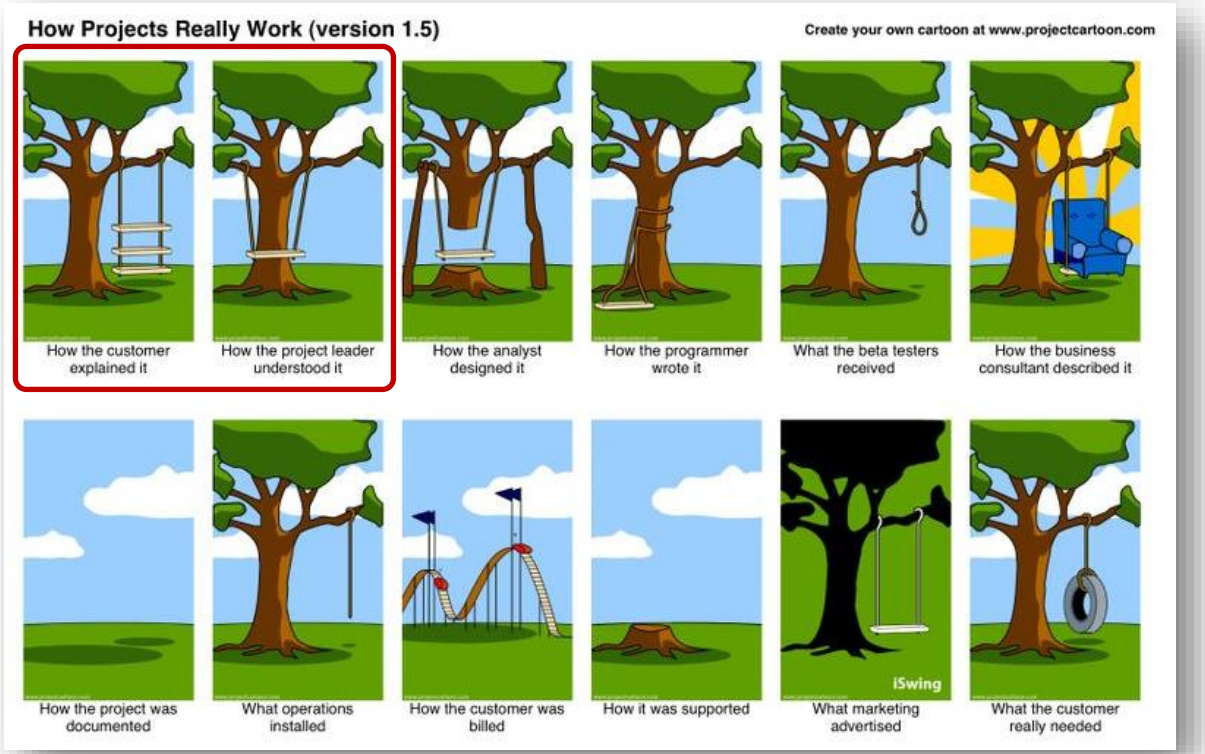
Software Security Governance

Visit https://kahoot.it and enter the game pin I'll share with you!

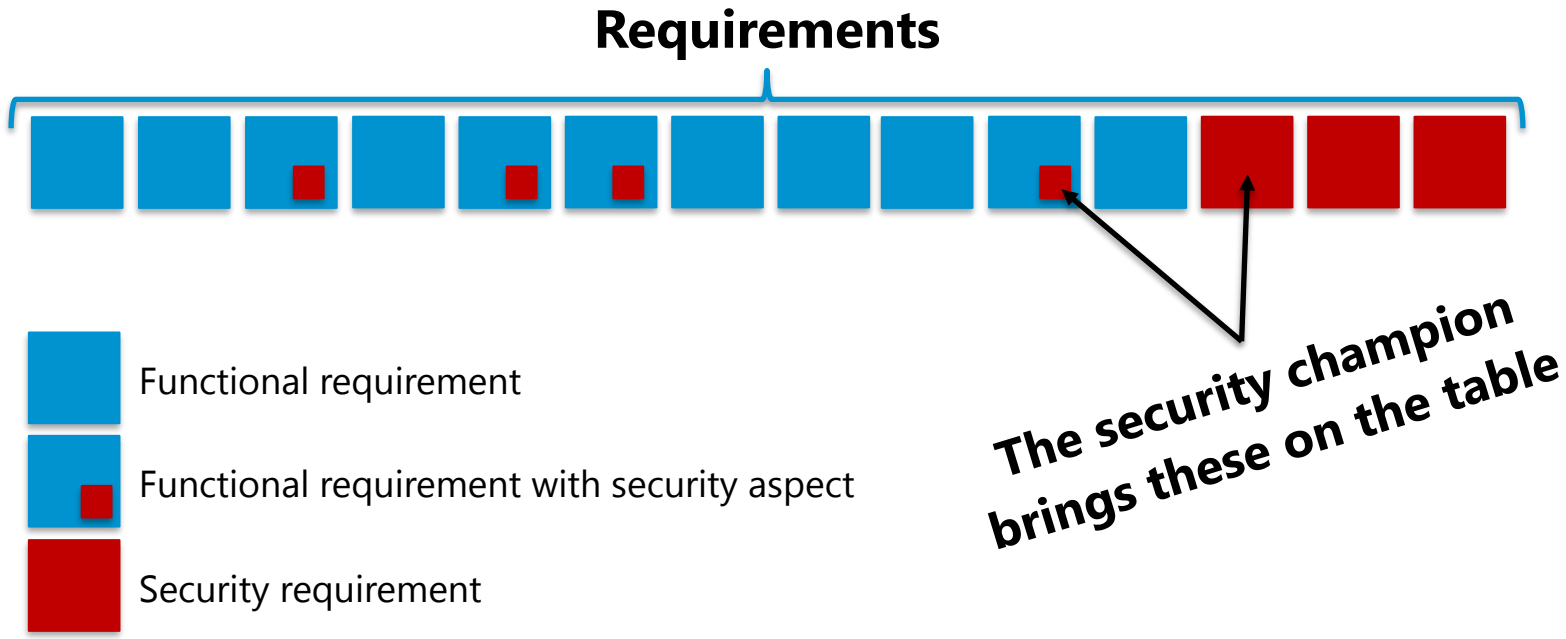
Software Security Requirements

Why, what, how

“Real-World Example”: Requirements



Give a Security Perspective



Sources of Security Requirements

Here are some common sources.

- Customer request and contracts
- Compliance regulations
- Internal (security) policies
- Security posture
- Standards (ASVS)
- Prior incidents
- Other security activities

OWASP ASVS

V6.2 Password Security

The requirements in this section mostly relate to § 5.1.1.2 of NIST's Guidance.

#	Description	Level
6.2.7	Verify that "paste" functionality, browser password helpers, and external password managers are permitted.	1
6.2.8	Verify that the application verifies the user's password exactly as received from the user, without any modifications such as truncation or case transformation.	1
6.2.9	Verify that passwords of at least 64 characters are permitted.	2
6.2.10	Verify that a user's password stays valid until it is discovered to be compromised or the user rotates it. The application must not require periodic credential rotation.	2

Good (Security) Requirements Engineering

Bad (security) requirements are ignored or misinterpreted. Good requirements …

- … describe the “what”, not the “how”.
- … are specific, measurable, reasonable, traceable
- … neither overdo nor underdo it.

Good (Security) Requirements Engineering

“Passwords need to be at least 12 characters long”

- Not “Passwords need to be secure” (not specific)
- Not “Passwords need to be at least 50 characters long” (not reasonable)
- Not “Passwords need to be sufficiently long to prevent brute-force attacks” (not measurable)

Requirements Example

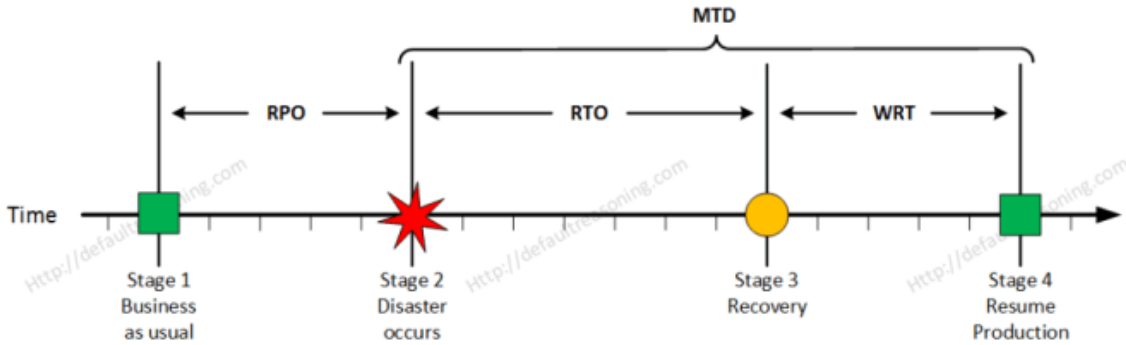
Category: Confidentiality and integrity

- The application contents must be personalized, and users must only access their own datasets.
- Users must have the ability to explicitly share datasets with other users of the system.
- The application must integrate with the organization's SSO system.

Requirements Example

Category: Availability

- The application must be able to handle 200 concurrent users.
- If data needs to be restored, no more than 2 hours of work can be lost.



Requirements Example

Category: Compliance

- Users must be able to independently download all the PII that the system processes of them.
- Users must be able to delete their account, including all PII. Upon deletion, the data must be permanently deleted.
- Users must be able to independently correct any of their PII stored in the system.

Requirements: Common Pitfalls

Non-functional requirements are the responsibility of the "other side".

- Business takes security for granted.
- Devs take requirements completeness for granted.
- The result is a gap of misunderstanding.
- Don't just quietly define hidden requirements.

Requirements: Common Pitfalls

Focusing only on vulnerability classes.

- It is common to see “we must protect against the OWASP Top 10”.
- This is rarely doable, nor worth doing.
- Relying only on this will leave your software open to flaws.
- You probably won’t spend your time wisely.

Requirements: Common Pitfalls

Throwing in whole standards.

- Same here: This is rarely doable, nor worth doing.
- This will let the effort estimation explode.
- Pick what's relevant.

Requirements: Common Pitfalls

Not mapping security issues back to requirements.

- Seeing the same security issue repeatedly might be a sign for misaligned requirements.
- Learn for the future.
- Share your experience.

Requirements: Common Pitfalls

Starting from zero for each project.

- Security requirements tend to repeat themselves within project types.
- Don’t start all over.
- Document blocks of common requirements and manage them centrally.

Requirements: Common Pitfalls

Forgetting vendors.

- There might be external code in your software.
- Don’t forget this when defining requirements.

Requirements: Common Pitfalls

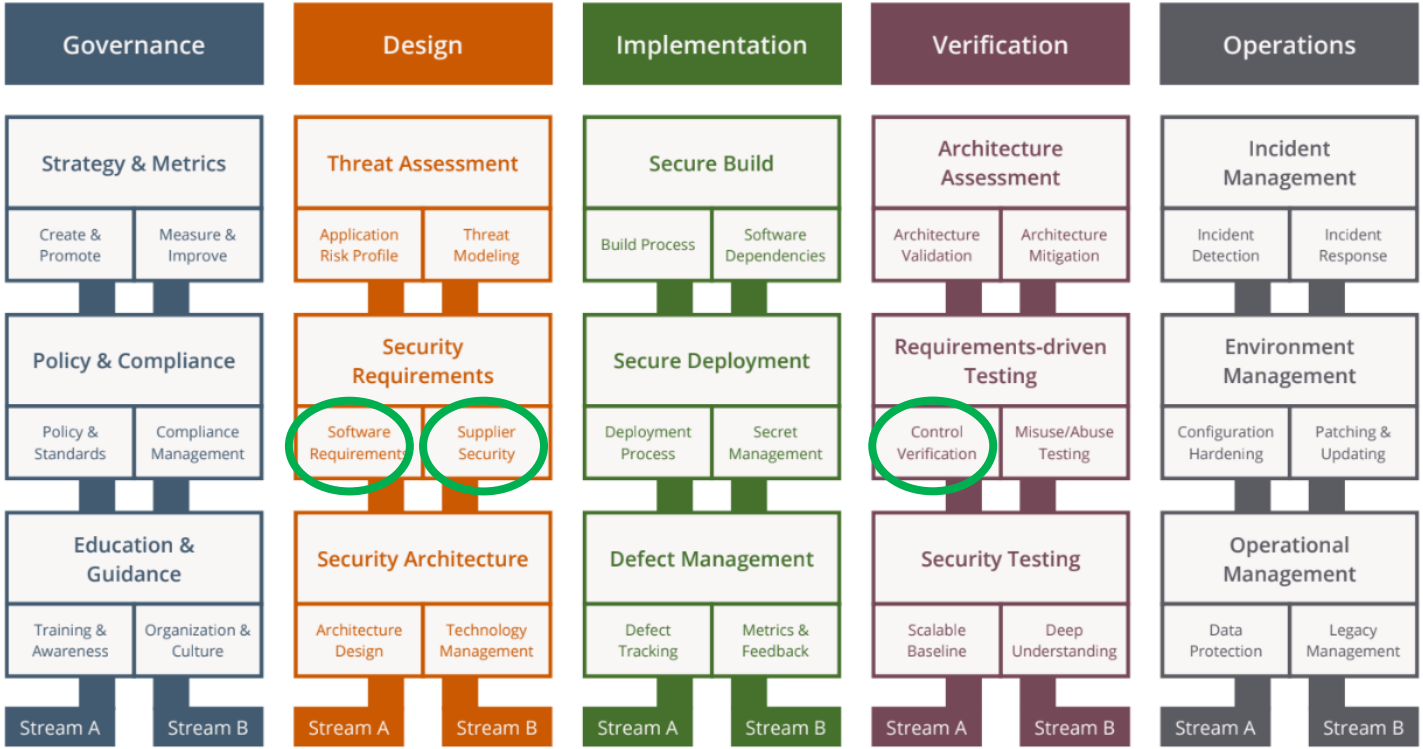
Not tracking your requirements.

- Have an eye on tracing requirements.
- Be aware that not all requirements end up in code being produced.
- Link the place of its implementation.
- Serves for proofing your implementation both to customers and auditors.

Requirements Traceability Matrix (RTM)

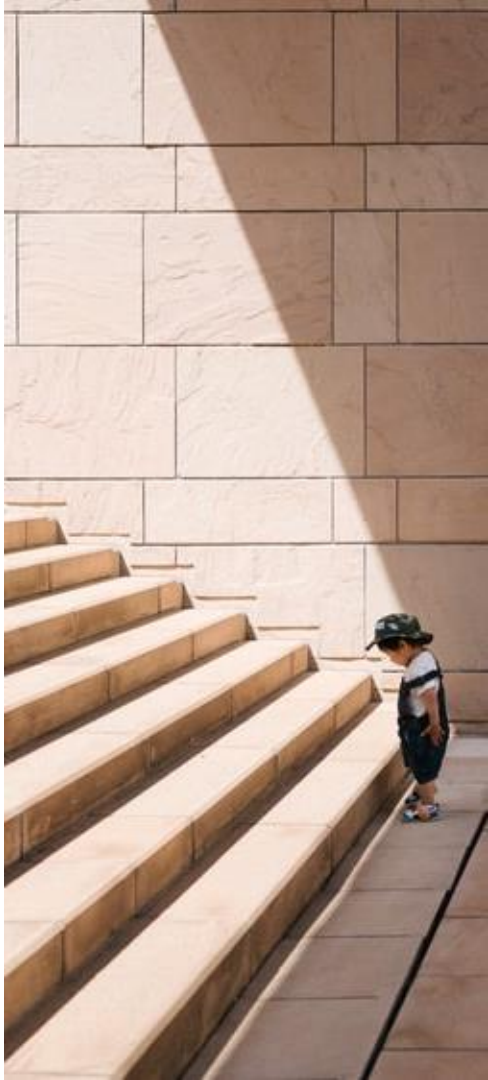
Requirement ID	Requirement Description	Category	Priority	Status	Linked Deliverables	Test Cases	Comments
REQ-001	Users must be able to log in using email and password.	Functional	High	In Progress	Login Module, User Manual	TC-001, TC-002	
REQ-002	Data must be encrypted when stored in the database.	Security	High	Planned	Database Schema, Security Policy	TC-010	Encryption method to be determined
REQ-003	The system should respond to requests within 2 seconds.	Performance	Medium	Open	Performance Improvement Plan	TC-020, TC-021	Requirement based on customer feedback
REQ-004	The system must support exporting reports in PDF format.	Functional	Medium	Under Review	Reporting Module, User Manual	TC-030	Must support PDF/A-1b for archiving
REQ-005	The system must ensure 99.9% availability per month.	Availability	High	In Progress	Maintenance Plan, SLA Documentation	TC-040	Including scheduled maintenance windows
REQ-006	Support multi-language user interface for English and Spanish.	Usability	Medium	Planned	UI Module, Localization Files	TC-050, TC-051	Important for market expansion
REQ-007	Implement role-based access control (RBAC) for system users.	Security	High	In Progress	Security Module, User Manual	TC-060, TC-061	Roles to be defined in collaboration with security team
REQ-008	System should integrate with external payment gateway XYZ.	Integration	High	Planned	Payment Module, Integration Test Plan	TC-070	Agreement with XYZ is pending
REQ-009	Allow users to reset their passwords via email verification.	Functional	High	In Progress	Login Module, Email Service Integration	TC-080	Must meet GDPR compliance
REQ-010	The system must be deployable on both AWS and Azure cloud platforms.	Deployability	Medium	Open	Deployment Scripts, Infrastructure as Code	TC-090	Ensure compatibility with both cloud services
REQ-011	Automated data backups should occur daily without user intervention.	Operational	High	Planned	Backup and Recovery Plan, Operational Manual	TC-100	Backup verification process required
REQ-012	The system must support at least 10,000 concurrent users.	Scalability	High	Open	Scaling Plan, Load Testing Results	TC-110	Based on projected growth over the next year

Where Are We In OWASP SAMM?



Essentials: Security Requirements

- ❑ Consider both security aspects of functional requirements and security requirements.
- ❑ Select requirements well.
- ❑ Make them measurable.
- ❑ Don’t reinvent the wheel.
- ❑ Involve the organization.
- ❑ Trace and link the implementation.



Quiz!

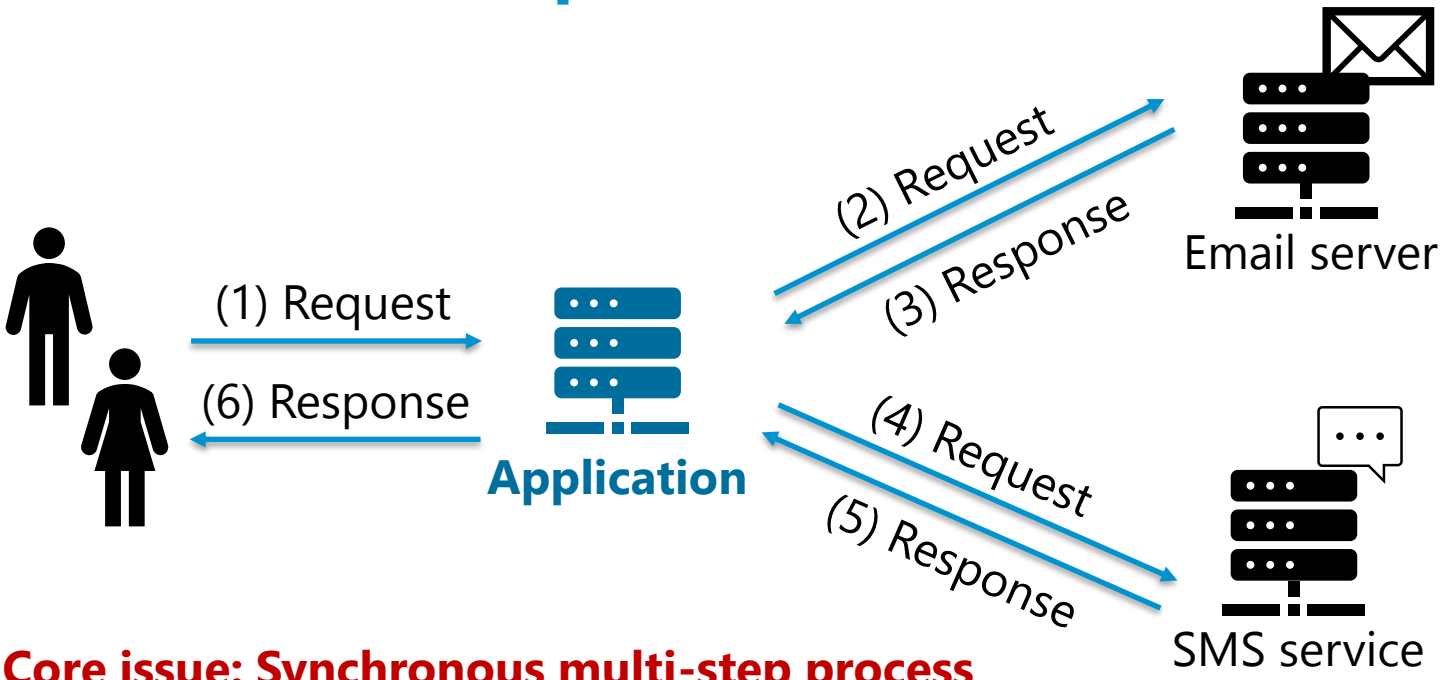
Security Requirements

Visit https://kahoot.it and enter the game pin I'll share with you!

Secure Software Design

The need for secure design, security design concepts, flaws vs. bugs, threat modeling

Real-World Example



Core issue: Synchronous multi-step process
Architecture is prone to Denial of Service

How Much Is Enough?

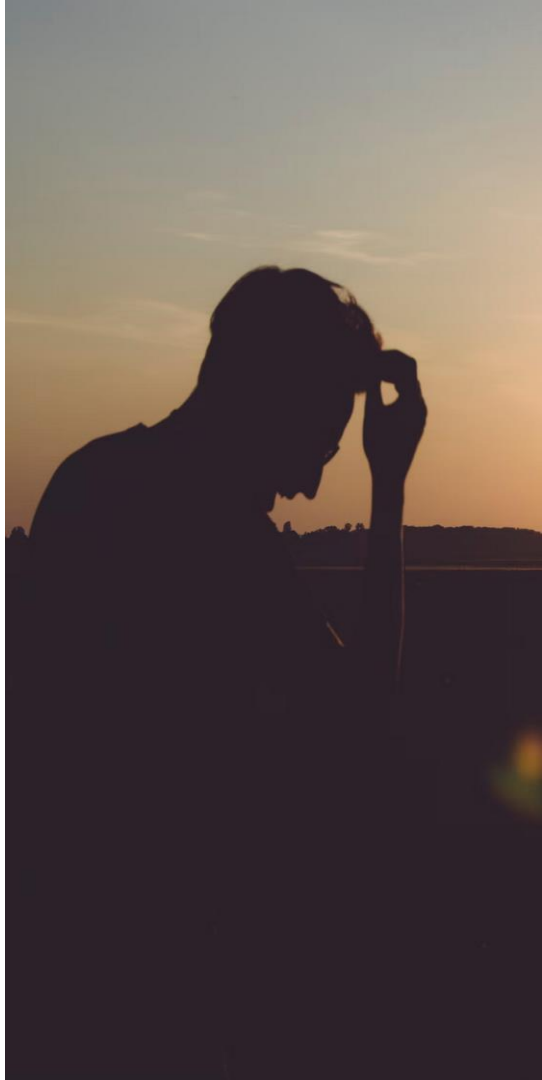
Prioritize your limited resources well!

- How critical is the software?
- How critical is the data processed?
- How big is the attack surface?
- How bad is it if it fails?
- …

Software Risk Profile

Classify applications according to risk

- Simple standardized set of questions
- “What happens if all goes wrong?” for CIA-triade
- Example: Mozilla’s Rapid Risk Assessment



Software Risk Profile

Data sensitivity

- Personally identifiable information (PII)
- Financial data
- Health data
- Other sensitive data

Software Risk Profile

Reachability and attack surface

- Internet
- Intranet
- Subnet
- Physical

Software Risk Profile

System and data availability

- Is it a threat to human wellbeing if the system goes down?
- How big would a financial loss roughly be after 1 minute, 1 hour, 1 day, 1 week?
- What is the maximum age of data restored from a backup?

The Wobbly Milking Chair

- This milking chair is wobbly on uneven grounds
- **Question**: What can we do to improve the chair?

https://i1.wp.com/berufe-dieser-welt.de/wp-content/uploads/2016/12/melkerin-kuh-1905.jpg?w=450&ssl=1

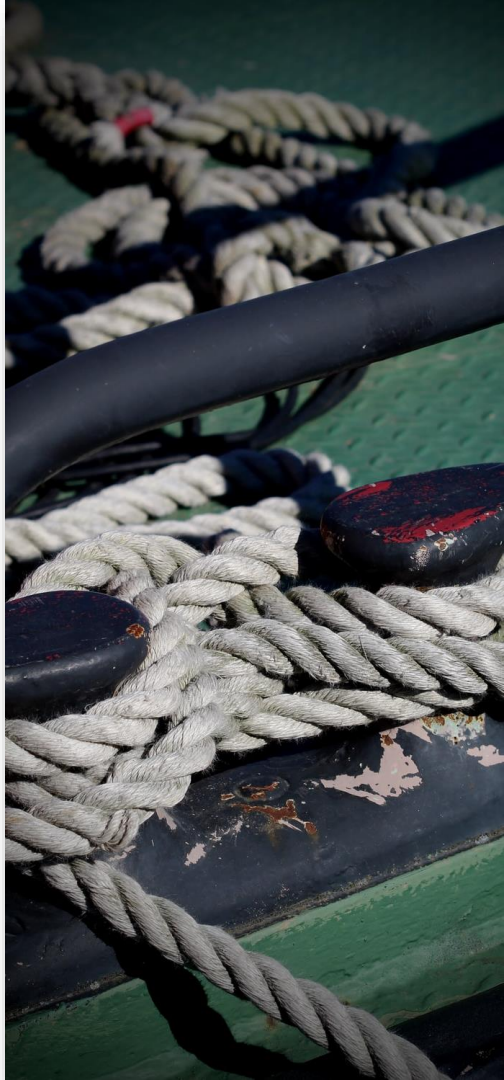
Solving the Wobbliness By Design

Possible by-design solution: Give it only three legs

“By Design” Means Negligible Room For Error

A by-design solution

- is not error-prone
- is secure by default
- requires an explicit decision to make it insecure



Threat, Vulnerability, Flaw, Bug

Vulnerability

- An abstract weakness
- Relates to implementation

Threat

- What an organization is defending against
- Can relate to design or implementation
- Needs a *Threat Actor*
- Based on a weakness

Threat, Vulnerability, Flaw, Bug

Flaw

- Design weakness
- Tends to be harder to fix

Bug

- Coding mistake
- Tends to be easier to fix

Flaw vs. Bug

https://www.britannica.com/topic/Tacoma-Narrows-Bridge

Implementation vs. Design Issues

	Implementation layer	Design layer
Defect name	Bug, vulnerability	Flaw
Best found	SAST, DAST, penetration testing, code review	Threat modeling, architecture review
Best solved	In code, via configuration	At design, in architecture, via technology choice
Ease to fix	Easy	Hard

Security Principles

- **Design Security Concepts**
  - Least Privilege
  - Separation of Duties
  - Defense in Depth
  - Fail Secure vs. Fail Safe
  - Economy of Mechanisms (a.k.a KISS)
  - Complete Mediation
  - Open Design
  - Psychological Acceptability
  - Weakest Link
  - Leveraging Existing Components

Security Principles

- **Usage**
  - Have a checklist of agreed upon *Security Principles*
  - Share and understand the list
  - Use it on every design decision

Photo by Carstens-Peters, https://unsplash.com/photos/RLw-UC03Gwc

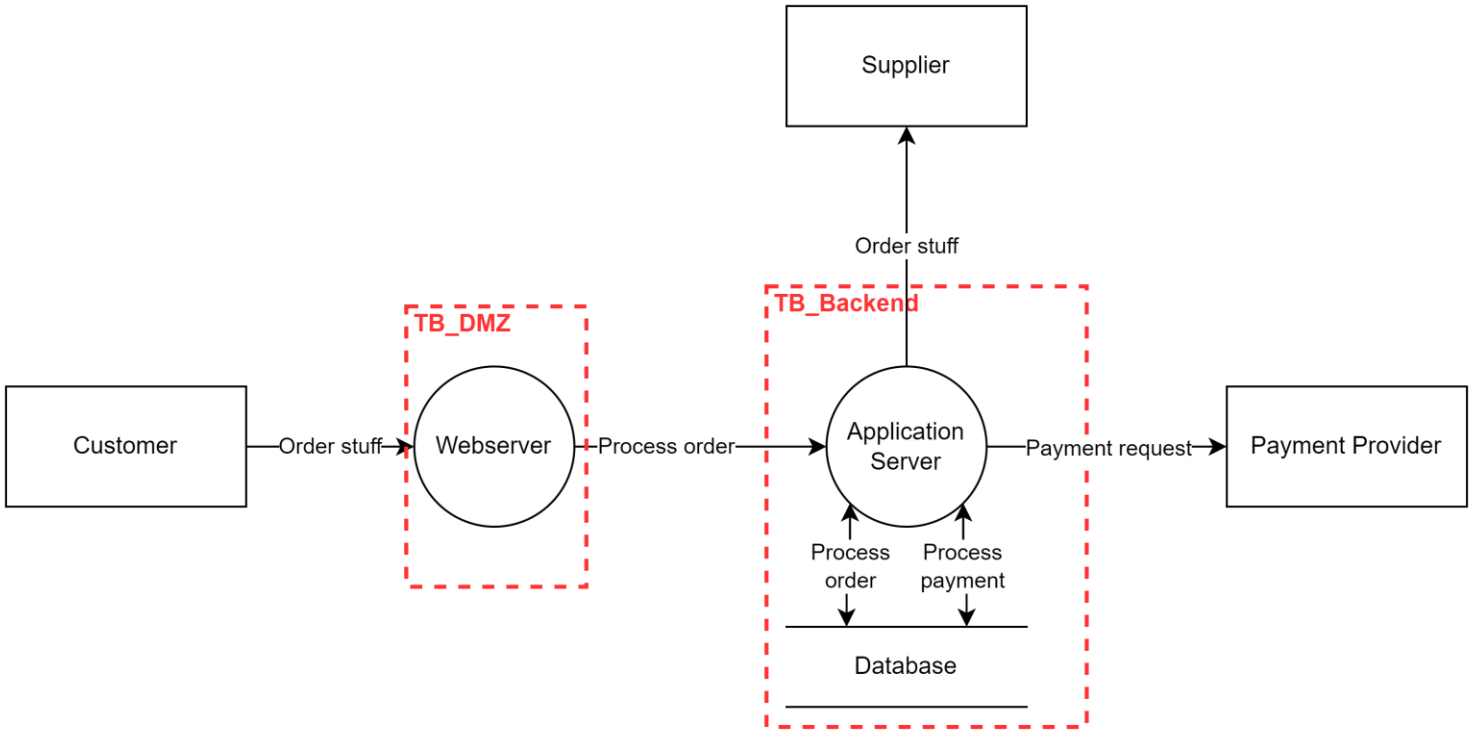
Threat Modeling

Asking these questions

- What are we building?
- What could go wrong?
- What are we doing about it?
- Are we doing a good enough job?

Threat Model: What Are We Building?

Dataflow Diagram



Threat Model: What Could Go Wrong?

Look for these threat classes wherever a **data flow crosses a trust boundary**.

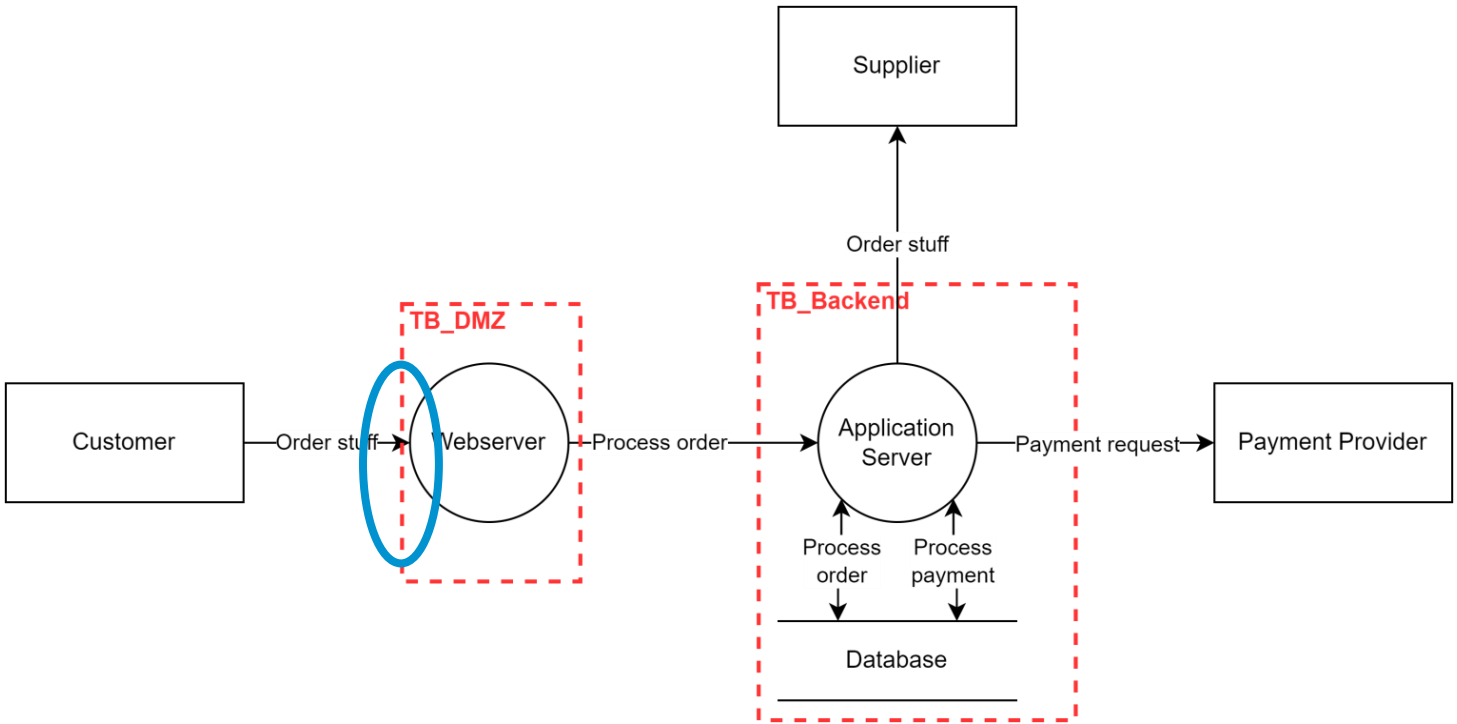
Threat (STRIDE)	Desired Security Property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-repudiation
Information Disclosure	Confidentiality
Denial of Service	Availability
Elevation of Privilege	Authorization

Desired Property?

It is not always that clear what's desired.

- "**Repudiation**" is a **threat** from the security perspective
- But it might be a **desired property** from a privacy perspective

Threat Model: Crossing Trust Boundaries



Threat Model Example: Account Security

Threat modeling as part of the design process

Threat	Threat Actor	Severity[1]	STRIDE	Countermeasures
Password guessing	Internet user	**High**	**S**poofing	(Temporary) user lockout, password policy, MFA, transparency (device lists and notifications, with Device Tokens)
Account lockout	Internet user	**Medium**	**D**enial of Service	Selective lockout (with Device Tokens)
Misuse of known passwords (public lists, other apps, ...)	Internet user	**Medium**	**S**poofing	MFA, Have I Been Pwned (HIBP-API)
Someone dumps the DB on the Internet	DB-Admin	**Medium**	**S**poofing	Proper hashes (Argon2id)
Enumerating valid usernames	Internet user	**Low**	**I**nformation disclosure	(Generic error messages, constant timing on all requests containing the username)

[1] *The severity really depends on the classification of your data. Don't see them as absolute and unchangeable values.*

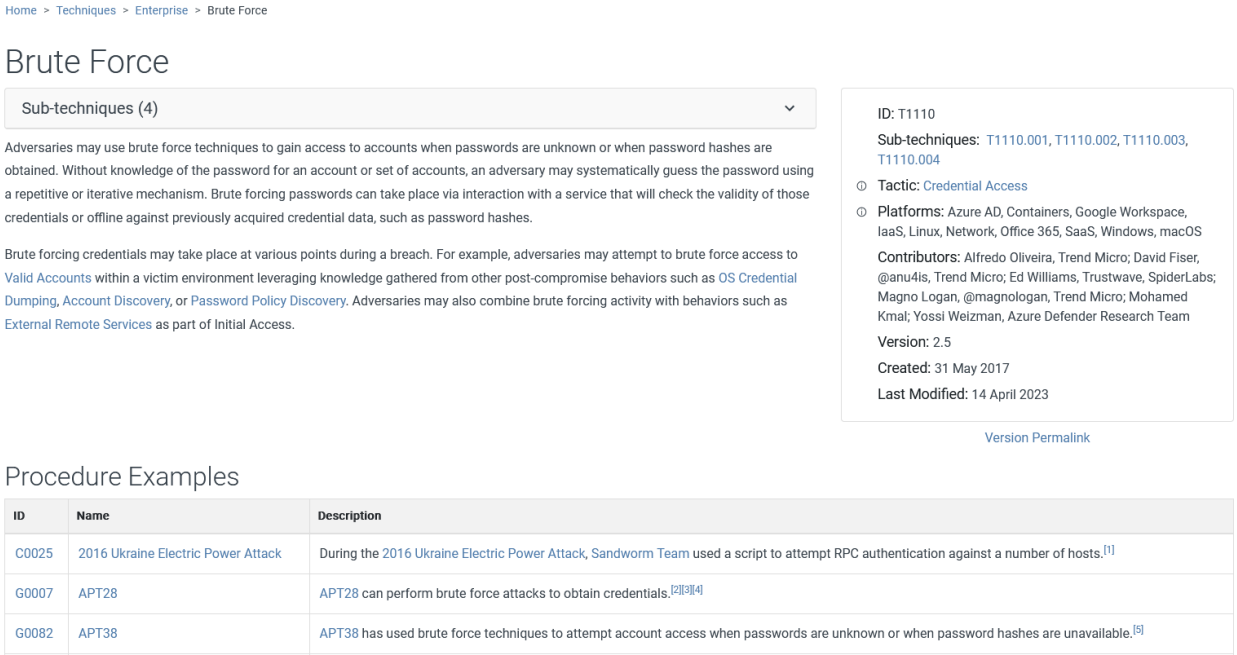
ATT&CK Framework

MITRE ATT&CK® is a globally-accessible knowledge base of adversary tactics and techniques based on real-world observations. The ATT&CK knowledge base is used as a foundation for the development of specific threat models and methodologies in the private sector, in government, and in the cybersecurity product and service community.

https://attack.mitre.org

ATT&CK Framework

Describes *tactics, techniques* and *procedures* of real world threats



Home > Techniques > Enterprise > Brute Force

Brute Force

Sub-techniques (4)

Adversaries may use brute force techniques to gain access to accounts when passwords are unknown or when password hashes are obtained. Without knowledge of the password for an account or set of accounts, an adversary may systematically guess the password using a repetitive or iterative mechanism. Brute forcing passwords can take place via interaction with a service that will check the validity of those credentials or offline against previously acquired credential data, such as password hashes.

Brute forcing credentials may take place at various points during a breach. For example, adversaries may attempt to brute force access to Valid Accounts within a victim environment leveraging knowledge gathered from other post-compromise behaviors such as OS Credential Dumping, Account Discovery, or Password Policy Discovery. Adversaries may also combine brute forcing activity with behaviors such as External Remote Services as part of Initial Access.

ID: T1110

Sub-techniques: T1110.001, T1110.002, T1110.003, T1110.004

Tactic: Credential Access

Platforms: Azure AD, Containers, Google Workspace, IaaS, Linux, Network, Office 365, SaaS, Windows, macOS

Contributors: Alfredo Oliveira, Trend Micro; David Fiser, @anu4is, Trend Micro; Ed Williams, Trustwave, SpiderLabs; Magno Logan, @magnologan, Trend Micro; Mohamed Kmal; Yossi Weizman, Azure Defender Research Team

Version: 2.5

Created: 31 May 2017

Last Modified: 14 April 2023

Version Permalink

Procedure Examples

ID	Name	Description
C0025	2016 Ukraine Electric Power Attack	During the 2016 Ukraine Electric Power Attack, Sandworm Team used a script to attempt RPC authentication against a number of hosts.[1]
G0007	APT28	APT28 can perform brute force attacks to obtain credentials.[2][3][4]
G0082	APT38	APT38 has used brute force techniques to attempt account access when passwords are unknown or when password hashes are unavailable.[5]

Threat Modeling: Common Pitfalls

Burying the spirit under a pile of tools.

- Start with a sheet of paper
- Put people into the center
- Optimize only when optimization is due
- Learn as you go

Threat Modeling: Common Pitfalls

Skipping the scoping part.

- Start small
- More abstract is usually better than more concrete
- Model a security-critical subsystem to gain confidence
- When the diagram gets too big, break it up into sub diagrams

Threat Modeling: Common Pitfalls

Loosing yourself in super-unlikely threat scenarios.

- The nation-state attacker is likely not your biggest problem
- Stay within your area of influence
- Time-box your efforts, especially when getting started
- Document threats you won’t mitigate to stop circulating around them

Threat Modeling: Common Pitfalls

Loosing yourself in discussions of just one threat.

- It is important to be aware of threats
- Not to have the perfect severity rating / description / categorization
- Start small and moderate the discussions

Threat Modeling: Common Pitfalls

Wanting to mitigate each threat.

- A known, accepted risk is better than an unknown or ignored one
- Make deliberate recommendations about risk acceptance and let business make a written decision
- Abstract the matter as much as possible before presenting it to stakeholders

Threat Modeling: Common Pitfalls

Do it once and never again.

- The threat model may change with your requirements, architecture, code, environment, users, and test results
- Do it on a regular basis
- Make it a part of your daily tasks

Common Software Security Design Patterns

Pattern name	Problem(s) targeted	STRIDE
Message queueing	Backpressure (DoS)	STRI**D**E
Data transfer objects	Mass assignment, excessive data exposure	S**T**R**I**DE
Device tokens	Account lock-out, credentials guessing, non-transparency	**STRI**DE
Single sign-on	Phishing susceptibility, account threats	**STRI**DE
Replication	Availability problems	STRI**D**E
Sandboxing	High attack blast radius	**STRIDE**

Design Pattern: Message Queueing

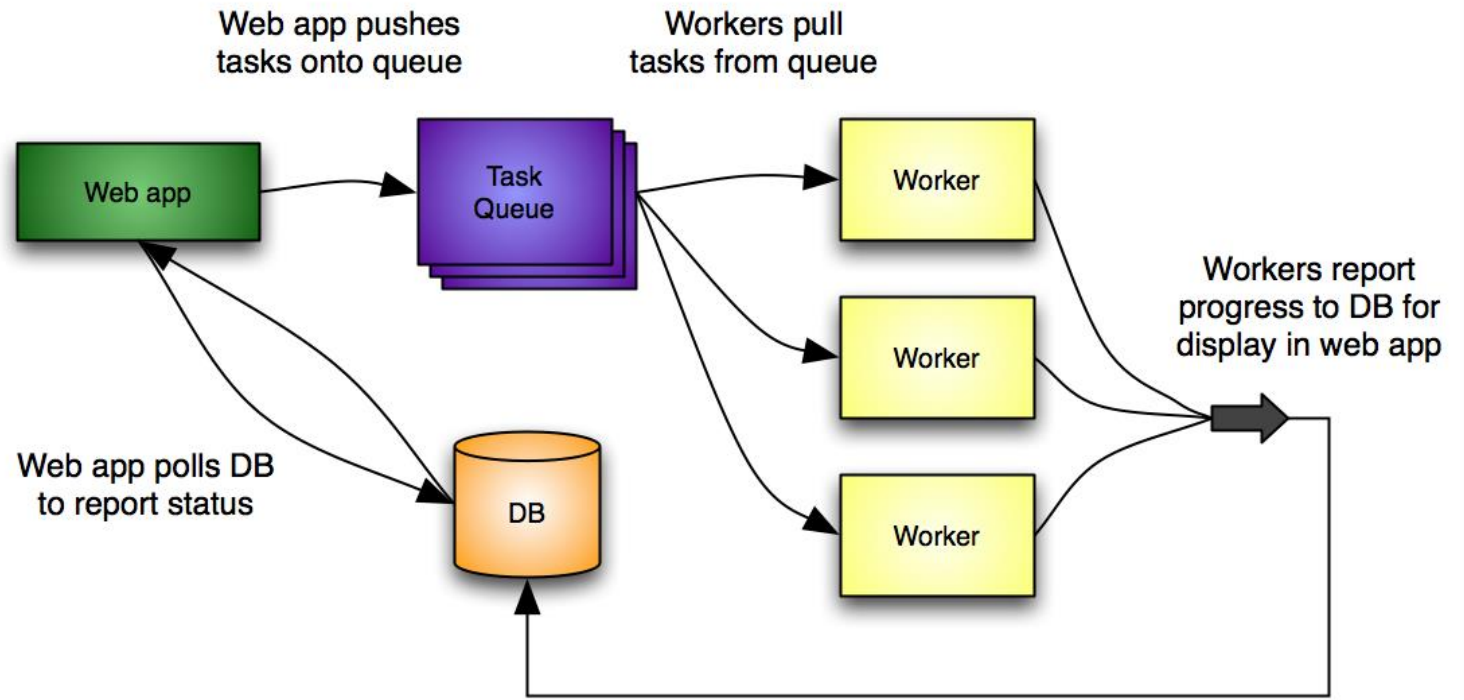
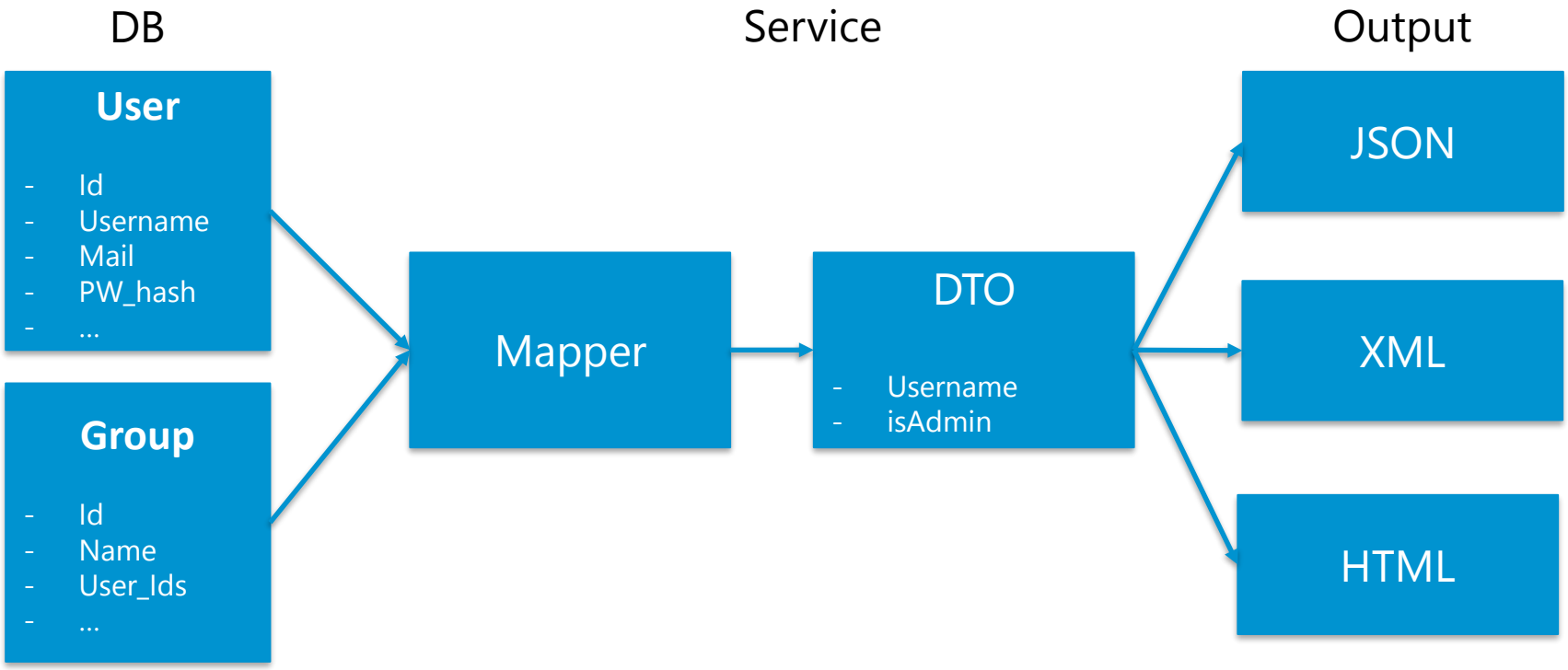
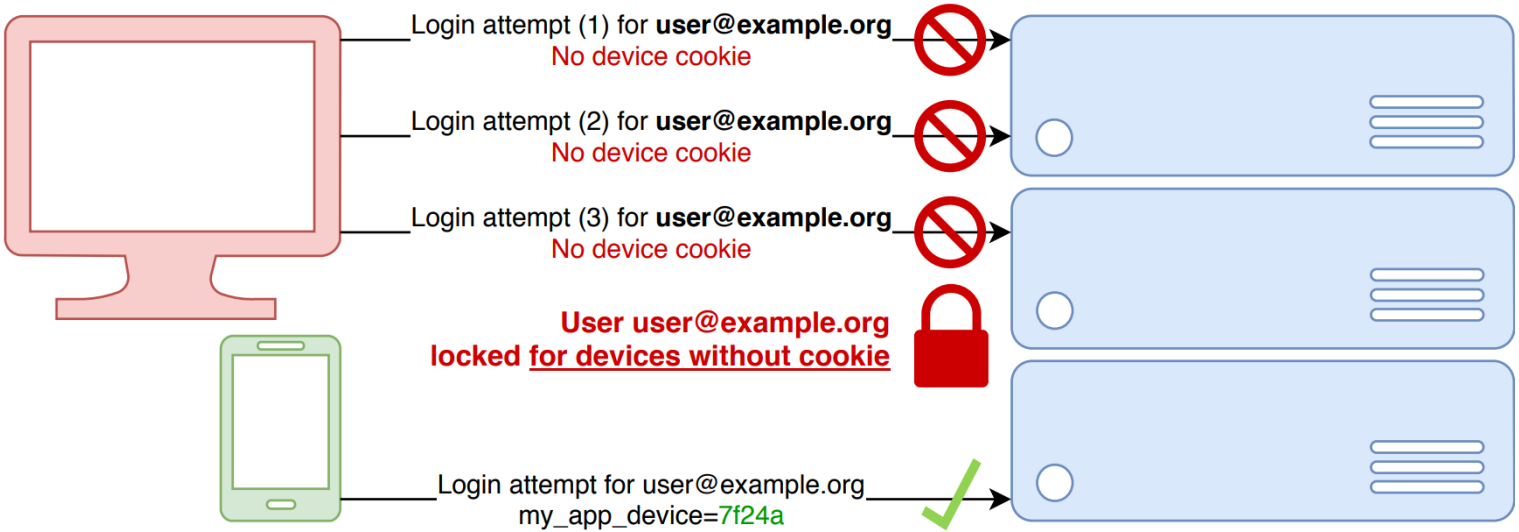


Image source: https://digitheadslabnotebook.blogspot.com/2010/10/message-queues-with-python.html

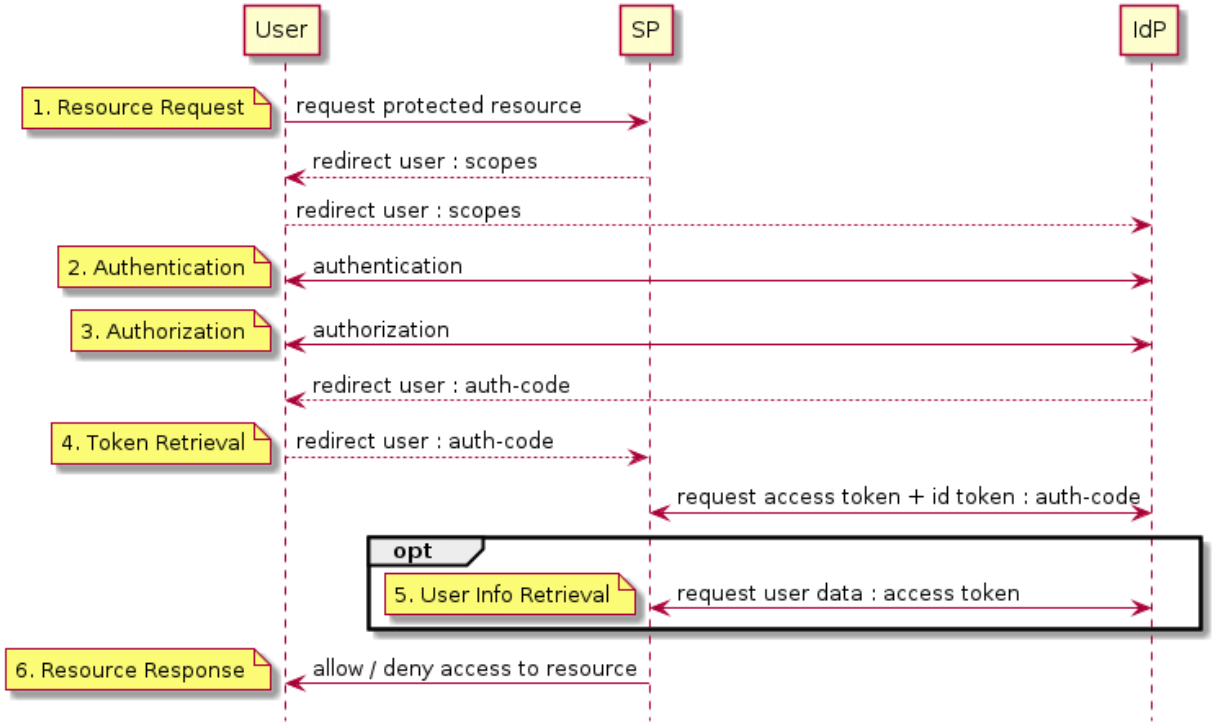
Design Pattern: Data Transfer Objects (DTO)



Design Pattern: Device Tokens



Design Pattern: Single Sign-On



Design Pattern: Replication

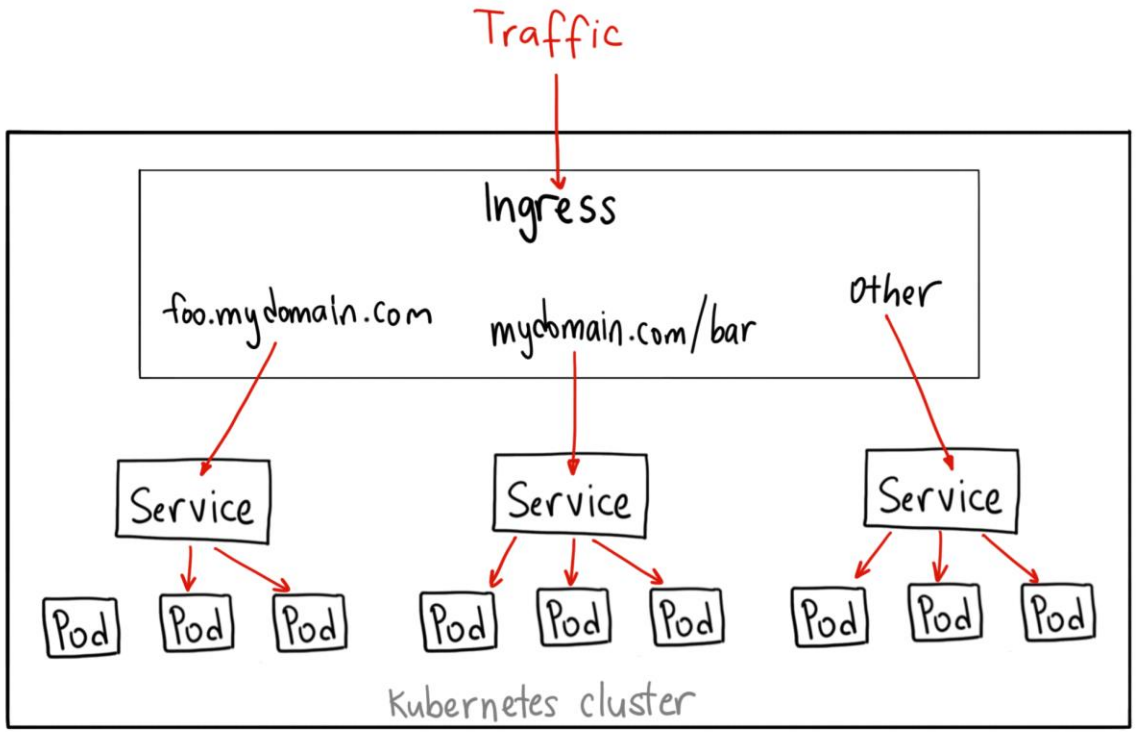
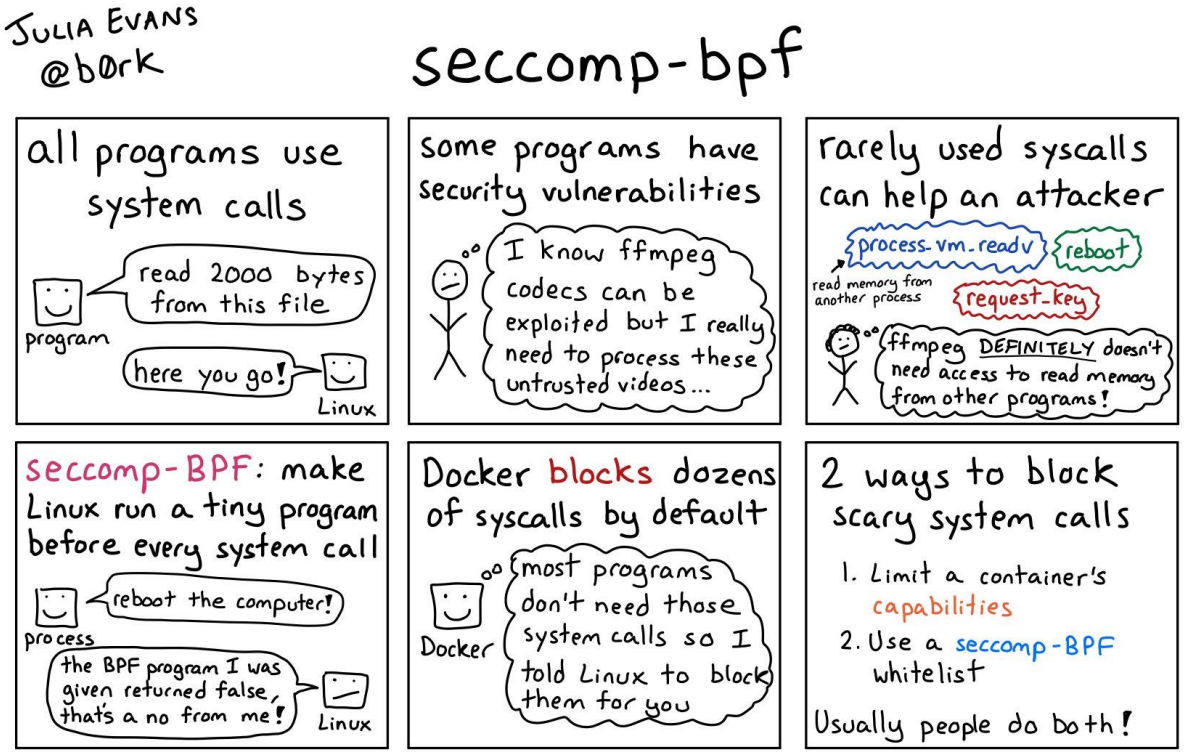


Image source: https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0

Design Pattern: Sandboxing

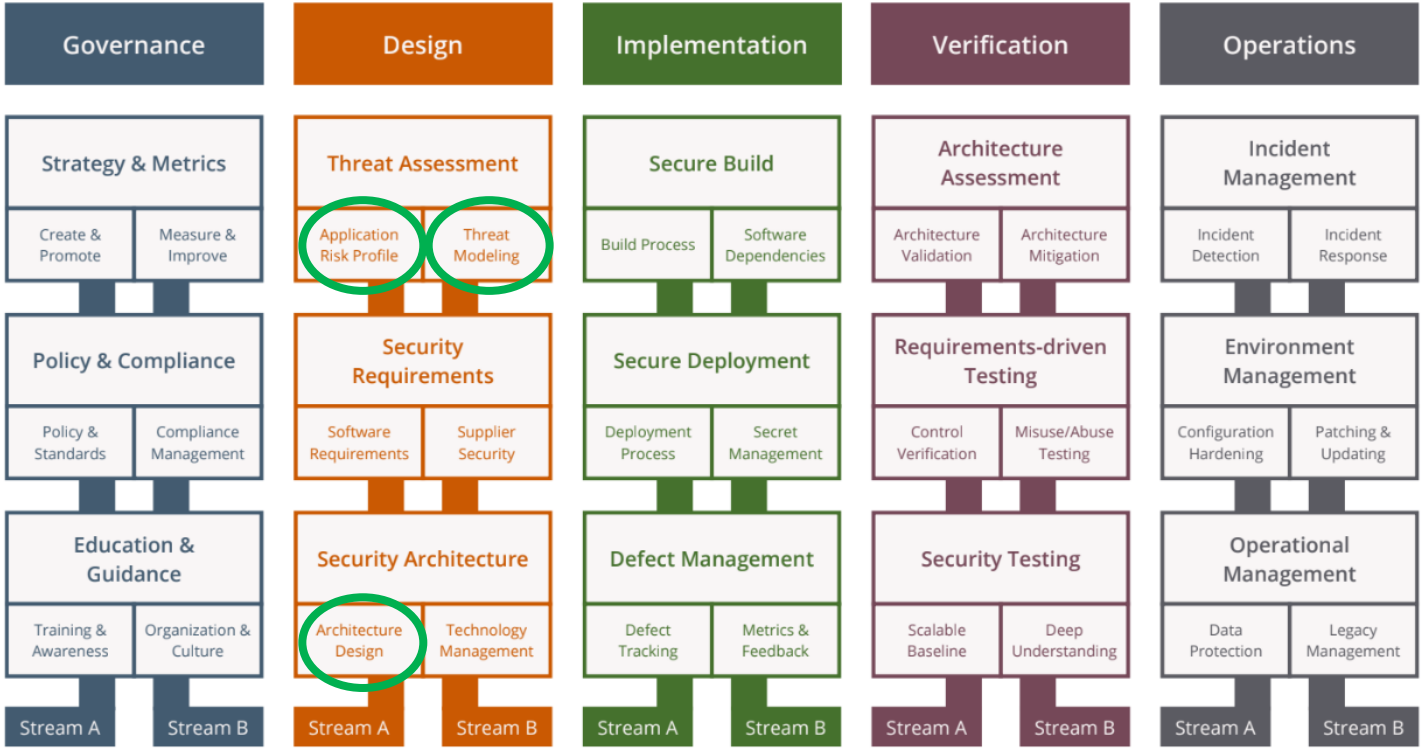


It’s Not Always About Threats

Less can be more

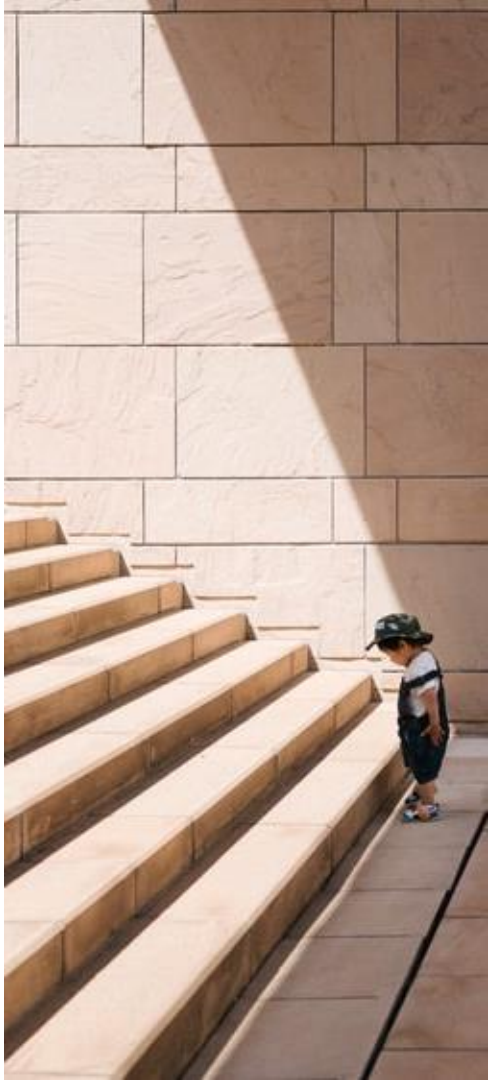
- “We don’t even see your data because we use end-to-end encryption”
- “We do not process your credentials as we use a trusted identity provider”
- “Your system is completely separated from that of other organizations, so their potential breach won’t affect you”

Where Are We In OWASP SAMM?



Essentials: Secure Software Design

- ❑ Create awareness about your software's risk profile
- ❑ Explicitly include security in your design considerations
- ❑ Have a threat model at least for security-critical subsystems
- ❑ Document and share patterns that solved security problems by design



Quiz!

Secure Software Design

Visit https://kahoot.it and enter the game pin I'll share with you!

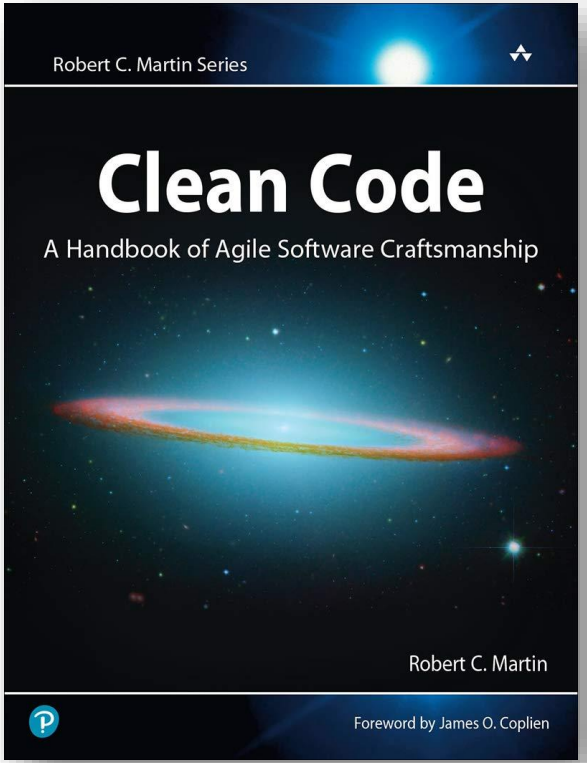
Secure Implementation

Security at the coding level

Secure Coding Practices

- Input handling
- Output handling
- Pitfalls in low-level languages
- The Principle of Complete Mediation
- Cryptography
- Session management
- Concurrency

Book Recommendation: Clean Code



Principles of Readable Code

1. Single responsibility
2. Well-structured
3. Thoughtful naming
4. Simple and concise
5. Comments explain „why“, not „how“
6. Continuously refactored for readability
7. Well-tested

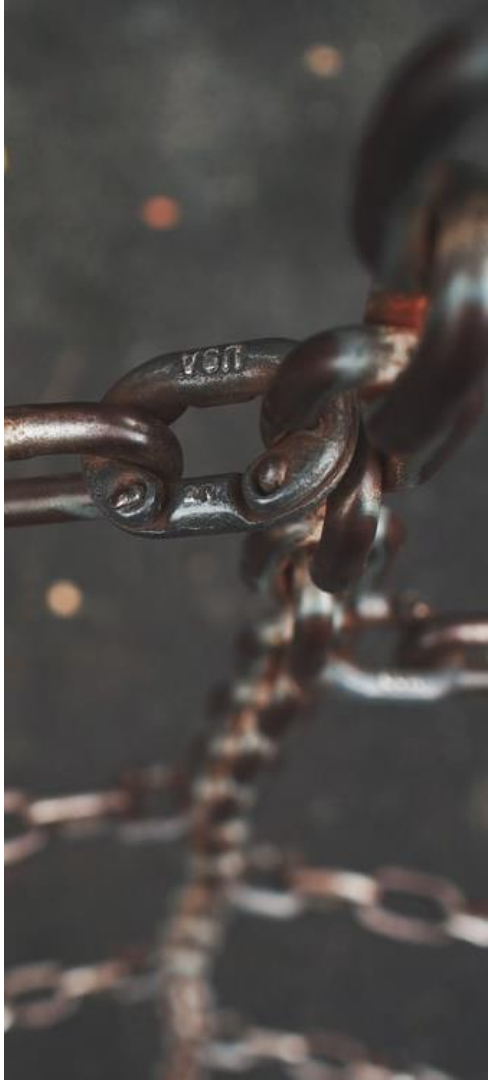
Source: https://blog.pragmaticengineer.com/readable-code/

Why Even Bother?

- **Unreadable code tends to be insecure**
  - Unreadable code is hard to understand
  - No understanding means creative thinking about how to circumvent security measures is basically impossible
- **Unreadable code adds to your technical debt**

Software Dependencies

- More specialization
- More division of tasks
- Longer supply chains with smaller chain links
- **This enables systemic risk!**



Modern Digital Infrastructure

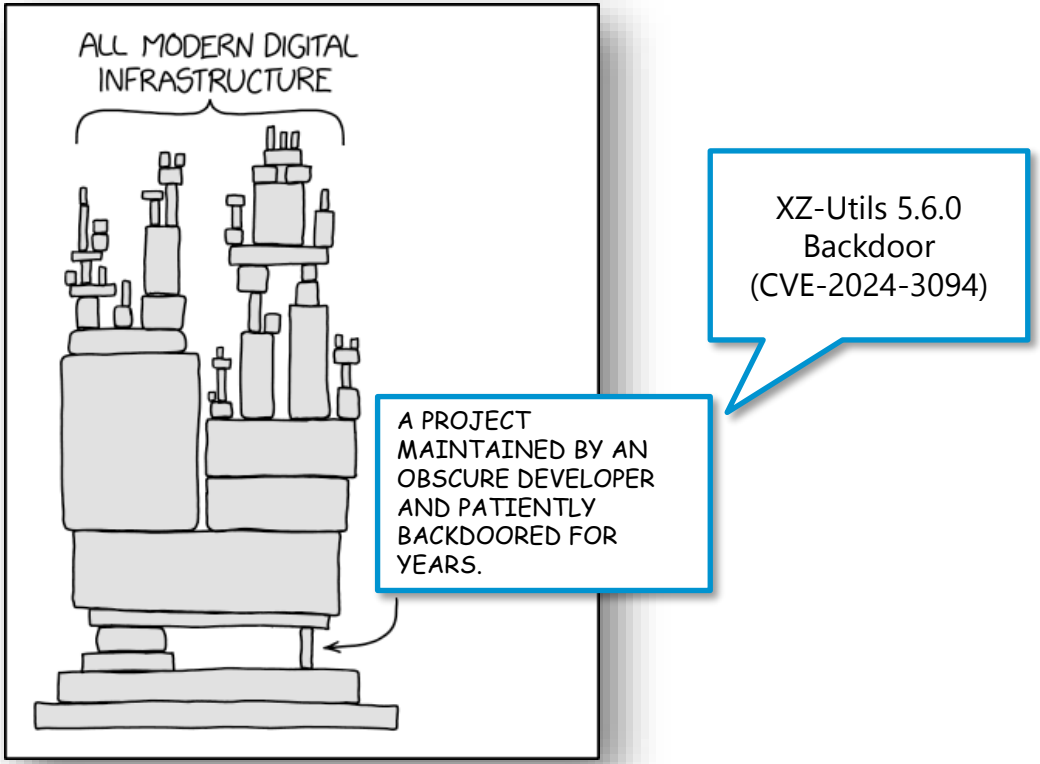
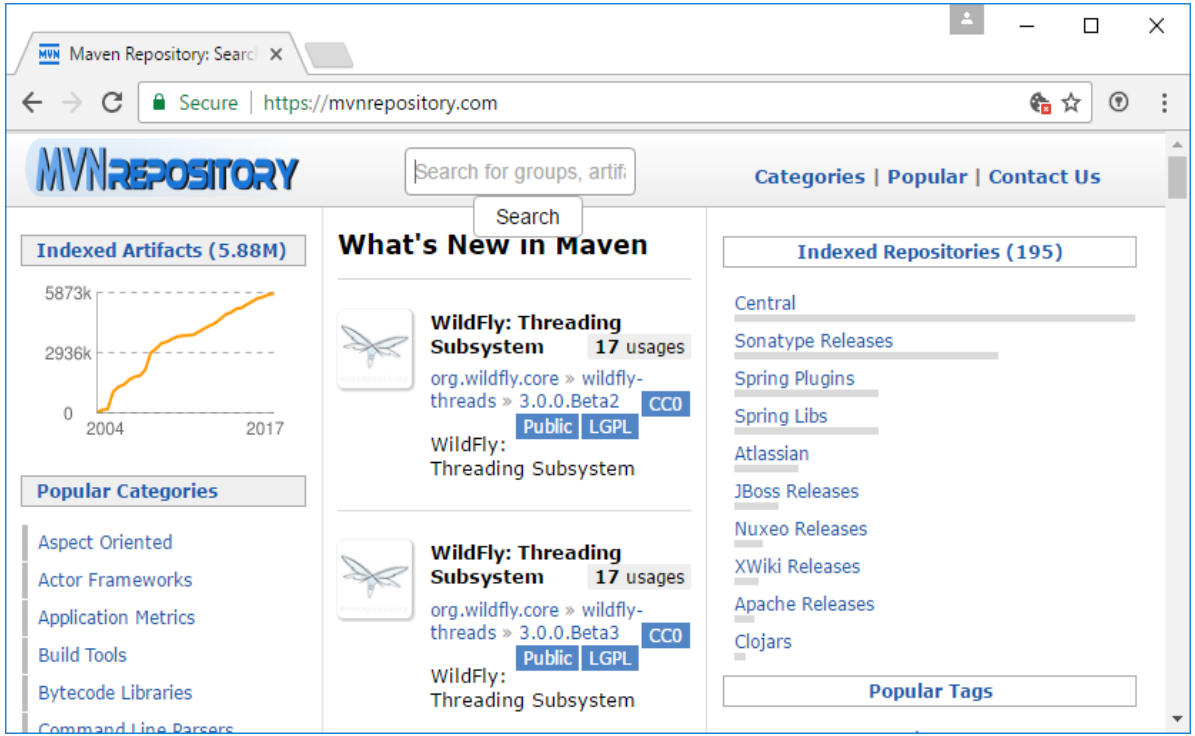
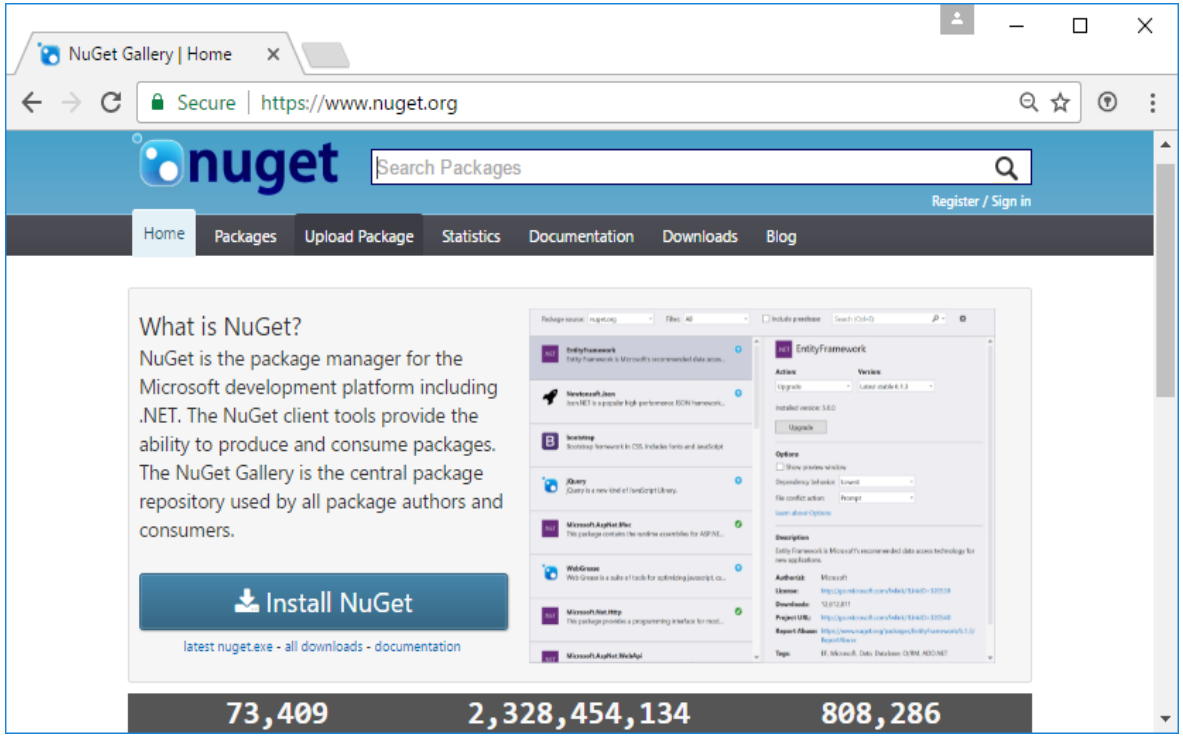


Image source: https://xkcd.com/2347/

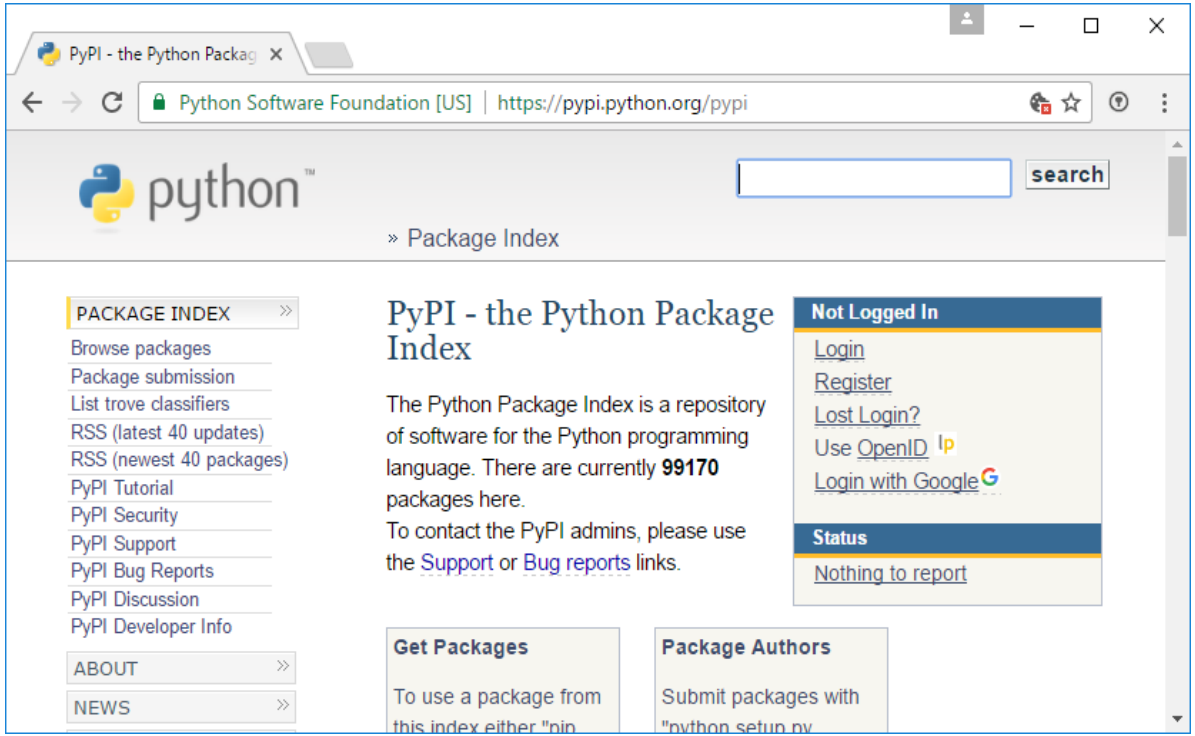
What Is A Software Component?



What Is A Software Component?



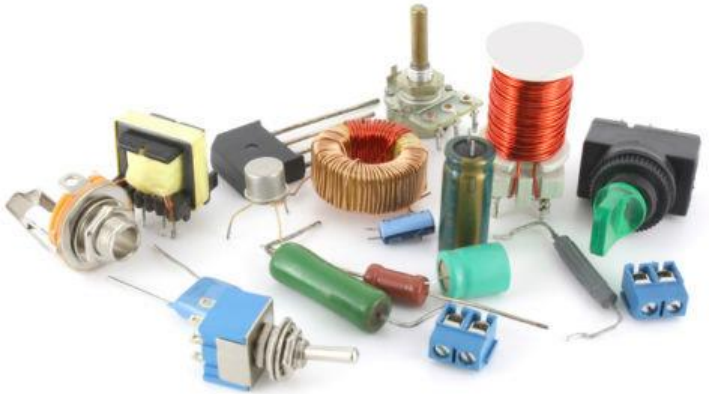
What Is A Software Component?



What Is A Software Component?

Components in software

- Libraries
- Frameworks
- Runtimes (e.g. JVM)
- Base images (e.g. Docker)



Foreign Code Usually Prevails

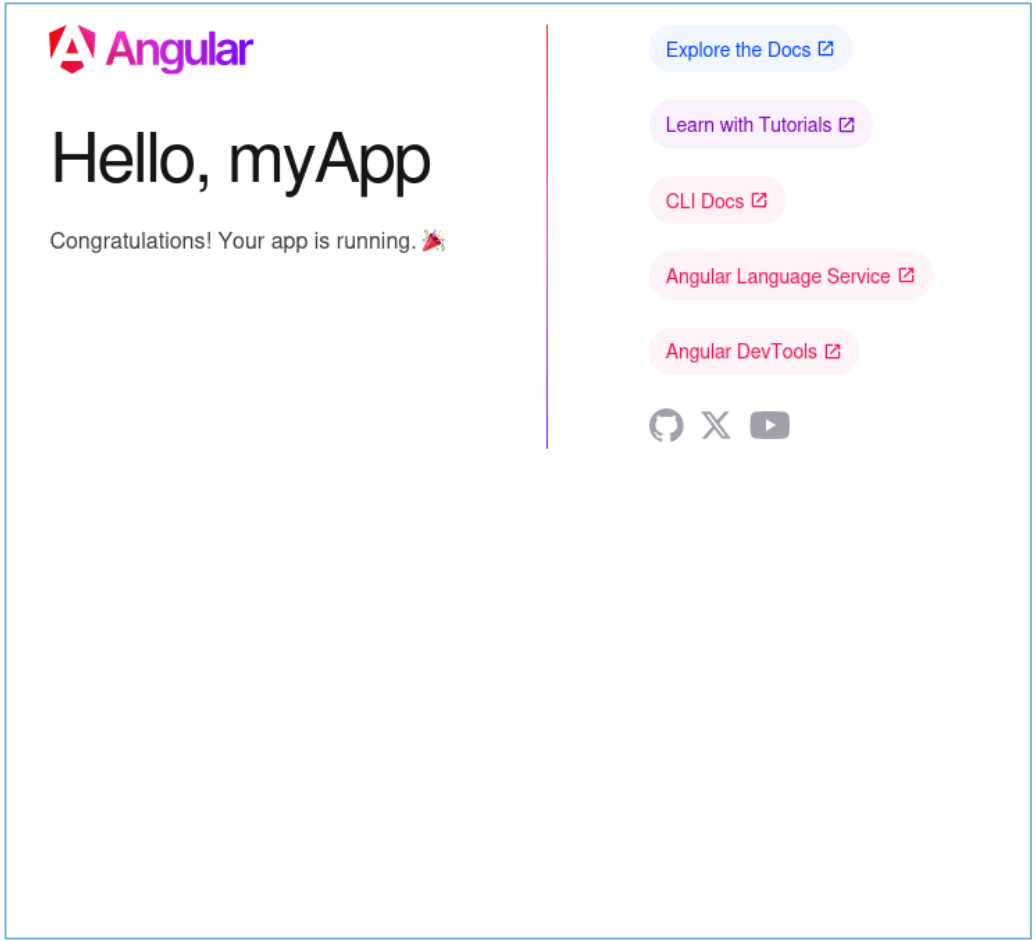
Foreign code usually makes up for > 50 % of running code!

- We cannot check every line of code
- But we can check them for known vulnerabilities
- Dependencies must be declared machine-readable!

Angular Project with Router and SCSS

```
$ npm install @angular/cli
$ node_modules/.bin/ng new --routing myApp
? Which stylesheet format would you like to use? Sass (SCSS)
? Enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? Yes
$ ~/go/bin/scc myApp/node_modules
```

Language	Files	Lines	Blanks	Comments	Code	Complexity
JavaScript	12278	2819766	112262	332911	2374593	382431
TypeScript Typings	3869	370244	14382	190720	165142	69572
JSON	1388	170634	148	0	170486	0
Markdown	1243	220145	60367	0	159778	0
TypeScript	466	43353	3307	13439	26607	5032



That's 3.2 M LoC in node_modules

https://angular.io/guide/setup-local

Step 1: Transparency

You can’t protect yourself, if you don’t know what you have

Good

- Machine-readable list of *declarative dependencies*

Bad

- DLL, .so, JAR, etc. files spread over the codebase and checked into the repository

Software Bill of Materials

- We can build a *Software Bill of Material (SBOM)* for our applications
- Multiple formats, e.g., CycloneDX
- Lists all components which are part of the application in a structured way
  - Contains the whole *dependency tree*
- Can be analyzed for known vulnerabilities
- Compliance requirement (CRA, NIS2, US laws, …)

Software Bill of Materials

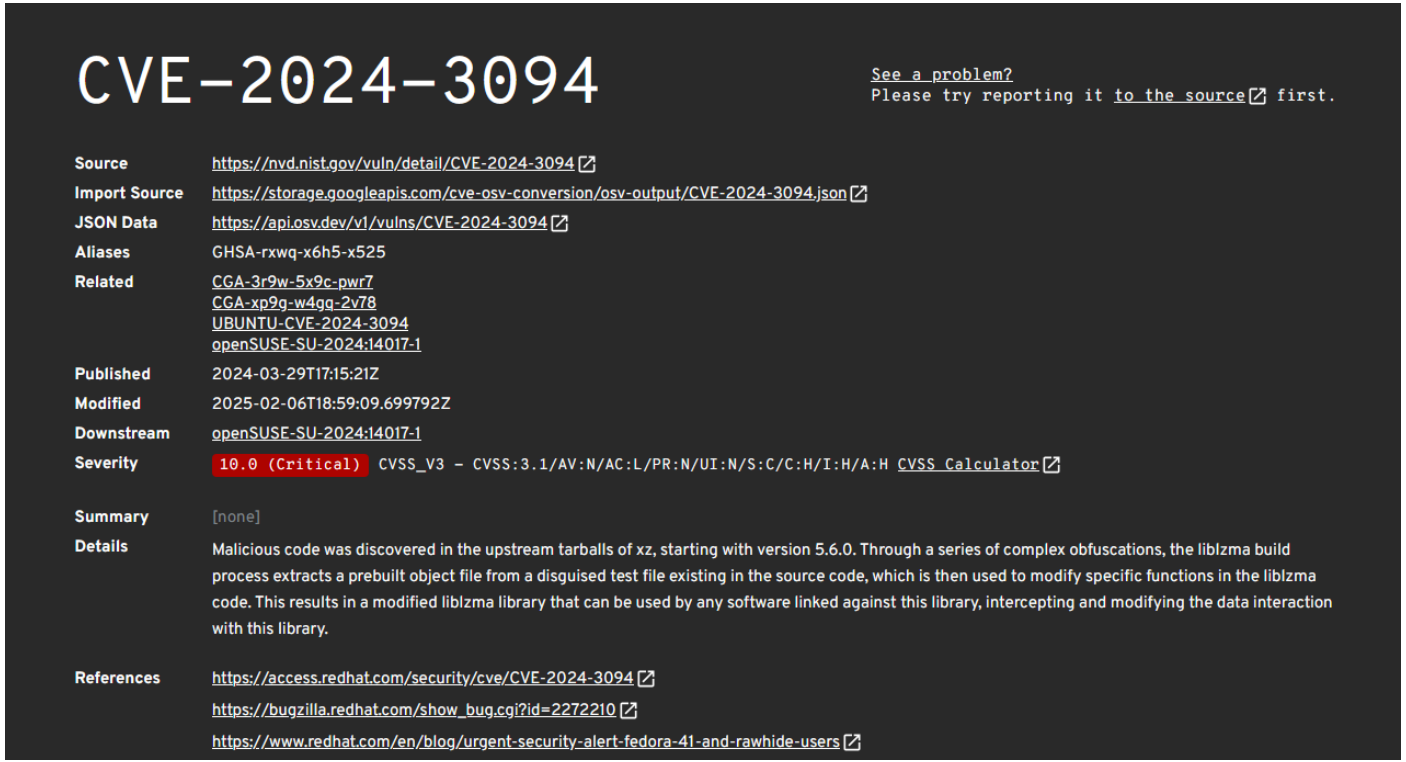
```
$ trivy repository --format=cyclonedx --output=bom.json .
$ cat bom.json

{
  "$schema": "http://cyclonedx.org/schema/bom-1.5.schema.json",
  "bomFormat": "CycloneDX",
  "specVersion": "1.5",
  "metadata": {
    "timestamp": "2024-04-08T09:31:57+00:00",
[…]
  "components": [
    {
      "bom-ref": "pkg:npm/%40angular/animations@17.3.3",
      "type": "library",
      "group": "@angular",
      "name": "animations",
      "version": "17.3.3",
      "licenses": [{"license": {"name": "MIT"}}],
      "purl": "pkg:npm/%40angular/animations@17.3.3",
```

Step 2: Identifying Problems

- All components can have *known vulnerabilities*
- Those are identified by a *CVE (Common Vulnerability Enumeration) number* across all databases
- Databases list all known vulnerabilities
  - E.g., https://www.cvedetails.com/, https://euvd.enisa.europa.eu/, https://osv.dev/

CVE



CVE-2024-3094

See a problem?
Please try reporting it to the source first.

Source	https://nvd.nist.gov/vuln/detail/CVE-2024-3094
Import Source	https://storage.googleapis.com/cve-osv-conversion/osv-output/CVE-2024-3094.json
JSON Data	https://api.osv.dev/v1/vulns/CVE-2024-3094
Aliases	GHSA-rxwq-x6h5-x525
Related	CGA-3r9w-5x9c-pwr7 CGA-xp9g-w4gq-2v78 UBUNTU-CVE-2024-3094 openSUSE-SU-2024:14017-1
Published	2024-03-29T17:15:21Z
Modified	2025-02-06T18:59:09.699792Z
Downstream	openSUSE-SU-2024:14017-1
Severity	10.0 (Critical) CVSS_V3 - CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H CVSS Calculator
Summary	[none]
Details	Malicious code was discovered in the upstream tarballs of xz, starting with version 5.6.0. Through a series of complex obfuscations, the liblzma build process extracts a prebuilt object file from a disguised test file existing in the source code, which is then used to modify specific functions in the liblzma code. This results in a modified liblzma library that can be used by any software linked against this library, intercepting and modifying the data interaction with this library.
References	https://access.redhat.com/security/cve/CVE-2024-3094 https://bugzilla.redhat.com/show_bug.cgi?id=2272210 https://www.redhat.com/en/blog/urgent-security-alert-fedora-41-and-rawhide-users

https://osv.dev/vulnerability/CVE-2024-3094

Software Composition Analysis

Software Composition Analysis (SCA) tools automatically identify vulnerable components

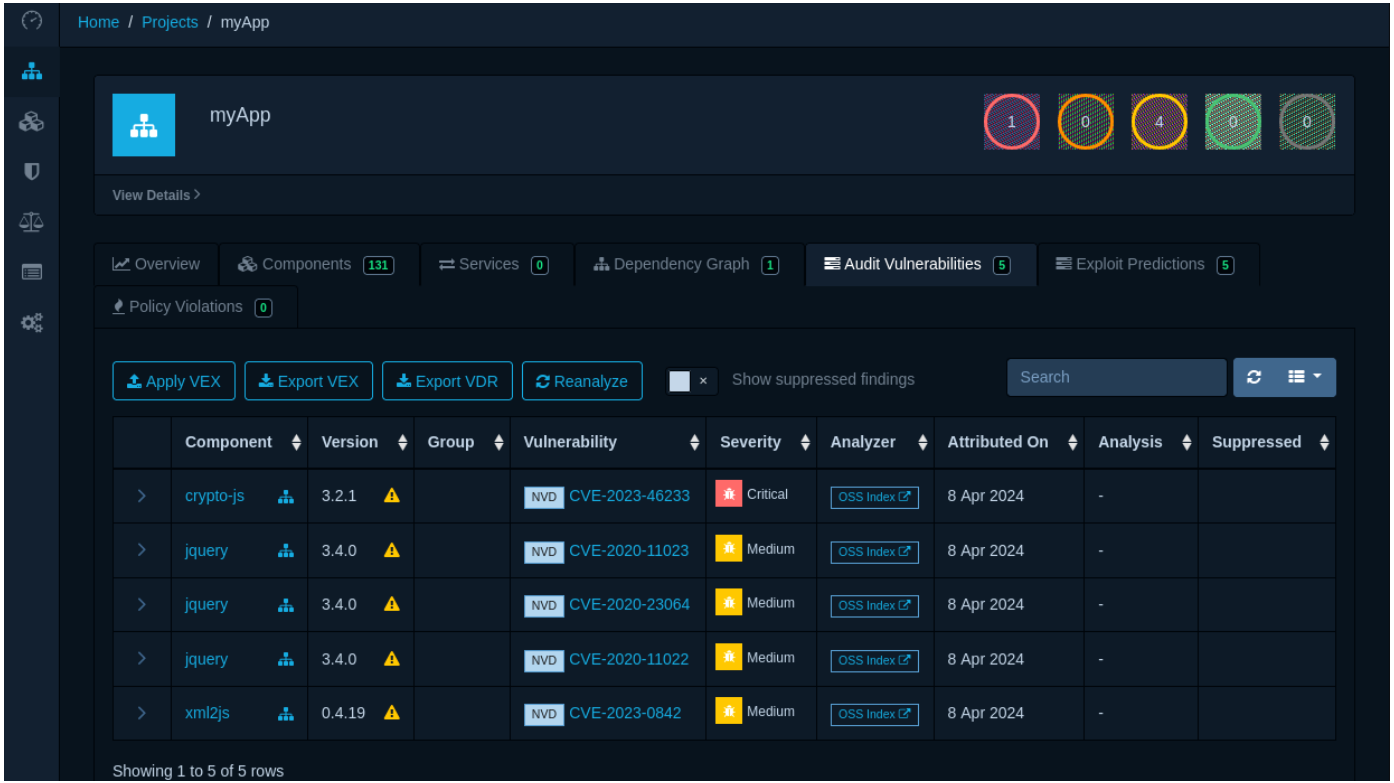
```
$ trivy image finalgene/hadolint

finalgene/hadolint (alpine 3.9.4)
Total: 26 (UNKNOWN: 0, LOW: 4, MEDIUM: 14, HIGH: 6, CRITICAL: 2)

┌──────────────┬─────────────────┬──────────┬─────────┬───────────────┬─────────┐
│   Library    │ Vulnerability   │ Severity │ Version │ Fixed Version │ Title   │
├──────────────┼─────────────────┼──────────┼─────────┼───────────────┼─────────┤
│ libcrypto1.1 │ CVE-2020-1967   │ HIGH     │ 1.1.1b-r1 │ 1.1.1g-r0   │ […]     │
├──────────────┼─────────────────┼──────────┼─────────┼───────────────┼─────────┤
│              │ CVE-2021-23840  │          │         │ 1.1.1j-r0     │ […]     │
├──────────────┼─────────────────┼──────────┼─────────┼───────────────┼─────────┤
[…]
```

Software Composition Analysis

Dependency Track



Software Composition Analysis

Tools

- Generic
  - Trivy (Open Source)
  - OWASP Dependency Track (Open Source)
  - OWASP Dependency Check (Open Source)
  - GitLab Dependency Checker (Ultimate Feature)
  - GitHub (through Dependabot)
  - Snyk Open Source Security Management (commercial)
  - JFrog Xray (commercial)
  - Sonatype Nexus (commercial)
  - Black Duck (commercial)
- Language specific
  - npm audit (NPM)
  - composer audit (PHP, Composer)
  - NuGetDefense, dotnet CLI (.NET)
  - Safety, PyRaider (Python)
  - govulncheck (Go)
  - cargo audit, cargo deny (Rust)
  - bundler-audit (Ruby)

Step 3: Automate

1. Generate SBOM in build pipeline
2. Check SBOM for known vulnerabilities with SCA tool directly in the pipeline or an external tool
3. Have a process to mitigate findings in a timely fashion
4. Bonus: Do it **regularly** even if no pushes happen!

Panic?

https://grafana.com/blog/2023/05/25/how-to-manage-cve-security-vulnerabilities-with-grafana-mergestat-and-osv-scanner/

Don’t Panic

- A vulnerability in a component is not necessarily a vulnerability in your application
- A vulnerability is not necessarily exploitable
- Severity ratings are probably not accurate

Step 4: Act

After establishing the problem, you can pursue multiple roads:

- Update dependency
  - Preferably automated (e.g., *Dependabot*)
- If an update is not possible: Analyze vulnerability and react
  - Patch vulnerabilities yourself
  - Remove or replace dependency
  - Accept risk

Analyze

Distinguishing the Signal and the Noise

- Remove *False Positives*
  - Package imported?
  - Function used?
- Prioritize
  - Likelihood of exploitation
    - Exploit Prediction Scoring System (EPSS)
    - Known Exploited Vulnerabilities (KEV) Catalog
    - Local *Computer Security Incident Response Team (CSIRT)* e.g., https://www.cert.at/
    - Age of vulnerability
  - Severity rating

Step 0: Minimize Problem

Be picky with your libs!

- Do not pull in a library for everything
  - Sometimes, Copy/Paste is your friend
- Invest some time in quality checks
- Check libraries for known vulnerabilities in an automated way

Problematic Dependencies

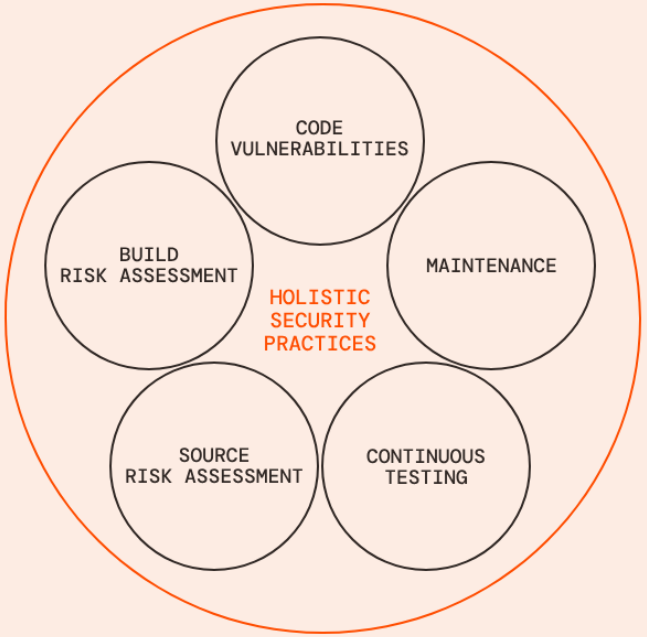
- Quality of components varies
  - Activity
  - Security contact
  - Reaction to previous security issues
- Many packages become unmaintained
- License changes

Problematic Dependencies

Security Scorecards

*Scorecard checks for vulnerabilities affecting different parts of the software supply chain including **source code**, **build**, **dependencies**, **testing**, and project **maintenance**.*

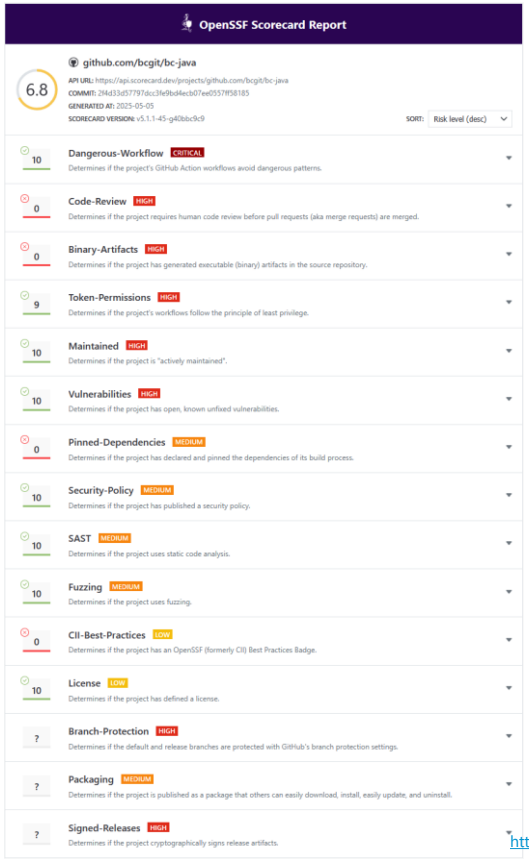
*Each automated check returns a **score out of 10** and a **risk level**. This score helps give a sense of the overall security posture of a project*



https://securityscorecards.dev/

Problematic Dependencies

Security Scorecards



https://scorecard.dev/viewer/?uri=github.com/bcgit/bc-java

Dependency usage cycle

Essentials: Dependencies

- ❑ Choose your dependencies wisely
- ❑ Have a declarative, machine-readable list of dependencies
- ❑ Produce and manage SBOMs
- ❑ Check your dependencies in an automated way
- ❑ Notify relevant stakeholders if there are severe vulnerabilities or incompatible license changes
- ❑ Rebuild and run checks regularly even if there is no push
- ❑ **Advanced**
  - ❑ Have a good test coverage
  - ❑ A bot submits a pull request with updates
  - ❑ Use Dependency Cooldown
  - ❑ Merge it automatically if tests are green

Secure Build
Build Automation

Have an automated, repeatable build process!

- This lowers the error-proneness.
- It fosters continuous improvement.
- It eases the introduction of verifiable builds.
- Allows continuous testing
- Enables regular usage of security tooling

Secure Build
Build Automation

Protect your code and build environment!

- Is it reachable only via VPN?
- Is MFA enabled for everybody?
- Are only those able to commit code who need to?
- Are build artifacts protected from manipulation?

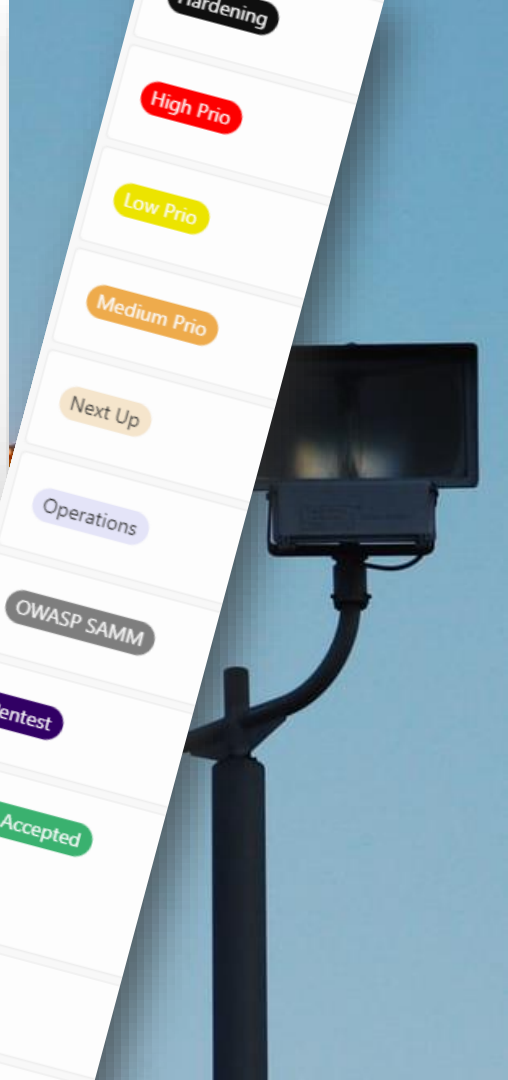
CI Risks

- CI systems are security critical!
- Especially if they do deployments as well
- [OWASP Top 10 CI/CD Security Risks]

Defect Tracking

Track security defects like any other defect, but label it as such.

- We’re all about making security an integral part of development, right?
- Easiest with the labelling functionality of your code management tool
- “What’s currently your most important security problem?”

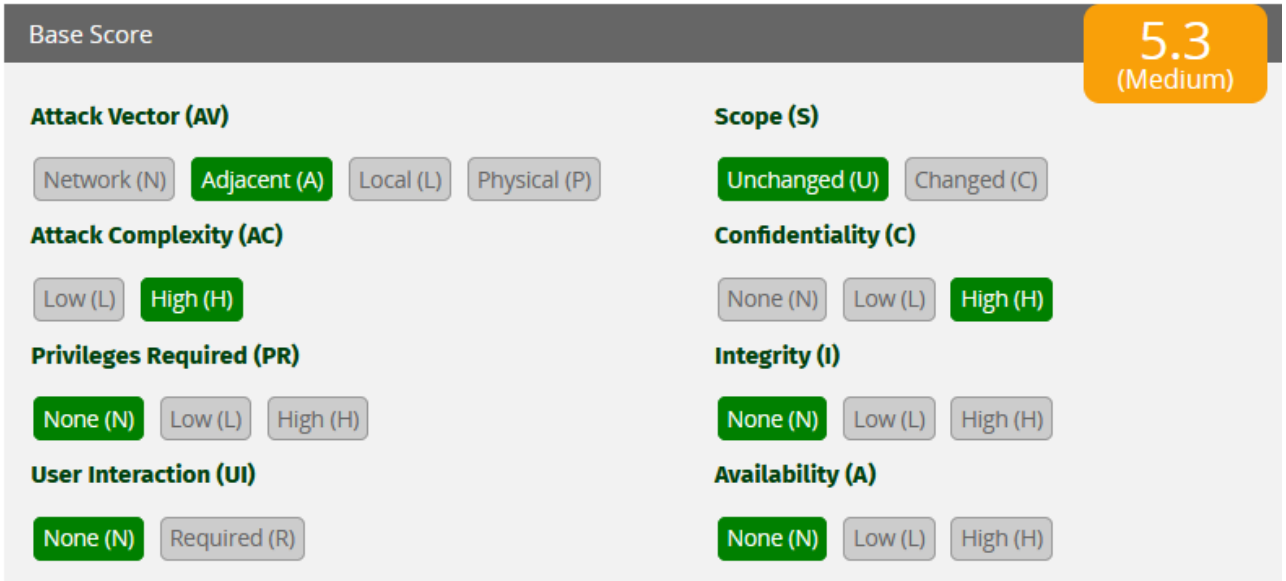


Defect Tracking
CVSS

- „Common Vulnerability Scoring System“
- Grades the **severity** not the **risk**
- Risk must be graded afterwards
- 3 parts
  - Base Score
  - Temporal Score
  - Environmental Score

Defect Tracking
CVSS Example

A document management system on the intranet makes its file readable to anyone knowing the link e.g. https://example.com/file/397rtfg2yisa8r3f (*Insecure Direct Object Reference*).



https://www.first.org/cvss/calculator/3.1#CVSS:3.1/AV:A/AC:H/PR:N/UI:N/S:U/C:H/I:N/A:N

Defect Tracking

Don’t take severity levels as business impact.

- Severity levels such as a CVSS 3.1 score are technical
- Tools cannot know the business impact
- Pentesters rarely know neither

Secret Management

Your build & deployment need *secrets*. Handle them well

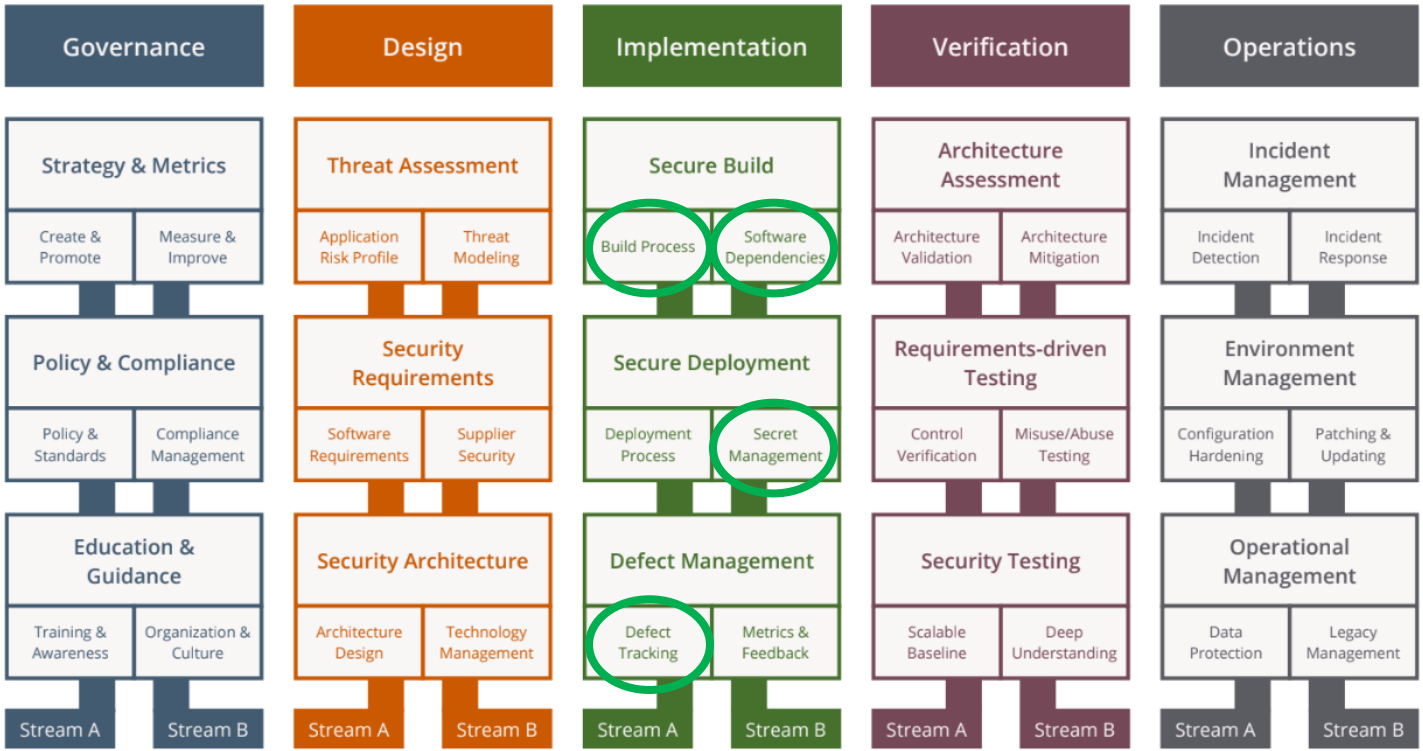
- Secure generation
- Secure Storage
- Access Rights
- Lifecycle management

Secret Management

No secrets in the source code

- Never check any secrets into your version control system (VCS)
- Inject secrets during deployment
  - Configuration file
  - Environment variable
- Use *Secret Detection* tool in your pipeline

Where Are We In OWASP SAMM?



Quiz!

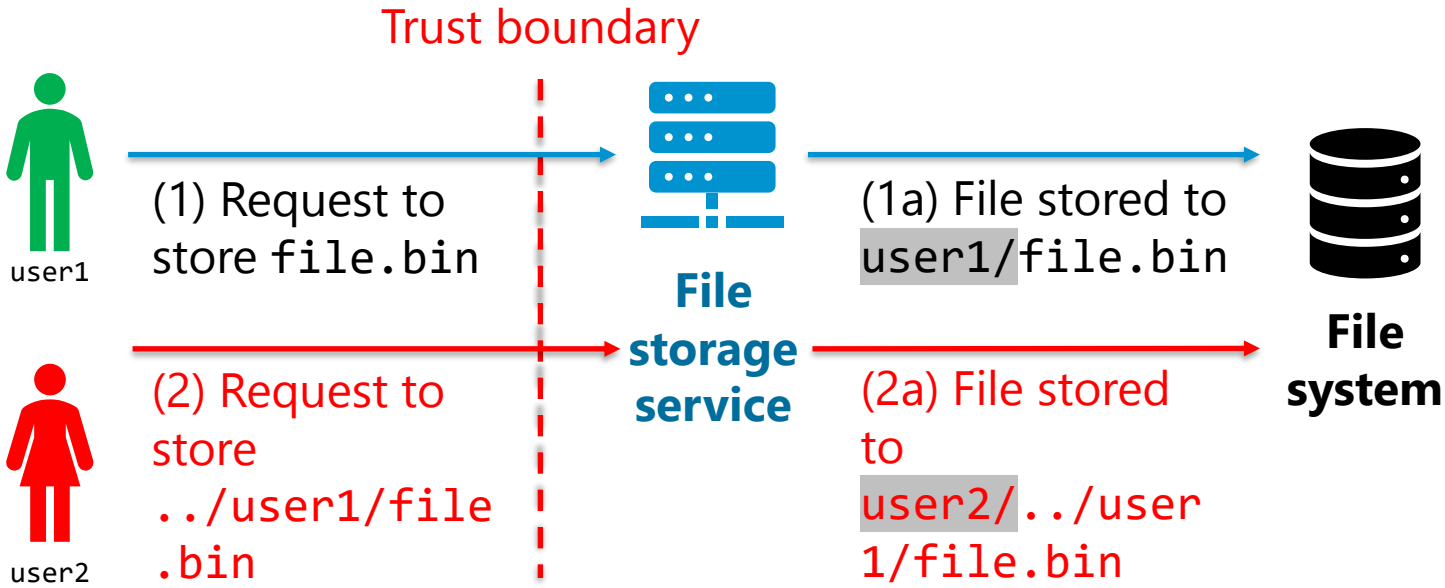
Secure Implementation

Visit https://kahoot.it and enter the game pin I'll share with you!

Software Security Verification

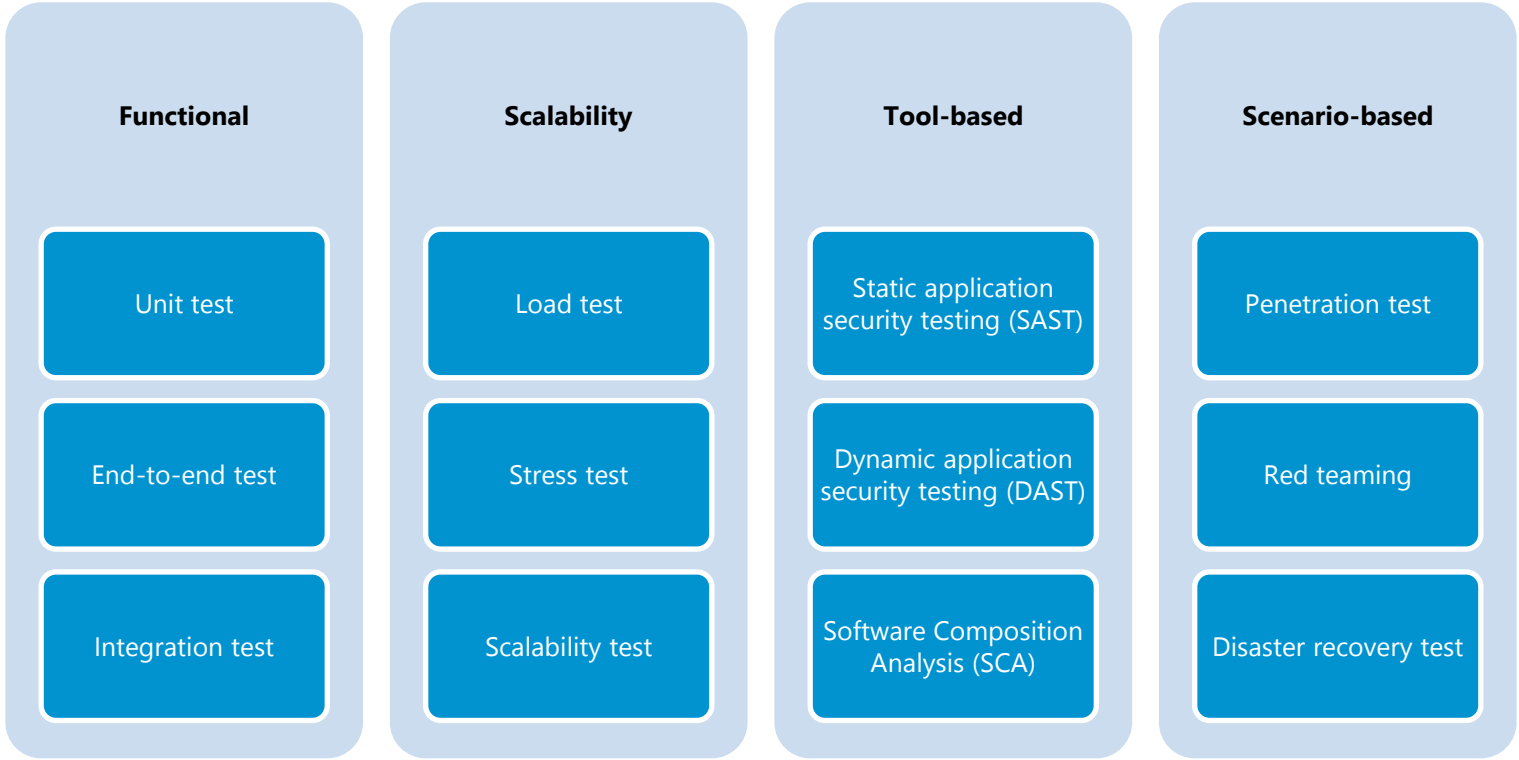
Types of software tests, strengths, weaknesses

Real-World Scenario



Core issue: Path traversal vulnerability
Detectable by static analysis

Types of Software Tests



Unit Test vs. Integration Test

Functional Tests

Unit tests and integration tests can (and should) be used for security purposes as well!

- Verify implementation of security requirements
- Authorization tests
- Don’t restrict yourself to the *happy path*
- Write and test *abuse stories* as well

Automated Tool Types

- **Static Application Security Testing** (SAST)
- **Dynamic Application Security Testing** (DAST)
- **Software Composition Analysis** (SCA)

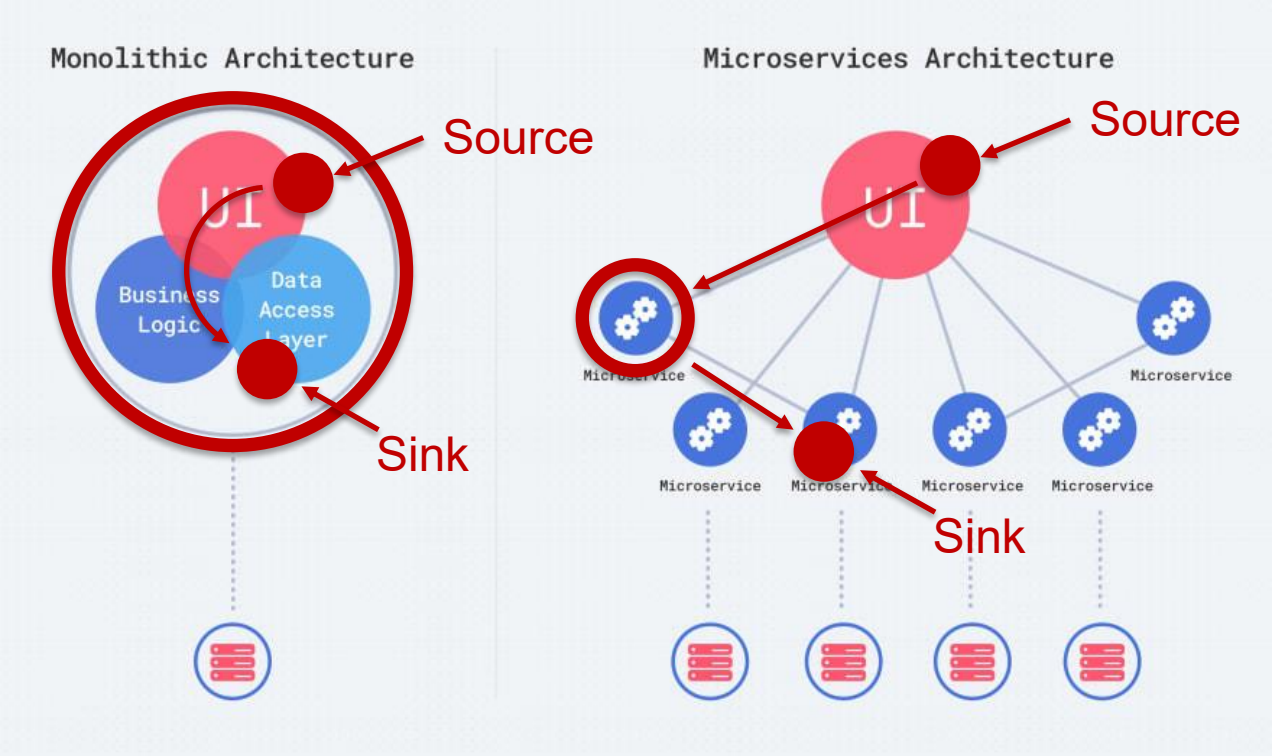
Static Application Security Testing (SAST)

- Scans the source code
- No running application required
- Builds a so-called Abstract Syntax Tree (AST)
- **Approach**: Input – way through your code – sink

Static Application Security Testing (SAST)

- **Advantages**
  - Reproducible results
  - Good code coverage
- **Disadvantages**
  - Usually only covers your own code
  - Can only detect a limited set of vulnerabilities
  - Lacks context when scanning microservices

Microservices and Vulnerability Context



Source: https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m

SAST: Common Pitfalls

Not knowing what the tool can really find in the own environment.

- Know the relevant vulnerability classes of your software
- Assess thoroughly whether SAST can find (some of) them
- Make a deliberate decision on whether this is worth it

SAST: Common Pitfalls

Turning on all the bells and whistles of the tool from the start.

- Expect the “security bump” to be big.
- Give your team digestible chunks of findings to work with.
- Start with critical findings only and work your way down to your goal.

SAST: Common Problem

- Classic tools are dinosaurs
- They’re big, clunky, slow, do way too much
- Hardly compatible with an agile approach and CI/CD pipelines

Semgrep: Lightweight SAST

- A bit like grep, but semantic-aware
- More lightweight
- Can be easily called on your code using a lightweight CLI tool or Docker container
- You must know what you're doing!

```
semgrep --
config="https://semgrep.dev/c/r/javascript.angular.securi
ty.detect-angular-trust-as-html-method.detect-angular-
trust-as-html-method"
```

Semgrep: Rules

```
- pattern-either:
  - pattern: |
      $RETURNTYPE $FUNC (..., @PathParam(...) $TYPE $VAR, ...) {
        ...
        new File(..., $VAR, ...);
        ...
      }
  - pattern: |-
      $RETURNTYPE $FUNC (..., @javax.ws.rs.PathParam(...) $TYPE $VAR, ...) {
        ...
        new File(..., $VAR, ...);
        ...
      }
```

Use with: semgrep CLI | Live Editor | GitHub

Deploy to Policy

Semgrep
Usage

- Prevent known vulnerabilities instead of finding unknown
- Fixate expected behavior
  - Vulnerability in application discovered
  - Find root cause in code
  - Create Semgrep template which detects the vulnerability
  - Run check on whole codebase
  - Integrate in build pipeline
- This allows you eliminate a *vulnerability class* instead of a single *vulnerability*

Semgrep
Learn

Semgrep Academy

All Courses Sign In

Secure Guardrails

Change your application security program from reactive to proactive with this course on how to create secure guardrails! Learn all about paved roads, secure defaults, and creating technical controls to ensure your developers stay on the secure path!

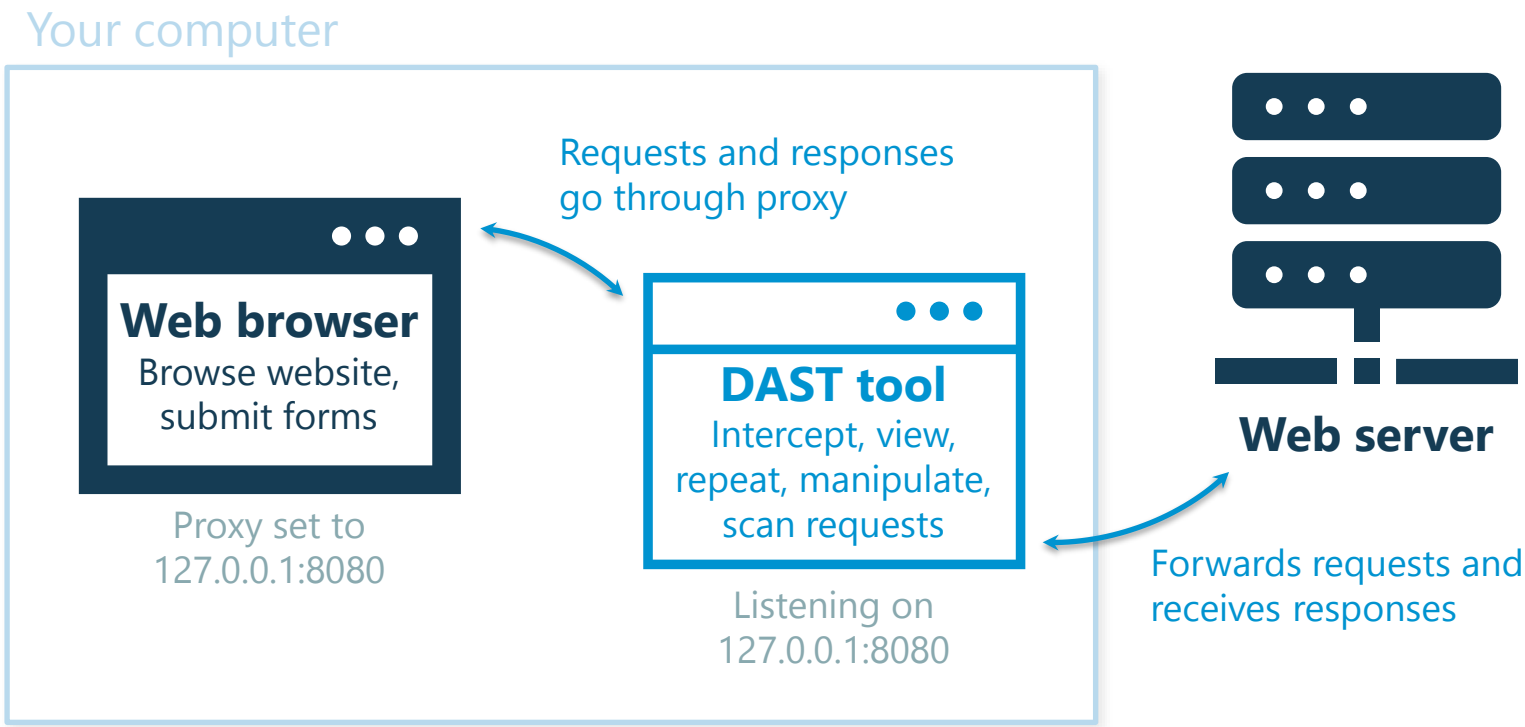
Enroll for free

https://academy.semgrep.dev/courses/secure-guardrails

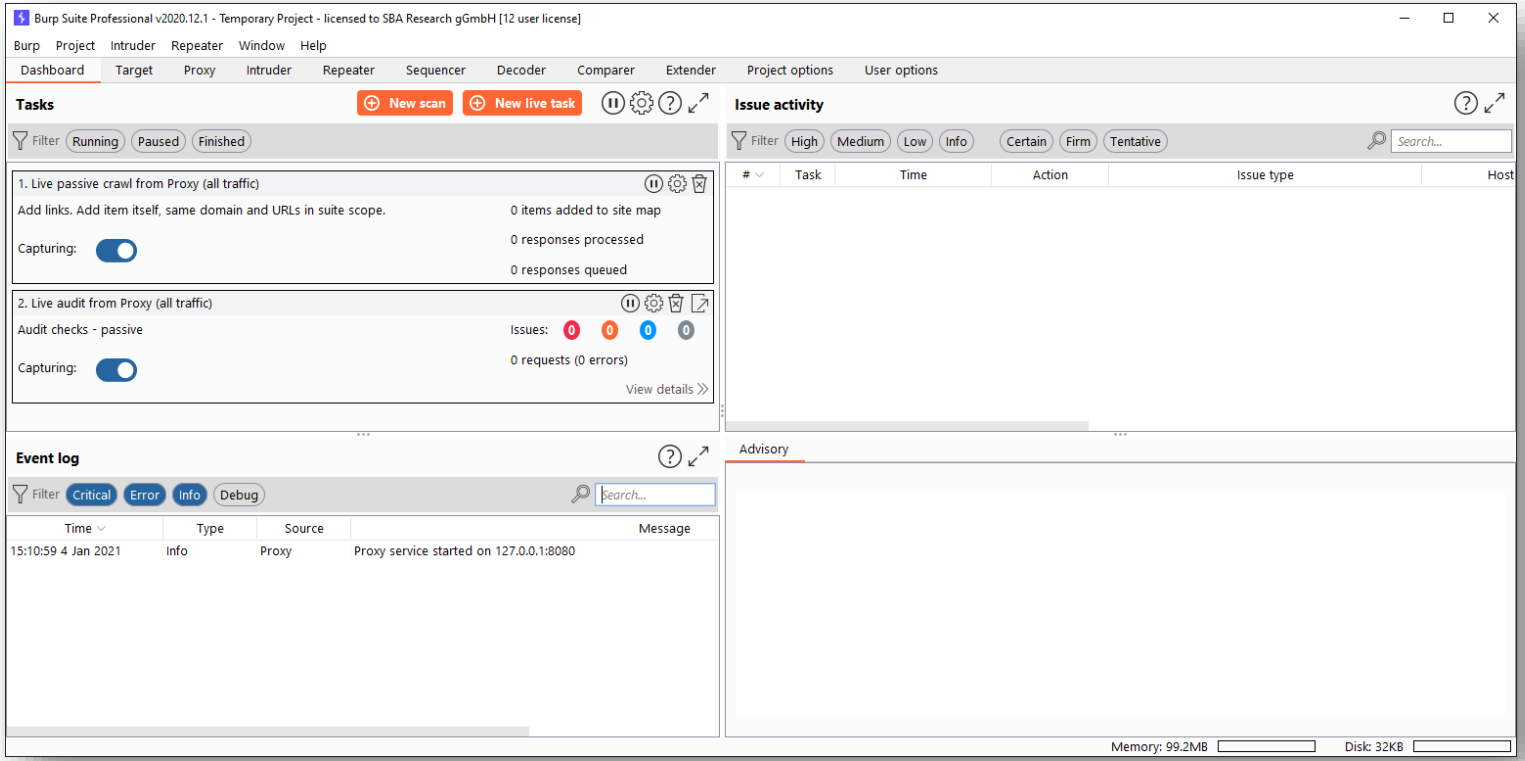
Dynamic Application Security Testing (DAST)

- Scans a running application
- Some tools have SAST elements built in
- **Approach**: Request – response
- Usage can be manual, automatic or semi-automatic

How DAST Works



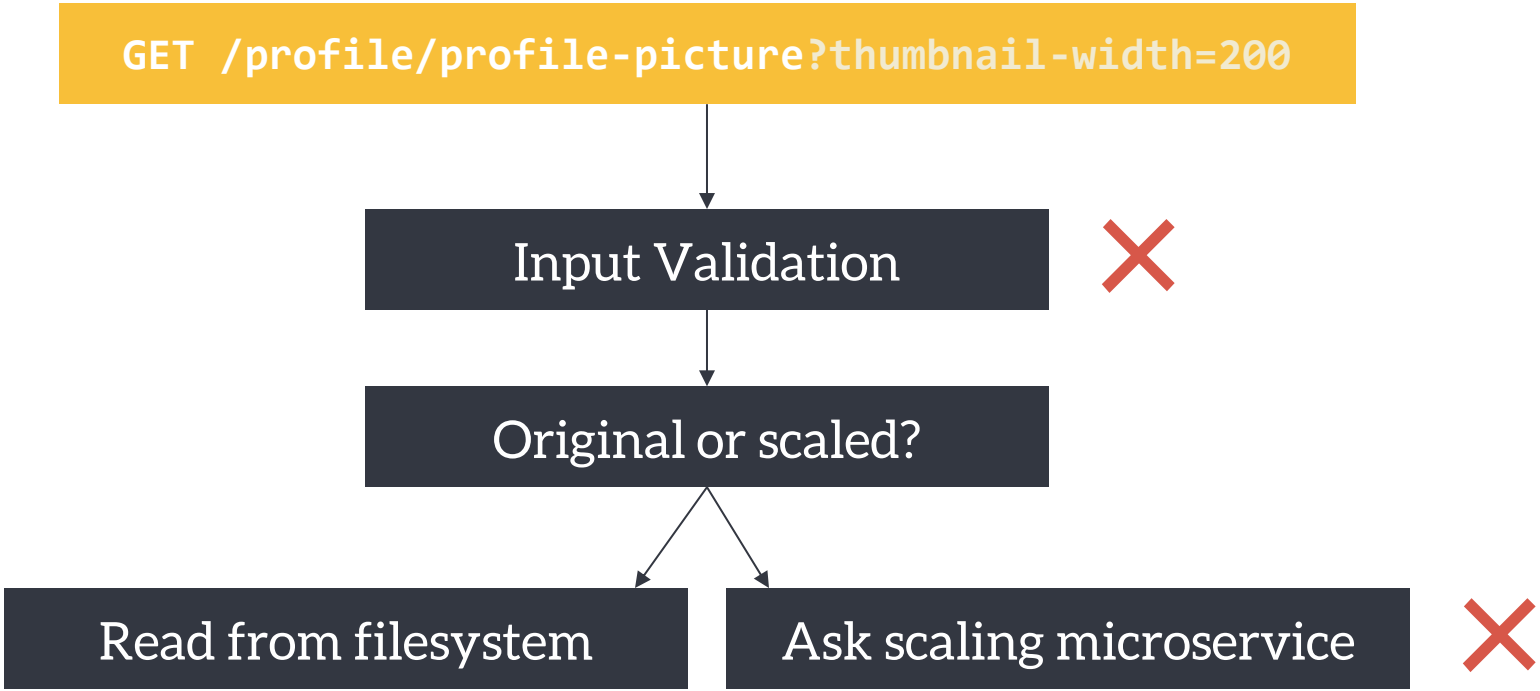
Burp Suite Professional



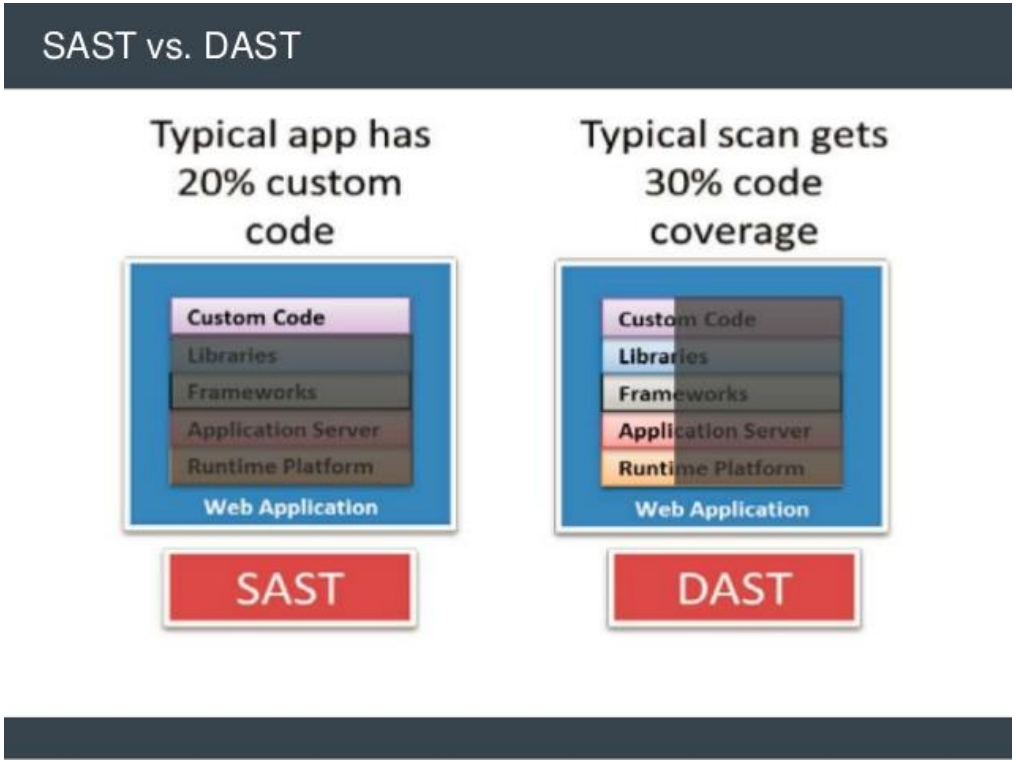
Dynamic Application Security Testing (DAST)

- **Advantages**
  - Touches more parts of your stack
  - Tends to have less false positives
- **Disadvantages**
  - Can only detect a limited set of vulnerabilities
  - SPAs require heavy lifting (headless browser)
  - Hard to get good code coverage

Dynamic Tests: Known-Good Requests



SAST vs. DAST


SAST vs. DAST
Typical app has 20% custom code
Custom Code
Libraries
Frameworks
Application Server
Runtime Platform
Web Application
SAST
Typical scan gets 30% code coverage
Custom Code
Libraries
Frameworks
Application Server
Runtime Platform
Web Application
DAST

DAST: Common Pitfalls

Ignoring code coverage.

- Make sure your tool can deal with authentication, different roles, multi-step forms, state-heavy functionality.
- Find the right method to get Known Good Requests.
- Don’t blindly trust, but measure.
- Keep it up with application changes.

DAST: Common Pitfalls

Overestimating horizontal coverage of DAST.

- With some vulnerability classes, DAST can have bad horizontal coverage.
- A common example is Stored XSS.
- Keep that in mind when creating your AST strategy.

DAST: Common Pitfalls

Just fixing that one occurrence.

- DAST rarely has 100 % horizontal code coverage.
- Treat results as symptoms, not as the disease.
- Invest time in finding all occurrences in your code.

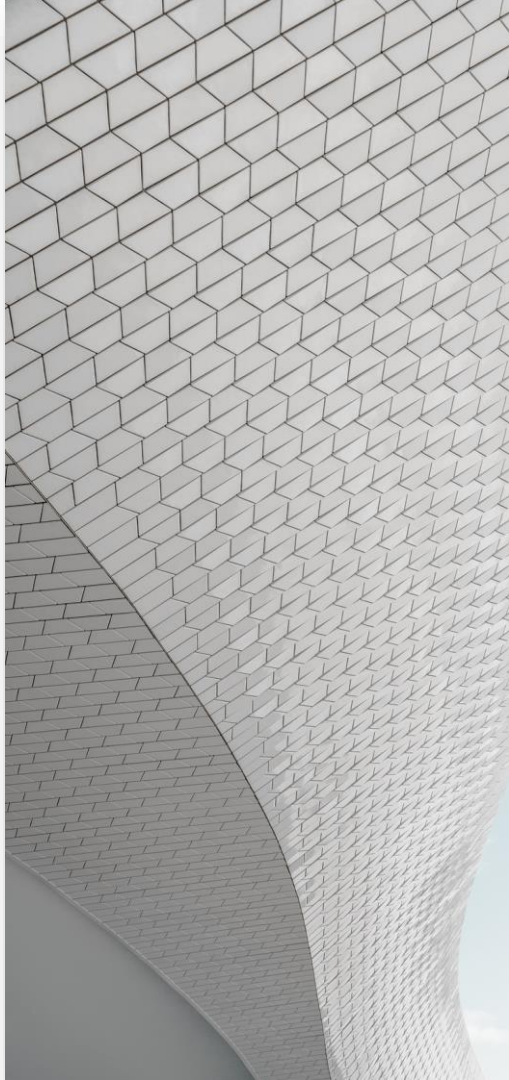
What Are The Demands?

- This is often unclear
- **Commonly mentioned vulnerability lists**
  - OWASP Top 10
  - OWASP API Security Top 10
  - CWE Top 25

Vulnerability Dimensions

It's not one-dimensional!

1. Error-proneness
2. Technology Support
3. Automated Testability
4. Verifiability
5. Stack Distribution
6. Availability of Defense in Depth
7. Specificity
8. Place of Best Mitigation
9. Place of Best Assurance



Automated Testability of Vulnerabilities

Vulnerability Class	Testable via automated means (green = reliably testable)	Rough severity rating (green = low severity)	Susceptability to *False Positives* (green = hardly susceptible)	Prevalence (green = rare)
SQL Injection				
Command Injection				
Password Security				
Brute-Force				
Session ID Randomness				
Account Recovery Security				
Weak TLS Configuration				
XML External Entities (XXE)				
Missing Function-Level Access Control				
Missing Object-Level Access Control				

Automated Testability of Vulnerabilities

Vulnerability Class	Testable via automated means (green = reliably testable)	Rough severity rating (green = low severity)	Susceptability to *False Positives* (green = hardly susceptible)	Prevalence (green = rare)
Reflected XSS				
Stored XSS				
DOM-Based XSS				
Insecure Deserialization in Java				
JavaScript Libraries with Known Vulnerabilities				
Backend Components with Known Vulnerabilities				
Cross-Site Request Forgery (CSRF)				
Security Headers and Cookie Flags				

General Tool Weaknesses

Design flaws cannot be detected

- Tools might know that “injection is bad” but not that “this user must not see this dataset”
- How would a tool know what functionality a role can call?

A More Sustainable Approach

Embrace the fact that vulnerabilities aren't single-dimensional!

1. Determine the impact of security incidents (BIA).
2. Document your tech stack.
3. Determine relevant vulnerability types for each component.

Steps Towards More Targeted Protection

For each vulnerability class, document

1. how, if at all, technology helps,
2. how, if at all, it could still go wrong,
3. which is the best place for mitigation,
4. which is the best place for quality assurance,
5. whether defense in depth is available and in use,
6. what residual risk there is,
7. and how the vulnerability class relates to docs, standards, guidelines, and requirements (internal or external).

Steps Towards More Targeted Protection

Use tools to

1. Fixate known good behavior
2. Prevent regressions
3. Identify all instances of a vulnerability

Example: Angular and XSS

- SPAs introduce good separation of concerns
- Only a small set of vulnerabilities is relevant
- **In short: It's mostly XSS**
  - via our own code, and
  - via dependencies

Mitigation Tracking: Angular and XSS

Vulnerability name	Cross-Site Scripting (XSS)
Threat type (C/I/A/N)	C/I/-/N
Qualitative severity	Medium to high
How does technology help?	Angular does automatic HTML encoding by default and comes with a sanitizer for `[href]` and `[innerHTML]` [1].
What are edge cases?	`bypassSecurityTrust*` [2] and direct DOM usage.
Automated checks	- We use SAST to disallow `bypassSecurityTrust*` [3] - We use a Linter to disallow DOM XSS sinks [4]
Defense-in-depth measures	We use a strong Content Security Policy [5].
Residual risk	Developers use untested insecure DOM APIs [6] directly.
Links and references	*Links above, internal policies and guidelines, requirements documents, ...*

Advantages of This Approach

- You only mitigate what's relevant
- It serves as compact coding guidelines
- Countermeasures are traceable

Automation: Common Pitfalls

Overestimating what tools can do.

- Have a clear picture of what the tools are capable of.
- Only the baseline is scalable.
- Tools cannot take away the requirement of deep understanding of your software and environment.

Automation: Common Pitfalls

Expecting the possibility to "outsource security".

- It is tempting to buy an expensive tool and then blame it on the tool if it does not find a certain problem.
- Don't do that.
- Focus on tools that do little but do it well.

Automation: Common Pitfalls

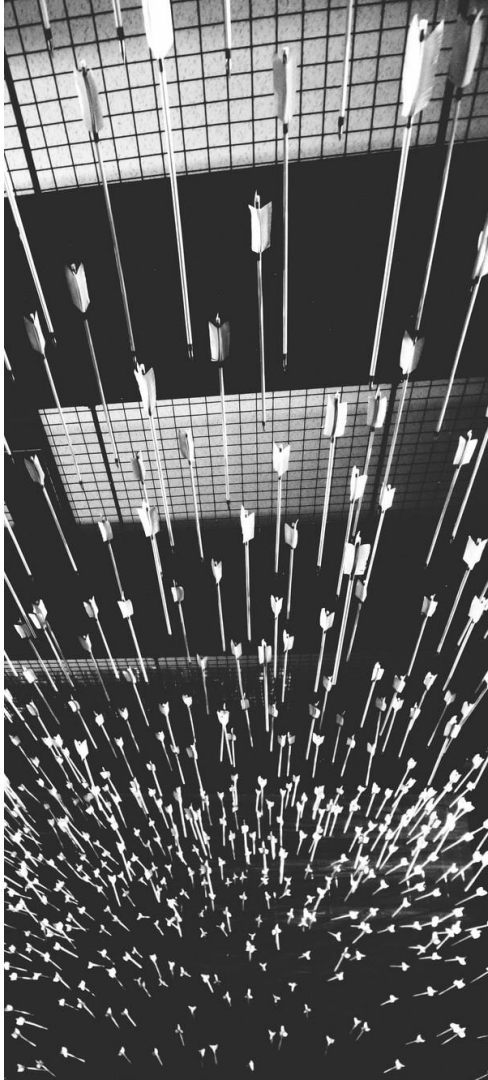
Not feeding back the findings to other security activities.

- Don’t let the same vulnerabilities fool you repeatedly.
- Often, they are symptoms for an underlying, bigger problem.
- Feed the results back into requirements, architecture changes, design documents, technology decisions.
- Share what you have learned.

Penetration Test

A penetration test is an attack simulation.

- Active system analysis
- Perspective of a potential adversary
- Vulnerabilities often exploited for demonstration purposes



Penetration Test

A penetration test is

- A method of quality assurance
- Simulating the paths of least resistance

A penetration test is not

- A test with 100 % code coverage
  - “Absence of evidence” ≠ “evidence of absence”
- A means to “test your system secure”

Penetration Tests: Common Pitfalls

Not giving the testers any information because “an attacker would not have it neither”.

- Right, but the attacker has way more time.
- We’re here to find as many issues as possible.
- Share what you know.

Penetration Tests: Common Pitfalls

Not taking enough time to make the testers understand the system *functionally*.

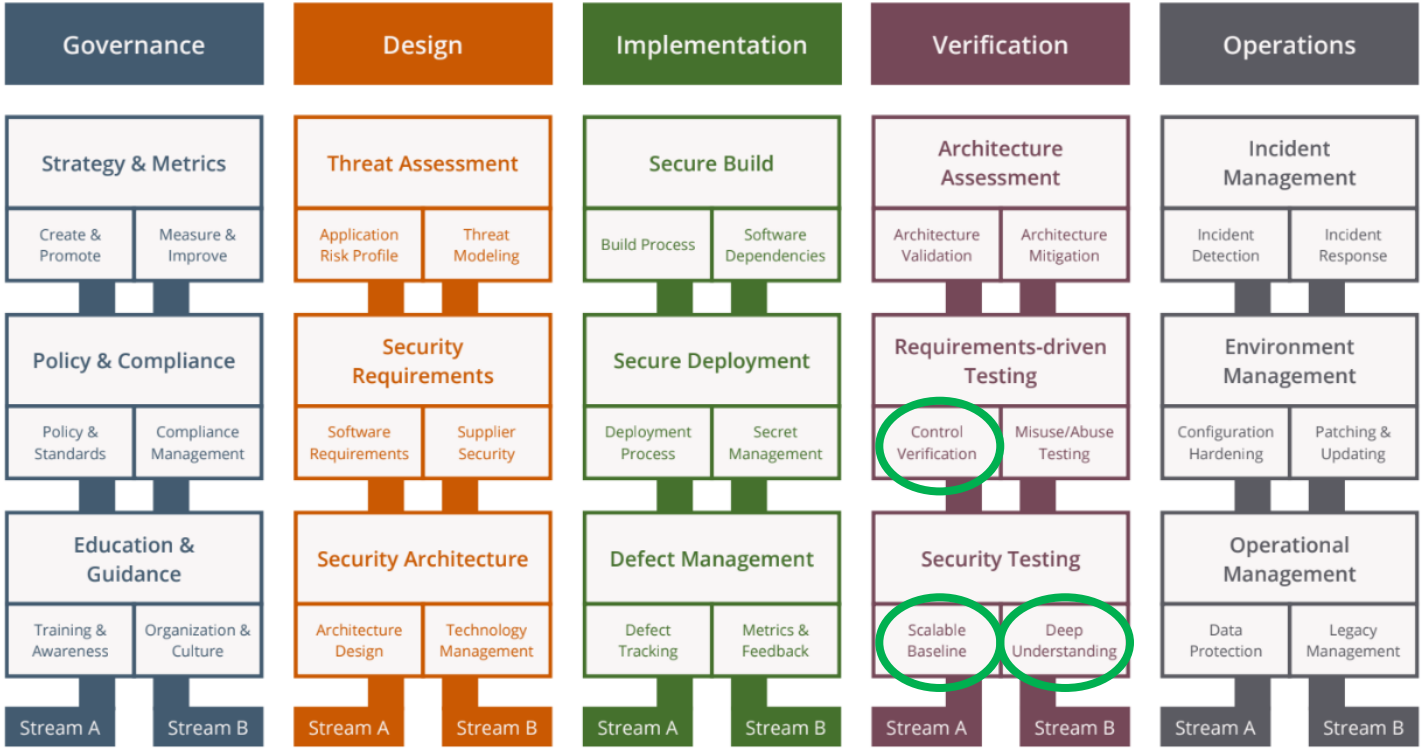
- No functional knowledge leads to wrong prioritization of testing efforts.
- What would an attacker be looking for?
- It also leads to worse coverage.

Penetration Tests: Common Pitfalls

Taking only the severity levels of the findings for prioritization.

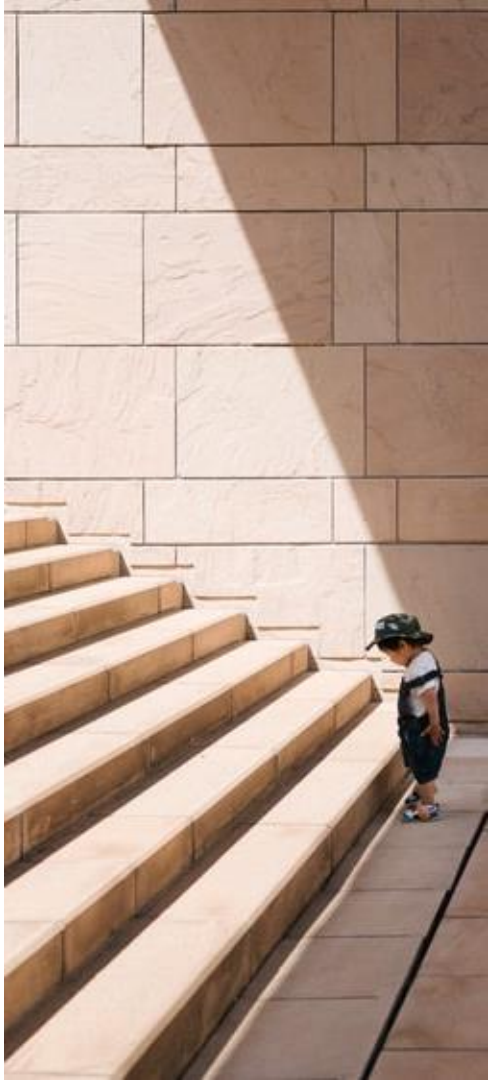
- It is a severity level, not a business risk.
- Translate the severity into what it means to your organization.
- Don’t be afraid to accept risk but do it formally and involve stakeholders.

Where Are We In OWASP SAMM?



Essentials: Software Security Verification

- ❑ Know and document the relevant vulnerability classes for your software
- ❑ Know the strengths and weaknesses of automated tools
- ❑ Focus on relevance, speed, and integration
- ❑ Use regular penetration testing for checks and balances



Quiz!

Software Security Verification

Visit https://kahoot.it and enter the game pin I'll share with you!

Operations and Maintenance

Security in the running environment

Visibility

You cannot react on what you do not see.

➔ Logging

➔ Log collection

➔ Events

➔ Monitoring

➔ Alerts

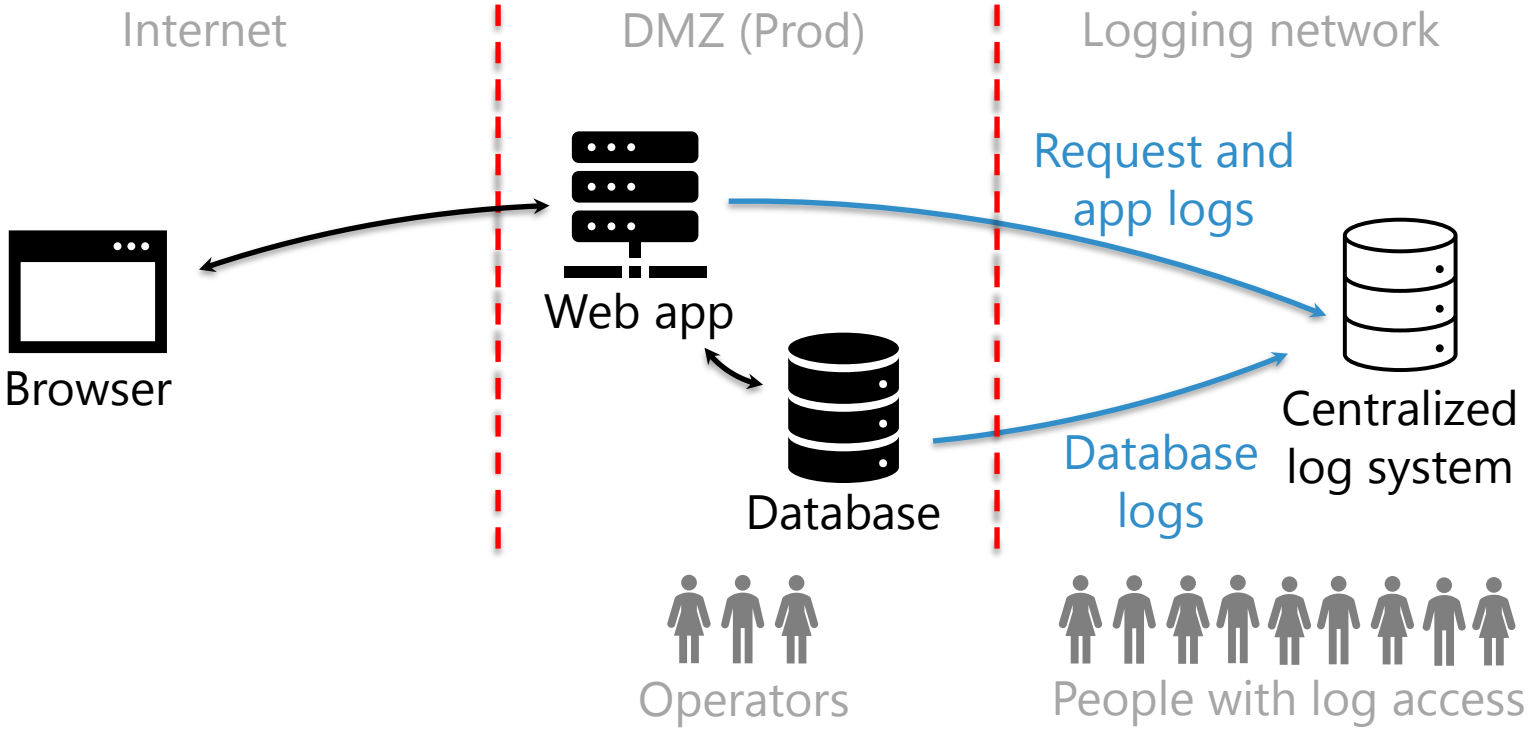
➔ Incidents

➔ Incident response

Security-Relevant Events

- Validation failures
- Authentication events
- Authorization failures
- Session management failures
- Exceptions
- System start-ups and shut-downs
- Use of high-risk functionality
- Legal opt-ins

Logs Outside the Trust Boundary



What *Not* To Log?

Logs might end up outside the trust boundary of your system. Don't log

- PII
- Request bodies
- Session and security tokens
- API keys, passwords, connection strings, and other credentials
- Encryption keys and other secrets
- Data above the criticality level the logging system is designed for

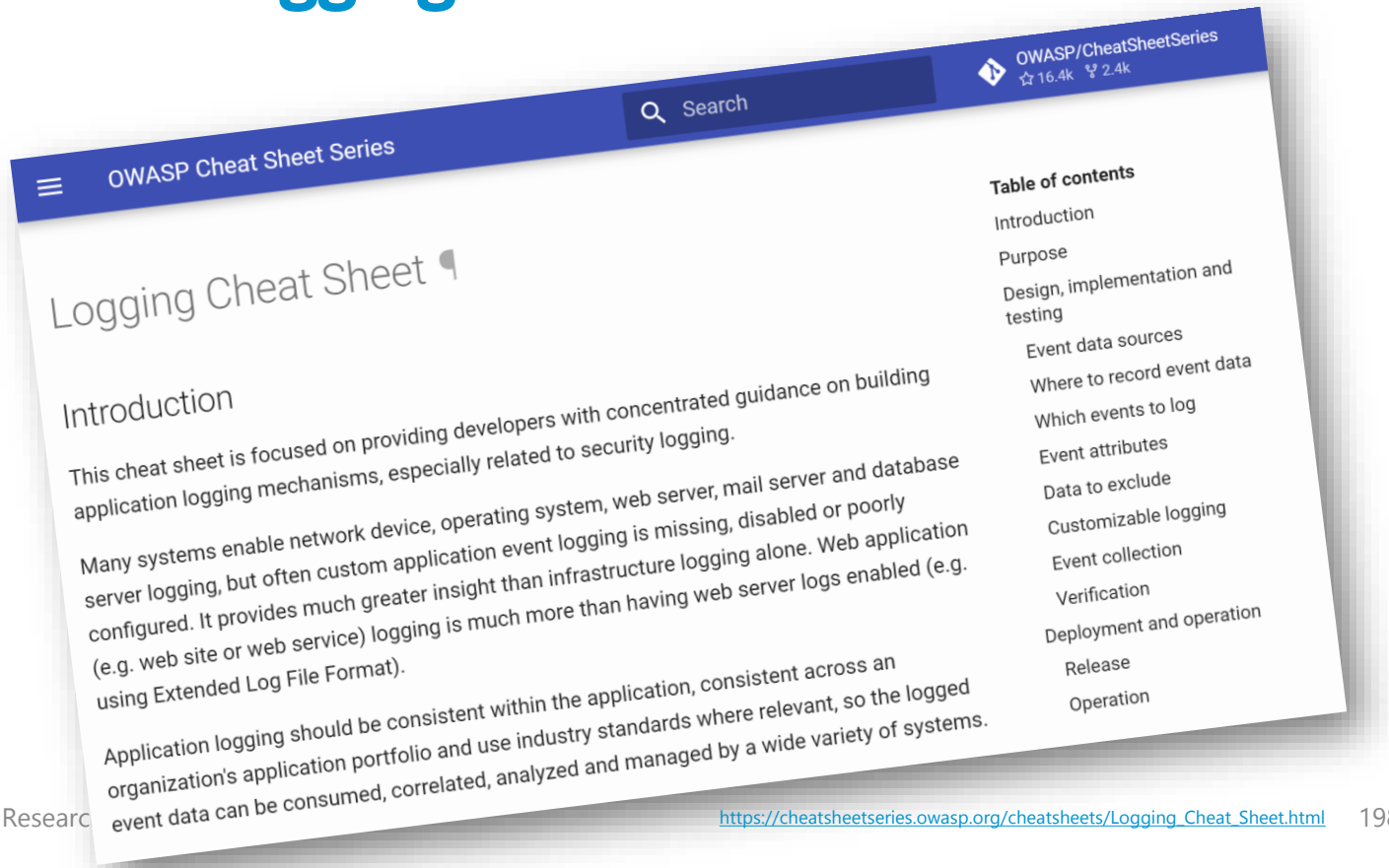
How To Log?

1. **When**: `2021-05-10 16:52:01 UTC`
2. **Where**: Web shop instance on `10.0.1.25`, HTTPS via port `443`, `/login` URL, `POST` method
3. **Who**: Public IP address `1.2.3.4`, user id `1389751`
4. **What**: Level `INFO`, event ID `auth.login_successful`, “`User #1389751 logged in.`”

Logging: General Rules

1. Log in a structural format (e.g., JSON)
2. Standardize the timestamp format
3. Identify the origin via mandatory fields
4. Allow applications to add their own arbitrary data
5. Synchronize time over systems
6. Use request IDs

OWASP Logging Cheat Sheet



Logging: Common Pitfalls

Not standardizing and synchronizing time.

- Correlating logs can be a pain if timestamps differ
- The exact time is especially important when lots of logs are produced on different systems
- Strive for an organization-wide standard

Logging: Common Pitfalls

Taking the wrong source IP address.

- The source IP address of the TCP packet might be your reverse proxy
- Look out for the `X-Forwarded-For` header
- Make sure to take the right address

Logging: Common Pitfalls

Logging too much.

- Analysis is harder under a pile of meaningless logs
- Commercial log systems tend to get expensive on high log load
- Focus on what might be helpful for log analysis

Logging: Common Pitfalls

Not seeing logs as production data.

- Some sort of sensitive data will end up there.
- Centralized logging systems are rarely within the trust boundary.
- Account for that.

Monitoring

If an attack happens, is anyone going to notice it?

- On average, compromises stay unnoticed for ~200 days on average [IBM]
- Logs are the base
- Monitoring and alerting brings logs to life

Monitoring

When should I alert?

- Security-related events that should rarely happen on normal usage
  - Access denied errors
  - Rare states (log them as `WARN`)
  - Unexpected exceptions

Monitoring

When should I alert?

- Security-related events that should not happen in huge amounts
  - Failed log-in attempts
  - “Not found” errors
  - Resource exhaustion

Monitoring: Common Pitfalls

Not alerting at all.

- Logs are good, but they’re just the base.
- Ask yourself: Would we detect an attack?
- Start with few, important events.

Monitoring: Common Pitfalls

Over-alerting.

- Alerting too often will make people ignore the alerts.
- Start simple.
- Aim at detecting the obvious first.

Incident Management

1. Preparation
2. Identification
3. Containment
4. Eradication
5. Recovery
6. Lessons Learned

Incident Management: Essentials

Here are some essential steps.

1. Responsibilities must be clear.
2. Guidance on what to do in the case of an incident must be accessible.
3. Regular trainings and drills will increase the efficiency.

Configuration Hardening

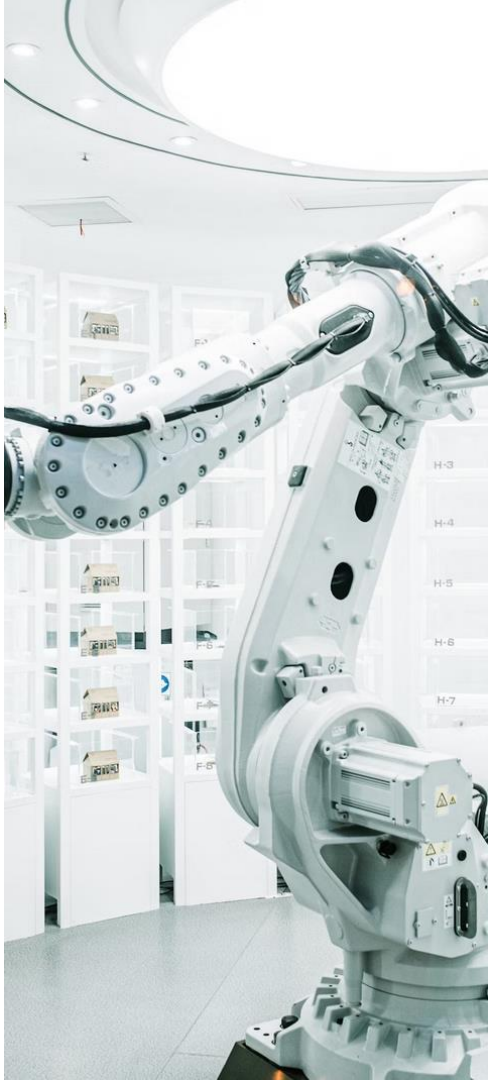
- Perform configuration hardening of your key components
- Use security best practices from vendors
- Follow established baselines
  - E.g., various CIS Benchmarks (Apache, nginx, Debian, Red Hat, Kubernetes, AWS, Azure, ...)

Patching & Updating

- Introduce patch process
  - Regular (at least once a month)
- Risk-based approach
  - Patch critical systems first
  - Patch critical components first
  - Patch critical systems more often
- Know your software components (SBOM, CBOM, HBOM, …)
- Track known vulnerabilities (threat intelligence)

Automate Your Patching

- Unattended OS updates
- Use automation tools e.g., Puppet or Ansible
- Use Infrastructure as Code (IaC)
- Use multiple environments
  - E.g., Test, Staging, Production
- Use automated regression testing



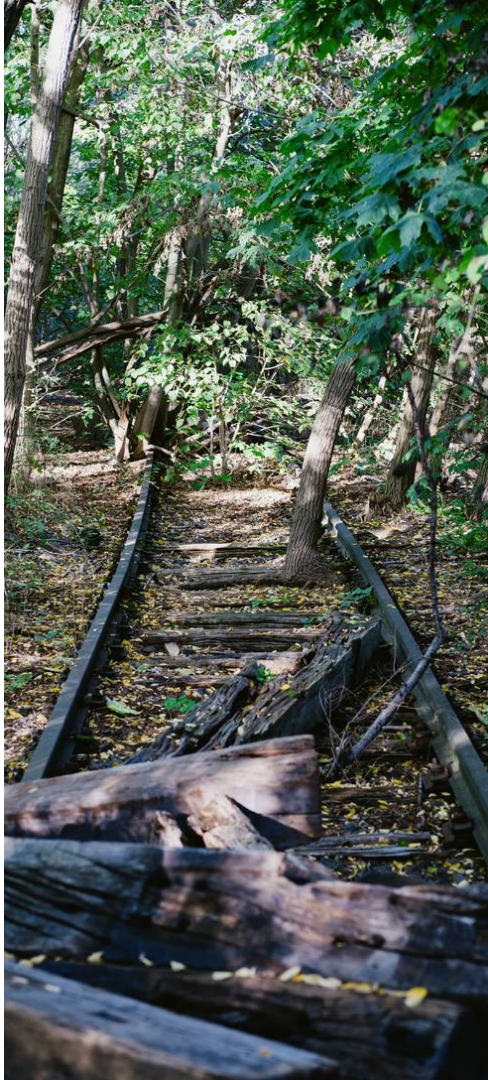
Courses of Action

After analyzing the problem, you can pursue multiple roads:

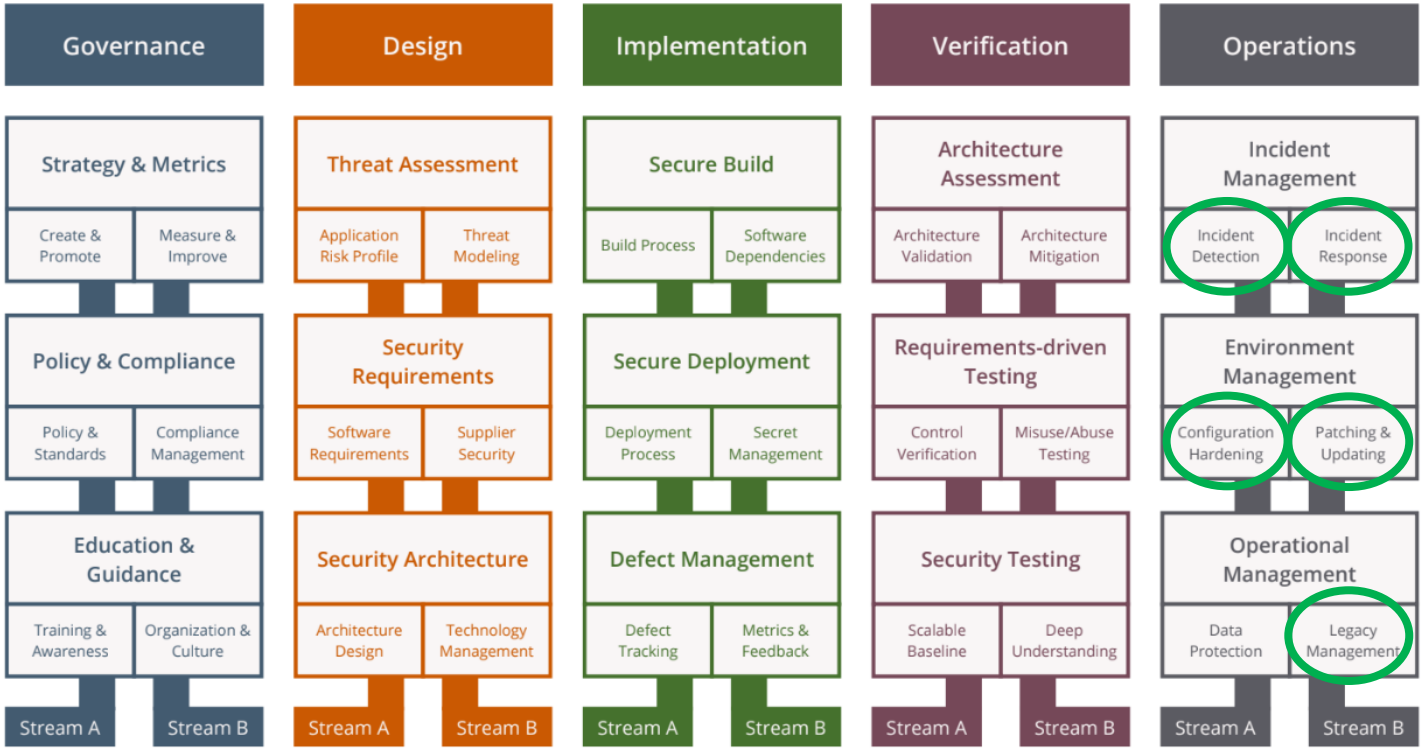
- Update component or install vendor patch
- Remove or replace component
- (Patch vulnerabilities yourself)
- Implement compensating controls
- Accept risk

Decommission & Legacy Management

- Identify unused components
- Track end-of-life (EOL) dates of your major components
- Plan your replacements and major upgrades in advance
- Migrate or disable legacy systems

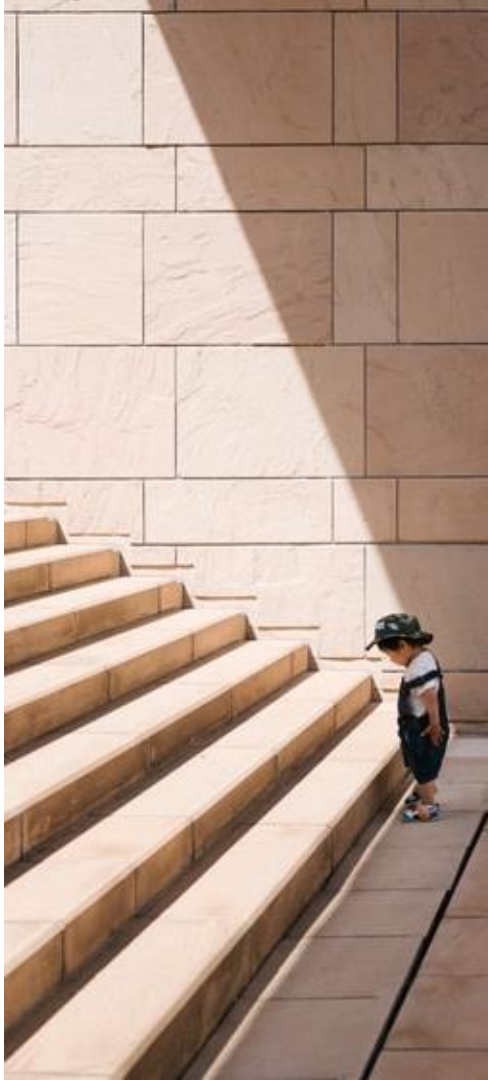


Where Are We In OWASP SAMM?



Essentials: Operations and Maintenance

- ❑ Agree on how and what to log; Centralize logging
- ❑ Normalize time and timestamps
- ❑ Treat log data as production data
- ❑ Alert on essential events, but don’t over-alert
- ❑ Establish responsibilities, guidance, and training for incident response
- ❑ Introduce regular patching & updating
- ❑ Automate patching process
- ❑ Keep an eye on your legacy systems



Quiz!

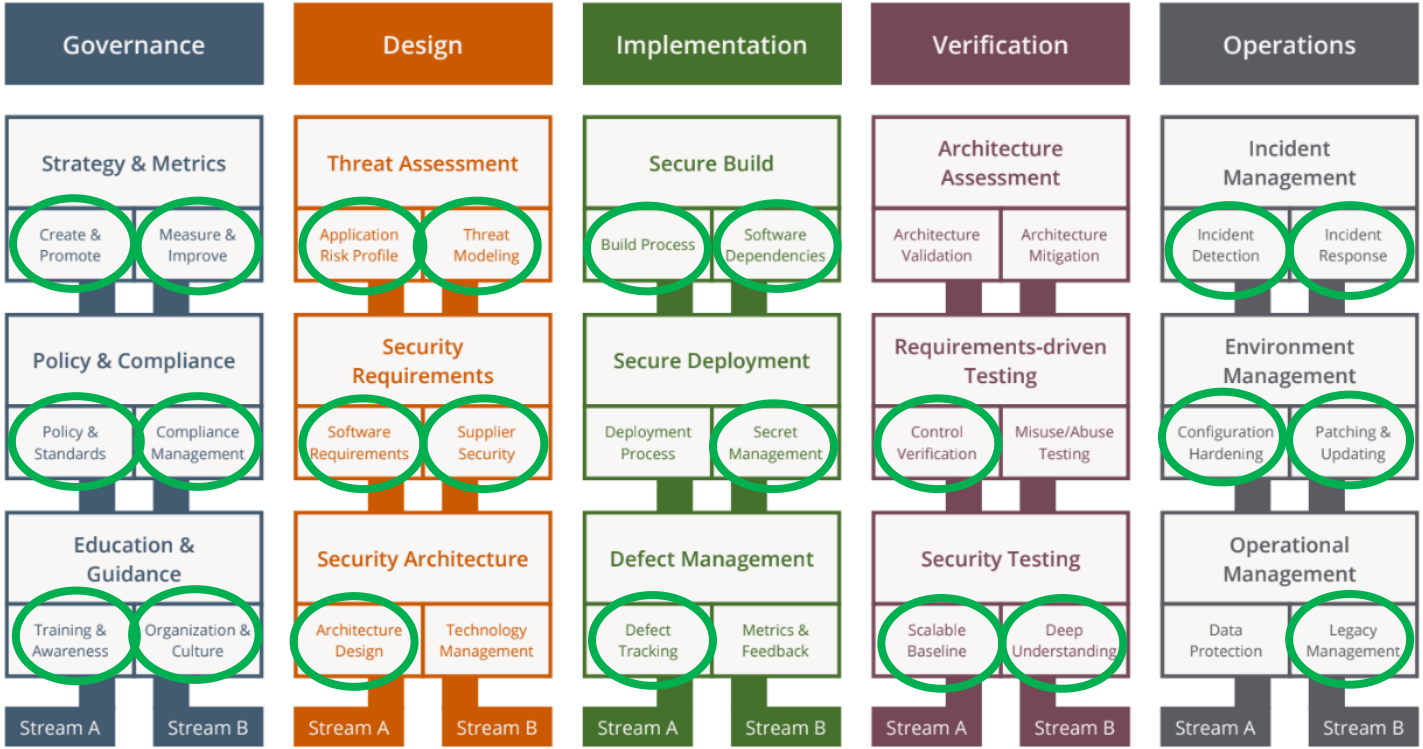
Operations and Maintenance

Visit https://kahoot.it and enter the game pin I'll share with you!

Wrapping Up

Areas covered in OWASP SAMM, key take-aways

What Did We Cover?

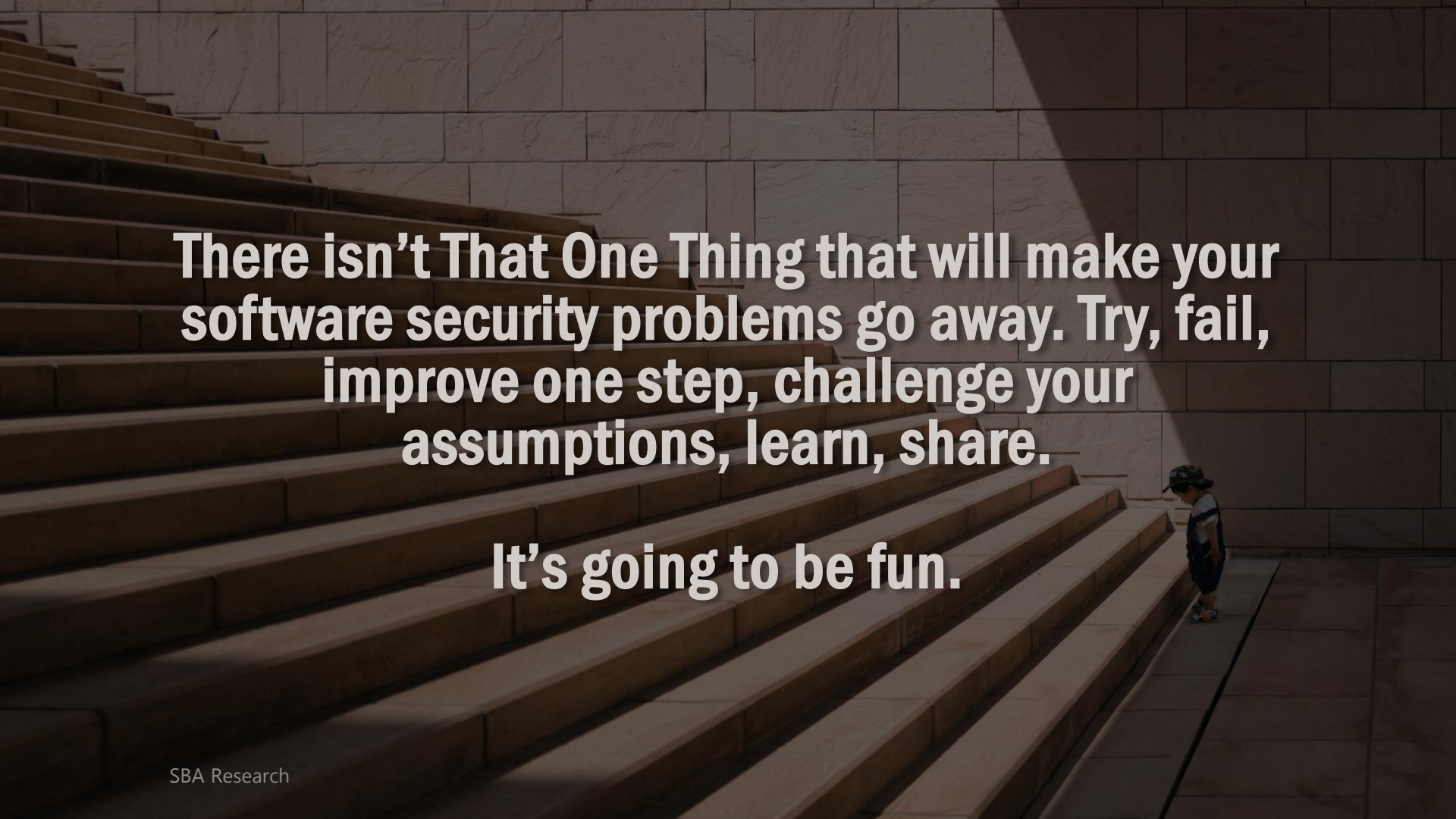


Key Take-Aways

1. Education has a good ROI.
2. Well-selected and well-tracked security requirements are a key to success.
3. Distinguishing bugs from flaws is key to understanding holistic software security.
4. Establishing trust boundaries is a necessity for many other security activities.
5. Tracking security issues outside the “normal” issues fosters a culture of treating security as “something separate”.

Key Take-Aways

6. Knowing what tools can and cannot do for your stack will save you from mis-spending time and money.
7. A penetration test is a great assurance tool at the *end* of your test chain.
8. Logs must be treated as production data, unless one can prove otherwise.
9. An effective incident response process consists *at least* of responsibilities, guidance, and trainings.
10. There isn't That One Thing that will solve all your software security problems.


There isn’t That One Thing that will make your software security problems go away. Try, fail, improve one step, challenge your assumptions, learn, share.
It’s going to be fun.

Andreas Boll

SBA Research

Floragasse 7, 1040 Vienna

aboll@sba-research.org

Federal Ministry Innovation, Mobility and Infrastructure Republic of Austria

Federal Ministry Economy, Energy and Tourism Republic of Austria

FFG Promoting Innovation.

vienna business agency | For the City of Vienna

European Commission

FWF Austrian Science Fund

netidee FÖRDERUNGEN