Secure Development Lifecycle

Secure Development Lifecycle

❌ Identifying errors in late lifecycle phases makes them more expensive to fix or mitigate. [^1]

❌ Having an unpublished or informal Secure Development Lifecycle will not be successful. [^2]

✓ Security must be embedded into all stages of the Software Development Lifecycle to be effective. [^2]

✓ A close connection with the right expert and management drive from the beginning are both mandatory. [^3]

Spaghetti Analogy

Sprinkling security on insecurely written software is equivalent to sprinkling salt on spaghetti 🍝 after cooking them in unsalted water 🚰.

Estimates of Relative Cost Factors of Correcting Errors

Introduction of Error	Requirements / Design	Coding / Unit Test	Integration / System Test	Early Access / Beta Test	Post-Release
Requirements / Design	x1	x5	x10	x15	x30
Coding / Unit Test		x1	x10	x20	x30
Integration / System Test			x1	x10	x20

NIST: The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002

Example: Microsoft SDL

Phase	Practice
Training	Core Security Training
Requirements	Establish Security Requirements, Create Quality Gates/Bug Bars, Perform Security and Privacy Risk Assessments
Design	Establish Design Requirements, Perform Attack Surface Analysis/Reduction, Use Threat Modelling
Implementation	Use Approved Tools, Deprecate Unsafe Functions, Perform Static Analysis

Phase	Practice
Verification	Perform Dynamic Analysis, Perform Fuzz Testing, Conduct Attack Surface Review
Release	Create an Incident Response Plan, Conduct Final Security Review, Certify Release and Archive
Response	Execute Incident Response Plan

Security Requirements

Derive Security Requirements from Business Functionality

- Gather and review functional requirements
- For each functional requirement derive relevant security requirements
  - Lead stakeholders through explicitly noting security expectations
    - e.g. data security, access control, transaction integrity, criticality of business function, separation of duties, uptime etc.
  - Follow the same principles for writing good requirements in general
    - i.e. they should be specific, measurable, and reasonable

Security and Compliance Guidance for Requirements

- Determine industry best-practices that project teams should treat as requirements
  - e.g. publicly available guidelines, internal or external guidelines/standards/policies, or established compliance requirements
- Do not attempt to bring in too many best-practice requirements into each development iteration
- Slowly add best-practices over successive development cycles

Secure Design Principles

Secure Design Principles

Minimize Attack Surface Area	Don't trust Services
Establish Secure Defaults	Separation of Duties
Principle of Least Privilege	Avoid Security by Obscurity
Principle of Defense in Depth	Keep Security simple
Fail securely	Fix Security Issues correctly

Minimize Attack Surface Area

- **Every feature** that is added to an application **adds a certain amount of risk** to the overall application
- The aim for secure development is to **reduce** the overall risk by reducing **the attack surface area**

Establish Secure Defaults

- The "**out-of-box**" experience for the user **should be secure**
- It should be up to the user to reduce their security - if they are allowed

Principle of Least Privilege

- Accounts have the **least amount of privilege required** to perform their business processes
- This encompasses user rights and resource permissions, e.g.
  - CPU limits
  - memory
  - network
  - file system

Principle of Defense in Depth

- Where one control would be reasonable, **more controls that approach risks in different fashions** are better
- In-depth-controls can make severe vulnerabilities extraordinarily difficult to exploit

Fail securely

- Whenever a transaction fails or code execution throws an exception it should **always "fail closed"** and never "fail open"

Don't trust Services

- Third party partners more than likely have differing security policies and posture
- Implicit **trust of externally run systems is not warranted**
- All external systems should be treated in a similar fashion

Separation of Duties

- Separation of duties is a key **fraud control**
- **Administrators should not also be users** of an application they are responsible for

Avoid Security by Obscurity

- Security through obscurity is a weak security control, and nearly always fails when it is the only control
- The **security** of key systems **should not be reliant upon keeping details hidden**

Keep Security simple

- **Attack surface and simplicity go hand in hand**
- Prefer straightforward and simple code over complex and over-engineered approaches
- Avoid the use of double negatives and complex architectures when a simpler approach would be faster and simpler

Fix Security Issues correctly

- Once a security issue has been identified, it is important to develop a test for it, and to **understand the root cause** of the issue
- It is likely that the security issue is widespread amongst all code bases, so **developing the right fix without introducing regressions** is essential

Data Factors

A04:2021 – Insecure Design

CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Avg Weighted Exploit	Avg Weighted Impact	Max Coverage	Avg Coverage	Total Occurrences	Total CVEs
40	24.19%	3.00%	6.46	6.78	77.25%	42.51%	262,407	2,691

Secure Coding Guidelines

❌ Secure Coding Guidelines

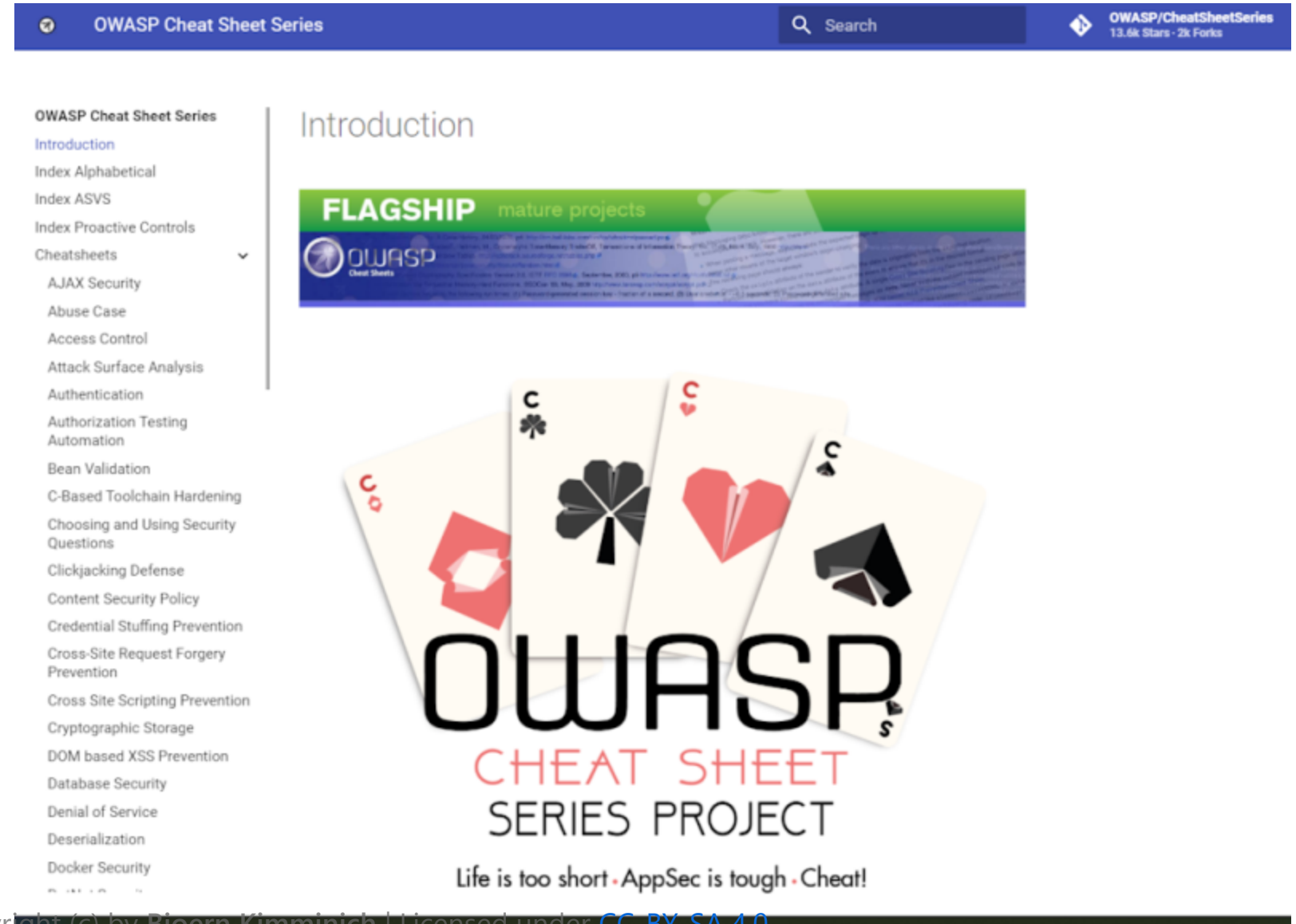
> - Overlooked by developers
> - "Static and not helpful"
> - 100+ pages that can be language specific
> - *Can* be successful if collaborative/Wiki format and regularly updated [^4]

ℹ️ *Secure Coding Guidelines are often mandated by contracts, compliance requirements or as a certification precondition.*

✓ Secure Coding Checklist

> Checklists are good. Big secure coding books are bad. In know this, I wrote one. Didn't help. [*Jim Manico @ dotSecurity 2017*, ^5]
>
> - Simple 1-2 page document
> - All checklist items must be relevant
> - Brief document must be backed up with deeper resources and code samples [^5]

✓ OWASP Cheat Sheet Series



Security Testing

Static Application Security Testing (SAST)

> Static application security testing (SAST) is a set of technologies designed to analyze application source code, byte code and binaries for coding and design conditions that are indicative of security vulnerabilities. SAST solutions analyze an application from the "inside out" in a nonrunning state. [^6]

💪 Strengths of SAST

- Scales well - can be run on lots of software, and can be run repeatedly (as with nightly builds or continuous integration)
- Useful for things that such tools can automatically find with high confidence, such as buffer overflows, SQL Injection Flaws, and so forth
- Output is good for developers - highlights the precise source files, line numbers, and even subsections of lines that are affected

👎 Weaknesses of SAST

- Many types of security vulnerabilities are very difficult to find automatically (e.g. authentication problems, access control issues, insecure use of cryptography, etc.)
- Cover only a relatively small percentage of application security flaws
- High numbers of false positives
- Cannot find configuration issues not represented in the code
- Difficult to *prove* that an identified security issue is an actual vulnerability

Open Source SAST Tools (Examples)

	Tool	Supported Language
	SonarQube	20+ languages
	FindSecBugs plugin for SpotBugs	Java, Android, Groovy, Scala
	Bandit	Python
	Brakeman	Ruby on Rails

Dynamic Application Security Testing (DAST)

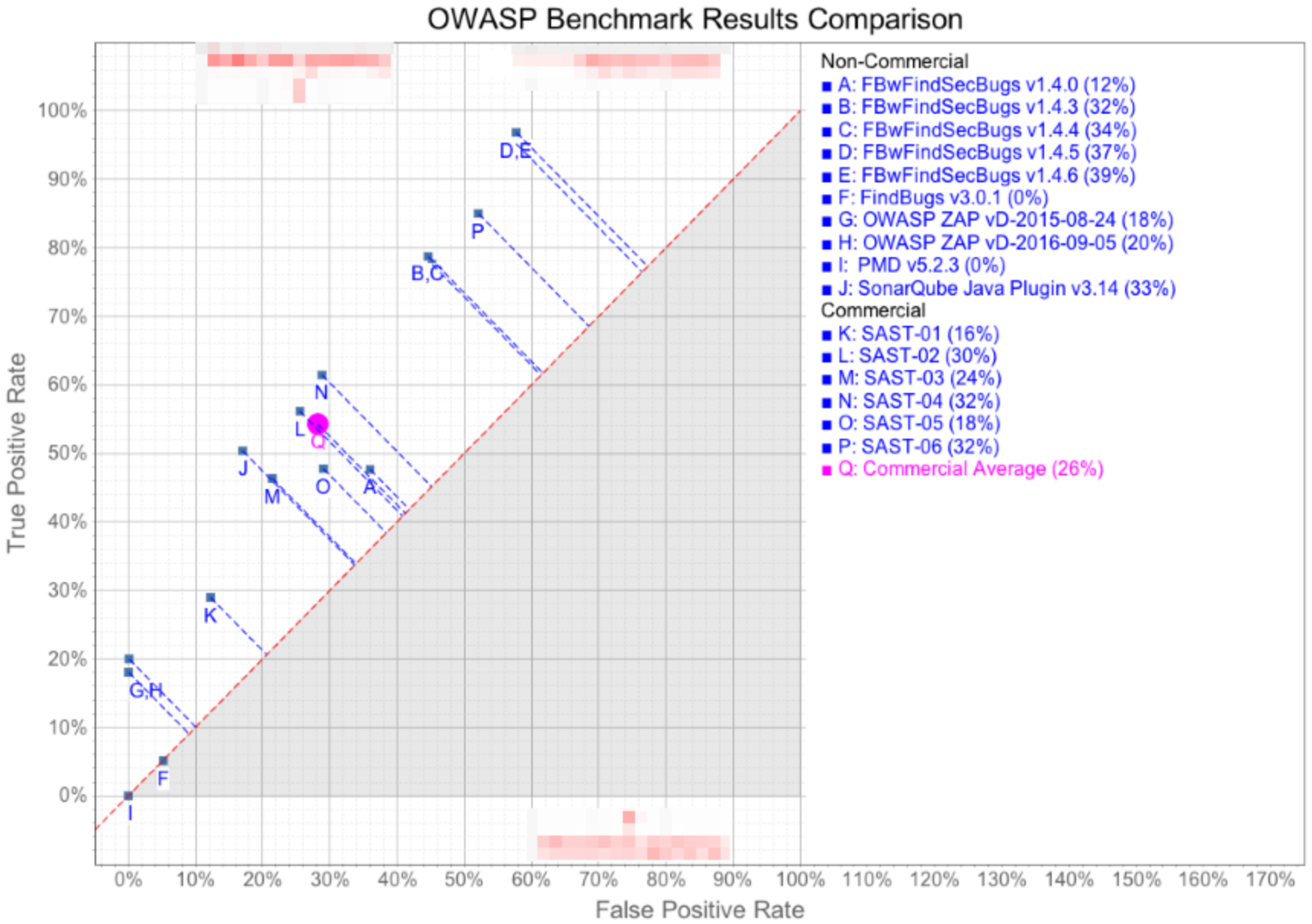
> Dynamic application security testing (DAST) technologies are designed to detect conditions indicative of a security vulnerability in an application in its running state. Most DAST solutions test only the exposed HTTP and HTML interfaces of Web-enabled applications; however, some solutions are designed specifically for non-Web protocol and data malformation (for example, remote procedure call, Session Initiation Protocol [SIP] and so on). [^7]

Open Source DAST Tools (Examples)

	Tool	Proxy	Scanner
	Zed Attack Proxy	✓	✓
	w3af	✗	✓
	Wapiti	✗	✓
	Nikto	✗	✓

OWASP Benchmark

> The OWASP Benchmark for Security Automation (OWASP Benchmark) is a free and open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services (henceforth simply referred to as 'tools'). [...] There are four possible test outcomes in the Benchmark:
>
> 1. Tool correctly identifies a real vulnerability (True Positive)
> 2. Tool fails to identify a real vulnerability (False Negative)
> 3. Tool correctly ignores a false alarm (True Negative)
> 4. Tool fails to ignore a false alarm (False Positive)



Exercise 9.1

Assign the following statements to a corresponding section of the OWASP Benchmark result diagram on the previous slide:

- *Ideal* vulnerability detection
- Tool reports *everything* is vulnerable
- Tool reports *nothing* is vulnerable
- Tool reports vulnerabilities *randomly*
- *Worse* than random

Security Logging & Monitoring

Security Logging and Monitoring Failures

- Exploitation of **insufficient logging and monitoring is the bedrock of nearly every major incident**
- **Attackers** rely on the lack of monitoring and timely response to **achieve their goals without being detected**
  - Most successful attacks start with vulnerability probing
  - Allowing such probes to continue can raise the likelihood of successful exploit to nearly 100%

Examples of Insufficiencies

- Auditable events, such as logins, failed logins, and high-value transactions are not logged
- Warnings and errors generate no, inadequate, or unclear log messages
- Logs of applications and APIs are not monitored for suspicious activity
- Logs are only stored locally
- Appropriate alerting thresholds and response escalation processes are not in place or effective
- Penetration testing and scans by automated tools do not trigger alerts
- The application is unable to detect, escalate, or alert for active attacks in real time or near real time

Data Factors

A09:2021 – Security Logging and Monitoring Failures

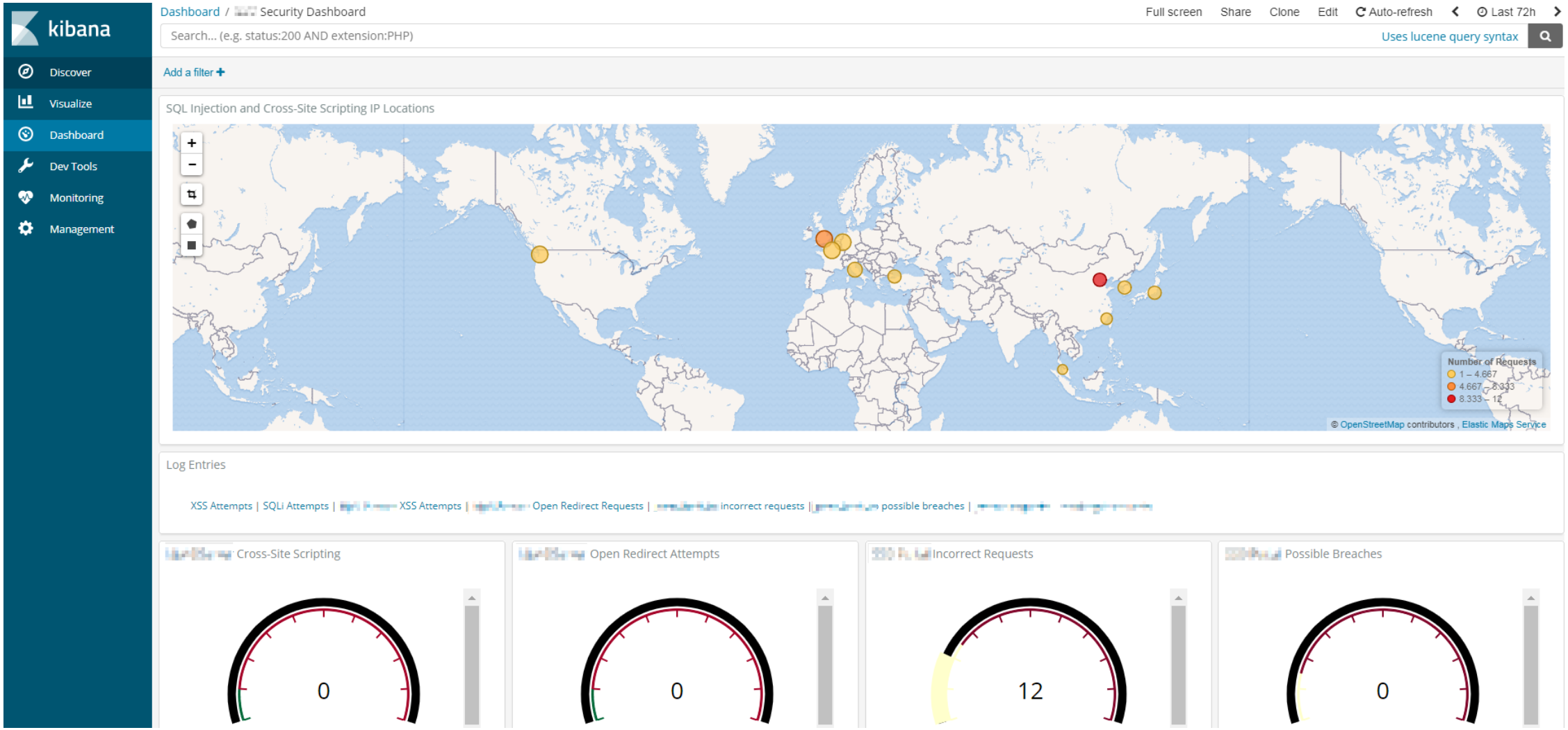
CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Avg Weighted Exploit	Avg Weighted Impact	Max Coverage	Avg Coverage	Total Occurrences	Total CVEs
4	19.23%	6.51%	6.87	4.99	53.67%	39.97%	53,615	242

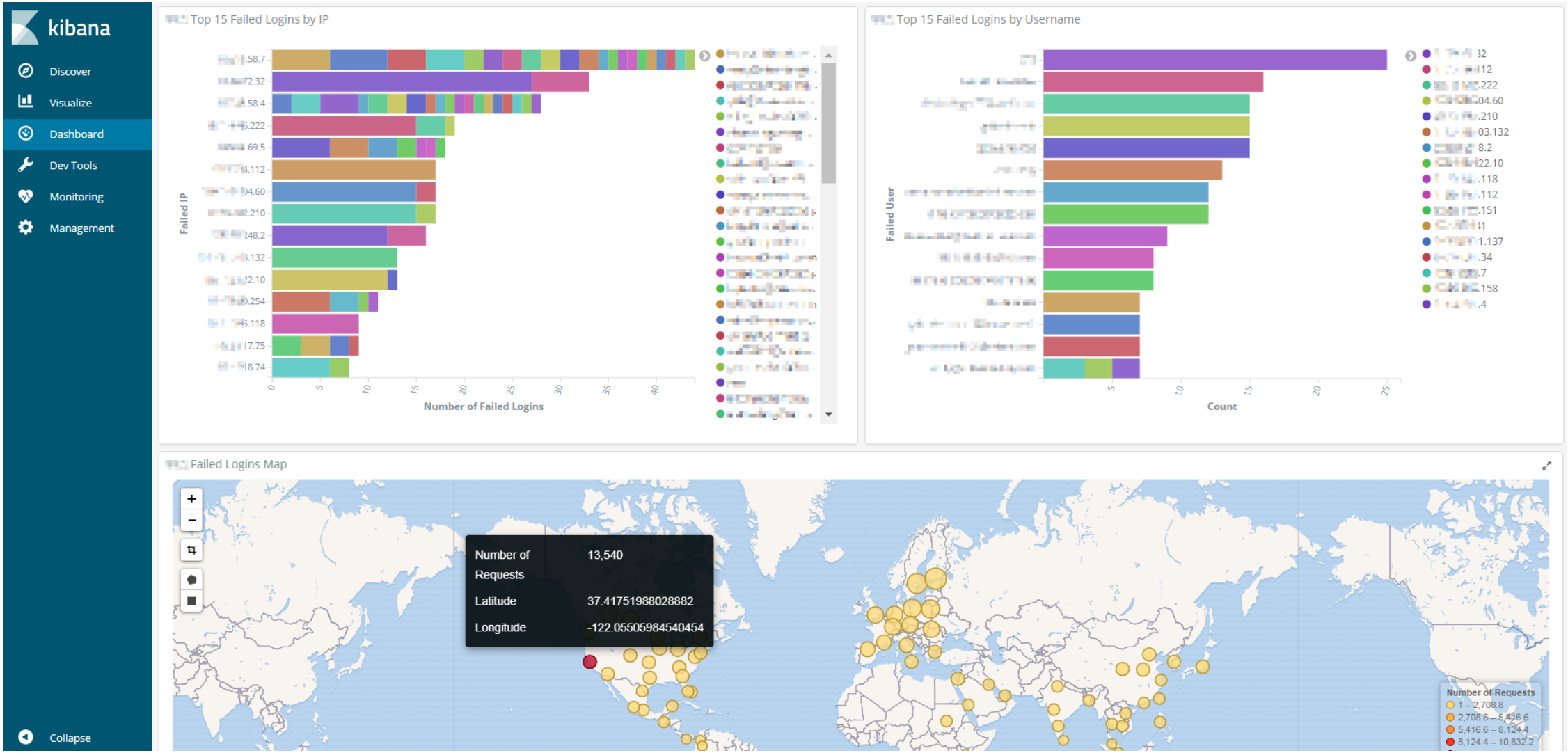
Prevention

- Ensure all login, access control failures, and server-side input validation **failures can be**
  - **logged with sufficient user context** to identify suspicious or malicious accounts
  - **held for sufficient time** to allow delayed forensic analysis
- Ensure that logs are generated in a format that can be easily consumed by a **centralized log management solution**
  - e.g. Elastic Stack (Kibana, Elasticsearch, Logstash & Beats)

- Ensure **high-value transactions have an audit trail** with integrity controls to prevent tampering or deletion
  - e.g. append-only database tables or similar
- Establish effective monitoring and alerting such that **suspicious activities are detected** and responded to **in a timely fashion**
- Establish or adopt an incident response and recovery plan

Example Kibana Security Dashboard

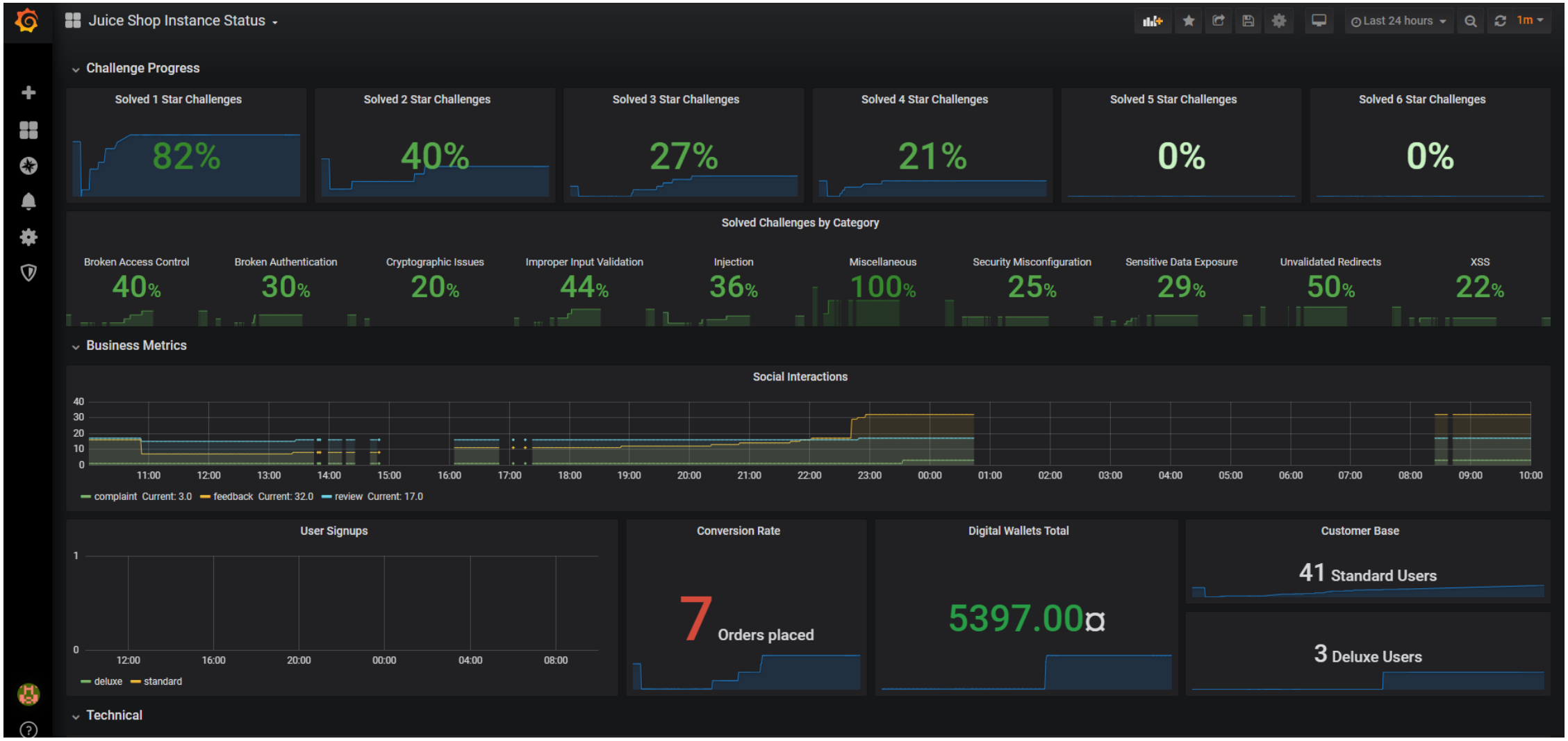




Exercise 9.2

1. Find the Prometheus monitoring endpoint offered by any running OWASP Juice Shop instance (⭐)
2. Locate and retrieve a SIEM-signature file misplaced in the `/ftp` folder of the application (⭐⭐⭐⭐)

Grafana Dashboard for OWASP Juice Shop

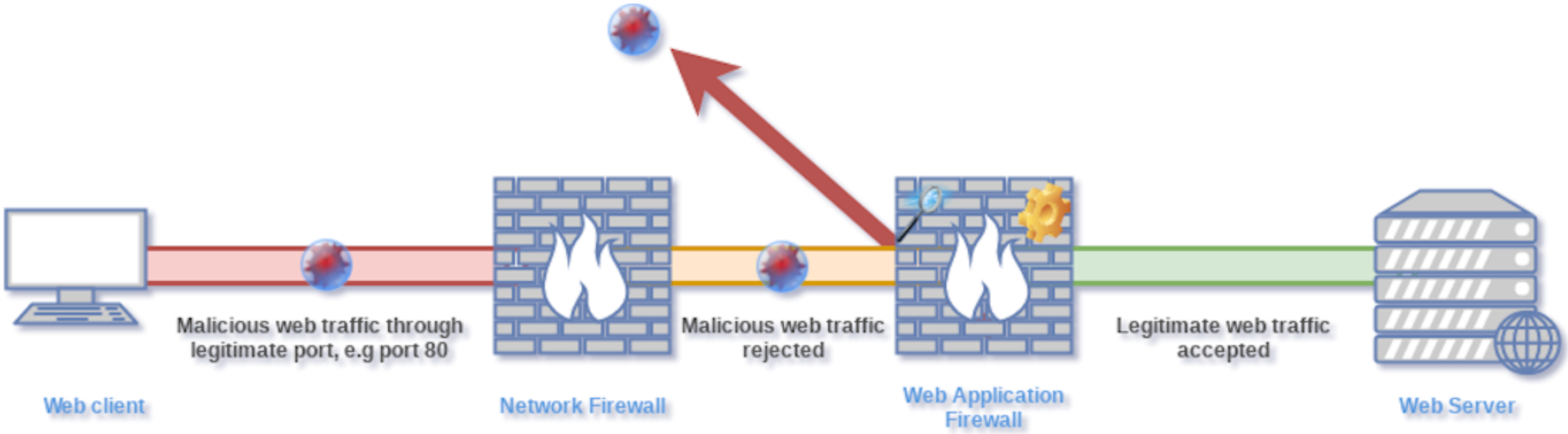


Web Application Firewall (WAF)

Web Application Firewall

> A web application firewall (WAF) is an application firewall for HTTP applications. It applies a set of rules to an HTTP conversation. Generally, these rules cover common attacks such as cross-site scripting (XSS) and SQL injection.
>
> While proxies generally protect clients, WAFs protect servers. A WAF is deployed to protect a specific web application or set of web applications. A WAF can be considered a reverse proxy. WAFs may come in the form of an appliance, server plugin, or filter, and may be customized to an application. The effort to perform this customization can be significant and needs to be maintained as the application is modified. [^8]

WAF Deployment in the Network



💡 *An application should be able to protect itself! Use a WAF only as a secondary defense mechanism to achieve Defense in Depth! For legacy systems (with no feasible way to patch directly) a WAF can be the main protection mechanism.*

Risk in the use of WAFs

- "Yet-another-proxy" (increased complexity of the IT infrastructure)
- Organisational tasks
- Training the WAF on each new release of the web application
- Testing
- False positives (which may have a significant business impact)
- More complex troubleshooting
- WAFs also have/generate errors
- Responsibility for system-wide error situations
- Cost-effectiveness

WAF Modes

- **Blocking Mode**: Normal operational mode where the WAF blocks requests it identified as malicious.
- **Monitoring Mode**: The WAF logs alerts but does not block the corresponding requests.
- **Learning Mode**: The WAF learns from good traffic (e.g. by allowlisted IPs) what the normal use cases and input are.

💡 *Learning Mode might lead to false positives on new application releases when the WAF did not learn any traffic for new functionality.*

OWASP ModSecurity Core Rule Set

> The OWASP ModSecurity Core Rule Set (CRS) is a set of generic attack detection rules for use with ModSecurity or compatible web application firewalls. The CRS aims to protect web applications from a wide range of attacks, including the OWASP Top Ten, with a minimum of false alerts.

Exercise 9.3 (🏠)

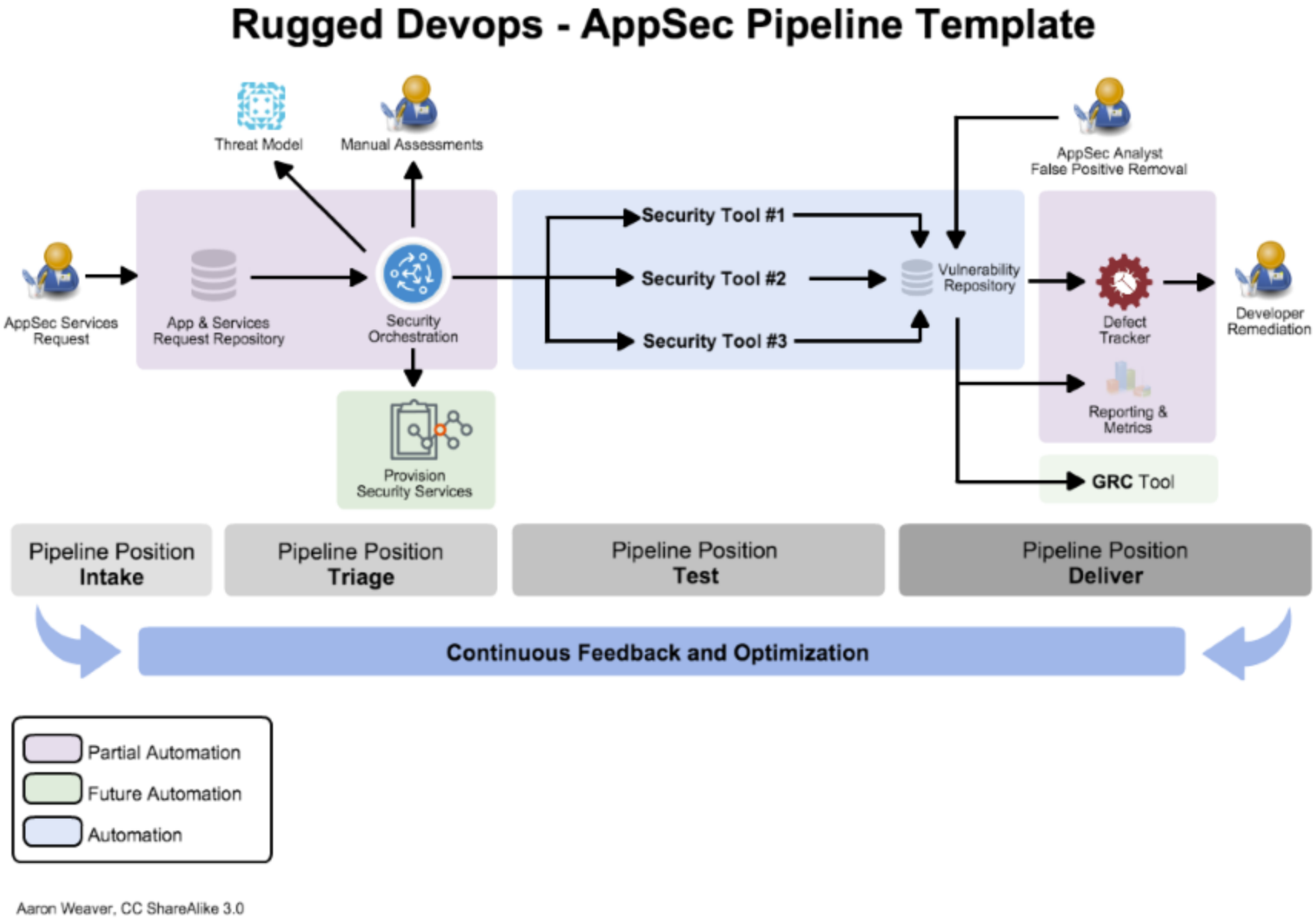
1. Find out if your university uses a Web Application Firewall
2. Find out which product/vendor is being used
3. Find out the number of web applications in your university
    - in total
    - accessible from public Internet (i.e. without VPN)
    - protected by a WAF
4. Repeat steps 1-3 for the company you work at (if applicable)

AppSec Pipeline

AppSec Pipeline

- Applies **principles of DevOps and Lean** to application security
- Designed for **iterative improvement** and has the ability to **grow in functionality organically** over time
- Provides a **consistent process** for the application security team and the constituency (devs, QA, product & senior managers)
- Throughout the process flow each activity has well-defined states
- **Relies heavily on automation** for repeatable tasks

ℹ️ *Start with the area of greatest pain and work on a reusable path for all the AppSec activities that follow.*



Security Champions

> Security Champions are "active members of a team that may help to make decisions about when to engage the Security Team". They act as a core element of security assurance process within the product or service, and hold the role of the Single Point of Contact (SPOC) within the team.

🔮 It's all a question of scalability...

Developers will outnumber Security staff in almost every organization. There is *no way* to scale this up without either heavy automation or offloading security responsibility to the development teams!

Security Champions playbook

- Enumerate products and services
- List teams per each product
- Identify Product manager (responsible for product) and team manager (working directly with developers)
- Write down technologies (programming languages) used by each team

- Measure current security state among the teams and define security goals you plan to achieve in mid-term (e.g. by using OWASP SAMM)
- Identify the places where champions could help (such as verifying security reviews, raising issues for risks in existing code, conducting automated scans etc.)
- Write down clearly defined roles, as these will be the primary tasks for newly nominated champions to work on

- Introduce the idea and role descriptions and get approvals on all levels - both from product and engineering managers, as well as from top management
- Together with team leader identify potentially interested candidates
- Officially nominate them as part of your security meta-team

- Make sure to have an easy way to spread information and get feedback
- While differing from company to company, this usually includes chats (Slack/IRC channel, Yammer group, ...) and separate mailing lists
- Set up periodic sync ups - bi-weelky should be fine to start with

- Build a solid internal security knowledge base, which would become the main source of inspiration for the champions
- It should include security meta-team page with defined roles, secure development best practices, descriptions of risks and vulnerabilities and any other relevant info
- Pay special attention to clear and easy-to-follow checklists, as it's usually the simplest way to get the things going

- Develop your ways or choose one of the below to keep in touch and maintain the interest of the champions
- Conduct periodic workshops and encourage participation in security conferences
- Share recent appsec news (e.g. Ezine) via communication channels
- Send internal monthly security newsletters with updates, plans and recognitions for the good work
- Create champions corner with security library, conference calendar, and other interesting materials

Exercise 9.4 (🏠)

1. Download the SANS Cloud Security and DevSecOps Best Practices / SWAT Checklist poster
2. Compare your company's development & operations tools with the ones proposed on the *Secure DevOps Toolchain* page:
    i. In which of the 5 phases do you have the most overlap?
    ii. Where do you have the least?
3. For the last development project you worked on, which items from the *Securing Web Application Technologies (SWAT) Checklist* did the team take into account?