SECURE SOFTWARE DEVELOPMENT LIFECYCLE DEMYSTIFIED

Tools and lessons from building financial applications

The average cost of data breaches in 2017 was 3.62 million dollars[1] and Equifax alone lost 439M dollars[2] in the aftermath of one of the worst data breaches in history.
The recent EU GDPR regulation pushes those numbers even further: companies may be fined up to 20M dollars or 4% of their global annual revenues in case of data breaches.

So the common misconception that implementing security in application is costly, is becoming simply unsustainable: building INsecure applications can cost organizations orders of magnitude more than building secure ones. Other common misconceptions are that securing applications means building fences (firewalls and any other sort of perimetral defenses) around your ecosystem of applications, or buying magical “black-box” products from big vendors that solve all of your security issues in 6 months.

In the following we will discuss practices, tools, and lessons learned from our experience in helping our customers build secure applications.

[1] Source IBM: https://ibm.co/2tMp7ek
[2] Source Reuters: https://reut.rs/2QsGivt

1. Secure Software Development Lifecycle Tenets

In our experience

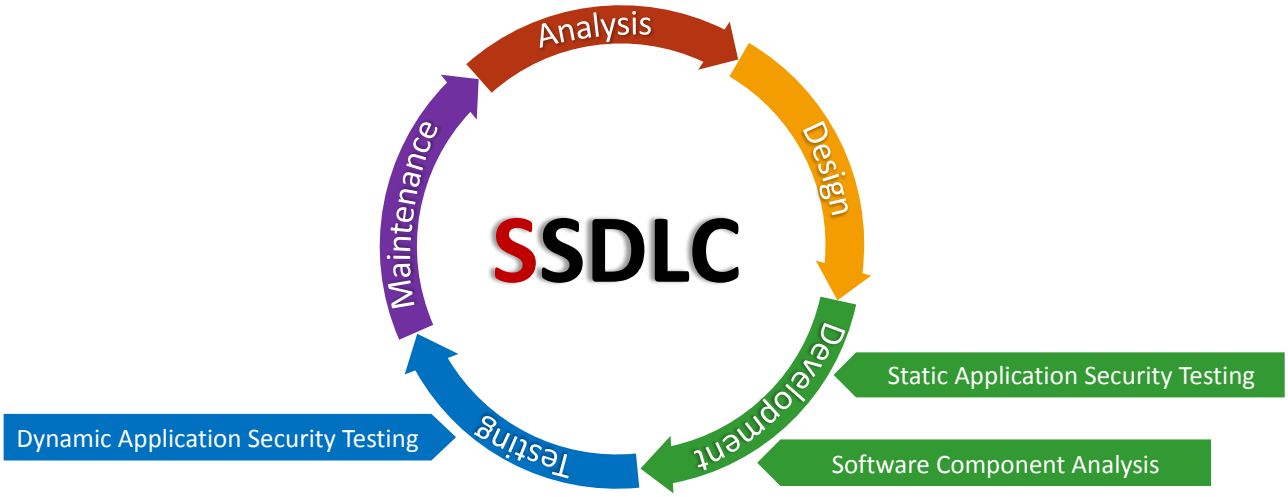
- **Security** should be a **process**, i.e., it should be an integral and pervasive part of your software development and delivery lifecycle;
- **Security** is a matter of **tools and people,** i.e., tools can help you, but do not rely on them blindly because they alone can fail; use tools instead to build **observability** in your process and to **educate** people to the consequences of bad programming choices;
- **Security** should be **testable** in an **automated** fashion, i.e., anyone should be able to enforce rule checking to assess application security at any given time (possibly at every developer commit), and block application artifacts to be built in the first place, let alone be deployed to production, if relevant vulnerabilities are discovered;
- **Security** should be **measurable**, i.e., you should turn all tests and security rule checking activities into actual numbers, and build a set of coherent KPIs that
    - let you clearly understand how your application ecosystem security varies over time (and you definitely want your KPIs to turn better and better at any commit)
    - let you enforce technical and business/legal Service Level Agreements

Security as a Process

In a traditional software development process, security is often thought about mainly in the maintenance phase, when your application is in production. Malicious attackers find vulnerabilities in your application and while your organization begins firefighting technical, business, and legal issues, teams soon start blaming each other, patches pile up in a poorly coordinated fashion, stocks value plunge and attackers may already have stolen sensitive information or compromised your business.
Sometimes, vulnerabilities are the result of bad design choices and become so deeply tangled in your product that you can only mitigate them up to a certain point, but not remove them completely.
A typical example in the hardware world comes from the Meltdown and Spectre vulnerabilities that affect the last generations of microprocessors: those

vulnerabilities are the result of poor design choices, and dramatically impact any modern microprocessor, making patches and fixes in many cases infeasible.
This is why you should build application security right from the inception phase of any development process.
Here we will focus on a few relevant practices and tools that you can integrate in the early phases of your development and delivery lifecycle to automatically enforce security

The landscape of secure development practices and tools is wide, and the following sections will focus on a some relevant practices and tools you can easily integrate in you process right now. These practices and tools focus on earlier phases of the development and delivery process, and especially relate to the development and testing phases.

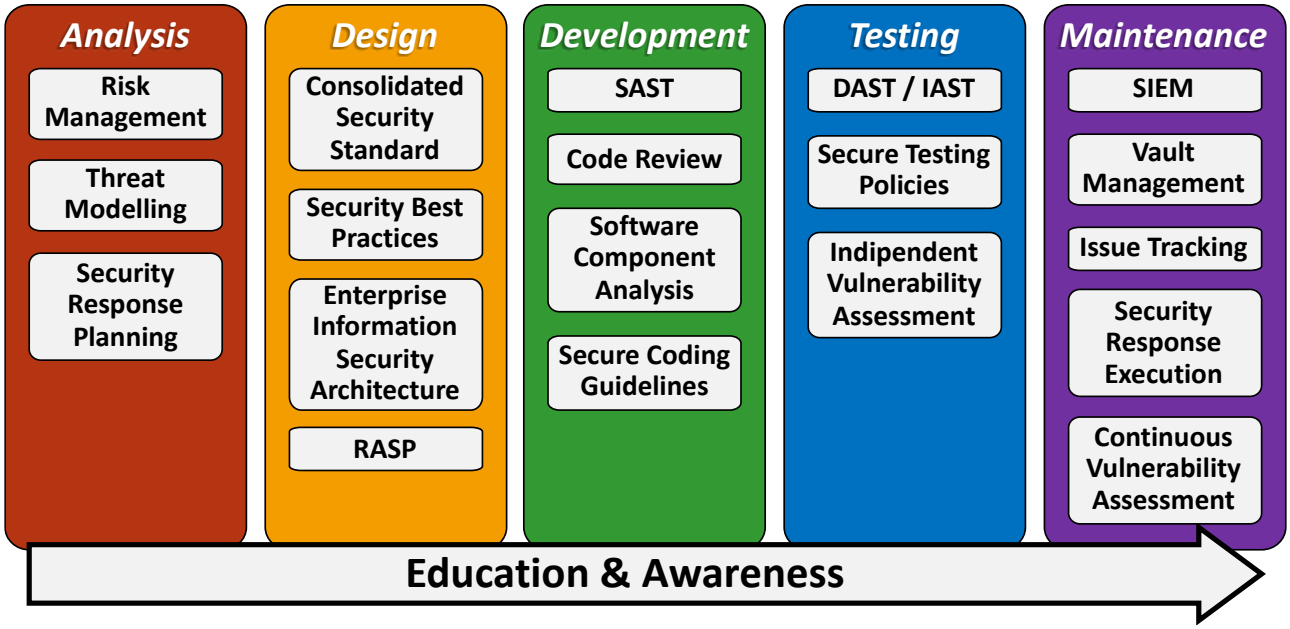


SSDLC -Secure Software Development LifeCycle – some key practices

Security in Software Development Lifecycles should entail much more practices (more on some of them in following blog posts), with different levels of maturity, in any phase on the traditional lifecycle.

One key point we want to stress, though, is that the common denominator for any SSDLC initiative should be to focus and invest on education and awareness, not only

of your development teams (both internal and external), but also of your organization as a whole.



Secure Software Development LifeCycle conceptual framework

Security should be testable

Here we will focus on three main practices to test and enforce security during the development phases, namely

- Software Component Analysis
- SAST – Static Application Security Testing
- DAST – Dynamic Application Security Testing

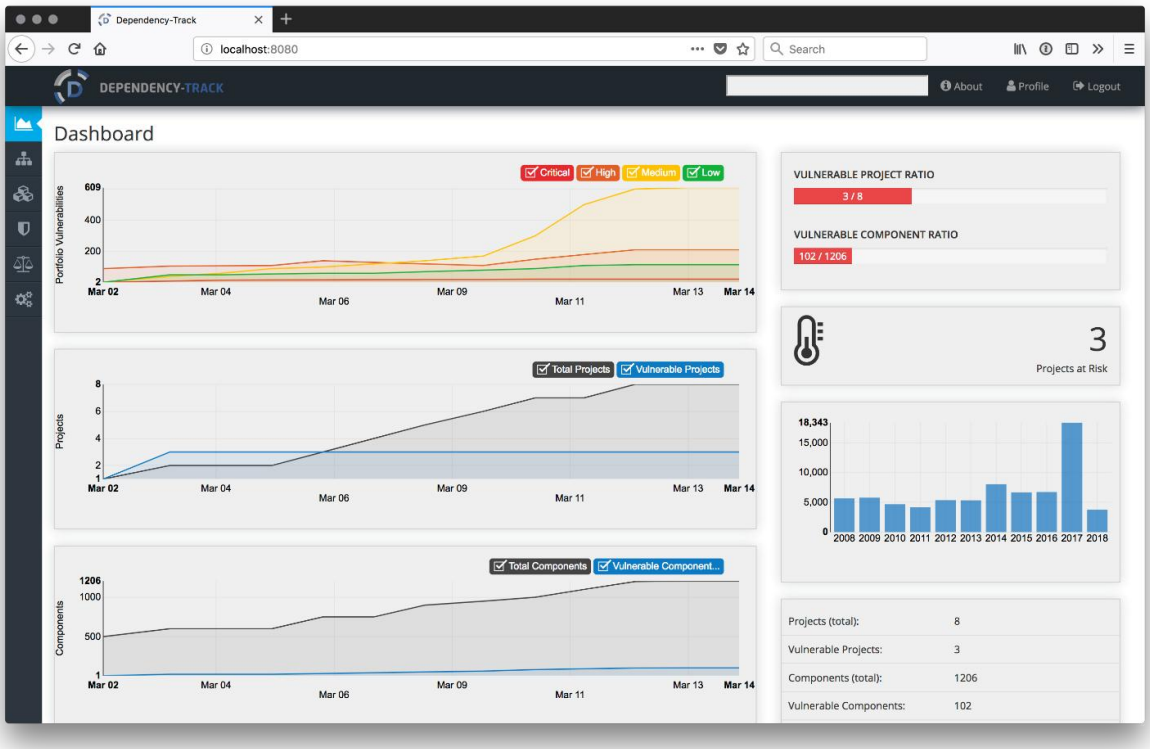
Software Component Analysis

The Equifax data breach[3] responsibility was not on Equifax code itself, but rather on a third party, widely used open source library, the Apache Struts 2.x web application framework, that was extensively used on Equifax web portal. Such vulnerability allowed malicious users to steal 145.5 M user accounts, and resulted in a massive public backlash, and a subsequent dramatic plunge on Equifax stock value.

[3] EQUIFAX is a large credit reporting agency in U.S.A.; on march 2017 a serious input parameter validation flaw was disclosed on Apache Struts 2.x (CVE-2017-5638). On may 2017, malicious users exploited such vulnerability in the Equifax web portal to steal 145.5M users data.

In the aftermath of the data breach, many customers began asking us: “Are we using the Apache Struts 2.x library too? Where in our application ecosystem do we use it?”
Software Component Analysis tools allow to answer such questions, and Dependency Track (https://docs.dependencytrack.org/) is an open source alternative that has recently seen a dramatic increase in popularity and widespread adoption.
This practice becomes even more relevant in contexts where the application ecosystem is composed of hundreds or even thousands of applications, both developed internally, and by a range of external contractors; in such situations, organizations should be able to be informed as soon as possible about the following

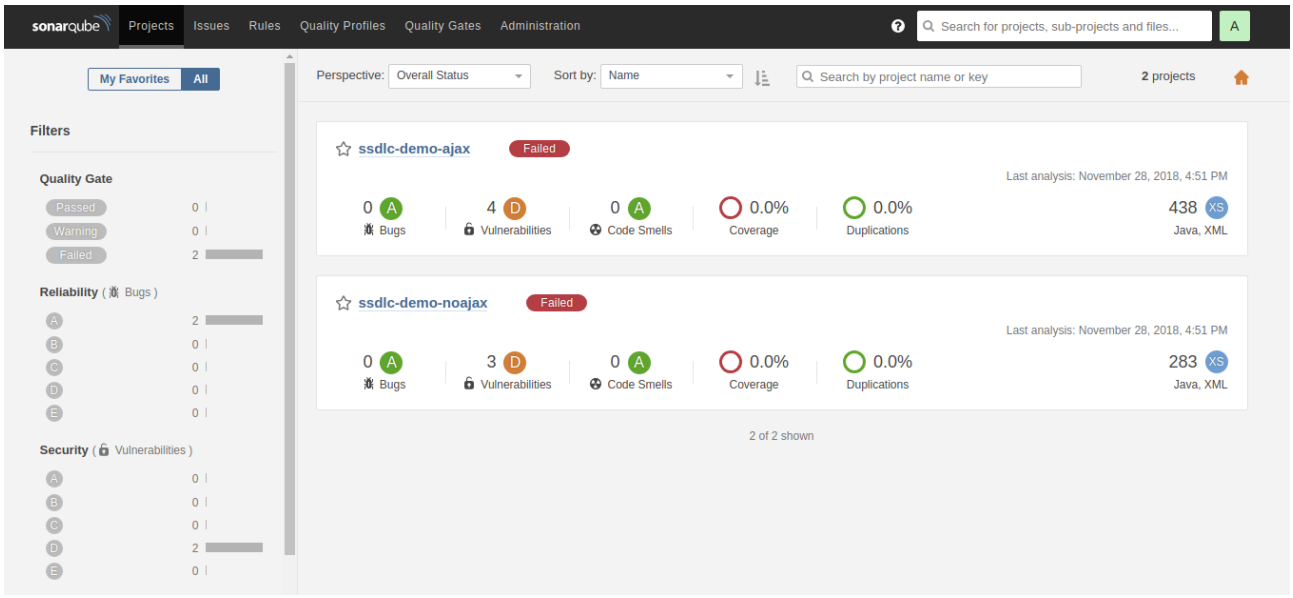
- internal/external developers mistakenly introduced critically vulnerable dependencies to an existing/running application;
- a new application enters the organization ecosystem, with relevant vulnerable dependencies;
- a new vulnerability on dependencies used in the organization gets published.



Dependency Track dashboard examples

SAST – Static Application Security Testing
This practice entails analyzing source code to identify patterns of bad programming habits, such as missing SQL query parameter validation that may lead to SQL injections[4].

SONARQUBE (https://www.sonarqube.org/) is an open source SAST tool, with widespread adoption, a strong community, and a myriad of plugins and rules to enforce security validations. Findbugs (https://github.com/spotbugs/sonar-findbugs) is one such plugin that extends default rules with more security-oriented ones.
Vulnerabilities identified by SONARQUBE fall into different ranges of severity and are directly identifiable in the source code. Quality gates can then be configured to force build failures in case a relevant number of critical vulnerabilities are found, so as to prevent insecure applications to reach any environment, let alone be deployed to production.
SONARQUBE rules and quality gates allow developers to receive fast feedback about vulnerabilities, **promoting awareness and education through observability**.



SONARQUBE dashboard examples

[4] SQL Injection is a specific attack pattern that lets a malicious user inject SQL code in a poorly validated database query, for instance to bypass query filters or to exfiltrate data from the database

DAST – Dynamic Application Security Testing
This practice entails **testing the application runtime** to identify vulnerabilities, for instance via penetration testing techniques. ZAP – Zed Attack Proxy (https://www.zaproxy.org/) – is an open source Web Application penetration testing tool that allows you to perform active and passive scans on web applications.
Passive scans simply inspect the HTTP traffic that goes through ZAP to identify patterns of common vulnerabilities, such as cross-site scripting or SQL injections, whereas active scans perform real a penetration tests, following common attack patterns malicious users typically exploit (e.g., trying to inject SQL code via HTML form parameters).

All of the above tools are open source, easily automatable via APIs and extensible via plugins that allow for custom behavior to fit specific organization needs.

Security should be measurable

When trying to measure security within your application landscape, two needs emerge:

- The need to evaluate vulnerabilities in objective, comparable ways, so as to rank them and focus the efforts of solutions for the most relevant and impactful ones;
- The need to score your source code against vulnerabilities, to see how well it performs in time (e.g., new vulnerabilities solved/introduced at every code commit).

The Common Vulnerability Scoring System (CVSS) is a standard framework that captures the principal characteristics of a vulnerability and produce a numerical score reflecting its severity. The numerical score can then be translated into a qualitative representation (such as low, medium, high, and critical) to help organizations properly assess and prioritize their vulnerability management processes.
The CVSS is composed of a set of metrics that, combined together, form a unique indicator; these metrics fall into three groups:

- **Base metrics** take into account of both the *Exploitability – characteristics of the thing that is vulnerable* (e.g., the Attack Vector – the channel one attacker may conduct exploits, could be the Internet or a reserved protocol) and *Impact - consequences to the thing that suffers the impact* (e.g., the Availability Impact evaluates how the vulnerability may affect service/system uptime and continuity); these metrics typically are static and do not change over time;

- **Temporal metrics** take into account characteristics that may change over time, for instance becoming less relevant (e.g., a new official patch for a vulnerability may decrease the overall CVSS over time).
- **Environmental metrics** group characteristics of a vulnerability that are relevant and unique to a particular application environment, e.g. to mitigate consequences, as well as to promote or demote the importance of a vulnerable system according to business risk.

The **Weighted Risk Trend – WRT** scores vulnerabilities in the source code, and specifically attributes different weights to classes of issues (higher weight to more critical ones); the weighted sum of all issues constitutes the WRT index. This index is specifically interesting since it can track the security "performance" of a codebase over time: a progressively shrinking WRT index means the source code grows safer, whereas an increasing WRT index means new vulnerabilities appear in code without developers taking actions to mitigate them.

---

2. Security in Continuous Integration and Delivery pipelines

Practices, KPIs, and tools we mentioned above should be part of an integrated Continuous Integration and Continuous Delivery pipeline, so that potentially every developer commit can trigger SAST, DAST and software component analysis, and prevent applications to be ever built, let alone deployed to any running environment.
Automating security testing and measuring applications security against given KPIs allows developers and organizations alike to feel increasingly more confident in the code they write and put into production, and to create a common ground for educating teams through awareness of good/bad programming habits, security-wise.

3. KEY TAKEAWAY POINTS

Our experience in devising Secure Software Development Lifecycles for our customers taught us the following:

- Security can **be a process** and **can fit** and **extend** your software development process naturally;
- In order to be constantly enforced, security should be **testable and measurable** in an automated fashion: tools and conceptual frameworks exist to let you automate vulnerabilities detection and scoring them;
- Automated Tools for vulnerabilities detection (e.g., SAST, DAST, and Software Component Analysis) are extremely helpful; however, tools can fail. They are actually written by humans, and humans fail; they usually rely on heuristics, and heuristics may fail; you have to deal with false positives, and missing vulnerabilities detections; so any organization should **use tools wisely**, and complement them with **periodic code review sessions**, to identify undetected issues;
- Use automation, scoring and reporting, as well as human code reviews to promote **culture and knowledge** about security across the organization.

Imola Informatica S.p.A. · via Selice 66/a · Imola (BO)

+39 0542 32640 | www.imolainformatica.it | blog.imolinfo.it