An Open Source Static Code Analyzer for Security Code Review

Wanchun (Paul) Li

- Project Name: spec-sca
- Goal

Build a static code analyzer for
security code review with low false positive

- Github repository

https://github.com/paulur/spec/tree/master/spec_sca

Roadmap of the Presentation

- Code Review in SDLC
- Design Principles to Reduce False Positive
- High Level View of Architecture Design
- Project Phases

Overview of SDLC

Development Phase	Requirement	Design	Implementation	Testing
Expected Input to Security Team	Business Logic Use-case	Architecture Diagram	Source Code	Deployed code .jar, .war files
Security Assessment	Mis-use Case Analysis	Architecture Analysis	Security Code Review	Penetration Testing

Comparison of

architecture analysis, code review, penetration testing

	Architecture Analysis	Security Code Review	Penetration testing
Effort	Low	High	Medium
Coverage	High	High	Low

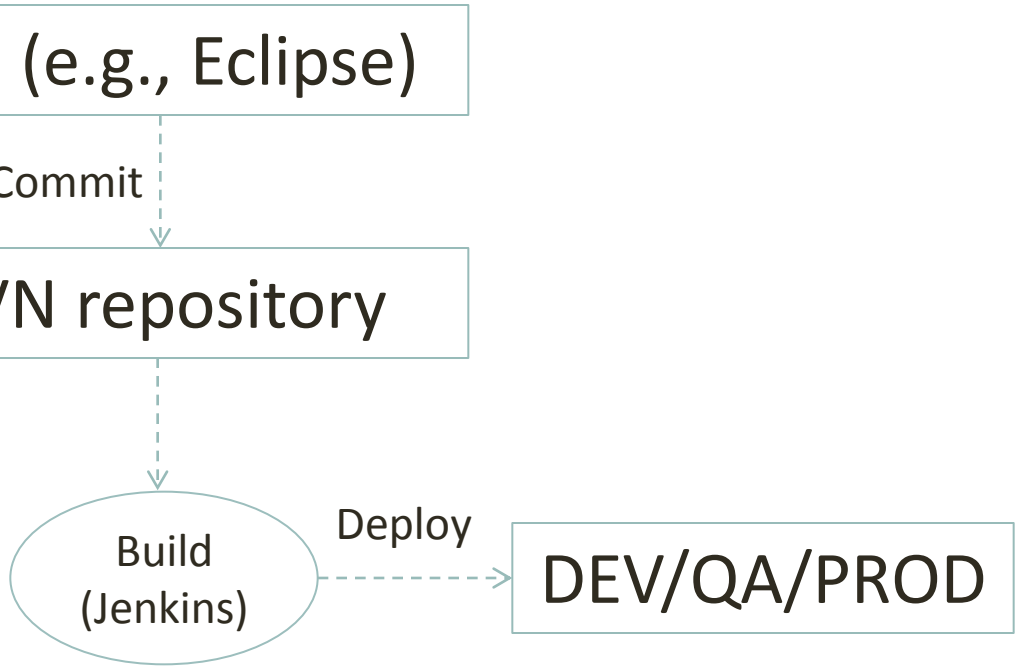
Static Code Analyzer (SCA) for Security Code Review

After Developer has the implementation,
we need a SCA tool for security code review

Security Code Review
High
High

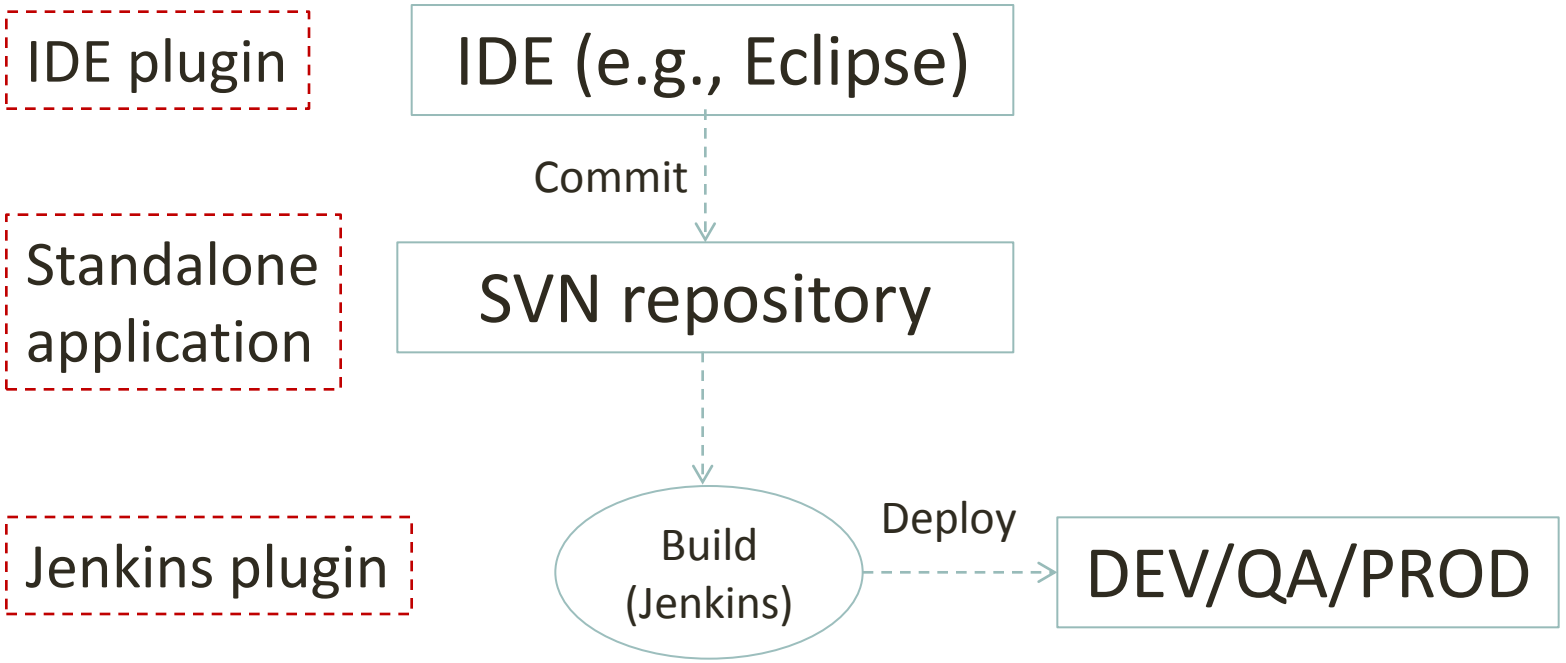
Key Challenge is to reduce false-positive

The Lifecycle of Code



Code Review Touch-points

A code review tool can be used as a plugin or a standalone application



Design Principles to Reduce False Positive

Principle 1: Use SCA for the "Right" Bugs

Use our security weapons (i.e., pen-testing and SCA) right. For example,

- XSS are right for penetration testing,
  but difficult for SCA with low false positive

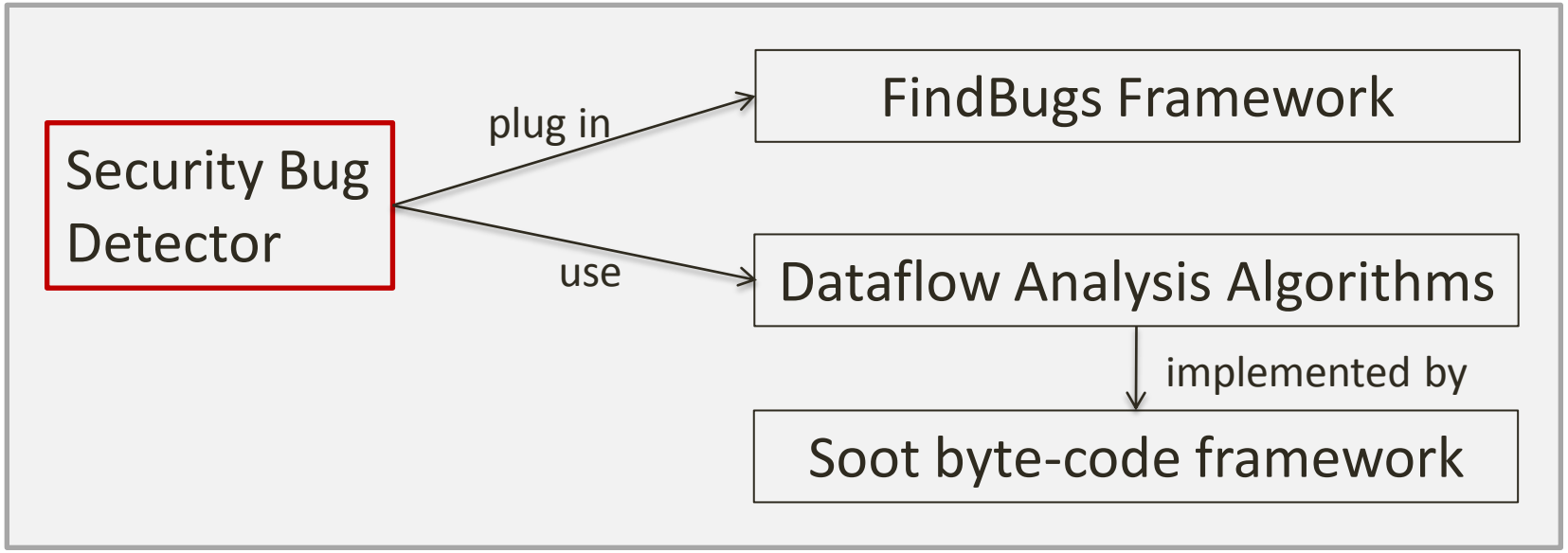
- Encryption bugs are right for SCA,
  but almost impossible for penetration testing

Design Principles to Reduce False Positive

Principle 2: Apply Program Analysis Algorithms

- Program analysis algorithms (e.g., data flow analysis) can provide more accurate context to detect bugs
  - Apply Inter-procedure/class/module algorithms
- Apply these algorithms in unsafe approximation[1] fashion to reduce the algorithm complexity
  - Combine security bugs features in the approximation

High Level Architecture



- The core component is the Security Bug Detector
- Detectors are plugins to Findbugs [2]
  for bugs reporting and integrating with the build process
- Detectors use dataflow algorithms to detect bugs
- Dataflow algorithms are implemented using Soot [3]

Project Phases

- Product Design
  - Determine the right bugs for SCA
- POC
  - Select the right techniques
  - Decide the high level architecture
- Implementation
  - Three phases of implementation
  - Later phases are to reduce the false positive and improve the accuracy of earlier phases

Implementation Phase 1

- Implement detectors of the right bugs to detect sink points where could introduce bugs

  For example, raising a warning at the code where an encryption cipher is initialized

- These detectors can assist manual code review to report the sink points as the entry points of review

Implementation Phase 2

- Implement common dataflow algorithms used for detecting different types of bugs

  For example, inter-procedure/class/library dataflow algorithms for detecting constant values

- These algorithms can reduce false positives and improve the accuracy of detectors that share common detecting rules

Implementation Phase 3

- Determine and implementation dataflow algorithms used for individual types of bugs

- These algorithms can reduce false positives and improve the accuracy of individual detectors

Reference

[1] Unsafe approximation
http://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Slides/BasicAnalysis3.pdf

[2] Findbugs http://findbugs.sourceforge.net/

[3] Soot https://github.com/Sable/soot