Controlling Usage and Security Risks of OSS in Applications Ecosystems

Stan Zajdel, Independence Health Group, Philadelphia, USA

Diego Elias Costa, LATECE Lab, Univ. Québec à Montréal, Canada

Hafedh Mili, LATECE Lab, Univ. Québec à Montréal, Canada

The good

What we said 30 years ago	Today's reality

The bad

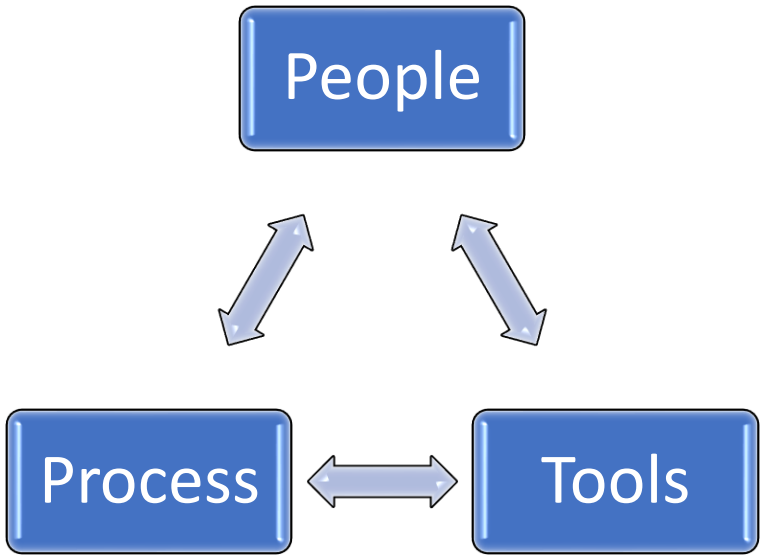
Open source software (OSS) is

In theory	In practice
• Developed by a large number …	
• Of highly competent, sophisticated	
• altruistic	
• … intensive users	
• … whose contributions are vetted by yet more competent, sophisticated and altruistic users	
• Good quality because of inherently large test coverage	

The ugly

- Color and Faker
  - Disgruntled maintainer introduced an infinite loop
- node-ipc:
  - Political protest
- Log4shell
- The kitchen drawer syndrome: a framework that does a lot more than:
  - You need
  - It should

Approach: DevOps ➔ Dev**Sec**Ops



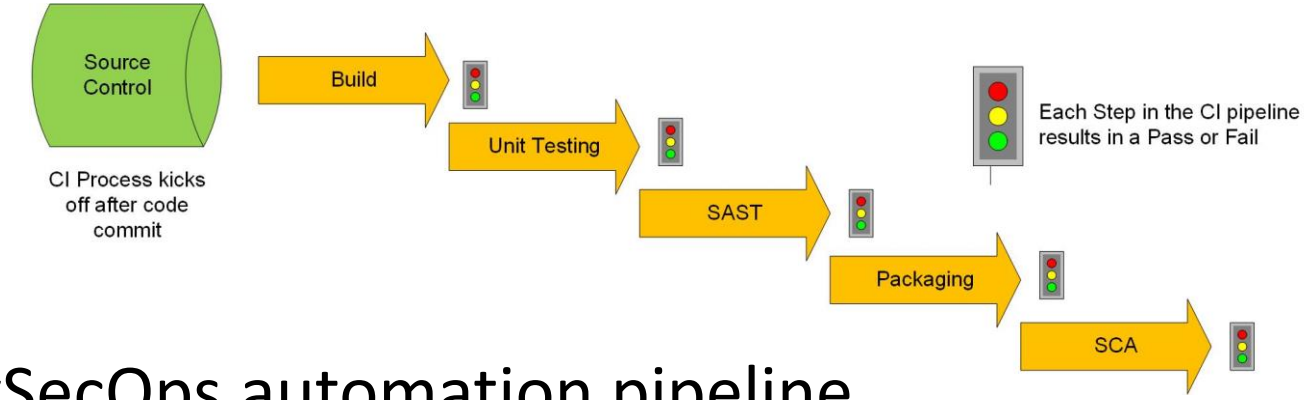
- People:
  - Don’t interfere with their « creative workflow »
    - Validate *after* the fact
  - Educate them on the security risks of unfettered reuse
- Tools:
  - Source Composition Analysis (SCA) early in the DevOps pipe
    - Read/Query a vulnerability database
  - Flag or block problematic builds
- Processes
  - Software vetting process for populating the vulnerability database
  - Research, categorize, reduce

Outline

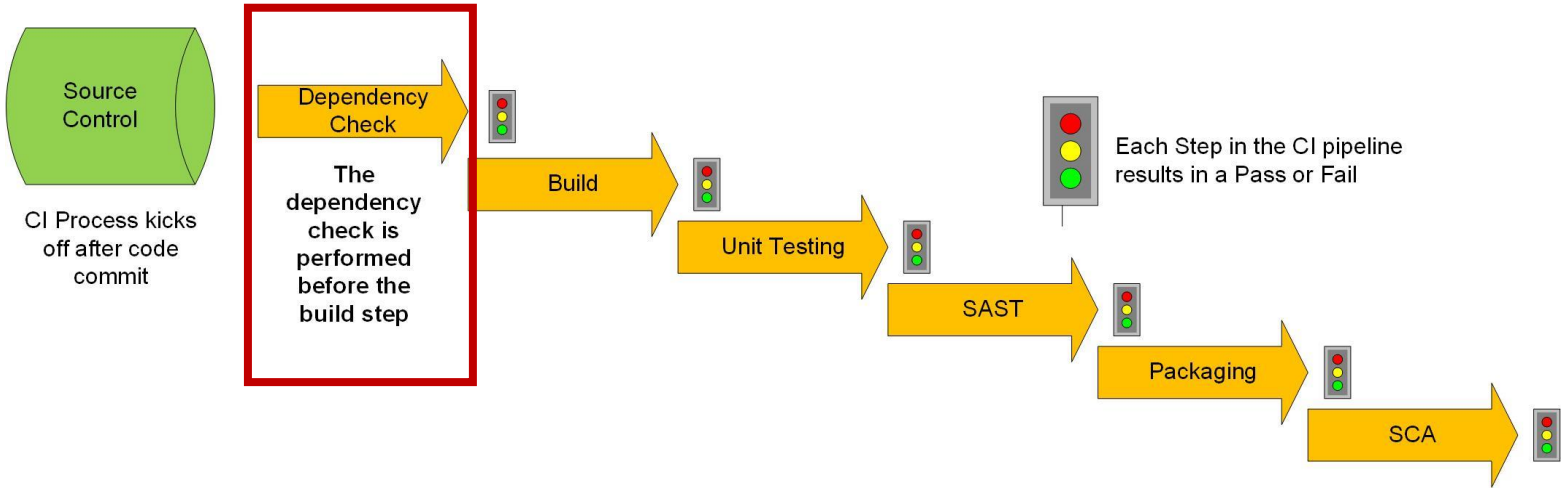
- ***The tools***
- Preliminary results
- What next

The Tools

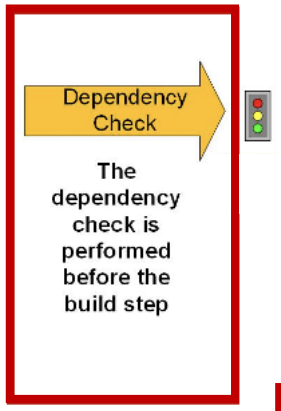
- Traditional DevOps automation pipeline

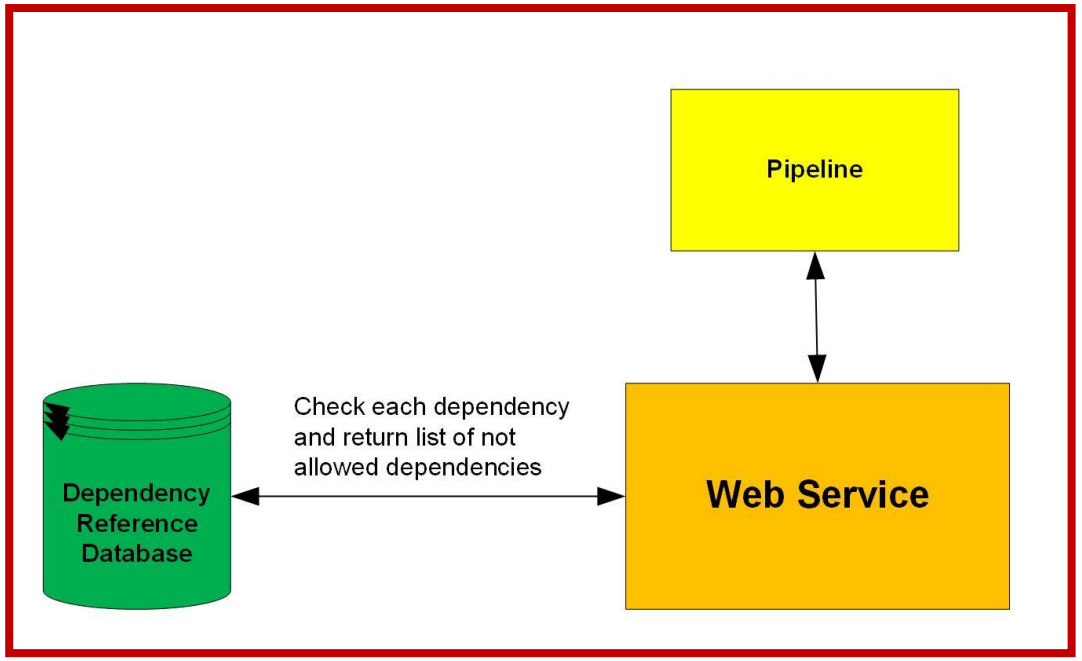


- Our DevSecOps automation pipeline



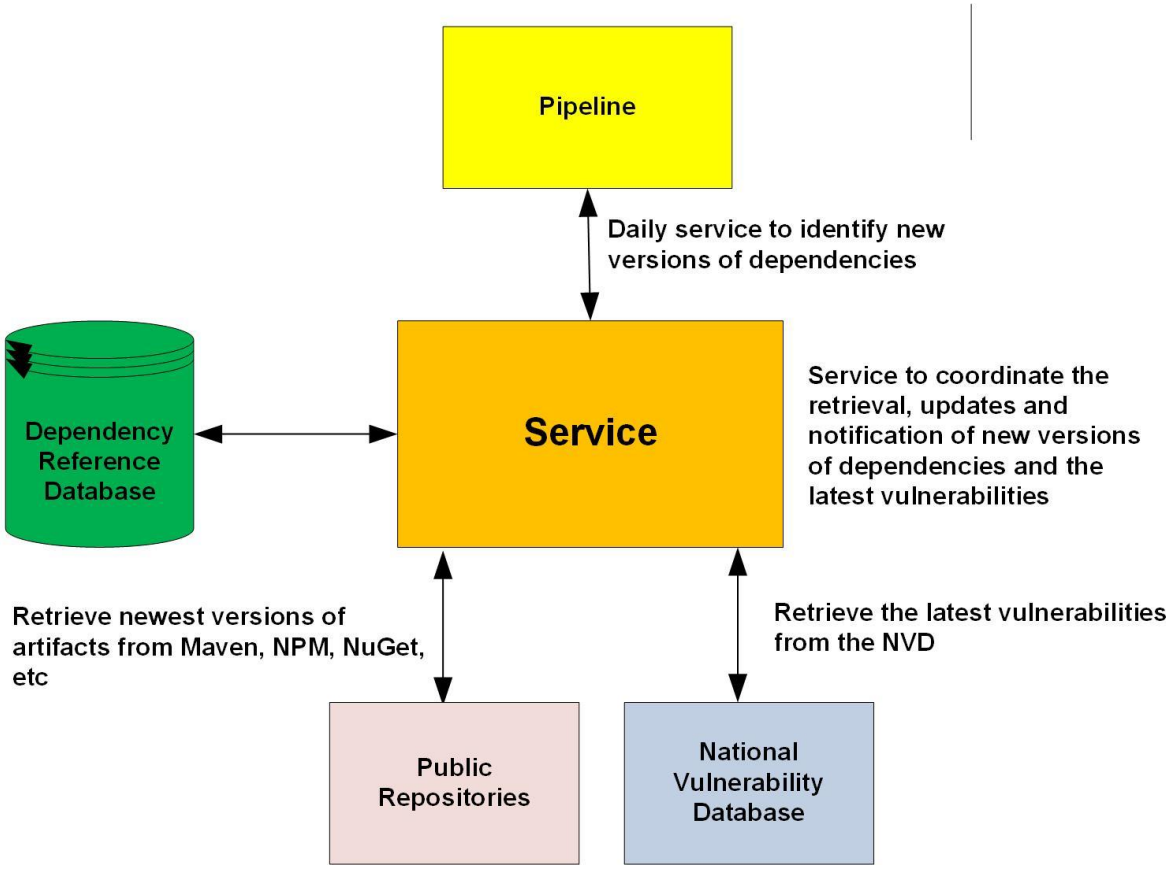
The dependency check





- Upon commit, a web service call with repo name
- Web service:
  - Checks build type to locate build config file
  - Invokes config file specific parser to extract *direct dependencies*
    - Package/library
    - Version
  - Queries Dependencies Reference Database

Updating the Dependency Reference Database



- Different versions of libraries may be used by different teams, or at different times
- New (versions of) libraries appear regularly in builds
- Ideally/ultimately:
  - New libraries/versions are put on probation until properly vetted
  - Builds involving libraries with documented critical vulnerabilities fail after a grace period

Outline

- *The tools*
- ***Preliminary results***
- What next

Preliminary results

Basic metrics

- ≈ 400 developers
- 780 code repositories:
  - 527 Java repositories
  - 211 .Net repositories
  - 42 Javascript repositories

How bad is it?

- 527 Java repositories
  - 1986 unique direct dependencies on Java libraries/versions

Preliminary results – how bad is it?

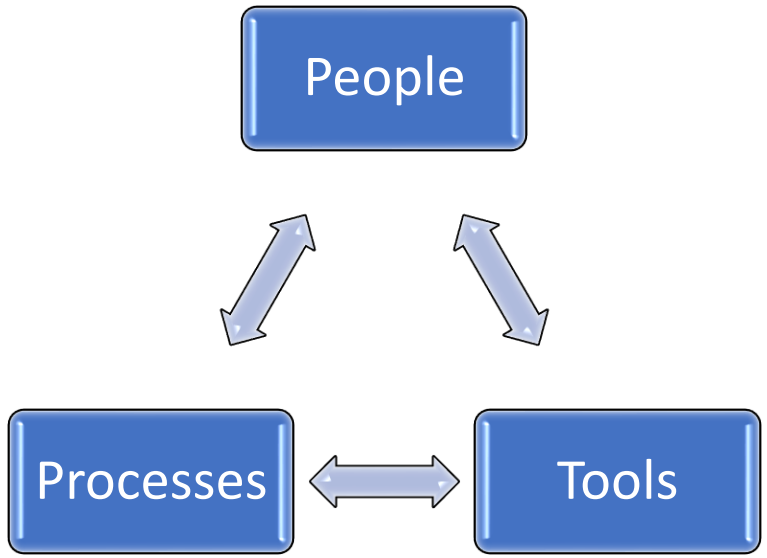
Library Domain	# of Different Libraries
Web Frameworks	60
Logging	54
Database Connectivity	52
REST framework	43
SOAP	30
PDF	34
Email	21
ORM	21
XML Parser	18
Encryption	16
JSON Parser	12
Date/Time Parser	8
Charting	7
Caching	6

XML and JSON Parsers	# Vulns	# Versions
xstream	6	4
xmlsec	6	3
jackson-dataformat-xml	3	13
dom4j	1	2
jdom	1	1
xom	1	1
xmlbeans	1	3
xalan	1	2
xmlschema	0	1
xerces	0	1
sax	0	1
xml-aps	0	2
xmlpublic	0	1
aalto-xml	0	1
javax.xml.stream	0	1
xmlpull	0	1
xpp3_min	0	1
xmlsec	0	1

Outline

- *The tools*
- *Preliminary results*
- ***What next***

People – Processes - Tools



- Tools:
  - Part of the answer
    - Human element + vetting requires people …
    - But there is room for more automation
  - Help enforce agreed-upon processes
- Processes:
  - Who is responsible for vetting?
    - Qualifications but also time management
  - Should derogations be possible?
- People
  - Inform
  - Educate
  - Engage

Thank you for your attention