Aalto University
School of Science
Master's Programme in Computer,
Communication and Information Sciences

Juha Kuusela

Security testing in continuous integration processes

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science (Technology)

Espoo, May 25, 2017

Supervisor: Tuomas Aura, Professor
Thesis advisor: Antti Vähä-Sipilä, M.Sc.(Tech.)

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:
Juha Kuusela

Title of the Thesis:
Security testing in continuous integration processes

Number of pages:	**Date:**
ix + 78	May 25, 2017

Major:
Data Communications Software

Supervisor:
Tuomas Aura, Professor

Thesis advisor:
Antti Vähä-Sipilä, M.Sc.(Tech.)

Modern software development processes in which changes can be deployed to production multiple times a day present a challenge from the software security point of view.

In this work we explore the possibility of using existing software security testing methods and tools in continuous integration to achieve a basic level of continuous security testing. We review existing software security testing methods and tools to determine their applicability to continuous security testing. In four case studies we made selected security testing tools a part of real life software development projects' continuous integration systems and development processes.

We found that continuous security testing is feasible using current security testing methods and tools. Multiple different, complementary approaches to implementing it are available depending on the level of expendable effort and security expertise at hand. Dependency verification is in most cases the best starting point for implementing continuous security testing. Good dependency verification tools, which require minimal effort and security testing expertise from the user, are available for most major programming languages.

Keywords:	**Publishing language:**
Security, Testing, Continuous, Integration	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:
Juha Kuusela

Työn nimi:
Tietoturvatestaaminen jatkuvan integraation prosesseissa

Sivumäärä:	**Päiväys:**
ix + 78	25. toukokuuta 2017

Pääaine:
Tietoliikenneohjelmistot

Valvoja:
Tuomas Aura, Professori

Ohjaaja:
Antti Vähä-Sipilä, Diplomi-insinööri

Modernit ohjelmistokehitysprocessit, joissa muutoksia voidaan viedä tuotantoon useita kertoja päivässä, ovat haastavia kehitetyn ohjelmiston tietoturvan varmistamisen kannalta.

Tässä työssä tutkimme miten olemassa olevia tietoturvatestausmetodeja ja -työkaluja voitaisiin käyttää jatkuvan integraation järjestelmissä perustason jatkuvan tietoturvatestauksen saavuttamiseksi. Käymme läpi olemassa olevia tietoturvatestausmetodeja ja -työkaluja määrittääksemme niiden soveltuvuuden jatkuvaan tietoturvatestaukseen. Testaamme myös valikoitujen tietoturvatestaustyökalujen lisäämistä neljän ohjelmistokehitysprojektin jatkuvan integraation järjestelmiin ja ohjelmistokehitysprosesseihin.

Havaitsemme, että joidenkin osa-alueiden jatkuva tietoturvatestaus on mahdollista olemassaolevien tietoturvatestausmenetelmien ja -työkalujen avulla. Tarjolla on monta erilaista, toisiaan täydentävää lähestymistapaa, joista kukin vaatii eri määrän työpanosta ja tietoturvaosaamista. Havaintojemme perusteella useimmissa tapauksissa paras tapa aloittaa jatkuva tietoturvatestaaminen on kehitettävän ohjelmiston riippuvuuksien verifiointi. Siihen tarkoitettujen tietoturvatestastyökalujen saatavuus eri ohjelmointikielille on hyvä, ja niiden käyttöönotto vaatii hyvin vähän työpanosta ja tietoturvaosaamista.

Asiasanat:	**Kieli:**
Tietoturva, Testaus, Jatkuva, Integraatio	Englanti

Acknowledgements

I wish to express my gratitude to my supervisor and instructor for their encouragement, guidance and feedback during the writing process.

My thanks also go out to all my former colleagues at Avaus Consulting with whom I had constructive discussions on the topic of this work (or software development in general), or who participated in the case studies.

I extend my congratulations to all the fine folks at #kku who accepted my challenges to a graduation race and bested me. Who knows how you would have done without such a good pacesetter.

My deepest gratitude goes to my family for their patience, understanding and occasional prodding during this process. Furthermore, some special thank yous:

To my parents, for their early support of my interest in computers and later of my studies at Helsinki University of Technology. It has been essential to getting here.

To Elisa, for building a loving family with me.

And to Miika, for turning my life upside down.

Espoo, May 25, 2017

Juha Kuusela

Abbreviations and Acronyms

.NET	A software framework developed by Microsoft
API	Application programming interface
AWS	Amazon Web Services
BDD	Behaviour Driven Development
C	A general purpose programming language
CI	Continuous Integration
CLASP	Comprehesive, Lightweight Application Security Process
CLI	Command line interface
CMS	Content management system
CPE	Common Platform Enumeration
CVE	Common Vulnerability and Exposure
DOM-XSS	Document Object Model based Cross-site scripting
DSL	Domain specific language
DevOps	Development and operations. A movement that aims to improve software quality and shorten development cycle times by setting shared goals for software developers and operations personnel and creating a culture that encourages to sharing of knowledge, tools and practices
E2E	End-to-end
GUI	Graphical user interface
IDE	Integrated Development Environment
IaC	Infrastructure as Code
JS	JavaScript
JSON	JavaScript Object Notation
JVM	Java virtual machine
LSAT	Linux Security Auditing Tool
MD5	Message Digest 5, a hashing algorithm
MitM	Man-in-the-Middle

NIST	National Institute of Standards and Technology
NPM	Node Package Manager
NVD	National Vulnerability Database
OOPSLA	Object-Oriented Programming, Systems, Languages& Applications
OWASP	Open Web Application Security Project
PHP	PHP: Hypertext Preprocessor (recursive acronym)
REST	Representational state transfer
SBT	Scala Build Tool
SDL	Security Development Lifecycle
SQL	Structured Query Language
SSL	Secure Sockets Layer
w3af	Web Application Attack and Audit Framework
XML	Extensible Markup Language
XSS	Cross-site scripting
ZAP	Zed Attack Proxy, a penetration testing tool by OWASP
zapr	A CLI wrapper for Zed Attack Proxy

Contents

Chapter 1

Introduction

From the 1990s, there has been a trend in software development methodologies towards shorter development cycle times[1]. Agile methods stress the importance of delivering working software frequently, on a scale of weeks instead of months as before. The Scrum methodology brought with it closer collaboration between business people and developers. Lean software development emphasizes the importance of eliminating waste in the development process in order to deliver software faster. The DevOps — short for "Development and Operations" — movement takes this progress even further by promoting, among other things, more extensive use of automation and better collaboration between different actors in the delivery pipeline, including but not limited to developers and operations personnel. In practice one of the intended effects is to shorten development cycle times.

For the developers of web applications and Software-as-a-Service offerings, these shorter cycle times have been especially pronounced because the centrally hosted nature of the software makes deployment of new versions relatively easy and quick.

In the web application industry, frequent deployments have been a normal practice for many years now. For example, already in 2009 at Flickr software changes were deployed to production 10 times a day[1], in 2012 at Etsy there were an average of 25 deployments a day[24], and in 2013 at Facebook new code was usually deployed to production daily[13].

The developers of desktop software have had to put a bit more effort into converting the faster pace of development into more frequent releases, as traditional release channels for desktop software have not been very accommodating to releasing software frequently. Desktop software release channels have been improving, however, and currently many operating systems come

[1]Cycle time meaning the time it takes to go from an application/feature/change idea to a having it implemented and deployed into production.

with package managers and underlying software repositories that make it relatively easy for both developers to release new versions of software and users to keep their versions up to date. On operating systems that do not feature such a capability, desktop software now commonly includes an updater to keep the end users' versions up to date.

As for mobile software, all the major platforms provide their own application release channels, though it might take some time for an update to be reviewed and green-lighted for distribution to end clients. For example, between September 2013 and August 2014, the average review delays in Apple's iOS App Store fluctuated between 4 and 8 days, with some apps experiencing as long as 30 day review delays during the first half of August 2014.[23]

The trend of ever more frequent deployments or releases is also evident in current version numbering practices. In the past, desktop software was versioned using a scheme such as semantic versioning[20] with at most a major, minor and patch version numbers. Currently, it is common to include a build number in addition. In many web applications and Software-as-a-Service offerings, the user might not even know which version of the software they are using, or the version number displayed might just be a version control commit id hash.

This fast pace of modern software development presents some challenges from the software security point of view. How useful is a yearly or quarterly security audit, or other similarly infrequent security activity, in the software development process in which a new feature can be ready for deployment into production within days of being just an idea in someone's head? How long do the findings of an audit stay valid in such fast-paced development processes? Conversely, how to maintain a reasonable level of assurance that an application is secure enough when deploying new code to production multiple times a day?

1.1 Problem statement

Modern, fast-paced software development methodologies fit poorly together with traditional security practices. Since in most cases the benefits of current software development methodologies outweigh the perceived value of traditional security practices, those practices are largely relegated to rubber stamps with little actual value. We need new ways of ensuring software security. This means a shift from traditional control points that are few and far between to a more continuous workflow that takes place in parallel with regular development.

In this thesis, I will explore one way of doing software security work on a

continuous basis, namely how to use the currently available security testing tools paired with continuous integration systems to provide a basic level of confidence that the software is safe to deploy regardless of the speed of the development process.

1.2 Research goals

This thesis has the following goals:

1. Find methods and techniques for integrating software security testing tools into a modern software development process implementing DevOps practices such as infrastructure automation and continuous deployment.

2. Analyse the challenges related to continuous security assurance. Present lessons learned from analysis of security testing tool documentation, related literature and case studies in integrating tools to development processes.

3. Identify the characteristics of a security testing tool that determine whether it is applicable to continuous security testing.

In the process, we will also review existing security testing tools to find ones that are suitable for continuous testing. We will then test some of the tools in case studies to verify the validity of our selection criteria and evaluate the usefulness of the tools.

Our primary interest lies in achieving continuous security testing of even the most basic kind, preferably with minimal effort and security expertise required on the part of the software developers. The test suites can then be extended for more comprehensive coverage and other desirable qualities of the actual security testing.

1.3 Structure of the thesis

The rest of this thesis is organized as follows. Chapter 2 provides background and context for the problem scrutinized in this thesis, as well as initial reasoning for the approach taken. Chapter 3 lists the different types of automated security testing methods and available tools of each type that were considered to have potential for continuous security testing, with a brief review of each. Chapter 4 covers methods and techniques of integrating the different

security testing methods into the development process for continuous security testing. Chapter 5 presents the results of case studies in adding security testing tools to the continuous integration processes of software development projects. Chapter 6 contains reflections on the types of tools tested and on the effectiveness of different ways of integrating them into the development workflow. Chapter 7 summarizes the work and provides conclusions.

Chapter 2

Background: security testing and agile development

In this chapter, we briefly cover some of the most significant advances in software development methods and their implications from the software security point of view. We will also note some existing models for secure software development that have arisen to cope with the later software development processes. Finally, we will briefly discuss the possibilities of continuous security testing.

2.1 Security implications of the progress of software development practices

Software development methodologies that have evolved since the 1990s have among other aspects incorporated consecutive changes to processes, practices and tools meant to shorten development cycle times.

Many changes brought on by this evolution have also affected software security work. In this section, we note some of the most significant aspects of different development methodologies from the software security point of view.

2.1.1 Waterfall

Before the agile revolution, waterfall projects were the standard approach to software development. The stages of a waterfall project, presented in figure 2.1, are performed one after the other with little or no overlap between consecutive stages.

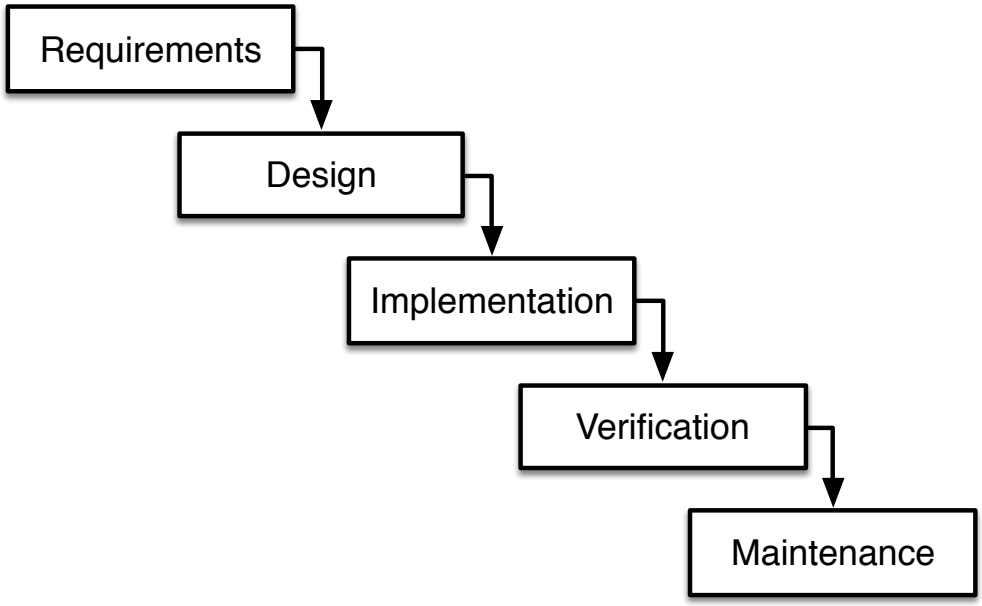

Figure 2.1: Stages of the traditional waterfall development model.

Typical project lengths and release intervals for waterfall projects are anything from months to years.

Security work in a waterfall project typically involves one-time practices such as a security audit of the architecture in the Design stage and security testing and auditing of the completed implementation in the Verification stage.

2.1.2 Agile

Around the 1990s, recognition of software project failures caused by the rigidity of the traditional waterfall model gave rise to an agile approach to software development. The failures occurred in many cases because successfully eliciting all the requirements before implementation in a complex project is extremely difficult, and the waterfall model did not allow for effectively responding to changing requirements once implementation had started.

The agile approach lays out a lightweight software development methodology focused on the ability to respond to changes in requirements. The principles of this approach were codified in the Agile manifesto[8] published by the Agile Alliance in 2001.

Two of the agile principles state that:

> "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
>
> "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

This focus on delivering software early and continuously does not fit well together with the way security work has been done in waterfall projects. To achieve the same security impact in an agile project, architecture audits would conceivably need to be done every time the architecture changes, and implementation audits and security testing before each delivery.

2.1.3 Scrum

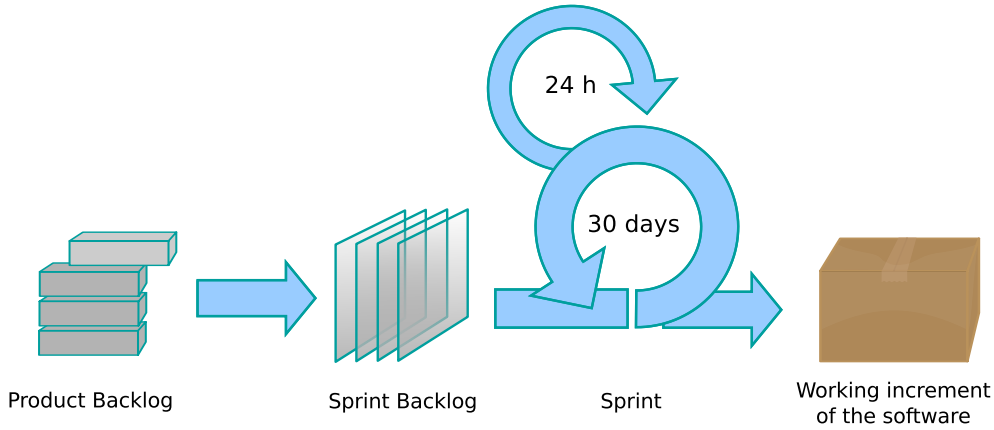

Figure 2.2: The Scrum process. Image by Lakeworks from the Wikimedia Commons (CC-BY-SA 3.0)

Scrum, arguably the most famous and widely adopted agile development framework, was first introduced to the world in the OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) conference of 1995 [25]. A book describing the method in more detail was published in 2001 [22].

A simplified illustration of the scrum model of software development is presented in figure 2.2. In contrast to the waterfall model, the scrum model splits the whole project into smaller iterations that normally last 1-4 weeks. Each of these iterations, called sprints, is effectively a mini-project containing

work from all the phases of a waterfall project and resulting in a shippable increment of software.

The frequency of security work required is quite different than in a waterfall project. An architecture security audit would be required once in the early stages of the project, like in waterfall, after the drafting of the initial architecture, but a re-audit would be required in each sprint where the architecture is changed. Security testing, on the other hand, would be necessary in each sprint since the software changes implemented in each sprint are intended to be deliverable to production.

2.1.4 Lean

Lean software development[19] is a subculture within the agile software development scene. It is an adoption of the lean manufacturing principles of the Toyota Production System[18] into the software development domain.

The most relevant Lean principles from the security point of view are:

1. Eliminate waste
3. Decide as late as possible
4. Deliver as fast as possible

Principle 1, "Eliminate waste", can be used to justify not following security practices that are perceived to have too little payoff compared to the time and resources required, because waste as intended in this context includes delay in the software development process.

Principle 4, "Deliver as fast as possible", can put pressure on doing security work sloppily in order to not hold back delivery of the software.

Finally, principle 3, "Decide as late as possible", means that the software architecture is expected to evolve during development. This means changes affecting security can and will happen at any point during the lifetime of the software, so an unknown amount of security audits would be required. Neither is it necessarily known much in advance when an audit is required.

2.1.5 DevOps

DevOps – short for Development and Operations – is an ideology that promotes collaboration and communications between stakeholders in all stages of software development, from idea conception to production. Automation of software deployment and infrastructure setup is also an essential part of DevOps.

A series of "DevOps days" in Belgium in 2009 popularized DevOps as a term. Many of the core principles and practices of DevOps did exist even before that, but after "DevOps days" the movement begun picking up considerable global momentum. DevOps has since become a major influence on software development practices.

In this thesis, we are mainly interested in the movement's contributions to faster cycle times via infrastructure, testing and deployment automation and the implications that these have for software security.

The "Infrastructure as Code" principle promoted by the DevOps movement opens up new possibilities for continuous testing during software development. Automated server provisioning and configuration combined with automated application deployment provides the capability to easily set up full-fledged, disposable testing and staging environments. This capability can drastically reduce the setup costs involved in security testing, thus making it feasible to do more frequently.

The emphasis on sharing of tools where it makes sense means that security testing tools, or at least their results should be available to all relevant personnel in addition to the security experts. In practice this would mean developers and operations personnel.

The principle of spreading knowledge via open communication and working together calls for security experts to disseminate their expertise to developers and operations personnel by working closely together.

Relevant, fast feedback to interested parties in this context means that the results of security testing should be accessible to the team whose responsibility it is to react to them, as well as to the people in charge of prioritizing work.

These practices promoted by the DevOps movement form a foundation for a more continuous way of doing security work than has ever before been possible.

2.2 Existing methods for secure software development

Several different methods and frameworks for secure software development have been designed in response to the challenges imposed on software security work by modern software development methodologies. We cover briefly some of the most notable approaches in this section.

2.2.1 Checklist approach

The checklist approach to secure software development involves taking a checklist of software security practices and incorporating some or all applicable practices from it into your development lifecycle. Generally, these checklists only specify what should be accomplished, leaving the actual method to use up to the reader.

Many institutions and authors have published secure coding checklists, with some being very broad and general purpose and others targeted for a more specific audience or use case. For a general purpose example, see the OWASP Secure Coding Practices Quick Reference Guide[1].

2.2.2 Provable security

While the term is usually used in cryptography to refer to mathematical proofs, provable security can also be applied more broadly to software security. Such proofs involve modeling both an attacker and the system under scrutiny, and rely heavily on the correctness of the models. Optimally, the actual code of the system can be verified to match the model of the system, for example by using a static analysis tool.

2.2.3 Linus's law

Formulated by Eric S. Raymond in "The Cathedral and the Bazaar"[21] and embraced to a large extent by the Open Source community, Linus's Law states that "given enough eyeballs, all bugs are shallow", or more formally

> "Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone."

While this claim was not strictly made to concern security issues, it has been adopted as an approach for combating security bugs.

The same principle of distributed review can also be applied to other types of security work, such as architecture reviews and other methods for finding security flaws.

[1] `https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide`

Project phase	Security activities
Training	1. Core Security Training
Requirements	2. Establish Security Requirements
	3. Create Quality Gates/Bug Bars
	4. Perform Security and Privacy Risk Assessments
Design	5. Establish Design Requirements
	6. Perform Attack Surface Analysis/ Reduction
	7. Use Threat Modeling
Implementation	8. Use Approved Tools
	9. Deprecate Unsafe Functions
	10. Perform Static Analysis
Verification	11. Perform Dynamic Analysis
	12. Perform Fuzz Testing
	13. Conduct Attack Surface Review
Release	14. Create an Incident Response Plan
	15. Conduct Final Security Review
	16. Certify Release and Archive
Response	Execute Incident Response Plan

Table 2.1: Security activities of Microsoft SDL

2.2.4 Microsoft Security Development Lifecycle (SDL)

Microsoft SDL, outlined in the 2006 book "The security development lifecycle"[14], is a

> "software development process that helps developers build more secure software and address security compliance requirements while reducing development cost"

It maps several security practices to different software lifecycle phases, as seen in table 2.1. In addition to the book, there are comprehensive online resources, including tools, available for free.[2]

While the SDL advocates the use of various security testing practices, in an automated manner where applicable, it mainly stresses that they be performed at regular intervals instead of continuously.

Microsoft SDL also has an official guideline for adopting the SDL practices into an agile development process. The guideline maps all the security activities of the standard SDL into three different categories that fit into the agile development process:

[2] `https://www.microsoft.com/en-us/sdl/`

Every-sprint practices Essential security practices that should be performed in every release.

Bucket practices Important security practices that must be completed on a regular basis but can be spread across multiple sprints during the project lifetime.

One-Time practices Foundational security practices that must be established once at the start of every new Agile project.

An interactive representation of the mapping of security activities to the agile development process is available online[3].

2.2.5 Other

Other existing methods that we will not go into detail in this work, but will mention here so interested readers can look them up: Cigital's Security Touchpoints, Common Criteria, OWASP Comprehesive, Lightweight Application Security Process (CLASP), OWASP Software Assurance Maturity Model (SAMM), NIST Special Publication 800-64 (Security Considerations in the System Development Life Cycle).

Some agile security approaches can also be found from research literature, for example Baca and Carlsson[3] have presented an approach to secure agile software development in which they adopted the most suitable security engineering practices from Microsoft SDL, Cigital Touchpoints and Common criteria into an agile development process.

2.3 Continuous security testing

Many methods for secure software development exist, a few of which were mentioned in the previous section. Some of them involve testing or analysis of various kinds using automated tools. However, none of them really promote doing security testing, or any other security work, continuously.

It would seem that the security community has not yet really started to leverage the modern capabilities of infrastructure, testing and deployment automation. Granted, certain kinds of security work, such as architectural review, might not be feasible candidates for automation. Nevertheless, it is likely that at least some type of security testing could be performed automatically and continuously.

[3] `https://www.microsoft.com/security/sdl/discover/sdlagile.aspx`

According to Boehm et al.[9], the fixing of a software defect is both cheaper and faster the earlier in the development process it is caught. As security issues are a type of defect, it then makes obvious sense to try to detect them as early as possible. Using modern software development tools, "as early as possible" equates to pre-commit (via tool IDE integrations and pre-commit hooks), per-commit (tests run by CI system for all commits) and nightly (for time intensive cases, such as extensive E2E testing of large systems) testing. All of these can be considered reasonably continuous testing.

We do acknowledge that there are numerous possible issues and open questions regarding continuous security testing, some of which are:

- The feasibility of automation. Depending on the test case at hand and the testing methods suited to it, automation might not always be a realistic goal.
- The results produced by security testing tools are in most cases not pass/fail, instead often requiring further analysis and/or verification. This means they mostly cannot be handled as easily as unit tests in CI systems.
- How much security expertise is required for a basic level of security testing?
- How much effort goes into setting up automated continuous security testing? How about possible maintenance of the setup and handling the results produced?
- How frequently is it possible, and on the other hand meaningful, to perform security testing? For each commit, daily, weekly? On what factors does this depend on?

Despite the many open questions and uncertainty involved, we are confident that useful levels of continuous security testing are achievable with reasonable effort even with currently available methods and tools.

2.4 Summary

The evolution of software development methodologies since the 1990s towards faster development cycle times and more frequent releases has imposed numerous challenges on software security work.

More lately, the proliferation of infrastructure, testing and deployment automation has also brought with it previously unseen opportunities for doing security testing continuously alongside the actual development work.

Unfortunately, the software security community has not yet really started to exploit these new opportunities. Despite this, we are confident that a useful level of continuous security testing is achievable using currently available methods and tools.

Chapter 3

Security testing methods and tools

In this chapter, we analyse existing security testing methods and tools to determine the feasibility of integrating them into a CI system. The source material for this analysis are publications on security testing methods and tools that we have deemed to have potential for continuous security testing, and 29 selected security testing tools along with their documentation.

For each security testing method, we present a general description, an analysis of its applicability to continuous security testing based the characteristics outlined below, and present existing tools representing the testing method in question. From the presented tools we pick some for use in case studies in Chapter 5.

The high-level characteristics we have deemed to most affect the suitability of tools for continuous security testing are:

- Ease of integration into the development workflow. This breaks down into two parts, first of which is how easy it is to execute the tool from a CI job, and the second how easily the results produced by the tool can be interpreted and presented in the CI system. This is mainly a tool-specific measure.

- The amount of initial work required for meaningful results. How much configuration does the tool require, what and how many parameters, is it easy to learn. In case of adoption into an existing project does it produce many false positives and can they be easily checked and managed. The security testing method mainly determines this, but an individual tool's implementation also has an effect.

- The amount of continuous work alongside software development that the tool requires. How much effort goes into keeping the tools up to date to catch new vulnerabilities, how much work into processing the results

produced by the tool? Are there efficient mechanisms for verifying findings, managing false positives, handling the results of consecutive runs etc? Can some of this work be spread across projects and teams? This is also mainly determined by the method, but also influenced by an individual tool's implementation.

- How much security expertise is required for meaningful use of the tool. Ideally, we would want tools usable with little to no security expertise on part of the developers to generate meaningful results. This mainly depends on the security testing method.

It is worth noting that we limit our review mainly to open source / free security testing tools, even though commercial tools are often considered better according to various metrics like the thoroughness and accuracy of results [7]. We do this because we are mostly interested in the general principles of the technical and process integration of the tools into a development process. In the context of developing security-sensitive software, the benefits of using a commercial tool might well justify its cost.

3.1 Vulnerability scanning

Vulnerability scanning seems to have considerable potential for our use case. Scanners usually include some sort of a web application crawler that can be used to map possible attack vectors, and thus running some form of basic scan against a web application should require very little configuration or security expertise.

Despite this, the scans can usually be customized quite heavily to fit the specific needs of each use case. Meaningful scan customization, though, can require a significant amount of general security expertise, in-depth knowledge of the target app and expertise in using the specific scanner. Nevertheless, the end result is that these tools are versatile and can be utilized to different lengths depending on an organization's available expertise and expendable effort.

In addition, while meaningful scan customization requires security expertise, the reports generated by these tools generally provide understandable enough descriptions of vulnerabilities found that little security expertise is required for reviewing scan results.

Another advantage that vulnerability scanners have over some other security testing tools is that because they take a black box approach to security testing, the same scanner can be used regardless of the implementation details and technology choices of the target application. A downside to this

approach is that all scanners require a functioning test instance of the target application to be set up, which will require some effort and computing resources. This has effectively prohibited using vulnerability scanners for continuous testing during software development before the advent of modern infrastructure, testing and deployment automation methods.

Vulnerability scanners are reasonably good at detecting the most basic (and common) forms of vulnerabilities. Some examples are SQL injections and cross site scripting[7, 11], which rank as 1st and 3rd in OWASP's Top 10 Most Critical Web Application Security Risks 2013 [26]. As such, vulnerability scanners can be very useful for achieving a basic level of confidence in an application's security.

On the other hand, according to several studies[7, 11, 15] vulnerability scanners lack detection capability against more complex forms of XSS, SQL injection and other vulnerabilities. The studies in question were published in 2010, though, so there has been ample time for improvement since then. And indeed, multiple commercial products[1,2] have later claimed detection capability for more complex vulnerability types such as DOM-XSS.

For best results, the vulnerability scanner should be selected to fit the target, since most vulnerability scanners are geared toward detecting certain types of vulnerabilities. This is especially if an application's infrastructure incorporates some widely used components, such as a popular CMS platform. There are usually vulnerability scanners focusing specifically on detecting vulnerabilities in such components. Some examples are WPScan[3] for WordPress, CMS Explorer[4] for multiple CMS solutions and OWASP Joomla! Security Scanner[5] for Joomla.

There are some limitations, too. One is that scans can easily take a lot of time if not specifically crafted to fit the target application. For example, a crawler might want to visit each individual item's page in a web shop where such pages are dynamically generated and items can easily number in the thousands and upwards from there. This means that in most cases it is not practical to run scans like unit tests continually against every commit in CI to get immediate feedback, but instead maybe as nightly runs. Of course, the time taken for scans can be considerably shortened by running a more targeted scan.

[1] `http://blog.portswigger.net/2014/07/burp-gets-new-javascript-analysis.html`

[2] `https://dominator.mindedsecurity.com/`

[3] `http://wpscan.org/`

[4] `https://code.google.com/p/cms-explorer/`

[5] `https://www.owasp.org/index.php/Category:OWASP_Joomla_Vulnerability_Scanner_Project`

In addition, errors in application logic are not within a vulnerability scanner's ability to detect. For example, being able to order from a web store without paying is not something a vulnerability scanner will raise alarms about.

Another factor in scanning times is that the scanning can require quite a bit of infrastructure. For speedy scans, multiple scanning engine dispatchers are usually required, along with the actual scan orchestrator application and database to store results in. Moreover, after removing that bottleneck, the limiting factor becomes how well the application under scan is able to cope with all the requests resulting from the scan, so for optimal speed more resources might be required on that end, too.

Vulnerability scanners also add new detection capabilities via updates to the tool itself or via new plugins or updates to existing ones, so latest versions of the tools and their plugins should be used for best results.

Analysis of the characteristics laid out at the start of this chapter:

Workflow integration Easy to trigger from CI, but findings at best representable as separate HTML reports for each scan in CI. Only Arachni tracks findings across scans. Time taken by scans practically limits continuousness to nightly testing.

Initial work Some scanners require infrastructure for the scanner to be setup, some do not. All scanners require a working target application to be set up. All scanners can do a default scan without any initial configuration.

Continuous work Some work required periodically for keeping the tool up to date. Reviewing results can take a considerable amount of effort, especially if the scans are not well targeted.

Security expertise Required for effective scan customization. Results should be reviewable to some extent without much security expertise, as the tools provide generally provide good descriptions of the issues found. However, without security expertise it can be difficult in many cases to determine if a warning reported by a scanner is an actual issue or not.

3.1.1 Arachni

Arachni[6] is a vulnerability scanner focused on web application security.

[6] `http://www.arachni-scanner.com/`

It provides a convenient automation interface via its CLI tool. The results of a scan are exportable into an output format suitable for automated processing, so it should be relatively easy to integrate into a CI system.

In addition to the CLI, Arachni features a web interface that can be used for both running scans and reviewing the potential issues found in them. So, false positive management could be done through it.

Arachni also includes the concept of scan revisions, and through these can intelligently manage issues found across multiple successive scans of the same target. In a CI setup this should prove useful.

Another feature not found in most other scanners is the ability to utilize separate scan dispatchers, which should make Arachni scans easy to scale.

At the time of writing, the Arachni project is actively developed and overall seems very promising for our use case.

3.1.2 OWASP ZAP

The OWASP Zed Attack Proxy Project[7] is web application penetration testing tool. It is mainly intended for use through the provided GUI, but also features a CLI that can be used to either run the process as a daemon or to run simple one-off scans.

There is a ZAProxy Plugin[8] available for Jenkins, so CI integration for that case is simple. A running ZAP installation is needed by this plugin, though, so some intial setup effort is required.

On other CI systems either the CLI or a third party CLI wrapper called zapr[9] could be used in CI, although currently neither supports all the features available via the ZAP GUI.

Triggering simple scans in CI using zapr or the CLI is relatively easy and the results are available as JSON (from zapr), XML (from the CLI) or as a structured plaintext summary. Zapr does require a running and properly configured ZAP installation to be available to do the actual scanning, though, so some setup effort is needed. In contrast, the CLI can be used to run scans without a separately running ZAP instance.

For more complex scanning the ZAP API can be used programmatically, and results are also available through it in XML and JSON formats. The downside is that only the core features are available through the API. ZAP is actively developed, however, and future versions are intended to steadily increase the functionality available via the API.

[7] `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`
[8] `https://wiki.jenkins-ci.org/display/JENKINS/ZAProxy+Plugin`
[9] `https://github.com/garethr/zapr`

ZAP does not appear to support any notion of successive scans, and as such does not provide a way for effective management of the results of scanning in CI.

3.1.3 Nikto

Nikto[10] is a popular open source web server assessment tool. It is aimed at finding vulnerabilities in web servers and associated infrastructure, not in actual web applications served by the web servers.

Since Nikto is a Perl script, no initial setup beyond having Perl installed is required. It has a mechanism for automatically updating itself.

Running basic scans via CLI is easy, and scans can be customized to perform faster scans doing only the desired tests.

The results can be written in multiple different formats including XML, and as such should be possible to integrate into a CI system.

No mechanism for tracking issues across scans or managing findings exists in Nikto, so a separate solution for those functionalities would be required for effective use in CI.

3.1.4 w3af

W3af[11] stands for "Web Application Attack and Audit Framework". It is an open source project, and intended to be a platform for web application vulnerability assessment and penetration testing. It includes functionality for crawling a web application for attack vectors, auditing the found attack vectors for vulnerabilities and for exploiting the found vulnerabilities.

The w3af documentation includes good instructions for getting started, and the installation is straightforward.

Scans can be run using either the provided GUI or CLI. The CLI commands can be automated using the scripting functionality of the tool. W3af also exposes a REST API using which the scanning can be automated using any programming language of choice. Multiple output formats, including XML, are available via plugins. As such, integration into a CI system is doable with some effort.

W3af also has a feature for self-updating and supports crawling and auditing JavaScript, Flash and Java applet targets.

[10] `http://www.cirt.net/Nikto2`
[11] `http://w3af.org`

3.2 Static vulnerability analysis

Static vulnerability analysers search for signs of Common Vulnerabilities and Exposures (CVEs)[12] or other dangerous code patterns. Since new detection sets are added via tool updates, a scheme for keeping these tools up to date should also be put in place when adopting them for running in a CI system.

A potential disadvantage with these tools is that they are often programming language specific, although commercial ones often support multiple languages. Relatively good analysers exist for older and established languages like C or Java, but more recent languages or ones that have only recently begun to grow in usage and importance, such as JavaScript or Go, may lack these tools completely.

The input of a static analysis tool can be either source code, bytecode (for languages that compile into bytecode) or machine code.

This means that a static analysis tool that analyses a language's bytecode can also analyse the code of newer languages if they compile to the same bytecode format. So for example, newer JVM-based languages can be analysed using Java bytecode static analysis tools.

3rd party libraries could also conceivably be run through static analysis if extra care was desired in evaluating the security of an application's dependencies.

Despite dynamically typed languages being more challenging to analyse than statically typed ones, the methods of static analysis have fortunately progressed far enough so that static analysis tools are currently available for both types of languages.

Even static analysis that is not strictly security-focused is usually well worth performing from the security point of view. This is because security issues often arise as a side effect from bugs or bad coding practices, against which static analysis can be a valuable tool. [10]

A potential issue with using static analysis tools is that they can produce considerable amounts of false positives. The proportion of false positives to actual valid findings varies by tool and case, but can be at least as high as 50 false positives for each true positive. Because of this, an efficient method of managing false positives is vital for the usefulness of these tools. [31]

Analysis of the characteristics laid out at the start of this chapter:

Workflow integration Easy to trigger from CI and results can be represented and reviewed through existing static analysis plugins or features of CI systems. Largely usable in developer IDEs to detect issues even before CI.

[12] `https://cve.mitre.org/about/index.html`

Initial work Most tools are trivial to adopt into use. SonarQube requires infrastructure for its server application. Some initial configuration of desired detection rulesets can be required. Reviewing the findings of the first analysis run can take a considerable amount of effort if tool is adopted mid-project.

Continuous work Some continuous work is required for reviewing findings and managing false positives. The tools must also be kept up to date, which may require a small amount of periodic work if the tool is not capable of self-updating.

Security expertise Can be used to a large extent with little to no security expertise, as the analysis results produced generally contain good descriptions of the vulnerabilities the warnings are about. Security expertise would make results review considerably more efficient, though.

3.2.1 FindSecurityBugs

FindSecurityBugs[13] is an open source FindBugs[14] extension with a focus on security issues. It mainly detects potential vulnerabilities in Java code, but since version 1.2.0 released in October 2013 it has also included detection sets for Groovy and Scala.

This seems to be the favoured open source tool for static security analysis in the Java community. It is actively developed.

Integration to development environment should be extremely easy for projects built with Maven, since it is only available as a plugin for the FindBugs maven plugin. The downside is that for non-Maven projects it is considerably harder. Integration into CI environment should likewise be easy since the tool's results are exportable into the xUnit format, which is universally understood by CI systems. Additionally for Jenkins, the Jenkins FindBugs plugin[15] can be used for presenting the results.

An interesting detail about FindSecurityBugs is that it analyses JVM bytecode instead of Java source code, and could thus be used for analysing any JVM-based language.

In addition to CI, FindSecurityBugs also integrates easily into popular Java IDEs including Eclipse and IntelliJ IDEA, so it could be used to highlight possible security issues while coding. This should reduce the amount of issues that are caught and must be dealt with later, in the CI stage.

[13] `http://h3xstream.github.io/find-sec-bugs/`
[14] `http://findbugs.sourceforge.net/`
[15] `https://wiki.jenkins-ci.org/display/JENKINS/FindBugs+Plugin`

3.2.2 SonarQube

SonarQube[16] is a static code analysis platform with support for numerous programming languages. It involves a server application into which client applications can send analysis results. The server application provides a web application interface for reviewing results and configuring the system. As such, some resources and initial effort is required for setting up the server application as well as database into which the results should be stored and, if desired, customizing the analysis rules.

Integration into CI should be simple, as most CI systems have dedicated client plugins for SonarQube. SonarQube also supports analyzing FindSecurityBugs results, so it could be used for better visualization and analysis of them.

There was a Security Rules Plugin[17] available for older versions of SonarQube, but latest version of that was released in January 2012 and deprecated in SonarQube version 4.2. Fortunately, they have started adding security-related rules to the normal language detection sets in 2015, so recent versions can do some static security analysis out of the box.

For Java, JavaScript and PHP, the SonarLint[18] tool provides basic analysis without customizable rules via CLI or IDE integration.

Reviewing and managing findings in SonarQube is easy, and it can calculate trends and track issues across multiple analyses.

Good descriptions of the found vulnerability types are provided when reviewing analysis results, in many cases with examples and instructions on how to fix the issue, so the tool is usable to large extent without much security expertise.

3.2.3 Brakeman

Brakeman[19] is a static code security scanner for Ruby, focused primarily on Ruby on Rails. It is open source and available as a Ruby gem, and thus running it in development or continuous integration environments is easy.

This appears to be the preferred open source tool for Ruby on Rails static security analysis and at the time of writing is actively developed.

In addition to static code analysis, it does dependency verification at least for public vulnerabilities in Ruby on Rails.

[16] `http://www.sonarqube.org/`
[17] `http://docs.sonarqube.org/display/PLUG/Security+Rules+Plugin`
[18] `http://www.sonarlint.org/`
[19] `http://brakemanscanner.org/`

Results can be output in multiple human- or machine-readable formats, but not in any commonly used test result format. Thus, integration into any CI feedback loop should not prove too difficult, but will likely require a custom results transformer. There is also a dedicated plugin[20] available for Jenkins that already does this, so if using Jenkins for CI the integration should be extremely easy.

The tool features an easy-to-use interactive false positive management mode, through which a configuration file for ignoring false positives can be created, loaded or modified.

3.2.4 phpcs-security-audit

phpcs-security-audit[21] is a security focused new ruleset to the PHP_CodeSniffer[22] static analysis tool.

Initial setup is easy. PHP_Codesniffer is available through PEAR, the PHP Extension and Application Repository, and phpcs-security-audit is installed by linking or copying the ruleset files into the PHP_Codesniffer installation directory.

It has general detection rules for possible security issues in core PHP code and specific sets for Drupals 7-8 and Symfony 2.

Running the tool is simple via a CLI and results are available in multiple machine-readable formats, so integration into CI systems should not require much effort.

3.2.5 eslint-config-scanjs

Adapted from ScanJS[23], which was a static security analysis tool for JavaScript, eslint-config-scanjs[24] is a security focused ruleset for ESLint[25].

Setup is easy and involves installing ESLint, which is available via NPM, the Node Package Manager, as well as some specific dependencies for eslint-config-scanjs, which are also available via NPM. After that, the scanjs rules are specified as a command line parameter for ESLint.

CI integration is easy, since ESLint can generate reports in a general static analysis XML format. The XML report can then be presented in the CI

[20] `https://wiki.jenkins-ci.org/display/JENKINS/Brakeman+Plugin`
[21] `https://github.com/Pheromone/phpcs-security-audit`
[22] `https://github.com/squizlabs/PHP_CodeSniffer`
[23] `https://github.com/mozilla/scanjs`
[24] `https://github.com/mozfreddyb/eslint-config-scanjs`
[25] `http://eslint.org/`

system using any static analysis presentation plugin, such as the Checkstyle plugin for Jenkins.

IDE integration is also possible, at least for IntelliJ IDEs.

3.2.6 JSPrime

JSPrime[26] is another static security analysis tool for JavaScript.

Unfortunately, it only offers an analysis interface via the web browser, and the results are represented as an html page. On cursory examination, the implementation of the tool is heavily tied into the model of input and output through html pages. Because of this, logic extraction or refactoring to enable reasonable CI integration would require a non-trivial amount of work.

A potentially more viable workaround would be to utilize Selenium or some such tool to pass input and fetch output from the html page interfaces of the tool in a CI system.

Overall, JSPrime is not a suitable candidate for continuous testing.

3.3 Configuration checking

Security issues can also result from the misconfiguration of systems or applications. For instance, production servers of an application most certainly should not allow remote root login, nor should production databases allow connections from anywhere but the application servers.

The OWASP Top 10 Web Application Security Risks 2013[26] recognize security misconfiguration as the 5th most significant risk. Though generic web vulnerability scanners can be used to detect some security issues resulting from application-level misconfiguration, configuration checking is a more effective detection method [12].

Configuration issues have traditionally been the domain of sysadmins and sysops, and thus far removed from the actual software development process. However, with the rise of DevOps and the Infrastructure as Code paradigm we are able to include the code for automating the provisioning and configuration of infrastructure for a developed application into the same development process and CI feedback loop as the actual application we are developing.

We are able to use infrastructure and deployment automation tools such

[26] `https://github.com/dpnishant/jsprime`

as Chef[27], Ansible[28] and Capistrano[29] to develop and automatically deploy the whole technology stack of our application. With that capability, we can bring up new instances from scratch for each change, and run configuration checking tools on the end result to verify that our web servers, firewalls etc are securely configured.

Since the tools for configuration checking are non-invasive and require little processing power and resources, they can also be run on production machines. This is especially useful if it is not feasible to provision new machines on which to run configuration checking tools in the CI process.

In addition to checking the configurations of individual applications, it is also possible to check and enforce security policies. Security policy compliance checking could be a useful continuous security testing method in cloud environments where virtual machines are created and destroyed continuously. In such a scenario, we could conceivably check at regular intervals that each running machine conforms to our defined security policy to detect intrusions. It would also be relatively simple to also enforce the security policy by shutting down every non-compliant machine when detected.

Analysis of the characteristics laid out at the start of this chapter:

Workflow integration Easy to trigger from CI, but no ready solutions for representing results in CI.

Initial work No effort required for setting up since the tools are just scripts or standalone binaries. Some effort is needed to implement custom results parsing into some format that a CI system can present and track.

Continuous work Very little continuous work, as the results can mostly be interpreted into pass/fail cases. The tools should be kept up to date, but that also requires very little work.

Security expertise Some security expertise can be required to accurately determine remedial actions for some warnings. Many of the checks are straightforward and fixable even without security expertise, though.

3.3.1 Lynis

Lynis[30] is an open source auditing and hardening tool for Unix/Linux based systems. It provides security-related feedback on system components and

[27] `https://www.getchef.com/chef/`
[28] `http://www.ansible.com/`
[29] `http://capistranorb.com/`
[30] `http://cisofy.com/lynis/`

configuration, such as user accounts, groups, shells and such.

There is also a commercial enterprise version available, with added automation, reporting and hardening information features.

Running is easy via CLI, and results are produced in a machine-readable structured text format. Integration into CI feedback loop should pose no problem, but will require implementing a parser for automated processing of the results.

3.3.2 MySQLTuner

MySQLTuner[31] is a configuration checking script for improving MySQL performance and stability. It also has a section on security, although that only contains a check that no passwordless users exist.

Nevertheless, database performance and stability issues can also cascade into security issues. For example, a misconfigured database can allow an attacker to overload itself, opening up a vector for a Denial of Service attack.

This tool is run via CLI and provides output in a structured text format that is machine-readable, so integration into a CI feedback loop should be relatively simple but would require implementing a custom parser.

3.3.3 SSLyze

SSLyze[32] is a tool for identifying SSL misconfigurations in a server by connecting to it.

Running the tool is easy via its CLI and results output is available in machine-readable plaintext and XML. Some of the checks done by the tool produce a pass/fail result that could easily be used in a CI system, but most parts of the results require further analysis to determine if they are security issues.

With some parsing and results processing code it could be used in CI to validate the sanity of SSL configurations and absence of critical vulnerabilities such as heartbleed[33].

3.3.4 unix-privesc-check

unix-privesc-check[34] is a tool that checks for misconfigurations that could lead to escalation of privileges.

[31] `http://mysqltuner.com/`
[32] `https://github.com/iSECPartners/sslyze`
[33] `http://heartbleed.com/`
[34] `http://pentestmonkey.net/tools/audit/unix-privesc-check`

It is a shell script with dependencies to a few system tools, and as such easy to trigger in CI. The output is in structured plaintext, so it can be interpreted with simple parsing rules.

3.3.5 Linux Security Auditing Tool

Linux Security Auditing Tool (LSAT)[35] is another tool for detecting system-level misconfigurations that can result in vulnerabilities.

The tool comprises of 37 small modules, which in total generate 66 small reports on various aspects of the system. The format of the reports unfortunately varies so much that most would require separate result parsing logic for meaningful usage in continuous testing. Additionally, the output of many modules requires case-by-case interpretation and cannot be automatically parsed by the CI system as an issue or not.

While the required effort of writing separate parsing logic for each report format can make this tool unsuitable in many cases, some subset of the reports could also be chosen to utilize the checks deemed most useful with considerably less effort.

3.3.6 Security monkey

Security monkey[36] is a tool that monitors the policies and configurations in an AWS account. It is intended for alerting about insecure configurations.

This is a standalone application and does not currently integrate into any CI system. It also requires quite a bit of setup and configuration effort to get started.

While security monkey only generates warnings, it would be easy to adapt into a tool that enforced the given security rules and policies by immediately shutting down non-conforming instances.

3.4 Security verification

Most security testing tools, like vulnerability scanners and static code analysers, can only act as "badness-ometers" in that their results only tell you where you are on the range from "deep trouble" to "who knows" [17]. Even if they find no issues, you still do not know if your software is secure. Security verification tools aim to provide a way of specifying security requirements for an application and then testing that it adheres to all of them.

[35] `http://usat.sourceforge.net/`

[36] `https://github.com/Netflix/security_monkey`

Ideally, security verification tools could be used to codify the security requirements of an application and verify them continually during development. They could also be used for writing regression tests for found and fixed vulnerabilities to prevent them from quietly re-surfacing.

All of the three tools we found take the behavior-driven development (BDD) approach to security testing.

A drawback of these tools is that they require a relatively large amount of security expertise from the organization, since the verification tests need to be created specifically for each application. A good understanding of both the security requirements of each application and how to test them is needed.

Another thing to consider with tools of this type is that they rely on other, more specialized tools like nmap[37] or sslyze to do the actual testing, and thus require the availability of multiple tools in addition to themselves for meaningful use.

In addition to continuous testing during development, security verification tools can also be used for verifying that production instances of applications adhere to the security requirements codified in their tests.

Analysis of the characteristics laid out at the start of this chapter:

Workflow integration Very easy to trigger from CI, results presentable and trackable across builds using existing CI functionality.

Initial work A little effort is required for installing the framework and desired tools it leverages. Considerable effort can be required to write necessary test cases, but one can also start with just a few and add more continuously.

Continuous work Some continuous work is required for writing more test cases as the software develops. A little work is also required for keeping the framework and leveraged tools up to date. Very little effort required for reviewing test results.

Security expertise Considerable security expertise can be required for writing good test cases. Little to no expertise is required for reviewing the test results.

3.4.1 Gauntlt

Gauntlt[38] is a security testing framework that provides a DSL to write tests utilizing tools like nmap. This means it can be used to write basic security

[37] `http://nmap.org/`
[38] `http://gauntlt.org/`

verification tests, like what ports should or should not be open on a server.

This tool is available as a Ruby gem, so installation and running in development and CI environments should be relatively simple.

Gauntlt uses cucumber[39] and its gherkin[40] syntax for writing and running tests.

Integration into CI feedback loop should also be easy, and the results are available in a machine-readable text format. Cucumber is also able to export results into JUnit format, but doing so via gauntlt is currently (March 2016) broken. It can be fixed with a minor patch to gauntlt code, though, and there is an open issue concerning it. So either patching gauntlt or writing a custom parser is required if automated results parsing is desired on a finer grain than the exit code of the tool.

3.4.2 BDD-security

BDD-security[41] is another framework for testing the security requirements of an application.

The tool is written in Java and uses JBehave[42] as its test framework.

A promising aspect of this tool is that it separates the security requirement definitions from the application-specific logic needed to test them. Some basic scenarios based on common security requirements, such as authentication and session management, are even included in the tool.

This could be an advantage especially in larger organizations, since security experts could conceivably craft common security requirements and test scenario descriptions based on them, and software developers could re-use the same tests in various projects just by implementing the functionality of application-specific key words like "login".

BDD-security provides a convenient CLI for automation, and using plugins the results are exportable to a unit test format so CI integration should be easy.

3.4.3 Mittn

Mittn[43] is a recent addition to the group of security verification frameworks. Like Gauntlt, it provides a way of writing security assurance tests in human-

[39] `http://cukes.info/`
[40] `http://cukes.info/gherkin.html`
[41] `http://www.continuumsecurity.net/bdd-intro.html`
[42] `http://jbehave.org/`
[43] `https://github.com/F-Secure/mittn`

readable language, leveraging existing tools like SSLyze and Radamsa to do the actual testing.

It is written in Python and uses behave[44] as its BDD framework.

It differs from the other two security verification frameworks in that it saves test results into a PostgreSQL database. Storing the results into a database allows for different ways of handling the test results, which can be advantageous. For example, false positive management is done via the database.

Triggering the tool via CLI in CI is easy, and the results are exportable to a unit test format so presenting them in CI should not pose a problem.

3.5 Dependency verification

Since practically all modern applications make use of third party libraries, be they free/open source or proprietary, the risk of security vulnerabilities resulting from included dependencies is a growing concern. This concern seems justified as included dependencies account for some 80% of the code in modern applications. The seriousness of this issue is also evidenced by a study concluding that in 26% of cases, libraries downloaded from Maven central contain publicly disclosed vulnerabilities.[30]

OWASP has also noted this issue in their "The Top 10 Most Critical Web Application Security Risks 2013" listing by including "Using components with Known Vulnerabilities" as a new category in the Top 10. The listing's release notes state regarding it [27]:

> "This issue was mentioned as part of 2010-A6 — Security Misconfiguration, but now has a category of its own as the growth and depth of component based development has significantly increased the risk of using components with known vulnerabilities"

As such, it makes sense to try to verify that the third party libraries included as dependencies contain no known security vulnerabilities. For this purpose, various dependency verification tools have emerged.

Dependency verification tools do not perform any active analysis. Instead, they only check if the tool's vulnerability database contains any known vulnerabilities in the versions of dependencies used by the application under scrutiny. Because of this, the processing overhead involved in using these tools tends to be quite small.

[44] `http://pythonhosted.org/behave/`

While these tools can produce false positives, it does not happen often and requires either misidentification of a library or a false vulnerability entry in the vulnerability database utilized by the tool.

For many of these tools, the vulnerability database is an online one. This needs to be taken into account when planning tool usage in a CI system, as some environments might not allow traffic from the CI server to public internet. An upside to this is that these tools often automatically update their local copy of the vulnerability database, or at least make automating the update extremely easy.

Some of these tools can also be run in production environments, and as such could be used to signal a higher priority alert if a vulnerable dependency is detected in production.

These tools share a disadvantage with static security analysers: both are language-specific, and as such, dependency verification tools mainly exist for established programming languages with widely used dependency management systems.

Another note-worthy aspect of these tools is that their usefulness is heavily dependent on the vulnerability database they use. Tools using more actively maintained vulnerability databases are of course more useful. The vulnerability databases generally contain a mapping from specific library versions to publicly disclosed CVE entries.

In addition to checking dependencies for publicly disclosed vulnerabilities, there are tools for adding checksum verification to dependency management systems missing such a feature. This offers a way to prevent MitM attacks.

It is also possible to check if the installed operating system level packages contain known vulnerabilities or have pending security updates to install with tools such as debsecan[45].

Analysis of the characteristics laid out at the start of this chapter:

Workflow integration Easy to trigger from CI. Output of tools can be interpreted as a "pass/fail" or "number of warnings" result in CI with a little effort.

Initial work Tools are trivial to adopt into use. A little initial effort is required for meaningful results presentation in a CI system.

Continuous work Basically no effort required for reviewing results. If tool or vulnerability database updates are required, they are trivially automatable.

Security expertise No security expertise required.

[45] `http://www.enyo.de/fw/software/debsecan/`

3.5.1 Bundler-audit

Bundler-audit[46] is a Ruby gem that does patch-level verification for gems declared as dependencies.

Running it in development or CI environments is easy, and while the output is provided only as text, it is structured in such a way that integration into CI feedback loop should not prove too much work.

It uses the ruby-advisory-db[47] project as its vulnerability database and provides an easy mechanism for updating the local copy, so keeping the vulnerability detection set up to date is extremely easy to automate.

Ignoring some advisories is possible by passing arguments to the tool invocation on CLI.

Triggering the tool via CLI in CI is easy, and the results are provided in a machine-readable text format so only a bit of parsing is required for rudimentary integration of the results into the CI feedback loop.

3.5.2 HolePicker

Like bundler-audit, HolePicker[48] checks the Ruby gems declared as dependencies of an application for known vulnerabilities.

Although the development road map of the tool includes switching to use ruby-advisory-db for vulnerability data, the latest version uses a self-maintained database. Based on a cursory examination, the database seems extremely limited in comparison to ruby-advisory-db, so the usefulness of this tool will be enhanced once the switch is implemented.

Like with Bundler-audit, running is easy in both development and CI. Output is likewise provided as structured text, and can thus be integrated into CI feedback loop with a small effort.

One interesting feature of this tool is the ability to scan dependencies of all applications served by a given web server instance by pointing the tool at apache/nginx configs. HolePicker can also easily integrate into the application deployment tool Capistrano, and so could be used to perform a dependency vulnerability check automatically before each deployment.

Unfortunately, development of the tool was discontinued in fall 2014. The online database has been disabled since then, and a new maintainer has not been found. As such, the tool cannot be used.

[46] `https://github.com/rubysec/bundler-audit`
[47] `https://github.com/rubysec/ruby-advisory-db`
[48] `https://github.com/jsuder/holepicker`

3.5.3 Dawnscanner

Dawnscanner[49] is a static web application code security scanner for Ruby. It is targeted specifically for use with the popular frameworks Ruby on Rails, Sinatra and Padrino. It checks the framework, Ruby version and declared dependencies for vulnerabilities against it's own vulnerability database, but also contains some static code analysis functionality for finding SQL injection and XSS vulnerabilities in code.

It is available as a Ruby gem with a convenient CLI so running should be easy both in CI and on developers' machines. Results are output in a structured plaintext format, so it should be relatively easy to parse for processing in CI.

It is actively developed at the time of writing in March 2016, with the focus of future development in improving its static code analysis features.

3.5.4 OWASP Dependency Check

OWASP Dependency Check[50] checks a Java or .NET application's dependencies for publicly known vulnerabilities.

Integration into a any CI system should be easy, since a CLI exists, as well as separate plugins for the popular Java build tools Maven and Ant. An actively maintained Jenkins-plugin is also available.

It has a mechanism for managing false positives by specifying the warnings to ignore in an XML file, and the XML block to suppress a given warning can be automatically generated from the tool's report page. Thus, not much continuous overhead should result from management of false positives.

It relies on National Vulnerability Database (NVD) feeds provided by the National Institute of Standards and Technology (NIST) for vulnerability data. The tool automatically keeps itself up-to-date using the feeds. The documentation states that the initial complete download of vulnerability data can take more than 15 minutes, which can be too long for continuous testing in some cases. However, if the tool is run at least once every 7 days, it will only require downloading a small update file.

[49] `https://github.com/thesp0nge/dawnscanner`
[50] `https://www.owasp.org/index.php/OWASP_Dependency_Check`

3.5.5 pliers-npm-security-check

Pliers-npm-security-check[51] is a tool for checking dependency npm packages for publicly disclosed vulnerabilities for the pliers[52] JavaScript build tool.

It uses the nodesecurity.io validation REST api to do the actual vulnerability checking.

Installation is easy, as is triggering the tool in CI via either the command line or a pliers task. The results are output in machine-readable plaintext, so presenting them in CI requires some simple parsing logic to be implemented.

3.5.6 nsp

Nsp[53] is the nodesecurity project's[54] tool for checking npm dependencies against their vulnerability database.

Installation via npm is simple for both CI and developer machine environments, and the tool is run via simple CLI. Results are ouput in a machine-readable plaintext format, so rudimentary CI integration of results is easy with a bit of parsing.

Nsp can also be easily triggered via the popular JavaScript task runner Grunt[55] by using a grunt plugin available via npm.

3.5.7 SensioLabs Security Advisories Checker

SensioLabs Security Advisories Checker[56] checks for known vulnerabilities in the dependencies of PHP projects that use Composer[57] for dependency management.

It utilizes an online security advisory database maintained by SensioLabs, so no user actions are required to keep advisories up-to-date.

It is usable via CLI, so triggering in a CI system is easy. Results are output in either JSON or machine-readable plaintext, so integrating them back to the CI feedback loop can be achieved with a bit of parsing logic.

[51] `https://www.npmjs.org/package/pliers-npm-security-check`
[52] `https://github.com/pliersjs/pliers`
[53] `https://github.com/nodesecurity/nsp`
[54] `http://nodesecurity.io`
[55] `http://gruntjs.com/`
[56] `https://github.com/sensiolabs/security-checker`
[57] `https://getcomposer.org/`

3.5.8 Versions Maven Plugin

While not a security testing tool as such, the Versions Maven Plugin[58] can be used to check if there are updates available for any of the dependencies declared for a project that uses maven for dependency management.

Considering that many years old versions of libraries and frameworks that contain publicly disclosed vulnerabilities are still downloaded in considerable amounts[30] by dependency management systems, using a tool such as this sounds prudent.

Running this tool in the CI cycle of a maven project should be extremely easy. Handling the results would require some work, though, as human review and potential dependency update would probably be wanted each time it detects out-of-date dependencies. Some processing of the results might also be wanted, for example to ignore some dependencies when checking for new versions if the specific dependency could not be updated for some reason.

A potential limitation of this tool is that running the checks for a project with many dependencies can take a while because the tool checks for available versions for each dependency from each configured maven repository one at a time.

The tool also provides a way to ignore available updates in selected dependencies, or restrict the considered version range, which can be useful if upgrade into a new major version, for example, is not feasible. The mechanism works by specifying rules for handling versions in an XML file.

The tool's output format is structured plaintext, so some parsing would be required for automated analysis of results. It claims to also have the capability to output the results in a report format of some sort, but at least in version 2.1 this functionality was broken.

3.5.9 Retire.js

Retire.js[59] is a tool for identifying JavaScript libraries with known vulnerabilities. A grunt plugin, grunt-retire[60], also exists, allowing for easier automation in projects that use grunt.

No method of managing false positives is provided, but it is possible to ignore some paths completely, which might in some cases be usable for the same purpose.

The tool's output format is structured plaintext, so some parsing is required for automated analysis and presentation of results.

[58] `http://mojo.codehaus.org/versions-maven-plugin`
[59] `https://github.com/bekk/retire.js`
[60] `https://www.npmjs.org/package/grunt-retire`

Retire.js is also available as a chrome or firefox extension, and could be used to check frontends for use of vulnerable libraries even without access to the development process. Such checking could conceivably be automated using Selenium or some other browser automation tool. This could prove useful for gathering overall vulnerability status for example when developing only a part of a larger site made up of interconnected web applications developed by different companies.

3.5.10 Gradle witness

Maven Central[61], the main distribution channel of dependency libraries for JVM languages, does not by default provide download access to the dependencies over SSL. This opens the clients using dependency managers such as maven, gradle[62], SBT[63] or leiningen[64] up for Man-in-the-Middle (MitM) attacks.[29]

Gradle witness[65] is tool for verifying that the checksums of downloaded dependencies match those expected, thus preventing MitM backdooring of them.

Using this tool requires adding the checksum of a dependency library into the dependency manifest each time a new dependency is added. The tool includes a feature for easily generating and adding the checksums of all currently declared libraries, so it should be easy to adopt if one is willing to trust the authenticity of their currently downloaded dependencies.

The tool integrates itself into the gradle dependency manager, so after installing it will automatically run every time dependencies are fetched. The installation process itself is not as simple as adding a dependency, but still relatively simple and should not pose a problem. Presumably, the tool will cause dependency resolving to fail if checksums do not match. For meaningful use and presentation in CI, some logic would be needed to differentiate failures caused by this tool from other failures and highlight them as such.

3.6 Summary

At the start of this chapter, we recognized the following high-level characteristics of security testing methods and tools to affect the suitability for

[61] `http://central.sonatype.org/`
[62] `http://www.gradle.org/`
[63] `http://www.scala-sbt.org/`
[64] `http://leiningen.org/`
[65] `https://github.com/whispersystems/gradle-witness`

continuous testing:

- Ease of integration into the development workflow
- Initial work required for meaningful results
- Continuous work required alongside software development
- Security expertise required for meaningful use of the tool

To determine more specifically which aspects of a security testing method or tool most affect the above characteristics, we reviewed publications on the selected security testing methods and the documentation of 29 security testing tools representative of the methods. A simplified overview of each covered method's suitability for continuous security testing, based on the analysis, is presented in table 3.6. We found the following more specific factors to most affect the suitability of a security testing method or tool for continuous testing:

- The infrastructure required by the tool affects the amount of initial work required. Vulnerability scanners can require multiple nodes for scanning and corresponding capacity for the system under test to be able to handle all the scanning traffic, whereas dependency verification tools require very little resources.

- The time it takes to run the tool has an effect on development workflow integration. If it takes hours for a single run, the tool is probably unsuitable for running after each build in a CI system to get immediate feedback. In such a case, nightly runs may be a practical alternative.

- Human effort required in reading reports and verifying findings largely affects the continuous work required. Especially for small development teams it is often not feasible to spend much time reviewing reports.

- Tool maintenance and setup costs in effort and money affect both the initial and continuous work required. Larger organizations may be able to spread these costs over multiple projects, while smaller organizations may not be able to spend much effort or money on the tool.

- Available tools and processes for managing false positives and duplicate reports can greatly affect the continuous work required. Handling false positives and duplicate reports can cause considerable overhead in even non-continuous security testing processes, and continuousness of the testing can only make this problem worse. As such, effective tools and processes for handling the false positives and duplicate reports are essential.

	Workflow integration effort	Initial work	Continuous work	Security expertise required
Vulnerability scanning	difficult	high	medium	medium
Static vulnerability analysis	very easy	low	medium	medium
Configuration checking	easy	low	low	low
Security verification	very easy	high	medium	high
Dependency verification	easy	very low	very low	low

Table 3.1: Analysis of security testing method characteristics that determine suitability for continuous security testing

- Required security expertise is largely dependent on testing method. The results of most of the covered tools can be reviewed to a large extent with little or no security expertise. Considerable expertise can be required for initial testing configuration or setup beyond installing the tool, however. For example, utilizing vulnerability scanners with customized scan profiles or security verification tools with custom-written tests specifically targeting the application at hand can require a considerable amount of security expertise. On the other hand, dependency verification tools do not require any security expertise to use.

Chapter 4

Security testing methods in continuous integration

In this chapter, we present methods of and challenges in adopting security testing into CI systems and development workflows. We first go through some general guidelines and issues related to work management, tool adoption, testing and results handling that apply for adopting any of the reviewed security testing methods into a CI process, and then review the case for each method separately.

4.1 General guidelines

In "Secure Programming with Static Analysis", Chess and West[10] state that three questions need to be answered for a software development organization to adopt a security testing tool successfully. Though the book only deals with static analysis, the questions are general enough to apply to all security testing tools. The three questions are:

- Who runs the tool?
- When is the tool run?
- What happens to the results?

In the case of continuous security testing, the answers to the first two questions are largely fixed.

The "who" is the CI system, and by extension the developers. For even faster feedback cycles, in cases where it is feasible, developers should also be able to run the tools in their development environments. Chess and West[10] propose having developers run the tools in a mode that only produces high-confidence results and leaving more thorough reviews to be done by security

experts. This is a great option for cases where security expert support is available, and may be a sensible choice in many cases even if such support is not available.

The "when" depends somewhat on the security testing methods and tools selected. For truly continuous security testing the acceptable range would be from "continually in developers' IDEs", through "once for each commit in the CI system", to "nightly test runs in the CI system" for the most time-consuming tests, the preference being for the former end of the spectrum.

For the last question, no single answer exists. How the results of continuous security testing can best be processed depends on the case, with the selected security testing methods and tools affecting the feasibility of different results processing options.

4.1.1 Work management

All work involved must be made visible by going through the normal work management process. This goes for initial planning and setup required for tool adoption, as well as for any continuous work required for results processing and fixing found issues. The aim is to ensure that all stakeholders are aware that security requires real effort, and thus make sure that adequate resources are allocated to have a reasonable chance of success.

Adopting certain tools involves initial costs in the form of infrastructure setup. It is a good idea to try to spread these costs across projects/teams. In many cases, the same tool installation can be used for multiple projects or teams. In cases where sharing a tool instance is not feasible, the setup can be automated using modern infrastructure automation and configuration management tools to avoid duplicate work.

4.1.2 Tool adoption

It is important to select security testing methods and tools that are suited to the case at hand. Many security testing tools are good at catching certain types of vulnerabilities, and some tools are even designed for detecting vulnerabilities in a specific framework or application. Selecting the most suitable tool will lead to more valuable results. Higher perceived value of the security testing in turn increases the chances that the team will take ownership of it.

It is a good idea to start small, as Chess and West recommend[10]. Especially when adopting a security testing tool mid-project the first results can be overwhelming and lead to developer apathy towards security testing if not managed properly. A good adoption process only enables a few, easily

understood vulnerability checks at the beginning, and increases the amount of tests and tools gradually at a pace that's comfortable for the developers.

Security and tool training for developers is a good idea. Both general security expertise and knowledge on the specific security testing tools improve developer efficiency when reviewing testing results. The same goes for accuracy when deciding what actions the results necessitate. Additionally, by improving the value gained by using the tools, continuous security testing adoption success chances are improved.

4.1.3 Tool maintenance

A strategy for keeping the security testing tool up to date is needed. Static analysis tools for instance often add new detection sets in tool updates. In many cases, this can and should be automated.

Monitoring the maintenance state of selected tools to notice if their development or maintenance stagnates is also important, especially for utilized open source tools. Continuing to use an abandoned tool can lead to a false sense of security, so using one should always be a conscious decision.

4.1.4 Test targeting

CI systems are normally used to run tests for development and release candidate versions of the developed software. However, security tests are different from other kinds of testing in that new vulnerabilities can be found after security testing tool or database updates. Therefore, to prevent security issues that have already been fixed in development or release candidate versions from secretly persisting in production, the current production version should also be regularly security tested if deployments are not done daily.

It is not enough to run security tests for each commit when developing the software. Vulnerabilities in the technologies and third party libraries can surface years after the application has entered maintenance mode. Security testing tool updates and periodic security tests should continue also after active development has ended, for the whole lifecycle of the application.

4.1.5 Results handling

A system for efficiently managing the results of tools in continuous use is essential. The two major issues that must be solved are ignoring false positives, and tracking findings across test runs. Most of the currently available tools are intended for manual one-off use, and so they rarely include any mechanism for keeping track of and managing findings across multiple runs.

If security experts are available, a good process to at least start with for handling the results of security tests would be to utilize a security expert - developer pair for both verifying and fixing found issues. In addition to ensuring the accuracy of issue verification and fixes, it would also help disseminate security expertise among the developers.

In most cases, it would be possible to integrate the handling of the security testing results into the normal development workflow by automatically creating a new issue in an issue tracker for each finding by the CI system. The process of verifying and handling the issues would then proceed from there like any other work. Involving an issue tracking system can cause considerable overhead, though, according to a study by Baca et al.[5], and can severely lengthen the processing time for findings.

Another, lighter-weight approach to development workflow integration is to utilize pull request reviews for processing the security testing results. This approach is suitable to most modern development processes, since development through feature branches and pull requests is the de-facto standard practice. The security testing results can be posted either as comments to the pull request or as separate test case statuses by the CI system. This approach is mainly useful for preventing regressions by limiting the analysed results to the set of differences between the feature branch and the master branch. The drawback is that some other approach is required for handling possible baseline issues.

The way that the security testing results factor into the build status of the whole application needs to be determined based on the security requirements of application. In some cases, even a potential finding can be grounds to prevent deployment to production, while in others it can be desirable to deploy even with verified vulnerabilities.

4.2 Vulnerability scanning

While continuous vulnerability scanning can be an effective tool for improving the security of software during development, the technical and process integration of the scanners into a project's CI loop is not trivial.

Vulnerability scanners require infrastructure to be set up for both the scanner and the target to be scanned. The effort involved can make this approach to continuous security testing unfeasible, especially in cases where infrastructure and application deployment automation is not yet utilized. On the other hand, if the project adheres to the Infrastructure as Code principle and has an automated mechanism for deploying the application, the setup of the target application should be almost effortless. Additionally

the infrastructure setup of the scanner can be automated, and as such, the work effort costs associated with it can be spread across many teams and/or projects.

In most cases, the actual application's infrastructure setup automation is more important than that of the scanner. This is because there should not be a need to redeploy the scanner after initial setup. In contrast, the test instance of the developed application will need constant updating as the application and associated infrastructure are developed further. Even complete redeploys from scratch may be needed, for example if an aggressive scan breaks the application irreversibly.

For the most thorough results, it is important to keep the scanner updated to the latest available version, as new detection capabilities and improvements to existing ones are often added via tool updates.

While vulnerability scanning in CI is the most feasible approach to continuousness, using these tools in development environments is possible especially if the project adheres to the IaC principle and virtualization can be leveraged. However, since vulnerability scanning can require a considerable amount of computing resources for both the scanner and the scanned application, it might be prohibitively slow to do on developers' machines.

The computing resource requirements of vulnerability scanning are also an issue from the continuous security testing perspective, as scans can easily take hours of time. Factors that affect the time taken are the attack surface of the target application, whether or not the scan profile has been crafted to fit the target application, and the infrastructure and resources given to both the scanner and the target instance of the application. While it may be possible to get the scanning times short enough to enable true continuous testing with enough effort and resources, it is probably not feasible in most cases. A practical alternative is to run vulnerability scans at reasonably frequent intervals, for example nightly.

An opportunity for automatically creating comprehensive attack vector maps specifically targeted at the application under scan is presented by modern web application testing automation tools such as Selenium[1]. Some scanners feature an active scanning mode as an alternative to using a crawler to map attack vectors. While the active scanning mode is intended to be used with a user manually browsing the target web application, it could also be combined with existing web application tests. This way a customized attack vector map for the application could be effortlessly created without time-consuming and potentially error-prone blind crawling. As web application tests are a common practice in modern software development, this approach

[1] `http://www.seleniumhq.org/`

should be applicable in most cases.

While vulnerability scanners can be operated without much security expertise by simply using a default comprehensive scan profile that runs tests against all possible vulnerabilities that the scanner can detect, this is in most cases very inefficient and time-consuming. To better utilize the scanner, some security expertise is required to create a scan profile that only tests for the vulnerability types that are relevant to the target application.

Once the technical integration work to get the vulnerability scanner up and running in CI is done, a process for dealing with the results is needed. Since vulnerability scanning results always require manual verification, some continuous effort for verification should be reserved. Security expertise might also be needed in many cases to correctly and efficiently verify the vulnerabilities reported.

The high-level variables for a results processing strategy are:

- Continuousness of results verification
- Continuousness of fixing verified issues
- Who does the verification
- Who does the fixing

To get the greatest benefit out of continuous security testing, the preference would be to do both results verification and fixing of verified issues continuously, immediately following each scan. Additionally, since security experts are not necessarily always available, it would be best if developers could do the verification and fixes. However, that is probably not a realistic model for most cases, especially considering that developers are not very good at detecting false positives from among security issues[6].

A more widely applicable approach would be to split the security testing, including results handling, into two complementary parts. First, a minimal scan profile for detecting only critical high-confidence issues could be run for each commit in CI, if the scan duration can be gotten short enough. The results of these scans should then be verifiable by the developers immediately without security expert support. For the verified findings, fixes should be implemented immediately, as the issues should all be critical.

The second part of the process would be to have a less frequent, for example nightly, more comprehensive scan. The results of this scan would be verified preferably with security expert support, or even just by security experts. A reasonably continuous verification frequency could range from daily to weekly. The verified issues would then be prioritized and fixed through the normal work management process. Developers would implement the fixes, preferably with security expert support on more obscure issues.

One major issue for the continuous use of vulnerability scanning tools is that most of them lack a method for reviewing and managing findings across successive scans. Since the findings cannot reasonably be mapped to a unit test like pass/fail format, the core CI functionality cannot be used for cross-scan issue tracking either. Of the vulnerability scanners investigated in chapter 3, only Arachni featured a mechanism tracking findings across scans. This suggests that most vulnerability scanners need an external solution.

4.3 Static vulnerability analysis

The technical integration of static vulnerability analysis into CI is easy and straightforward, as the existing good tooling and best-practices of general-purpose static analysis in CI can be used as-is. For an even shorter feedback loop during development, static vulnerability analysis can also easily be integrated into development environments using existing IDE plugins or by implementing a version control pre-commit hook. A third possibility for development processes that utilize code reviews via pull requests is to have the CI system run the tools for PRs and post the results to the PR as comments. For example, pronto[2] or SonarQube via its Github plugin[3] can be used for this.

A challenge for both successful tool adoption and continuous use is that static analysis can produce considerable amounts of false positives[31]. A high false positive production rate undermines the credibility and perceived usefulness of the testing. It can even cause developers to stop taking the warnings seriously and lead to abandoning the tool entirely. A further risk is introduced by the fact that developers are bad at identifying false positives from among findings[6]. If a false positive is mistaken for a valid finding, developers can even introduce a real vulnerability while trying to fix the non-existent one[5].

It is therefore essential to select a tool that produces few false positives and enables efficient handling of the ones it does produce. Tool configuration and analysis profile customization can further be used to lower the false positive production rate. The general guideline of starting small is an especially good idea in the case of static analysis, as adoption can be considerably easier if only some easy-to-understand high-confidence detection sets are enabled in the beginning.

Static analysis can also be difficult to successfully adopt in the middle of a project, since in that case, the initial run can produce a considerable amount

[2] `https://github.com/mmozuras/pronto`

[3] `http://docs.sonarqube.org/display/PLUG/GitHub+Plugin`

of findings. This can feel overwhelming and lead to apathy on part of the developers. In contrast, if adopted at the start of a project, the findings will trickle in small amounts and are much easier to manage. This means that when adopting static analysis at the start of a project, it is easier to commit to immediately verifying all findings and fixing those that are valid. When adopting in the middle of a project, a strategy for successfully dealing with the initial results is required.

One such strategy is to treat the initial findings as a baseline, and only commit to continuously processing new findings. The baseline findings can then be handled in several ways. One is to whittle them down a bit at a time whenever individual developers have some slack time. Another is to take on greater batches in concentrated efforts by the whole team or a part of it. Alternatively, a team member or external security expert can be dedicated to the task of going through all the baseline findings.

Another possible strategy is be to consider the total findings of both the baseline and new findings, but just track the delta between analysis runs and commit to steadily keeping a downwards trend.

Aside from the special case of how to handle overwhelming amounts of initial findings from a mid-project initial analysis, the general results processing strategies for static vulnerability analysis are largely the same the ones presented in section 4.2 for vulnerability scanning. Both methods produce findings that may be false positives and thus require manual verification, and that are characterized by a severity/confidence level. The main differences between the two methods are that static analysis is much faster to perform, and that effective mechanisms for false positive management and tracking findings across test runs exist for static analysis tools.

The speed of static analysis means that, in contrast to vulnerability scanning, a comprehensive analysis can be performed for each commit by the CI system. Nevertheless, the most reasonable results processing strategy is basically the same as the one outlined for vulnerability scanning. As the first part, have developers continuously verify and fix easy-to-understand high-confidence findings. Secondly, perform more comprehensive scans in CI less continuously, with their results verified by security experts with or without developers. Developers can fix the straightforward issues themselves, but for the more obscure ones a security expert should support the fixing developer.

4.4 Configuration checking

As configuration checking tools are extremely simple in their application, triggering one from a CI system is easy. In most cases, no installation is

required, just making the binary executable of the tool available on the CI server and pointing it toward your configuration file is enough.

In order to have the configuration files in CI, they must be programmatically generated. Therefore, the project needs to follow the IaC paradigm.

Configuration checking is also doable in development environments by utilizing virtual machines into which the environment is provisioned before doing the actual checks. For this purpose, light-weight virtualization options such as lxc[4] or Docker[5] would probably be most useful.

Interpreting and presenting the results of the tools in the CI system requires some work, though, as these tools have been created with manual use in mind so the output formats are mainly structured plaintext. Thus, parsing logic for the output is required for meaningful presentation of results in CI.

The simplest form of parsing and presenting the results would be to reduce the output of a tool into a binary pass/fail. This could be done with little effort for any threshold value of issues found. The amount of issues found should also be relatively easy to parse and represent in CI status and dashboard views.

More intelligent handling of the results, such as tracking the status of individual issues across successive runs, is possible by implementing custom logic into the results parser to uniquely identify the individual issues that the tool checks for, and translating these into some form that the CI system understands. For example, each check could be interpreted and represented as a unit test in the CI system. This way, the core functionality of the CI system could be leveraged to track the status of the check across successive runs of the tool. Such an approach is possible for configuration checking tools as they always check for the same issues and produce a status for each issue on every run.

4.5 Security verification

Security verification frameworks are by design easily usable in CI. They are easy to trigger and produce results in formats natively understood by CI systems. The results require no interpretation or verification, so they are simple to translate into a build status.

Development environment integration is also feasible, but has a couple of prerequisites. First, a functioning test instance of the developed application needs to be accessible. This can be a development instance served from inside

[4] `https://linuxcontainers.org/`
[5] `https://www.docker.com/`

a virtual machine, for example. Secondly, depending on the written tests, installation of leveraged security testing tools is needed.

If testing in a development environment against a development instance, not all of the security verification tests might be meaningful to run. For instance, SSL configuration checks of a development instance are a waste of time unless the webserver configuration is done identically to production via IaC.

The only challenge in using these tools is that considerable security expertise can be required to come up with the security requirements for the application tested, and to write good security test cases based on the requirements. From there on, these tools can be adopted into the development process like any other acceptance testing tool that produces binary results for test cases, and requires little to no security expertise on the part of the developers reviewing the test results.

4.6 Dependency verification

Continuous dependency verification is easy to set up from both the technical and process perspectives. On the technical side, dependency verification tools exist for all major languages with package managers. Some tools can also identify third party libraries even if no package manager is used.

Installing the tools from the relevant package manager is simple, as is triggering them, and they generally require no initial configuration. In most cases, the only configuration done will be to ignore some findings. Usually this is achieved by adding some filters or rules into a tool's configuration files.

False positives can be one reason for wanting to ignore some findings, although dependency verification tools rarely produce them. It is also possible that a found vulnerability concerns a part of the third party library that is not actually used by the application being developed, and so might not require any action. In these cases, the ignoring functionality can also be used to prevent further alerts from the same cause.

The output format of results is in most cases structured plaintext, so while not understandable by CI systems as-is they are easy to parse into a binary pass/fail or into a count of found vulnerabilities and/or count of dependencies with vulnerabilities.

With the same effort, the tools can also be integrated to development environments via version control system pre-commit or pre-push hooks. Preventing the commit or push if issues are found would make the feedback cycle for developers even faster than integration into the CI system would.

The simplest process for handling the results in the CI system would be

to fail the build if any vulnerable dependencies are detected. The team would then need to verify the issue and either fix it or, in case it does not warrant any action, ignore it from further results.

For finer-grained handling in CI, each found vulnerability or dependency with vulnerabilities can be interpreted as a unit test failure from the CI system's perspective. This way, with a bit of tool output parsing and transforming, existing CI functionality can be used to present the amount of findings in build status views and dashboards. Additionally the individual issues can be tracked using the CI system's core functionality.

Doing dependency verification for each commit during active development is not adequate. It should also be done at regular intervals throughout the lifecycle of the software because new vulnerabilities are disclosed continually, not just during the active development phase of an application.

4.7 Summary

We found several guidelines for work management, tool adoption, testing and results handling to ensure successful adoption of any reviewed security testing method into a CI process. In addition to these common guidelines, there are method-specific characteristics that need to be recognized for best results.

Both technical and process integration were found to be the most straightforward for security verification and static security analysis. For the former this is by design, since CI is central to the whole method. For the latter it is because the existing tooling and best practices of static analysis can largely be used as-is.

Configuration checking and dependency verification results can be reduced to unit/acceptance test format and handled as such with existing CI tooling and development process best practices. On the technical side, this just requires implementing a simple results parser.

The most challenging of the covered methods to apply to continuous security testing is vulnerability scanning. Practically no tooling to support its use in CI exists, and neither do established best practices. For most vulnerability scanners, the main challenge for continuous use is the absence of cross-scan findings tracking and false positive management mechanisms.

Chapter 5

Case studies

In this chapter, we will go through four case studies in which we integrate suitable security testing tools into a software project's CI feedback loop.

For each case, we give a general description of the project as well as a preliminary analysis based on which we decide what security testing methods and tools are suitable for use in the project's CI system. We also provide observations on both the technical and development process integration of the selected tools, as well as the actual security testing results achieved by using the tools.

In each case, we determined the most suitable tools to test via discussion with the team or team lead, taking into consideration the restrictions on effort and technology choices imposed by the project. We then handled the technical setup and integration of the selected tools into the CI system. After the tools were running as part of the CI system, we briefed the team on the CI setup and introduced the tools to them. Then, we discussed the possible ways of handling the output of the tools with the team or team lead, and had the team agree on a process for handling the security testing results. If necessary, we then made the relevant changes to the CI configuration to support the process that the team chose.

After the initial setup phase, our intention was to leave the team mainly to their own devices and monitor how the tools and their results were used over a period of some months. Unfortunately, we only managed to properly keep track of one of the projects after the initial phase.

5.1 Project A: a Ruby on Rails web application backend

The target project of this case study was a web application with a Ruby on Rails backend that exposes REST APIs that are utilized by an AngularJS frontend.

The Ruby portion of the codebase was approximately 5000 lines of code.

At the time of this case study in the spring of 2014, the project had been going on for nearly a year. The development team had consisted of 1-2 developers during the project lifetime. The development process was Scrum. Jenkins was used for the project's CI platform.

5.1.1 Preliminary analysis

Due to the small size of the development team, it could accept only minimal overhead from security testing. In light of this, our options were practically limited to dependency verification and static analysis tools. We decided to try static analysis, and see if the overhead from it would be acceptable. Brakeman was chosen for this case as the most potentially suitable static analysis tool.

5.1.2 Brakeman

Technical integration of the tool was easy, as it only required adding a gem into the project's Gemfile. After this, the tool could be run both on development machines locally and in the CI system. Since there was a Jenkins plugin available, results presentation in CI was also extremely easy. The CI plugin provided a convenient way of configuring the build to fail or become unstable based on how many warnings of different confidence levels (high, medium, weak) were produced. The thresholds could be configured to consider only new warnings or both new and old warnings.

The CI job was initially configured to determine build status based on both old and new warnings, with separate thresholds for warnings of different confidence levels.

For the results handling process, we decided that the team would immediately review and prioritize all high confidence warnings to be fixed as soon as possible. For these, the threshold for failing the build was set to 1. Warnings of lower confidence levels were to be reviewed and prioritized in a more leisurely manner, and they were configured to only set the build as unstable but never fail it.

5.1.3 Results

The technical integration of Brakeman was easy for both development environments and CI system.

The security testing results produced by Brakeman were encouraging: it produced a total of 26 (3 High, 3 Medium and 20 weak confidence) security warnings on initial run, all of which were eventually verified to be valid.

The workflow and development process integration of the tool required some iterating. The amount of warnings produced by the tool when it was initially run made the developers reluctant to verify and address the issues because they thought it would take too much time away from the actual development work. A contributing factor to this perception was a predisposition to believe the warnings would largely consist of false positives. This predisposition was the result of us initially warning the developers that static analysis tools may generate considerable amounts of false positives.

After a superficial review of the results of the initial run, the developers dismissed all the warnings as false positives. Afterwards they were reluctant to waste any more time by investigating the initial warnings more thoroughly or configuring them to be ignored. Since the CI job was configured to determine build status based on threshold values for warnings of each confidence level, the developers also quickly started ignoring the static security analysis job that was constantly flagging failures.

Once we recognized this initial tool adoption failure, the situation was improved by first going through the initial results more thoroughly with a developer to conclude that they were in fact all relevant warnings. This gave the developers greater confidence in the usefulness of the tool, and alleviated the concerns about how much time the verification of the findings would take.

Second, since the priority of taking action on the warnings was deemed too low even though the warnings were seen as relevant, the CI job was configured to only alert on new warnings. The team would then commit to verifying all new warnings immediately as they are detected. Any actions required by verified true positives would be placed onto the development team's backlog and planned, estimated and prioritized like any other development work.

In the end, the development process integration achieved was satisfactory. Some effort was required for iterating approaches to find a suitable process integration method, but the continuous work effort required for reviewing and managing the scanning results was low.

5.2 Project B: a Java + Scala web application backend

This project was a web application consisting of 8 independent, separately deployed and run modules, with most of the code written in Java. The smallest of the modules was written in Scala, and much of the frontend in ActionScript.

The project was at the time of this case study in spring 2014 in maintenance phase, with only small-scale development of new functionality going on in addition to support tasks. The size of the development and maintenance team was 2-3 developers using Kanban as the development process and Jenkins as their CI server. The project, and the oldest parts of the codebase, were 5 years old.

The total lines of Java code was approximately 55000. The Scala module was some 600 lines of code. The frontend contained 35000 lines of ActionScript, a considerable portion of which was automatically generated from the Java code.

5.2.1 Preliminary analysis

Some possible issues that could affect tool adoption were identified beforehand. One was budgetary concerns: since the project was in maintenance mode and working with a limited budget, not much work effort could be expended into this case study. Another issue was the relatively large existing codebase, which could produce lots of warnings and false positives when using static analysis tools.

Because of the budgetary and effort allocation constraints we could in practice only consider least effort approaches, namely dependency verification, configuration checking and static analysis. Of these, configuration checking was ruled out because the project did not have automated infrastructure setup. Thus, dependency verification and static analysis were the selected methods.

The tools selected for use were OWASP Depedency Check and FindSecurityBugs.

OWASP dependency check was the tool chosen for dependency verification because of the extremely low effort required to set up and maintain it.

FindSecurityBugs had the following points in its favor:

- Running the tool in either CI or development environments would be

extremely easy because the tool is available as a Maven plugin and Maven was the build tool for the Java modules of this project

- Presentation of results in CI would also be effortless since a FindSecurityBugs plugin is available for Jenkins
- Because FindSecurityBugs analyses JVM bytecode, it could also be used to analyse the module written in Scala

5.2.2 FindSecurityBugs

Integration into a Java module using Maven as a build tool only required the addition of a single plugin entry to each module's pom.xml-file to be able to run the tool in Jenkins CI jobs as well as locally on development machines.

For the Scala module, things were not so simple, since it used SBT as the build tool. However, with a bit of work we managed to get the tool running in the CI cycle. This required using SBT to generate a maven pom.xml file and post-process the generated file to:

1. include the FindSecurityBugs plugin declaration
2. properly handle the submodules that each of the project's modules was split into
3. include custom maven repository definitions to be able to access in-house dependencies
4. correctly resolve snapshot artifacts of dependencies under development

The results of each test run were easily presented and build status thresholds set in Jenkins using the FindBugs Jenkins plugin.

5.2.3 OWASP dependency check

Integration to maven modules was trivial, and only required the addition of a single plugin entry to each module's pom.xml-file. Running the tool in CI via maven was easy. Results presentation was also very easy, since a plugin for this purpose was available for Jenkins, the CI system used in the project.

For the Scala module built using SBT the process was identical to the FindSecurityBugs setup described in the previous subsection. We needed to generate a maven pom.xml-file containing the dependencies and to post-process the OWASP Dependency Check plugin declaration into it.

5.2.4 Results

The technical integration of both tools was easy, although some work was required to get the tools working with the Scala module.

FindSecurityBugs analysis resulted in over 200 warnings, and upon cursory examination, a large portion of them were very general. For example, each REST API exposed by the applications generated a general reminder warning to pay attention to user input validation. For the Scala module, FindSecurityBugs generated only one warning. The single warning was a false positive.

The large amount of warnings produced by the static analysis combined with the budgetary constraints of this project meant that it was not feasible to verify them immediately. Because of this, the process we decided on for handling the results was two-fold: the results of the initial analysis would be treated as a baseline and would be reviewed and fixed or ignored in small amounts over time, whereas new warnings would be reviewed and dealt with immediately.

OWASP Dependency check found tens of vulnerable dependencies across the whole project. Overall more than 100 separate publicly disclosed vulnerabilities existed in the dependencies.

The dependency verification results contained a lot of overlap. This was because we checked each module separately for vulnerable dependencies, and most dependencies were shared between modules. For example, all of them depended on the same database client library. As such, every module's dependency verification results contained the same warnings about vulnerabilities in the database client library.

For most cases, this is fine, but in a case like this one with a considerable part of dependencies being shared across modules, it would be better to aggregate the dependency verification results across all submodules or produce some kind of overview or other method of visualizing how many submodules use the same vulnerable dependency. That way the results would more accurately represent the amount of vulnerable dependencies, and prioritizing which of the dependencies is the most critical to fix would be easier.

For the same reasons behind the process for handling static analysis results, the agreed process for handling dependency check results was almost the same.

Firstly, to treat the initial findings as a baseline and have the CI system alert only on new vulnerabilities.

Second, start with a least effort approach to fixing the vulnerable dependencies. One by one, check which vulnerable dependencies can be upgraded without causing test failures in CI. Upgrade those without further investiga-

tion. The rest will require code changes, for example to conform to changed third party library APIs. Those will be investigated to get an estimate on the severity of the vulnerability and the effort required to upgrade the library. The fixes will then be prioritized through the normal work management process.

Overall, the development process integration of the selected security testing tools in this case went well. A working process for handling the sizable amount of warnings produced by the initial tool runs as well as the new warnings that would be produced in continuous use, while staying within the work effort restrictions imposed by the project, was successfully adopted.

5.3 Project C: a Scala + Java web application backend

This case was concerned with a web application backend written mainly in Scala, with parts in plain Java. The backend was one component of a larger web service. Active development of the backend with a 2-developer team started in November 2013.

The development process was Scrum. The team used Jenkins for their CI server.

At the start of this case study, in January 2014, the codebase was 500 lines of Scala code, and at the end in July 2014 9000 lines of Scala code and 3000 lines of Java code.

5.3.1 Preliminary analysis

This was a promising case since it dealt with a greenfield project in its early stages of development.

We determined that static analysis of Scala source code or JVM bytecode could be used for continuous security testing. Also since the application is a backend that only exposes different APIs for other parts of the larger web service to use, black-box scanners, plain fuzzing or other methods for attacking the APIs could be used. In retrospect, dependency verification methods would also have been a possibility and the most logical first step towards continuous security testing since the project used Maven and SBT for dependency management, but we had not yet come across dependency verification in our research at the time this case study started.

As for constraints, the small size of the development team meant that only minimal overhead could be accepted into the development process from

security testing that had not previously been factored into the project's development backlog.

We decided to start with adopting FindSecurityBugs to do static analysis, since by analysing JVM bytecode it could process both Scala and Java code. After seeing how the adoption of FindSecurityBugs went and how much overhead it incurred, we could later add other tools, too.

5.3.2 FindSecurityBugs

Integration into the Scala project required a little bit of work because it was built using SBT instead of Maven, but as this work had already been performed once for project B (see section 5.2) it did not take nearly as much effort this time.

The CI job was configured to alert on new vulnerabilities only, and these were to be checked and verified by the team immediately.

5.3.3 Results

The technical integration of FindSecurityBugs into a Scala project was easy, but would have required some more work if it had not already been once for project B in section 5.2.

Although we observed a good adoption of FindSecurityBugs into the development process with little overhead, we unfortunately never got back to integrating more security testing tools into the project's CI cycle.

FindSecurityBugs generated only four warnings during the duration of the case study, from January to July 2014. The team assessed all of the warnings to be false positives. The false positives were two cases of "use of weak hashing function (MD5) for message digest" and two of "potential path traversal vulnerabilities". We verified the assessments to have been accurate.

The developers felt that running FindSecurityBugs in CI was useful despite only having produced false positives. According to them, this was because the vulnerability information provided alongside a warning was good for raising developers' security awareness and because verifying the produced warnings took very little time away from development work.

5.4 Project D: a Java web application backend + AngularJS frontend

This case deals with a project whose web application has a Java backend and AngularJS frontend.

The backend consists of two separate Java projects, one for actual backend functionality and another that contains the API of a separate service utilized by the backend. Both of these used Maven as their build and dependency management tool. The API project code was imported from an external source and could not be modified by the team.

The team size was six developers in total, and the development process was Scrum. Jenkins was used as the project's CI server.

The size of the frontend codebase was approximately 2000 lines of JavaScript code. The backend was 9000 lines of Java code.

5.4.1 Preliminary analysis

All of the security testing methods we have covered were determined to potentially be applicable to this project. Static analysis and dependency verification tools were available for both languages, and the infrastructure automation of the project made vulnerability scanning and configuration checking possible. The team was also sizeable enough that it could conceivably support security verification tool adoption, if the required security expertise was available.

The main constraint imposed by the project was that no work effort budget had been allocated for continuous security work, and there was little slack in the developers' effort budgets. Some security activities were included in the later stages of the project's road map, though. Nevertheless, no security experts were continuously available to support the team in continuous security testing, even if the work effort were re-budgeted.

After discussing the options with the team's lead developer, we decided to start with adding dependency verification tools for both the backend and frontend into the CI cycle. Static analysis could also be applied, but vulnerability scanning and security verification frameworks were deemed to require too much effort and/or security expertise, and configuration checking inapplicable due to perceived low payoff.

5.4.2 OWASP Dependency check

Since the project's CI server was Jenkins and OWASP Dependency Check has a Jenkins plugin, integration into CI build cycle was as simple as installing the plugin, adding the proper Maven plugin declaration to the project's pom.xml-files and configuring a new Jenkins job for running the tool and presenting the results.

Process-wise, the CI jobs were configured to mark builds as unstable on normal or low level warnings, and to mark them as failed on high level

warnings. The procedure agreed upon for handling these warnings was to investigate and verify the warnings periodically and either fix the warning by updating the vulnerable dependency or if the warning is not considered relevant, mark it as ignored by using the repository's XML file intended for this purpose.

Since the team could not make any changes to the API project, the process for handling vulnerable dependencies found there would just be to notify their contact person for that project about the finding and hope that something is done about the issue.

5.4.3 Versions Maven Plugin

For this tool, there was no CI plugin available, but the actual tool itself was easy to add as a single plugin declaration in the project's Maven pom.xml-files. To present the results via build status in CI, we decided to simply search for a string that the tool always prints in its results if updates are found to at least one dependency and use that to decide the build status.

The tool needed to be configured to disregard SNAPSHOT, release candidate and other such potentially unstable versions when looking for newer versions of dependencies.

The CI jobs for this tool were configured to mark the build as unstable always if any newer versions of dependencies were available. The process for handling unstable builds was decided to be to review all dependencies for which it reports newer versions are available and either update the dependency or selectively ignore the alert by using the repository's XML file intended for this purpose.

5.4.4 Retire.js

For the frontend Angular.JS application, Retire.js was adopted using grunt-require to check the JavaScript libraries for known vulnerabilities.

The output of the tool was only structured plaintext, so some scripting was required to interpret and present the results in the CI system. We took the approach of looking for a string that is present whenever the tool finds any vulnerable libraries, and configured the CI job to mark the build as unstable in that case.

5.4.5 FindSecurityBugs

As the project already included FindBugs as a maven plugin to do standard static analysis in the CI cycle, setting up FindSecurityBugs proved challeng-

ing. This was because FindSecurityBugs limits the static analysis rules used by FindBugs to only security-related ones, so modifying the FindBugs configuration to enable FindSecurityBugs would have prevented the normal static analysis from happening.

We might have been able to bypass this issue with additional Maven project configuration, but unfortunately did not have time to investigate that possibility further. So instead, we left the normal FindBugs setup as it was and dropped FindSecurityBugs from this case.

5.4.6 Results

The technical integration of all tools except FindSecurityBugs went smoothly.

After integrating the tools to the project's CI system, we went through their usage with the lead developer of the team. He felt that using the tools was an excellent idea, but that because of current time and effort constraints the team would probably not be able to do anything about the warnings until much later.

One probable reason for this was that the security testing tool integration into the CI system of this project was done, although after discussing the topic with the lead developer and with his somewhat enthusiastic approval, on short notice and bypassing the normal work planning and prioritization process.

Since active adoption of the tools was not deemed to happen immediately, we went through the usage of the tools with another developer to spread the knowledge a bit more and increase the chances of successful adoption at some later stage. We also documented the tools, their usage and the reasoning behind using them into the project's wiki with the same developer.

OWASP Dependency Check found on its initial scan that 14 of the 130 dependencies declared in the API project had one or more publicly disclosed vulnerabilities. Unfortunately, the team could not do much about these, other than pass the information along and hope something is done about the issue. For the actual application project, 6 out of the 19 declared dependencies were found to have known vulnerabilities.

Maven Versions Plugin found on the initial run in the API project that out of the 130 overall dependencies declared, 17 had newer versions available. For the actual application project, the count was 15 out of 19.

One challenge we ran into with Maven Versions Plugin was that the time taken to check for updates could be prohibitively long for inclusion into the test chain of each commit in CI. For example, over 35 minutes was taken to check for updates for the 130 dependencies spread between 2 maven repositories (Maven Central and internal repository which contained non-public

artifacts) in the API project. While the repository package information was cached for subsequent runs, which were much faster, the run time would be quite long again once the cache eventually expired. This was a problem with the CI setup of this case, since it would regularly cause a CI job's build to take considerably more time than normal. Because of this, we had to separate running Maven Versions Plugin from the main build chain into a separate CI job so it would not block running other tests.

Retire.JS found 34 vulnerabilities among the dependencies declared, including dependencies of dependencies recursively.

The success of the development process integration of the security testing tools in this case was left unclear. We agreed with the team on how each of the tools would be integrated to the development process, and documented the tool usage, CI setup and workflow integration for later use. However, the actual handling of the testing results by the team was postponed to start at an undefined later date.

5.5 Common observations

We noted that the following things applied in all the cases:

1. The technical integration of the selected tools into each project's CI loop was easy.

2. The development process integration, especially of static security analysis tools, required more careful consideration and followup work.

3. Only a very limited amount of developer effort could be used for initial or continuous work.

4. The developers seemed to consider the adoption of continuous security testing into their project's development workflows useful and necessary.

The main challenge of the development process integration of static security analysis arises from the fact that, unlike with dependency verification, the results cannot reasonably be reduced to a unit test like pass/fail model. Instead, all results require manual review to first verify the accuracy of the finding, and then decide on what further actions, if any, are required. Unless properly managed, this can cause apathy among developers toward security testing in at least two major ways.

Firstly, developers can feel that the work involved is unnecessary overhead that slows development and makes them looks bad. This is especially

dangerous if adequate time and effort is not allocated to security work in the project's work planning. A good way to mitigate this danger is to drive all security work through the normal process, making it visible to all stakeholders and ensuring that work time is really allocated for it[28].

In addition, adopting continuous security testing practices in the beginning of a project is always easier than introducing them into a project that is well under way. The findings of security testing done continually from the start come as a steady trickle that's easy to manage, whereas introducing security testing later on can produce a large amount of findings, and thus unexpected work, at once. Another good mitigation technique is to only enable a few detection sets for easy-to-understand, relevant vulnerabilities, and gradually enable more detection sets at a pace that the developers feel comfortable with.

Secondly, accurate verification of static security analysis findings can require some level of security expertise on the part of the reviewer. Since most software developers are not security-inclined, they can feel, as well as really be, out of their depth when reviewing the findings. There are several ways to mitigate this danger. One is to have security experts separate from the developers to review the findings, although that might not be possible for all organizations or projects. Another is to always have a pair of developers review a finding together[5]. Finally, having the developers undergo security training helps, if they have any interest in the topic.

As an example of the pitfalls of findings verification, in the case study of project A, described in section 5.1, the development team was so strongly predisposed toward classifying static analysis warnings as false positives that they regarded all the initial findings as false positives. This was at least partly caused by us warning them beforehand that some amount of false positives are usually generated by static analysis tools and stressing that therefore verifying the warnings is essential before acting on them. When adopting security testing methods such as static analysis that require verification of findings, care should be taken to make the development team aware of the necessity for verification without causing such predispositions.

The budget or work effort available to us was very limited in all cases. Getting any security expert support was practically off the table, and teams were reluctant to take up practices that would continually demand developer effort. This issue was largely caused by how we conducted these studies: in an ad-hoc manner when opportunity presented itself, bypassing normal work prioritization and planning.

Since continuous security testing had not been entertained at all in the project proposals and agreements, no allowances for it had been calculated into budgets either. This meant that security expert support or major addi-

Dependency verification	Static analysis
OWASP Dependency Check	FindSecurityBugs
Brakeman	Brakeman
Retire.js	
Versions Maven Plugin	

Table 5.1: Security testing methods and tools used in the case studies

tional work by developers would require additional budget from the customer and thus further budget/contract negotiations.

Furthermore, because continuous security work was not taken into account during project planning, all developer effort towards it would effectively be taken from either other development overhead functions or feature development. As such, any significant effort spent on continuous security testing would lessen the chances of reaching the agreed project targets. This issue would probably not be as significant in internal software development activities, even though they are also usually budgeted and work effort managed in some way.

A better approach would have been to drive the adoption of the tools via the normal planning and work prioritization process, as well as include adequate resources for the security testing work in the initial project plan/proposal.

Despite the fact that developers in most cases felt that, due to project time and work effort allocation constraints, they would likely not be able to actually do anything about the testing results, they seemed to consider the adoption of continuous security testing into their project's development workflows useful and necessary.

5.6 Summary

In the four case studies, we adopted each of the security testing methods and tools listed in table 5.6 to at least one software development project's CI loop.

Despite being limited to only these two methods because of project constraints, we achieved satisfactory testing and integration results. Adopting and using these tools required little effort, but still produced valuable security testing results: multiple real vulnerabilities in the developed applications were uncovered.

Unfortunately, we managed to monitor the tool usage and results over several months only in the case study described in section 5.3. For the other

cases, actual relevant continuous security testing study was thus not really done. This being the case, we cannot offer more than anecdotal evidence for the long-term success of our chosen methods of the process integration of continuous security testing into the software development workflow.

The technical integrations of these tools into the CI systems were mostly smooth and effortless. This was expected, as the basis for selecting the tools was that they should give the most payoff for the least amount of effort.

The development process integrations required more careful consideration and followup work, especially for static security analysis tools. Fortunately, neither static analysis in general or even static security analysis specifically are not new phenomenons in the software development world. Both research as well as literature on good practices is available. For example, "Secure Programming with Static Analysis" by Chess and West[10] has some good pointers on adopting a static security analysis tool.

We had to make do with very little developer effort and no security expert support, largely because we conducted the case studies in an ad-hoc manner with a tight schedule when opportunity presented itself, bypassing normal work prioritization and planning. A better approach to the case studies would have been to drive the adoption of the tools via the normal planning and work prioritization processes of the projects. For future projects, continuous security testing should already be made part of the initial project plan/proposal to avoid work effort allocation and budgeting challenges.

Chapter 6

Discussion

In this chapter, we first present notable observations and open questions that have arisen during our work. Then we review the recognized challenges and limitations of continuous security testing, and acknowledge the limitations of this work. Finally, we present our ideas for future work related to continuous security testing.

6.1 Observations and open questions

The technical integration of all reviewed security testing methods into a CI system is easy. For some tools, the amount of work required is not trivial, but it is nevertheless not difficult work. This leaves the ease of development process integration to be the watershed when considering a method's suitability for continuous security testing.

The methods that are most suitable for continuous security testing are ones that perform a set number of tests whose results do not require further investigation or verification. The individual results either are inherently of a pass/fail nature or can be meaningfully interpreted as such. Configuration checking, security verification and dependency verification fall into this category. Because they can be reduced to a unit test like pass/fail results format from the perspective of the CI system, they can be integrated to the development process using established best practices and tooling like any other unit or acceptance testing type.

The rest of the reviewed security testing methods are characterized by producing an unforeseeable amount of findings for each test run. The produced findings are categorized to different confidence levels and always require manual verification. The development process integration of these methods is decidedly feasible, but significantly less straightforward than that

of the methods in the former category. This category contains static security analysis and vulnerability scanning.

One promising and lightweight approach to the workflow integration of static security analysis for development teams using pull requests is to automate the static analysis of a pull request's changes, and import the warnings as code review comments into the pull request. For example, pronto[1] and SonarQube via its Github plugin[2] can be used to easily post static analysis results as comments to Github pull requests. Unfortunately, we discovered this approach too late to try it out in our case studies.

We were pleased to note that static analysis and dependency verification tools exist for JavaScript, too. While vulnerabilities in the frontend code might not usually be as dangerous as ones in the backend, they can still involve for example serious XSS issues.

One question regarding static security analysis that we did not explore in this work is how to handle a situation where both general static code analysis and static security analysis are performed. Is it better to present and process all static analysis results through one report or strictly separate the security findings from other static analysis results? In a study, Baca[2] advocates having only a single static analysis report, because out of two reports the larger will easily get ignored. He also states that security relevant findings should still be clearly separated from other static analysis warnings to keep them from being lost among the usually considerably greater amount of general warnings.

Getting the development team to take ownership of the security testing is critical for continuous security testing implementation success. Tool adoption chances can be improved by developer education and good tool selection. Developers should be educated on efficient use of the tool as well as on the importance of catching and fixing issues as early as possible[5]. A good tool is one that fits easily into the development process and produces easily understandable, highly relevant and actionable warnings[2].

The DevOps movement promotes using tools and reports that are usable by and understandable to all relevant parties. For software security testing, these parties include at least security experts, developers, operations personnel and the managers deciding about work prioritization. Abiding by this principle when implementing continuous security testing seems like a good way of raising general security expertise and awareness of the related work among all the stakeholder groups. Even if the tools did not catch many vulnerabilities, the presence of easily accessible and understandable security

[1] `https://github.com/mmozuras/pronto`

[2] `http://docs.sonarqube.org/display/PLUG/GitHub+Plugin`

testing tools and reports in the CI loop can encourage stakeholders to self-improve their understanding of security issues.

6.2 Challenges and limitations of continuous security testing

It is important to recognize the inherent limitations of an automated, continuous approach to security testing. In general, the methods that lend themselves to continuous security testing are ones that find security bugs, but not design flaws. For example, security analysis of the software architecture is not considered automatable.

Not understanding the inherent limitations of continuous security testing can also lead to a false sense of security, which can be very dangerous. While low-effort continuous security testing is certainly better than no security work at all, it is no substitute for actual security design or secure coding practices.

As for the actual testing tools and CI system plugins that are freely available, their quality and maintenance varies greatly. Care must be taken when selecting a tool. Furthermore, it is prudent to monitor the maintenance status of the selected tool(s) so that stagnant tools can be replaced in a timely manner. We expect that this is a significantly smaller issue with commercial offerings than with the open source tools that we surveyed.

Of the actual security testing tools, only the security verification frameworks have been designed with CI integration in mind. So very few tools provide results in any common test result format that CI systems would be able to readily interpret and present. Some transforming of the test results thus is needed in almost all cases for example to present a summary of security test results on a CI job's status view. Some of the most common tool and CI system combinations have such functionality available, but most still require one to implement it themselves.

When planning the adoption of continuous security testing into a development process, it should be remembered that the possible testing frequency is largely determined by the selected testing methods and tools. Some, like static analysis or dependency verification, can be run for every commit while for example vulnerability scanners easily take hours to run and so are in most cases not feasible to run for every commit.

False positive identification and management is a deciding factor for the usefulness of methods that produce them. A study by Baca et al.[6] concludes that developers, regardless of general developer seniority, are bad at correctly identifying security vulnerabilities and false positives from static

analysis tool results. They also note that identification performance for security vulnerabilities is significantly improved by a combination of security training and static analysis tool expertise, but that neither improved the identification rate of false positives.

All of the static analysis tools we reviewed either solely detect security issues or have a specific category for them, which in our view makes the developers' poor performance in identifying security vulnerabilities from among static analysis results a non-issue.

A very low false positive rate is critical, then, for static analysis tool usefulness in developer hands. Fortunately, since static analysis findings are categorized by confidence level, it is possible to limit the analysis to high confidence findings in developer use to mitigate possible issues from false positive mis-identification. It is also important to avoid over-emphasizing false positives when introducing security testing tools to the developers. We found that doing so can significantly bias them towards categorizing findings incorrectly as false positives when reviewing security testing results.

Baca et al. also found in a separate study[5] that incorrectly identified false positives can even lead to actual security issues being introduced by developers trying to fix the non-existent vulnerability. In the study, they recommend having at least two developers cooperatively verify findings and determine how to fix valid ones to increase the likelihood of correct assessment. We concur with this, and note that pair review of results is also a good practice for spreading security expertise in addition to making individual assessments more accurate.

The same study[5] also notes that developers are by themselves reluctant to start using security testing tools, possibly because they do not immediately see the benefits. It also states the following:

> "We conclude that a configuration management approach where the tool is integrated in the development process as a mandatory part is the best adoption strategy that is only efficient if developers are educated in order to make use of the tool to correct identified vulnerabilities as soon as possible after detection."

This corroborates that our approach of integrating security testing tools into the development process via CI system is a sensible one, but also highlights the importance of developer education.

Lastly, it is not clear to us how the more general static security analysis warnings, such as reminders about proper input sanitation when encountering an exposed API, would best be handled in a continuous testing context. It is not good to process the alert for every test run, but neither is it smart to

verify that everything is fine on the first alert and ignore them from there on out. A method of detecting when the underlying implementation has changed and triggering a warning only in those cases would be optimal, but we did not encounter such functionality in the reviewed tools.

6.3 Limitations of this work

Our case studies were performed ad-hoc as the opportunities presented themselves. Because of this, we encountered resourcing and schedule challenges that most likely would have been avoided with a slower, better-planned approach. The case studies were also conducted in parallel with our research into security testing methods and tools, so in hindsight more suitable tools and better tool adoption processes could have been used in most of them.

Additionally, we monitored the tool usage and results over several months in only one of the four case studies. For the other cases, we have no observations on how well continuous security testing was adopted as part of the development process. As such, we cannot draw conclusions about the long-term successfulness of the workflow and process integration methods.

Furthermore, we tested the software development process integration of only dependency verification and static analysis tools in the case studies. Lessons learned from practical implementation experiences are thus missing for three out of the five investigated security testing methods.

Our tool availability analysis was only covered a section of available technologies, so we can't say anything about that for many notable languages such as C, Objective C, .NET and Go.

The static security analysis papers by Baca et al.[4][5][6] all deal mainly with analysing C code for security issues with general purpose static analysis tools. Therefore, their observations might not completely apply to higher level languages or situations where the static analysis tool only detects security issues.

Finally because the review of available methods and tools in 3 was performed in 2014, multiple years before the actual writing of this work was completed, the analysis of the tools may in cases be obsolete.

6.4 Future research

A common output format for the results of security testing tools would make it a lot easier to integrate multiple different tools into a CI cycle. While the output produced by security testing tools is in most cases not as simple as the

binary pass/fail of unit tests, some reasonable lowest-common-denominator format could still be conceived. Most output we have seen while working on this thesis could be represented reasonably well in CI status screens and dashboards with just a confidence level and a severity. For closer inspection in CI, a single field for free-form details could be specified to accommodate the diversity in information provided by different tools. For static analysis tools, an optional field for referencing the offending source code line would be good, too.

To complement the common results format, on the CI system's side some sort of common core library for security testing results management would be extremely useful. Something similar to the Jenkins static analysis core[3], that would provide functionality for tracking and management of findings across consecutive builds, false positive management, issue verification, setting build status, results presentation and possible other similar things.

Currently, lacking a common results format and CI core, the situation is OK for static security analysis and security verification: the former's results can be handled adequately using existing static analysis handling methods and the latter frameworks' results are by design easily handled in CI like any other acceptance tests. However, the results of configuration checking and dependency verification tools are easiest to handle by parsing them into some pass/fail unit test format for the CI system to digest. The most challenging results are produced by vulnerability scanning, for which no reasonable way of handling beyond showing an overall pass/fail in the CI system exists. For them we must instead rely on the tool's own issue management facilities, which in most cases do not address consecutive scans in any way.

We also feel that the continuous security testing methods that were in the end covered only lightly in this work would benefit from further study. Most notably the possibilities opened up by infrastructure as code and deployment automation combined with configuration checking and web application scanning tools, as well as combining existing automated API or end-to-end tests with a web application scanning proxy sound like promising avenues of investigation.

One of the challenges we presented in section 6.2 was how to best handle certain general static security analysis warnings, like reminders to properly sanitize API input. A possible way to efficiently handle these cases would be to combine static taint analysis, like utilized by Livshits and Lam[16] or Baca[2], with static analysis of code diffs like for example Pronto[4] enables.

[3] `https://wiki.jenkins-ci.org/display/JENKINS/Static+Code+Analysis+Plug-ins`

[4] `https://github.com/mmozuras/pronto`

We are not aware of any existing implementation for doing this, but the idea sounds feasible to us. It would amount to mapping access paths to/from the source code lines that originate the general warnings and issuing a warning only if the diff intersects with any of the paths.

6.5 Summary of findings

Regarding the research goals we set in Section 1.2, here are the most relevant findings.

6.5.1 Integration methods

In Chapter 4 we presented several guidelines for work management, tool adoption, testing and results handling to ensure successful adoption of any reviewed security testing method into a CI process. In addition to these common guidelines, we also covered several method-specific characteristics that need to be recognized for best results.

Both technical and process integration were found to be the most straightforward for security verification and static security analysis. For static security analysis, the most promising integration approach would be to automatically include the findings into the normal PR code review process, to be handled like any other comments.

Configuration checking and dependency verification were also found to be relatively easy to integrate, as their results can be reduced to unit/acceptance test format and handled as such with existing CI tooling and development process best practices.

Vulnerability scanning turned out to be the most challenging of the covered methods to apply to continuous security testing. The main causes for this are lack of tooling and best practices for CI.

Overall, we recommend taking a configuration management approach to adopting continuous security testing. If feasible, testing results should also be reduced to a unit/acceptance test format in which they can easily fitted into the CI workflow like any other tests.

6.5.2 Challenges of continuous security testing

In Section 6.2 we discuss the challenges and limitations of continuous security testing. First, it is important to recognize that in general the methods that lend themselves to continuous security testing are ones that find security bugs, but not design flaws.

It is also important to avoid a false sense of security that green build statuses of security jobs in CI may cause. Low-effort continuous security testing is certainly better than no security work at all, but it is no substitute for actual security design or secure coding practices.

Another thing is that since the quality and maintenance of freely available security testing tools and CI system plugins varies greatly, care must be taken when selecting a tool. Maintenance status of selected tool(s) should also be monitored so that stagnant tools can be replaced in a timely manner.

Few security testing tools have been designed with CI in mind, so some effort is likely required to integrate one into the CI process.

False positive identification and management is a deciding factor for the usefulness of methods that produce them. A tool should have a very low false positive rate when the results are to be processed by developers.

6.5.3 Characteristics of tools suitable for continuous security testing

In Section 3.6 we reviewed the characteristics we found to most affect a tool's suitability for continuous security testing. The following high-level characteristics of security testing methods and tools were found to most affect the suitability for continuous testing:

- Ease of integration into the development workflow
- Initial work required for meaningful results
- Continuous work required alongside software development
- Security expertise required for meaningful use of the tool

These high-level characteristics are the result of many specific features of the tools and methods. For example, the following features have a positive with regard to continuous security testing:

- Very low false positive rate
- Results are machine-reducible to pass/fail
- Mechanism for efficient management of findings (incl. false positives)
- Cross-scan issue tracking

Chapter 7

Conclusions

Our goal in this work was to find out if currently available security testing tools could be leveraged in CI systems to add a basic level of continuous security testing to a modern software development process. To this end, we reviewed different security testing methods and currently available tools representative of each method to find ones suitable for implementing continuous security testing. We also conducted four case studies, in which we added some of the reviewed tools to a software development project's continuous integration system and development process.

We found that continuous security testing can be done with current tools and methods. Multiple different, complementary approaches to implementing it were found to be available depending on the level of expendable effort and security expertise at hand. Some of these approaches have been recently enabled by the emergence of modern infrastructure and deployment automation techniques, while others have been possible for longer.

Of the security testing methods investigated, dependency verification is clearly the easiest starting point for continuous security testing for three reasons: it is the most straightforward of the available approaches, requires minimal effort and security expertise, but it still offers tangible results. Thus, we recommend employing at least continuous dependency verification in all software development projects from the start.

In our case studies, we were limited to minimum effort approaches and consequently only tested dependency verification and static analysis tools as part of the continuous integration systems of the project's development processes. Technical integration of the selected tools was found to be easy in each case, but success in development process integration and team adoption of the tools varied. Nevertheless, the overall results observed during the case studies were encouraging, as in each case some level of continuous security testing was achieved with relatively little effort expended and actual

vulnerabilities were uncovered.

Although the technical integration of the reviewed tools into a CI system was easy, very few of the tools had actually been developed with automation in mind. Some integration work was therefore usually required. Further research and development towards more seamless integration of security testing tools into CI systems – for example in the form of a common results format and CI system libraries or plugins for parsing the said format – could boost the adoption of these tools and methods considerably.

Bibliography

[1] Allspaw, J. 10+ deploys per day: Dev and Ops Cooperation at Flickr. `http://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr`. Accessed Aug 18, 2014.

[2] Baca, D. Identifying security relevant warnings from static code analysis tools through code tainting. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on* (2010), IEEE, pp. 386–390.

[3] Baca, D., and Carlsson, B. Agile development with security engineering activities. In *Proceedings of the 2011 International Conference on Software and Systems Process* (New York, NY, USA, 2011), ICSSP '11, ACM, pp. 149–158.

[4] Baca, D., Carlsson, B., and Lundberg, L. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2008), PLAS '08, ACM, pp. 79–88.

[5] Baca, D., Carlsson, B., Petersen, K., and Lundberg, L. Improving software security with static automated code analysis in an industry setting. *Software: Practice and Experience 43*, 3 (2013), 259–279.

[6] Baca, D., Petersen, K., Carlsson, B., and Lundberg, L. Static code analysis to detect software security vulnerabilities - does experience matter? In *Availability, Reliability and Security, 2009. ARES '09. International Conference on* (March 2009), pp. 804–810.

[7] Bau, J., Bursztein, E., Gupta, D., and Mitchell, J. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on* (May 2010), pp. 332–345.

[8] Beck, K. e. a. Agile manifesto. `http://www.agilemanifesto.org`, 2001.

[9] Boehm, B. W. *Software engineering economics*. Prentice-Hall, 1981.

[10] Chess, B., and West, J. *Secure programming with static analysis*. Pearson Education, 2007.

[11] Doupé, A., Cova, M., and Vigna, G. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Kreibich and M. Jahnke, Eds., vol. 6201 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 111–131.

[12] Eshete, B., Villafiorita, A., Weldemariam, K., and Zulkernine, M. Confeagle: Automated analysis of configuration vulnerabilities in web applications. In *Software Security and Reliability (SERE), 2013 IEEE 7th International Conference on* (June 2013), pp. 188–197.

[13] Feitelson, D., Frachtenberg, E., and Beck, K. Development and Deployment at Facebook. *Internet Computing, IEEE 17*, 4 (July 2013), 8–17.

[14] Howard, M., and Lipner, S. *The security development lifecycle*. Microsoft Press, 2006.

[15] Korscheck, C. Automatic detection of second-order cross-site scripting vulnerabilities. `http://www.korscheck.de/diploma-thesis.pdf`, 2010.

[16] Livshits, V. B., and Lam, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Usenix Security* (2005), vol. 2013.

[17] McGraw, G. *Software security: building security in*. Addison-Wesley Professional, 2006.

[18] Ōno, T. *Toyota production system: beyond large-scale production*. Productivity Press, 1988.

[19] Poppendieck, M., and Poppendieck, T. *Lean software development: an agile toolkit*. Addison-Wesley Professional, 2003.

[20] Preston-Werner, T. Semantic versioning 2.0.0. `http://semver.org/`. Accessed July 29, 2014.

[21] Raymond, E. The cathedral and the bazaar. *Knowledge, Technology & Policy 12*, 3 (1999), 23–49.

[22] Schwaber, K., and Beedle, M. *Agile Software Development with Scrum*. Prentice Hall PTR, 2001.

[23] Shiny Development. Average app store review times. `http://appreviewtimes.com/`. Accessed Aug 18, 2014.

[24] Snyder, R. Continuous Deployment at Etsy: A Tale of Two Approaches. `http://www.slideshare.net/beamrider9/continuous-deployment-at-etsy-a-tale-of-two-approaches`. Accessed July 23, 2014.

[25] Sutherland, J., and Schwaber, K. Business object design and implementation: Oopsla '95 workshop proceedings. 1995. *The University of Michigan* (1995), 118.

[26] The Open Web Application Security Project. The top 10 most critical web application security risks 2013. `https://www.owasp.org/index.php/Top_10_2013`. Accessed July 24, 2014.

[27] The Open Web Application Security Project. The top 10 most critical web application security risks 2013 release notes. `https://www.owasp.org/index.php/Top_10_2013-Release_Notes`. Accessed July 23, 2014.

[28] Vähä-Sipilä, A. Software security in agile product management. `https://fokkusu.fi/agile-security/Software%20security%20in%20agile%20product%20management.pdf`, 2011.

[29] Veytsman, M. How to take over the computer of any Java (or Clojure or Scala) developer. `http://blog.ontoillogical.com/blog/2014/07/28/how-to-take-over-any-java-developer/`, 2014.

[30] Williams, J. The unfortunate reality of insecure libraries. `http://www.aspectsecurity.com/research-presentations/the-unfortunate-reality-of-insecure-libraries`. Accessed July 23, 2014.

[31] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P., and Vouk, M. A. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on 32*, 4 (2006), 240–253.