

CSEC 793 CAPSTONE IN COMPUTING SECURITY
PROJECT REPORT

**MANDATING SECURITY WITH UNIT TESTS
IN A CONTINUOUS INTEGRATION
PIPELINE**

April 4, 2018

Ryan Whittier
Department of Computing Security
College of Computing and Information Sciences
Rochester Institute of Technology
rjw4910@g.rit.edu

1 Introduction

The project in this paper is the development of a security focused continuous integration (CI) pipeline. The goal is to make a pipeline that focuses on security in an attempt to improve web application security. This pipeline will automatically run a series of tests against a repository of configuration files and code. These tests will enforce correct configurations and security practices within committed changes.

CI is a technique that is used to lead to better development practices, but also to speed up development releases. Those benefits can also be tailored to improve security in development. Many penetration tests or bug reports revolve around the same information repeated for different cases. These reports often have a common solution. If we discuss web applications, one common vulnerability is cross site scripting (XSS). XSS vulnerabilities have the short term recommendation of encoding all user input to avoid users input being interpreted as part of the webpage. The long term recommendation is to set up a full Content Security Policy (CSP), limiting where scripts and HTML tags can be loaded from. Companies generally choose the short term route of fixing each individual case, but this will cost them time and money in the long term, eventually they decide to implement a long term solution. In this example companies will often implement encoding, but as they expand the web application, they find the same bug. The company eventually switches to creating a CSP and configuring it correctly so new expansions can not be affected by a XSS bug.

Everyday applications are getting larger and are handling enormous amounts of data. Many of these applications are services that companies provide, such as Facebook and Netflix. All of these applications need to provide security for both their internal company data and the end user's data. The problem is that as these applications grow, their logic gets more complex and mistakes will be made, potentially introducing vulnerabilities into the codebase without the developers knowing it. Sometimes these mistakes are simply forgotten flags on cookies and other times they can be complex flaws in the business logic of the application. These mistakes can result in damage ranging from defacement of a company site, to a complete breach of a company's internal network.

A bug does not need to be exploited by a malicious actor to cause financial loss for the company. Many companies offer bug bounty programs, in which a researcher can discover and report a bug for a reward. The rewards vary by company, but often times the researcher will receive a substantial monetary reward, depending on the severity of the bug discovered. This means that each time the company releases updates to their applications, researchers will be scavenging for more vulnerabilities, which can costly for the company. It may also cost the company's security team a significant amount of time, as they need to look into and verify each report that is submitted. Often times, researchers can find very common vulnerabilities in the application, that usually result from developers attempting to push out new features at such a rapid pace.

CI can be used to catch simple mistakes automatically, by ensuring that the common bugs are found before the changes are deployed into production. By finding these bugs

before they are introduced into production, the number of submissions from bug bounty programs will decrease dramatically. This means that the bug reports that are submitted by researchers will be of higher quality and provide more value to the company.

2 Literature Review

2.1 Introduction

Continuous Integration (CI) has grown in popularity, being used in both enterprise and open source projects. CI provides an increase in productivity to programming projects by creating an environment where building, testing, and often deployment is done automatically. When developers do not need to do these tasks manually they can spend more time programming and finding bugs that trigger a failed build or failed test. Most research into CI looks at increasing productivity of developers and the enforcement of development standards. The potential use of CI for security has been mostly overlooked with most research looking at exploiting and protecting a CI system or a CI pipeline and not at the benefits of a pipeline focusing on application security.

A separate pipeline for security could provide a number of benefits for a security baseline. It could force certain configurations such as Content Security Policy in web applications, or lack of use of depreciated crypto algorithms such as DES or TripleDES [14]. A pipeline that looks only at big win configurations such as these would serve a similar function as Automated Source Code Analysis Tools (ASCAT) do when looking at code meeting project coding guidelines [15]. The pipeline could also include ASCATs that search for security bugs in a warning stage to avoid wasted time with failed builds from false positive. Another potential feature is an inclusion of fuzzers, which were recently used against memory forensic tools to look into anti-forensics techniques through crashing the tools [3].

2.2 Test case generation

Test case generation is a discipline that focuses around automatically generating unit tests for Software Engineering projects. The discipline sometimes focuses on generating tests from a base, such as the source code or a config file, and generating a set of tests from the base or it focuses on taking existing tests and expanding them to cover more features.

One paper from the 2014 Automated Software Engineering conference looked at generating test cases for web from an existing selenium test suite [4]. This paper goes through the method that is used to generate the new test cases. The method starts by taking a selenium test suite and going through it keeping track of states, http element interactions, and test case assertions. It uses the base states and element interactions to create a State-Flow Graph, which is a graph of the states with the interactions that move from one state to the next and the assertions that are involved to a single state. It then uses a web crawler on each state to discover alternate paths and new states. It then makes use of assertions on the base states to generate new tests for the crawler discovered states. An example given in the paper is that an application that allows you to create notes is tested with selenium by logging in, clicking the create note element, and then verifying that an id element had specific text. The test suite was then expanded by crawling to find the edit note interaction and the delete note interaction. These were then clicked and tested using the assertion,

verifying that the id was present and changing the text to match the new interactions result text.

2.3 Test case selection

The idea behind test case selection is limiting tests run to save on time during builds. As a code base grows, the number of tests increase. The increase in tests make it more and more expensive and eventually becomes to big a load to do everytime [12, 13]. This has lead to ways of avoiding running all of the tests on every code change. The ways of running tests have historically been either rerun all of the tests or manually pick test cases to run based off of the changes the developer made.

A paper by Milos Gligoric looks at comparing manual test selection against research made into automatic test selection, where automatic test selection would only run the tests that test code affected by the changes a developer made [7]. The state of the automatic test selection tools were limited to Google and Microsoft at the time of the paper. The tools were impractical for use by smaller projects because they were either too imprecise, selected tests that were not affected by the code changes, or unsafe, failed to guarentee all tests not selected were unaffected by the code changes. The paper specifically compares 14 developers manual test choices against a tools automated test selection. The paper found that manual test case selection was most commonly done during debugging. It also found the 73% of the time manual RTS selected more tests than were necessary and 74% of the time some tests that were affected were missed.

2.4 Effective use of CI

Continuous integration has been a fantastic addition to Software Engineering practices by removing repetitive administrative tasks. The less steps a software engineer has to go through when producing new code, the more time they can spend on adding features, squashing bugs, or otherwise improving the project code base. There is plenty of research done into the use of CI including many case studies, papers and presentations looking at how to setup CI and Continuous Deployment (CD), and papers looking at specific configurations of CI.

CI can be integrated with all sorts of tools to increase productivity. The paper How Open Source Projects use Static Code Analysis Tools in Continuous Integration Pipelines by Fiorella Zampett looks at how Static Code Analysis tools in a CI pipeline effects projects [15]. The paper found that the most utilized feature of ASCATs was to ensure a project's code was consistent by ensuring the developers coding guidelines were met. The paper also found that ASCATs caused broken builds to be fixed quickly in an average of 8 hours and one build. There are a number of recommendations that this paper provides which outline how to set up ASCATs in a CI pipeline, what to think about when doing so, and what to expect to maintain the ASCAT. Adding a source code analysis tool will help keep a project

consistent while helping point out bugs that could be missed by developer made unit tests.

Two papers look at CI and how it affects projects in general. Continuous Integration and Quality Assurance: A Case Study of Two Open Source Projects by Jesper Holck and Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects by Michael Hilton both look at open source projects and how CI affects them [8, 9]. The papers found that CI often replaces the practice of having developers make formal design documents. The developers instead just pick from a list of tasks and works on completing them and merging them into the code base. The papers also found that most developers like CI and plan to use it again in future projects. For projects that did not use CI it was found that the reason was usually just that the developers were not familiar with how to set up and use CI. Both papers found CI to be successful in increasing productivity, causing projects to release twice as often, accept pull requests quicker, and have developers less worried about breaking the project.

2.5 Security in CI pipelines

Security conferences often look into how to break systems as a way of pointing out the flaws of a configuration. CI servers have also had their share of security professionals testing for exploits and looking at the consequences of compromise. CI has also had a little bit of research into how to secure a CI pipeline.

A presentation at DEFCON named Exploiting Continuous Integration (CI) and Automated Build Systems talked about the consequences of a CI enabled projects [11]. The presentation found that exploiting a repo that holds a CI integration ends with a huge amount of access. If the repo links into an internal CI server, then the attacker ends up with internal network access. If the repo links to a CI server with multiple CI instances or also runs the CD, then the attacker will get more source code and access to the deployment machines because the CI holds a way to connect to the deployment servers. Otherwise the attacker ends up with environment variables which often hold extremely sensitive information.

A paper by Len Bass called Securing a Deployment Pipeline looks at how to secure a CI pipeline to limit the damage in case of exploitation [2]. The paper details a way to break a pipeline down into trusted and untrustworthy parts, segmenting operations until an untrustworthy segment cannot be broken down any more. The CI pipeline then holds parts that are guaranteed to be trustworthy and run as expected, minus specific cases outlined in the paper, and parts that may provide untrustworthy output. By limiting the scope of untrustworthy parts, the rest of the pipeline can run as expected and it is possible to see where the most risk lies. Then the owners of the pipeline can work to limit access to the untrustworthy portions and look into solutions to make those portions trustworthy.

2.6 Use of CI for Security

There is little research into the use of CI for security. One presentation by Mozilla looks at the use of CI to tackle easy fixes in response to their bug bounty program [14]. The presentation looks at using CI to ensure that the production environment contains configurations that mirror best practices for common web application security bugs. Some examples include HSTS is enabled, CSP for XSS bugs, various X-Options headers, Cookies have secure, Cross origin sharing, and Subresource integrity checks. The presentation recommends figuring out a security baseline for a projects CI pipeline, drive testing from the CI pipeline, and empower the team to fix the issues. Another recommendation is not to break on deployment into production as that could break the production site if configured poorly. The end result of mozilla's CI setup was a large drop in bug reports that the CI tests aimed to fix.

2.7 Work and methodology

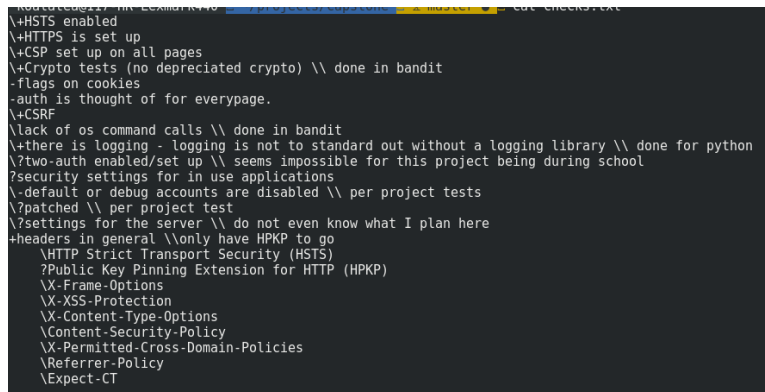
The work into CI so far has shown that CI is useful tool for developers to ease the creation of new features. CI has spread to open source projects and is deployed in most organizations that have at least one large code project. The security side of how a CI is dangerous if exploited and how to secure a pipeline against attacks has had little research. The most interesting thing that I think is lacking is the look into how CI can be used to improve the security the project that it is integrated on.

Most of the research does not mention how CI could help improve the security of a project. Some ways this could be done are through targeting common bugs that are already fixed. Some examples are ensuring that CSP is enabled, HSTS is enabled, parameterized queries are used, binary protections are enabled in compilation scripts, and unsafe functions are not used. Some of these are implemented already in some source code analysis tools, but these tools often have high levels of false positives. Another use could be in including fuzzing in a pipeline. I have not seen any research into automating fuzzing into testing an application. Depending on the project it could be a very useful tool to find bugs.

The research I am doing involves creating a pipeline with tests that focus on specific frameworks and are tailored towards an application instead of general language based tests. Most source code analysis tools look at specific languages instead of focusing on specific libraries or frameworks [1, 6, 5]. My tests will also do some dynamic testing instead of just focusing on source code analysis to verify that the protections are in place in a live application. I am following a similar path to Mozilla in using tests to mandate security configurations [14].

3 Project Implementation

The project's goal is a way to mandate security with low false positives. The start of this project involved creating a list of checks to implement as shown in Figure 1. The checks



```
+HSTS enabled
\+HTTPS is set up
\+CSP set up on all pages
\+Crypto tests (no depreciated crypto) \ done in bandit
-flags on cookies
-auth is thought of for everypage.
\+CSRF
\lack of os command calls \ done in bandit
\+there is logging - logging is not to standard out without a logging library \ done for python
\?two-auth enabled/set up \ seems impossible for this project being during school
?security settings for in use applications
\default or debug accounts are disabled \ per project tests
\?patched \ per project test
\?settings for the server \ do not even know what I plan here
+headers in general \only have HPKP to go
  \HTTP Strict Transport Security (HSTS)
  ?Public Key Pinning Extension for HTTP (HPKP)
  \X-Frame-Options
  \X-XSS-Protection
  \X-Content-Type-Options
  \Content-Security-Policy
  \X-Permitted-Cross-Domain-Policies
  \Referrer-Policy
  \Expect-CT
```

Figure 1:
The list of checks to implement

were chosen for both their ease of being checked and their security relevance. Tests that could be dynamic were prioritized over those that had to be static. The notation in Figure 1 checks is that a leading \ means the test is addressed, a + means that the test has been chosen to be implemented, a - meant the test would not be implemented, and a ? meant that there are still questions about the test. The majority of tests implemented are headers, with each one making specific bugs harder to exploit.

The implemented tests were all of the headers tests, and the logging test. The depreciated crypto test and the os test were both found to be done for python in the bandit source code analysis tools [1]. The HPKP header was dropped due to it's pending removal from chrome in favor of the new Expect-CT header [10]. The headers are chosen because they are easy to test for dynamically, the same rationale applies to csrf and https. The only static test that is created for this project is the logging test one because it does not require context like authentication being addressed for every web page. Figure 2 shows failing dynamic tests and Figure 3 shows passing dynamic tests. At the top of Figure 3 12 dots can be seen these represent the result of the tests in a failed case the . will instead be an F and if the test errors it will be an E.

The HTTPS test is tested by verifying that browsing to HTTP root redirects to an HTTPS root and then that doing a GET request against the HTTPS root returns a 200 response. Following the HTTPS test, HSTS is tested next by checking the Strict-Transport-Security header and that at least max-age is set, further parts could be included if the developer also wanted includeSubDomains or preload. CSP is tested in a similar way by checking that the Content-Security-Policy header is present and does not have unsafe-inline within any of the


```

=====
FAIL: test_Expected_CT (tests.headers_test.HeadersTest)
-----
Traceback (most recent call last):
  File "/home/koalateb/projects/capstone/base web apps/flask/tests/headers_test.py", line 63, in test_Expected_CT
    self.assertTrue('Expected-CT' in self.headers)
AssertionError: False is not true

=====
FAIL: test_Referrer_Policy (tests.headers_test.HeadersTest)
-----
Traceback (most recent call last):
  File "/home/koalateb/projects/capstone/base web apps/flask/tests/headers_test.py", line 54, in test_Referrer_Policy
    self.assertTrue('Referrer-Policy' in self.headers)
AssertionError: False is not true

=====
FAIL: test_X_Content_Type_Options (tests.headers_test.HeadersTest)
-----
Traceback (most recent call last):
  File "/home/koalateb/projects/capstone/base web apps/flask/tests/headers_test.py", line 57, in test_X_Content_Type_Options
    self.assertTrue('X-Content-Type-Options' in self.headers)
AssertionError: False is not true

=====
FAIL: test_X_FRAME_OPTIONS (tests.headers_test.HeadersTest)
-----
Traceback (most recent call last):
  File "/home/koalateb/projects/capstone/base web apps/flask/tests/headers_test.py", line 48, in test_X_FRAME_OPTIONS
    self.assertTrue('X-Frame-Options' in self.headers)
AssertionError: False is not true

=====
FAIL: test_X_Permitted_Cross_Domain_Policies (tests.headers_test.HeadersTest)
-----
Traceback (most recent call last):
  File "/home/koalateb/projects/capstone/base web apps/flask/tests/headers_test.py", line 60, in test_X_Permitted_Cross_Domain_Policies
    self.assertTrue('X-Permitted-Cross-Domain-Policies' in self.headers)
AssertionError: False is not true

=====
FAIL: test_X_XSS_Protection (tests.headers_test.HeadersTest)
-----
Traceback (most recent call last):
  File "/home/koalateb/projects/capstone/base web apps/flask/tests/headers_test.py", line 51, in test_X_XSS_Protection
    self.assertTrue('X-XSS-Protection' in self.headers)
AssertionError: False is not true

-----
Ran 12 tests in 0.391s

FAILED (failures=10)

```

Figure 2:
Example execution of failing tests

```

.....
-----
Ran 12 tests in 0.057s

OK

```

Figure 3:
Example execution of passing tests

Content-Security-Policy headers. The rest of the headers are implemented solely by checking for their existence. The headers have settings that are too different between projects so they are not tested for. The CSRF test is implemented using a list of dictionaries to keep track of end points. Each dictionary is of an endpoint that has a form to fill. The dictionaries have the endpoint, data to post, fail_text which is the text present if the form submission failed, success_text which is the text present if the form submission succeeds, a fail_status_code which is the status code for a failed submission, and a success_status_code if the submission succeeds. In the case of there not being one of the four success or fail codes or text then they are set to none. In the CSRF test for each endpoint the endpoint is queried and is

searched for an input html tag with an id of csrf.token. If the csrf.token exists the endpoint is POSTed to without the token. The response to this post is checked for each fail condition that is not none, and reach success condition that is not none is checked to make sure they are absent. Finally the csrf.token found earlier is added to the data and POSTed again with the checks from the POST without the token are reversed. The rest of the tests are static tests. Only one was implemented because it was found that the rest of the planned ones were already implemented. The os tests are done for potential inject points using os, which is slightly different from the original goal in the checks list, but the crypto one does exactly what is being looked for by checking for depreciated crypto. The written test was written as a plugin for bandit, it checks for any print statements and recommends using a logging library instead. This one exists partially because most developers already use logging libraries for most projects, the other reason is for proper logging management to avoid resource exhaustion.

All of the dynamic tests were easy to transfer over to java, only the url and the requests library had to be changed and the test functioned that same. The caveat is that the java application had to be dockerized so that it could be tested with the python code, while the python tests worked by using flasks built in debug client. The static tests are not as easy to transfer over. To use the static test the already implemented tests either needed to be found in another static code analysis tool or written again for java.

These tests were then added to a CI pipeline in Travis. To do so build stages were used for parallelizing the tests. In this project this made sense because there were two passing and two failing builds. Figure 4 is the travis.yml configuration file. The configuration involved install steps that installed docker-compose. This is required for running the dynamic tests against java, since it cannot just hook into the internal testing tools like with flask. There are also settings for what version of python this project uses, that sudo is required, and docker needs to be installed. Then the stages are defined. There are two stages for python and two for java. The TEST environment variables are set for visibility into what each build is doing from the travis webpage. In both python builds, first bandit is run against it with the tests that address the use of subprocess, os, and crypto. Next the unit tests are run against the flask. For the java, first docker is set up and then the unit tests are run against them. If any of the static tests trigger or the unit tests fail, then the entire build will fail. Figure 5 shows an example of the builds being ran in travis.

```
Activities Terminal Mon 21:01 vim:travis.yml
File Edit View Search Terminal Help
1 .travis.yml
sudo: required
language: python
python: "3.6"

env:
  DOCKER_COMPOSE_VERSION: 1.19.0


services:
  - docker

before_install:
  # Make sure everything's up to date.
  - sudo apt-get update -qq
  - docker-compose --version
  - sudo rm /usr/local/bin/docker-compose
  - curl -L https://github.com/docker/compose/releases/download/${DOCKER_COMPOSE_VERSION}/docker-compose.`uname -s`.`uname -m` > docker-compose
  - chmod +x docker-compose
  - sudo mv docker-compose /usr/local/bin/

install:
  - pip install -r requirements.txt


jobs:
  include:
    - stage: tests
      env: TEST=passing_python
      script:
        - find passing_apps/flask -name "*.py" -exec bandit -t B307,B304,B305,B505,B501,B502,B503,B504,B602,B603,B604,B605,B606,B607,B609 '{}' +
        - python passing_apps/flask/run_tests.py
    - env: TEST=pass_java DOCKER_COMPOSE_VERSION=1.19.0
      script:
        - cd passing_apps/java && docker-compose pull
        - docker-compose build
        - docker-compose up -d
        - cd ../..
        - sleep 10
        - python passing_apps/java/run_tests.py
    - env: TEST=base_python
      script:
        - find base_web_apps/flask -name "*.py" -exec bandit -t B307,B304,B305,B505,B501,B502,B503,B504,B602,B603,B604,B605,B606,B607,B609 '{}' +
        - python base_web_apps/flask/run_tests.py
    - env: TEST=base_java DOCKER_COMPOSE_VERSION=1.19.0
      script:
        - cd base_web_apps/java && docker-compose pull
        - docker-compose build
        - docker-compose up -d
        - cd ../..
        - sleep 10
```


Figure 4:
The travis configuration file


KoalaTea / capstone-security-unit-tests  Build **error**

Current Branches Build History Pull Requests **Build #24** More options

✗ travis fixed docker so that it maps correctly

↔ Commit aede4a3 

🔗 Compare f510d82..aede4a3 

🔗 Branch travis 

👤 koalatea authored and committed

→ #24 failed

🕒 Ran for 3 min 12 sec

🕒 Total time 7 min 32 sec

🕒 less than a minute ago

[Restart build](#)

Beta Feature Thank you for trying Build Stages! [We'd love your feedback](#)

🔍 Tests 🕒 3 min 12 sec

✓ # 24.1	🔗 Python: 3.6	🔗 TEST=passing_python	🕒 48 sec	🔄
✓ # 24.2	🔗 Python: 3.6	🔗 TEST=pass_java DOCKER_COMPOSE_VERSION=1.19.0	🕒 3 min 12 sec	🔄
✗ # 24.3	🔗 Python: 3.6	🔗 TEST=base_python	🕒 37 sec	🔄
✗ # 24.4	🔗 Python: 3.6	🔗 TEST=base_java DOCKER_COMPOSE_VERSION=1.19.0	🕒 2 min 55 sec	🔄

Figure 5:
Builds being ran in travis

4 Conclusions

The project involved creating two applications that were mirrors of each other in functionality. The applications were created as tests were decided to create appropriate functionality for the tests. One application was written in python with flask as the web framework, while the other one was written in java with spring as the web framework. Both applications had a home page and a page for POSTing data to a form which would return success if the form was POSTed successfully and failed if it was not. This was all the functionality needed for the tests involved. All applications were then dockerized. Then the tests were created, which lead to editing the base applications to also have a passing version. Finally the repository for this project was CI enabled using travis which a travis-ci.yml file. This was all to emulate a corporations environment, there would be an application hooked up to a CI pipeline. Every merge request to master would kick off a build job to run the tests and block on failure of those jobs. The goal of this project in this environment is to efficiently mandate security configurations on web applications in this environment. This would ensure expected configurations were in place and decrease attack surface of the application. To do so, the following is recommended when setting up a security focused pipeline.

- Prioritize creating dynamic tests, these can be used across projects and language
- Create policies that outline the expected configuration based on the dynamic tests and mandate them across projects
- Create a method for containerizing all applications developed so that they can easily be tested with the dynamic tests
- Include a static analysis tool that focuses on security and the language the project is in into the pipeline
- If static tests are needed write them using the static analysis tool chosen
- Seperate the security pipeline from the normal unit test pipeline so that it is obvious whether the unit tests or the security pipeline failed

Further work into this field could look into the rules around where to apply a security pipeline and how to have different levels besides just passing and failing builds. Another interesting field that would be helpful to this pipeline is automatic test selection so that when a change is made to the code, only the tests that are relevant to that code are run. Both of these would be helpful to furthering the use of CI for security testing.

References

- [1] *Bandit python source code analysis tool*. URL: <https://github.com/openstack/bandit>.
- [2] Len Bass. “Securing a Deployment Pipeline”. In: *IEEE/ACM 3rd International Workshop on Release Engineering* (2015), pp. 4–7.
- [3] Andrew Case. “Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks”. In: *DFRWS USA d Proceedings of the Seventeenth Annual DFRWS USA* (2017), S86–S93.
- [4] Amin Milani Fard. “Leveraging Existing Tests in Automated Test Generation for Web Applications”. In: *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2014), pp. 67–78.
- [5] *Find Security Bugs java source code analysis tool*. URL: <https://find-sec-bugs.github.io/>.
- [6] *FindBugs java source code analysis tool*. URL: <http://findbugs.sourceforge.net/>.
- [7] Milos Gligoric. “An Empirical Evaluation and Comparison of Manual and Automated Test Selection”. In: *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2014), pp. 361–372.
- [8] Michael Hilton. “Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects”. In: *International Conference on Automated Software Engineering (ASE)*. (2016), pp. 426–437.
- [9] Jesper Holck. “Continuous Integration and Quality Assurance: a case study of two open source projects”. In: *Australasian Journal of Information Systems 11(1)* (2012), pp. 40–53.
- [10] *Intent To Deprecate And Remove: Public Key Pinning*. URL: <https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/he9tr7p3rZ8/eNMwKPmUBAAJ>.
- [11] spaceb0x. *Exploiting Continous Integration (CI) and Automated Build Systems, DEF-Con 25*. URL: <https://media.defcon.org/DEF\%20CON\%2025/DEF\%20CON\%2025\%20presentations/DEFCON-25-spaceB0x-Exploiting-Continuous-Integration-UPDATED.pdf>.
- [12] *Testing at the speed and scale of Google, 2011*. URL: <http://goo.gl/OKqBk>.
- [13] *Tools for continuous integration at Google scale, 2011*. URL: <http://www.youtube.com/watch?v=b52aXZ2yi08>.
- [14] Julien Vehent. *Test Driven Security in Continuous Integration*. Presentation. Enigma. 2017. URL: <https://www.usenix.org/conference/enigma2017/conference-program/presentation/vehent>.

- [15] Fiorella Zampetti. “How Open Source Projects use Static Code Analysis Tools in Continuous Integration Pipelines”. In: *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (2017), pp. 334–344.