When Continuous Integration meets Application Security

Vasantharaju MS, Harish Krishnan

vasantharaju.ms@intel.com, harish.krishnan@intel.com

Abstract

Continuous Integration (CI) is not specific to extreme programming anymore, and most organizations which follow any form of Agile practice for product development adopt CI. The two major benefits that organizations derive from CI are build verification, and test automation. Test automation in CI usually covers Unit Testing, Functional Testing, Integration Testing, etc. Every check-in a developer makes is tested automatically to make sure the build is verified and functional. But what about the security of the product?

In most cases, security testing is not automated, but performed manually on a milestone build or, worse, at the end of project. As the number of companies offering Software as a Service (SaaS) grows, and reliance on a Continuous Deployment or Continuous Delivery pipeline increases, every change can be potentially deployed to production and there is an increased risk to companies in terms of application security. There is a pressing need to do security testing more often or, better yet, on every change.

What if you could run security test cases, security scanners and other security related build verifications during CI? This is where Continuous Integration meets Application Security. For example, static code analysis with security checkers or a defense to specific vulnerabilities like Blind SQL Injection, Cross Site Scripting (XSS) and other vulnerabilities can be verified in integration testing or an audit for vulnerable 3rd party libraries can be conducted on top of build created by CI. Automating security testing leverages the integration testing platform provided by CI to ensure application security. Of course, we can only deliver secure solutions as fast as we can test them, adding security testing to your existing CI capabilities will help in achieving this goal.

This paper covers the processes and tools required to automate security testing in Continuous Integration based on our learning and experience.

Biography

Harish Krishnan is a Product Security Champion at Intel Security Group with over 10 years of experience in Quality and Security domain. His primary responsibility is Product Security. His daily duties include performing security audits, architecture design review through Threat Model, Privacy review, Penetration testing, analyzing vulnerabilities and writing security Bulletins. He has also authored an in-house proprietary web application security scanner tool which many Intel security enterprise products run regularly to uncover any vulnerabilities.

Vasantharaju M.S. is a Senior Software Development Engineer at Intel Security Group with over 8 years of experience in developing enterprise-class software. His past experiences include; designing and implementing back-office solution to address a Dodd-Frank clause, enhancing web crawler and vulnerability detection in SaaS based vulnerability scanner and executing multiple Proof-of-Concepts for enhancing security management platform. He has a keen interest in defensive programming.

1 Introduction

Continuous Integration (CI) is a software development practice of merging all developer working copies to a shared mainline several times a day. Each iteration is verified by an automated build along with running several automated product feature tests to detect integration errors as quickly as possible.

There are several CI platforms available out there. In this paper we will be sharing our learning and experience working with JetBrains TeamCity [1] CI server.

Application security, in simple terms, is testing the security of an application, which can be achieved to some extent by running tests using a few security tools that are available. Our experiences that we are sharing in this paper include both commercial and free security tools. We use the following tools, which are described in much detail in later sections, to ensure the security of our Java based Web Application.

1. Static Analysis: Coverity [2] and FindBugs [3]
2. Network and Dynamic Analysis: Nessus [4] and some proprietary scanning tools
3. 3rd party component security audit: OWASP Dependency Check [5]

Though a few of the tools mentioned above are specifically for use with Java based applications, the process we describe can still be leveraged using alternative tools applicable to software applications built using a different technology stack.

When such tools are automated and are run as part of CI, then this is where Continuous Integration meets Application Security, detecting security issues as early as possible in the product development life cycle. When any security issue is found as part of CI tests run, then the build in TeamCity will be marked as failed indicating that the new code that was merged introduced this issue.

The benefit of having such CI setup for security testing is that many vulnerabilities are found and fixed early in the product development life cycle. Fixing issues earlier in the product life cycle is much easier and less costly in terms of rework needed and the number of engineers involved, than towards the end of project thereby saving us lot of time. Also during the product release, this gives us more confidence on the overall security of the product.

The paper is structure as follows. Section 2 talks about Static analysis, how to enable only security checkers in the FindBugs static analysis tool and how to run the tool. It also provides typical build steps needed for CI. Section 3 describes the Nessus scanner, a few examples on Nessus automation and again the typical build steps needed for CI. In Section 4, we mention some of the generic Web Application vulnerability scanners and the general build steps needed to integrate with CI. Lastly, in Section 5 we talk about how we perform the third party component security review.

2 Static Code Analysis with Security Checkers

Static code analysis is done using automated tools to examine source code that is instrumented and built to some form of object code without actually executing the program. Static code analysis tools are available for different programming languages like Java, C++ and also provide plugins for many IDEs (Integrated Development Environment), so that developers have a chance to run static code analysis during the development phase itself before the code is checked-in into the source repository. These tools find both quality and security issues in the code, but our focus here is only on security issues.

It is highly recommended to have a separate build configuration that runs static code analysis dedicated to running security checks. This separate build configuration can assure the software development team that for every check-in, the code is free from security flaws or malicious code. Additionally, when many such

security flaws are identified during this CI build, it will eventually encourage the developers to use plugin version of static code analysis tools to help them find such issues before the code is checked-in.

We use two static code analysis tools. Coverity, one of the commercial static code analysis tools and the other one is FindBugs, a very popular free tool for java static code analysis.

2.1 FindBugs

FindBugs is a free program which uses static analysis to look for bugs in Java code.

2.1.1 Configuring FindBugs to report only security issues

There is a useful feature in FindBugs called 'Filter files' using which the user can configure to include or exclude a particular bug instance/category in the report. A filter file is an xml document and here is an example filter file to report only items that match an issue category of 'security' when FindBugs static analysis is run:

```
<?xml version="1.0" encoding="UTF-8"?>
<FindBugsFilter>
  <Match>
   <Bug category="SECURITY"/>
  </Match>
</FindBugsFilter>
```

2.1.2 Running FindBugs

FindBugs static analysis tool can be run in multiple ways, either using the application GUI, the command line interface or using an Ant [6] task.

We run this using an Ant task method because TeamCity supports the Ant scripts as one of the build steps. Also, most of our other TeamCity build steps are Ant scripts so this way of running was our first choice to maintain the same pattern. Following is a sample Ant build script for running FindBugs static analysis:

```
<project name="FindBugs" default="findbugs" basedir=".">
<taskdef                                                name="findbugs"
classname="edu.umd.cs.findbugs.anttask.FindBugsTask"/>
<property name="findbugs.home" value="${basedir}/findbugs-3.0.1" />
  <target name="findbugs">
    <findbugs home="${findbugs.home}"
              output="xml"
              outputFile="result.xml"
      includeFilter="${basedir}/secFilter.xml" >   <!—this is the above created
      filter file to include only security issues→
      <sourcePath path="${basedir}/src" />
     <class location="${basedir}/lib/*.jar" />
    </findbugs>
  </target>
</project>
```

Store the above xml in 'build.xml' file and run the following command on the command prompt from the directory where this 'build.xml' is stored:

C:\> ant findbugs

2.2 Typical build steps needed for CI

- Checkout the code and build.
- Run Static code Analysis with security checkers enabled as shown above.
- Automatically analyze the scan result. Following are the details on how we achieve this -
    - The scan report is generated in xml format as we instructed in the ant build.xml file above. Also the report includes only security issues, if any.
    - Programmatically we parse this xml file and look for any bugs being reported. A typical FindBugs report includes a <BugInstance> element for every issue it finds. Following is an example from the report:

```
<BugInstance category="SECURITY">
   <Class classname="">
<SourceLine classname="" sourcepath="" sourcefile="" end=""
start=""/>
   </Class>
</BugInstance>
```

- Finally, mark the build PASS/FAIL based on the analysis done in the previous step i.e. fail the build if any bug instances found in the report file.

3 Network Scanners – Nessus

Nessus is a popular and widely used vulnerability assessment solution. This tool has a wide variety of tests to choose from and we mainly use it for Web Application testing. Writing a basic framework to integrate with Nessus API's to automate Nessus scanning provides an abstraction layer making it easy to perform scanning related operations. For our application, we used nessrest [7] framework to integrate with Nessus REST API's.

3.1 Example Usage of this python framework

Here are few examples using the APIs provided by the nessrest framework on how to connect to the Nessus server, how to run the scan and how to download the report. You can see with this framework now, how easy it is to connect to a Nessus server.

```
# Connecting to Nessus Server
from nessrest import ness6rest as nes

URL = 'https://' + NESSUS_HOST + ':' + NESSUS_PORT
log("Connecting to Nessus Server: %s" % URL)
try:
   # Insecure is set to True for allowing self-signed certificates.
   scan = nes.Scanner(url=URL,
                      api_akey=NESSUS_ACCESSKEY,
                      api_skey=NESSUS_SECRETKEY,
                      insecure=True)
   log("Successfully connected to Nessus")
except Exception, e:
   log("Could not connect to Nessus Server: %s" % str(e))
   sys.exit(1)
```

```
# Running the scan
scan.policy_set(POLICY)
scan.scan_add(targets=TARGETS, name=scan_name)
log("Running the scan....")
scan.scan_run()

# Downloading the scan report
content = scan.download_scan()
uid = scan.scan_uuid
output_path = os.path.join(os.getcwd(), 'reports', uid + '.nessus')
f = open(output_path, 'w')
f.write(content)
f.close()
log("Download complete - %s" % output_path)
```

3.2 Typical build steps needed for CI

- Checkout the code, build and deploy the application.
- Run the automation scripts written on top of the above python framework to trigger the Nessus scan.
- Once the scan is complete, automatically download the reports using the automation scripts. (Note: We store these reports in TeamCity server under Artifacts)
- Automatically analyze the report. Following are the details on how we achieve this -
    - Other than providing the APIs to download the scan report, the framework also provides an API for parsing the downloaded report.
    - The Nessus report lists vulnerabilities with following severities – Critical, High, Major, Low and Informational. You need to decide what type of severity issues you are interested in, so based on that you can PASS/FAIL the build.
    - We ignore Informational issues. Examples of Informational issues reported are, Hyper Text Transfer Protocol (HTTP) information, Self-Signed SSL (Secure Socket Layer) certificates are used, etc.
    - Here is an example snippet (not an exact working code) on how we achieve this –

```
report = PyNessusFramework.Report()

# parsing the downloaded report.
report.parse(downloadedReportFile)

# getting the target on which the scan was run.
target = report.targets[0]

# getting the list of vulnerabilities found on that target.
vulnerabilities = target.vulns

# creating empty list to store severities of the vulnerabilities.
severityOfVulnerabilities = []

# populating the list with severities of the vulnerabilities
for i in vulnerabilities:
    severityOfVulnerabilities.append(i.get('severity'))

# initial status
```

```
vulnerabilityFound = False

# If any vulnerability with severity >= Medium (i.e. med (2), high
(3), critical (4))
if    ('2'    in    severityOfVulnerabilities)    or    ('3'    in
severityOfVulnerabilities) or   ('4' in severityOfVulnerabilities):
    vulnerabilityFound = True
    log("VULNERABILITY FOUND = %s" % vulnerabilityFound)
else:
    log("VULNERABILITY FOUND = %s" % vulnerabilityFound)
```

- Finally, mark the build PASS/FAIL based on the analysis done in the previous step, i.e. fail the build if “VULNERABILITY FOUND = True” message is logged in the build logs.

4 Web Application Vulnerability Scanners

Web Application Vulnerability Scanners are tools that automatically scan web applications for known security vulnerabilities like cross-site scripting, SQL injection, cross-site request forgery, directory traversal, etc. Web Application Vulnerability Scanners can be configured to automatically crawl the web application whilst authenticated or unauthenticated and scan all discovered links and/or forms for vulnerabilities. There are many Web Application scanners that are available both in commercial and free versions.

Some of the well-known commercial web security scanners are Rapid7 [8], Burp Suite Pro [9], HP Web Inspect [10] and Acunetix [11]. Some of the free utilities also available are OWASP ZAP [12], Nikto [13] etc.

We have not integrated any of the above tools; instead, we use a proprietary web application security scanner to test our product. Like many other tools, even our proprietary scanner provides APIs so that scanning operations like triggering a new scan, downloading of reports, etc. can be automated using scripts.

We have automated the security tool in the PowerShell scripting language, which is supported by the TeamCity build steps.

Once we have an automated security-scanning tool, as mentioned in previous section, the following are the typical build steps for CI.

4.1. Typical build steps needed for CI

- Checkout the code, build and deploy the application
- Run the automated security scanning tool which will crawl the application links and analyze for vulnerabilities
- Automatically analyze the report similar to the previous section.

Finally, mark the build PASS/FAIL based on the analysis done in the previous step.

5 Third Party Component Security Review

Third party component security review involves identifying project dependencies and reviewing known or publicly disclosed vulnerabilities for identified project dependencies. Reviewing the software for third party components for known or publicly disclosed vulnerabilities is typically done in a milestone build and removing a third party library or framework after triaging published vulnerabilities involves a considerable

amount of time and effort. The major challenges in reviewing the third party components for known vulnerabilities are;

1. Identifying third party libraries, modules and frameworks and their corresponding version included in the software
2. Triaging published vulnerabilities for identified third party components

OWASP Dependency-Check tool automatically identifies project dependencies and reports any known or publicly disclosed vulnerabilities for the identified project dependencies. Currently Java, .NET, Ruby, Node.js, and Python projects are all supported by OWASP Dependency-Check. Having a build configuration in CI to run OWASP Dependency-Check helps to review project dependencies for known or publicly disclosed vulnerabilities and also helps in preventing vulnerable third party component to be added as project dependency.

Kicking off the build for every check-in would not be necessary since 3rd party libraries are not checked in frequently so kicking off a build for a check-in which includes 3rd party components will be more efficient and CI tools have provisions to trigger builds based on specific files based on Regex like *.jar in case of java project. Scheduling a nightly build also helps in finding recently published vulnerabilities for third party components which are already white listed, since there can be new CVE added for a third party component.

Triaging vulnerabilities published for known components certainly needs application security expertise to determine possible attack vectors and take a call to suppress the finding or remove the third party component and looking for alternatives. In the case of suppressing findings due to false positive results or ignoring for a valid reason after triaging, Suppression.xml can be used as below – this suppresses specific CVE (Common Vulnerability and Exposures) for a tomcat library catalina-ha.jar

```
<?xml version="1.0" encoding="UTF-8"?>
<suppressions
xmlns="https://www.owasp.org/index.php/OWASP_Dependency_Check_Suppression">
<suppress>
      <notes><![CDATA[
      file name: catalina-ha.jar
      ]]></notes>
      <sha1>d68cfd77bd369975cf4e51b07b0d66707a1af656</sha1>
        <cve>CVE-2013-2185</cve>
</suppress>
</suppressions>
```

OWASP Dependency Check reports have provisions to create the suppression xml dynamically based on the requirements so there is no need to create the suppression xml manually. After creation, this suppression xml is used to suppress findings in subsequent scans.

5.1 Typical build steps needed for CI

- Checkout the code, build and package.
- Explode the package and copy third party libraries like Jar/DLL into separate directory
- Run Dependency Checker tool to analyze third party libraries for published vulnerabilities.
    - Dependency-Check tool can be run in many different ways, one way is to run OWASP Dependency-Check Ant task as part of ant build script.

- Analyze the build report for failure conditions. For example OWASP Dependency-Check ant task can be configured to fail the ant task call if CVSS of finding is more than 4 or any number ranging 0-11.
- Finally, mark the build PASS/FAIL based on the analysis done in the previous step.

6 Conclusion

Most of the time security testing is performed manually on a milestone build or towards end of the project. If any vulnerabilities are uncovered later in the project, then it is very expensive to fix the issue as the product is already built to some extent, thereby consuming more time to resolve the issue.

As one can see from reading the information presented, automating security testing and then running those tests as part of a continuous integration workflow, many vulnerabilities can be found and fixed early in the product development life cycle, which is much easier than towards the end of project, thereby saving us lot of time and expense.

With around 10 - 12 weeks of efforts from 2 persons, we were able to build this entire CI system for security testing. Some of the immediate benefits we saw by having this setup were:

a) By performing a 3rd party security audit regularly as part of CI, new CVEs (Common Vulnerability and Exposures) that get published were tested almost every day. Around a dozen libraries that we were using was found to have some vulnerabilities and we fixed them immediately before it was too late.
b) In the past, we used to get many customer escalations on some security issues, which they found by running similar, or the same security tools on our product. This has been reduced by 95% as we are also proactively running these tools and fixing all the legitimate issues.
c) As part of performing static analysis, we have uncovered and fixed hundreds of resource leaks in our product, which when present, can lead to even DOS (Denial of Service) vulnerability in the product when all system memory is used up.

The trickiest part of running security tools in CI environment was how to automatically analyze the scan report. Even though we did our best in automatically analyzing the report and marking the build status accordingly, we have seen around 1/25 times, manual assessment was needed to make sure the build was marked accurately.

Overall, by performing the Application Security testing as part of CI, we are confident that we are releasing a more secure product than before

References

Web Sites

[1] TeamCity : https://www.jetbrains.com/teamcity/

[2] Coverity : http://www.coverity.com/

[3] FindBugs : http://findbugs.sourceforge.net/

[4] Nessus : https://www.tenable.com/products/nessus-vulnerability-scanner

[5] OWASP Dependency Check: https://www.owasp.org/index.php/OWASP_Dependency_Check

[6] Ant : http://ant.apache.org/

[7] Nessus Frameworks : https://github.com/tenable/nessrest
: https://python-nessus.readthedocs.io/en/latest/
: https://github.com/Adastra-thw/pynessus-rest

[8] Rapid7 : https://www.rapid7.com/products/appspider/

[9] Burp Suite pro: https://portswigger.net/burp/

[10] HP Web Inspect: http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast/

[11] Acunetix : http://www.acunetix.com/vulnerability-scanner/

[12] OWASP ZAP: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[13] Nikto : https://cirt.net/Nikto2