Testing Infer Quandary as a vulnerability analysis tool via confusion matrix perturbation analysis

ICT Risk Assessment final project

Giovanni Bartolomeo and Laura Bussi

1 Introduction

When dealing with security issues, we are provided a pletora of tools for the analysis of possible vulnerabilities in software. Many of them analyse the behaviour of the software at run time, but static analysis can also be performed: several tools can analyse source code and find possible flaws before the program is running.
Of course, both this kind of approaches cannot be 100% accurate. Most likely, they will provide as a result a set of possible vulnerabilities which intersect the set of the actual ones, i.e. for each pointed out vulnerability we will have four possible cases:

- **True Positive**: tool correctly identifies a real vulnerability
- **False Negative**: tool fails to identify a real vulnerability
- **True Negative**: tool correctly ignores a false alarm
- **False Positive**: tool fails to ignore a false alarm

From this classification, we obtain a 2×2 matrix, namely a confusion matrix. Confusion matrix can be a useful tool to benchmark a security analysis tool capabilities.

1.1 OWASP Benchmark

The OWASP benchmark project[1] is an executable web application, provided as a Maven project, written in Java using the javax framework. It contains several Java files (slightly less than 3000): each one of this files is a Java servlet wich can contain either a true vulnerability or a false one. As the project is provided as a public repository on GitHub, one can run both dynamic application security testing tool or static vulnerability analysis tools against it.
Of course, we are also provided with a csv containing the expected results for each test case, meaning that for each test we have:

- The name of the test case;
- The vulnerability area;
- A boolean flag indicating if the vulnerability is a true or a false positive;
- The CWE number of the vulnerability.

Once the analysis is complete, we are provided with a score, called the Benchmark Accuracy Score, in the range 0-100. This score is a Youden index, computed as:

$$J = sensitivity + specificity - 1$$

[1] https://github.com/OWASP/Benchmark

where in turn we have:

$$sensitivity = TP/P = TP/(TP + FN)$$

and

$$specificity = TN/N = TN/(TN + FP)$$

Thus, we have that sensitivity represents the ability of recognizing a true positive, while the specificity is the ability of correctly identifying true negative. Looking at the formulas, it is clear that a tool which label each line of code as a vulnerability has a very low sensitivity, as the number of False Negatives is very high. At the very same way, a tool which does not recognize any vulnerability has sensitivity 0, as the TP factor is nullified. A similar consideration can be done for specificity.

1.2 Infer Quandary

Facebook Infer[2] is an open source static code analyser written in OCaml. Based on abstract interpretation, it is not primarly intended to be used to discover security issues but, more generally, possible errors in the source code. This obviously means that it can shows several limitations for the former kind of purpose: however, it seems interesting to test it against the Owasp benchmark, in order to see how can it be used and improved for security.
Infer is provided with a plugin, namely Quandary[3], devoted to the taint analysis of the source code. Quandary can be configured by providing a JSON file, where one can define:

- a list of sources for the input data;
- a list of sink procedures, which may use tainted inputs;
- a list of sanitizer procedures;
- a list of endpoints.

As the OWASP benchmark is a web application, tainted inputs come from servlet procedures, which are not included in the Quandary configuration by default. This means that, at a first run, Quandary got a score 0: in the next sections we are providing a configuration which can improve Infer Quandary's performances in a web environment.

2 Project structure

This project is composed by a set of Python scripts that run the benchmark, analyse the results and create the confusion matrix along with some statistics. The generation of the final results can be accomplished in 2 steps running the following scripts:

[2] https://fbinfer.com/
[3] https://fbinfer.com/docs/next/checker-quandary/

1. `run_test.py` runs the Maven compilation of the benchmark together with the Infer quandary tool and exports the results. Note that Infer is launched with the option –quandary-only, thus the results are all and only those obtained by the Quandary plugin.

2. `confusion_builder.py` for each exported result from the previous step, prints statistics and builds the confusion matrix.

2.1 Custom Infer configurations

As mentioned before, it is possible to customise the analysis of vulnerabilities performed by Quandary providing an ad-hoc configuration file. Here it is possible to setup some custom sinks, endpoints, sources and sanitizers. In the project, we provide this configuration in the file *.inferconfig*. We focus here on the definition of sanitizing procedure, as the one for sources and sinks is quite straightforward.
We declare as sanitizers the following procedures:

- *decodeForHTML*, *decodeFromBase64*, *decodeFromURL*, *canonicalize* and *encodeForHTML* defined in org.owasp are pretty standard, and can be used to remove dangerous characters from the input;

- as unsecure random number generation is a flaw, the java standard Random is not considered to be a sanitizer. Instead, we allow as a sanitizer the *SecureRandom* procedure defined in the java.security framework;

- encryption using SHA is always considered to be a sanitizer. Note that, as we have no way to specify which SHA algorithm are going to use, Quandary is not able to report a vulnerabilty caused by the usage of an unsecure algortihm, as for instance SHA1. A similar consideration can be done for the last sanitizer, the *MessageDigest* procedure defined in java.security: more on this later on.

In order to run the test with a different configuration it is only needed to edit this configuration file and run again the tool as descrived above.

2.2 Compare the results

Each time the `run_test.py` script is run, a `csv/actual/actual.csv` file is generated or overwritten. The csv file mirrors the one provided in the owasp benchmark, hence having the fields:

- *filename* the name of the java test file;

- *vulnerability name* the name of the vulnerability considered by the test;

- *vulnerability presence* a boolean flag indicating wheter the vulnerability is present or not;

BenchmarkTest00001	pathtraver	true	22
BenchmarkTest00002	pathtraver	true	22
BenchmarkTest00003	hash	true	328
BenchmarkTest00004	trustbound	true	501
BenchmarkTest00005	crypto	true	327
BenchmarkTest00006	cmdi	true	78
BenchmarkTest00007	cmdi	true	78
BenchmarkTest00008	sqli	true	89
BenchmarkTest00009	hash	false	328

Figure 1: The expected results csv

- *CWE number* the number associated to the vulnerabilty in the Common Weakness Enumeration.

The structure of the file is shown in fig. **??**.

Note that, as the vulnerability name and CWE number are not useful for the analysis, we fill this fields with dummy values which will be ignored while building the results csv. In order to compare different results from different configurations of the Quandary tool and assess the results, it is possible to rename this file and run again the script: the new configuration is automatically recognized by Quandary. In this case, a new `csv/actual/actual.csv` file will be generated and the old one won't be overwritten. As the confusion matrix builder can build up the matrix for different csv files, running again the script will results in getting separate statistics according to all the `.csv` files placed inside the `csv/actual` folder, hence easing the comparison between different configurations.

3 Benchmark results

As said, we can set up Quandary in such a way that it can perform several tests on tainted inputs, and check whether they are sanitized or not, by defining a JSON configuration file. We run Quandary on the benchmark several times: as the first run didn't show any issue, thus scoring 0, the very last configuration obtained a score of 29.4 [Fig. 2]. This score has been obtained by iteratively upgrading the Infer's configuration file according to the analysis performed into the benchmark code. For each new sink, source or sanitizer discovered, the configuration file has been updated.

This value is still not satisfactory in absolute terms, hence it deserves to be analyzed deeper, in order to see if such a bad result is due to inherent limitations of Quandary or if we can refine the configuration further. First of all is important to notice that the number of correctly discovered vulnerabilities is high with respect to the undetected vulnerabilities, namely the false negative. What actually made the final score to drop significantly are the cases where a vulnerability has been detected erroneously.

	Positive	Negative
Total Population	1415	1325

	Condition Positive	Condition Negative
Predicted Condition Positive	1136	674
Predicted Condition Negative	279	651

Final Score:	29.41476098406559
Sensitivity:	0.803
Specificity:	0.491

Figure 2: The Confusion Matrix

Top 3 False Negative misclassification by Vulnerability type

Vulnerability name	Relative Incorrect classification %	Absolute Incorrect classification %
crypto	52.846	46.595
hash	12.712	10.753
securecookie	11.940	2.867

Top 3 False Positive misclassification by Vulnerability type

Vulnerability name	Incorrect classification %	Absolute Incorrect classification %
securecookie	37.313	3.709
hash	33.051	11.573
cmdi	32.271	12.018

Figure 3: Misclassification ranking

This is due to a limitation in the Quandary configuration, as we basically can only specify a sanitizer class withhout checking any of its parameters or properties. This obviously limits the possibility of checking that the sanitizing procedure is effective, then affecting the results.
In order to better understand which are the main sources of misclassifications, it is possible to look at the benchmark's vulnerability ranking generated as a result of the analysis [Fig. **??**]. From these ranking, we can get the list of the vulnerabilities which appears to be reported more frequently as a false negative or a false positive.

3.1 Securecookie

Let's begin with the analysis of the secure cookie vulnerability, classified as CWE-614. This vulnerability must be reported when

> "The Secure attribute for sensitive cookies in HTTPS sessions is not set, which could cause the user agent to send those cookies in plaintext over an HTTP session." [4]

This appears to be in the top three both as a false positive and as a false negative. Indeed, as we set up the sanitizers, we can only specify the name of

[4] https://cwe.mitre.org/data/definitions/614.html

some classes which will be accepted as sanitizers. What Quandary checks is if the tainted input is treated somewhere by one of these classes.
However, there are two cases which can be misleading:

- The cookie data pass through a sanitizer, but the cookie hasn't the secure flag activated. In this case, Quandary will not recognize a vulnerability (false negative);
- The cookie data does not pass through a sanitizer, but the secure flag of the cookie is activated. Quandary will report this as a vulnerability, hence giving a false positive.

3.2 Hash

The reversible one-way hash vulnerability CWE-328 happens when:

> "The product uses a hashing algorithm that produces a hash value that can be used to determine the original input, or to find an input that can produce the same hash, more efficiently than brute force techniques." [5]

The algorithm used for the hashing is a parameter that must be set to the MessageDigest class. Even if a list of the vulnerable hashing algorithm does exist, it is impossible from the quandary configuration file to discriminate a class from its internal state. In fact, the only possible configuration is the one where it's implied that the update method of a MessageDigest class is a sanitizer, even though it could use an insecure hashing algorithm. This lead to a high number of false negatives, because even if an input is going trough an insecure hash function, it is classified as sanitized. There are also a lot of false positive classifications for this vulnerability. For what we were able to analyse, mostly they are due to the fact that the benchmark logs back to the controller the un-hashed input value, this is recognised from the actual configuration as an attempt to expose sensitive data.

3.3 Crypto

The benchmark code can be affected from the CWE-327 because

> "The use of a broken or risky cryptographic algorithm is an unnecessary risk that may result in the exposure of sensitive information." [6]

In particular, inside the benchmark code, it is possible to find out several times the use of insecure algorithms for the KeyGenerator class. Once again, as for the CWE-328, we are not able to discriminate a sanitizer by its internal state and this leads to a high number of false negatives, as all the crypto algorithms are considered secure by default.

[5] https://cwe.mitre.org/data/definitions/328.html
[6] https://cwe.mitre.org/data/ definitions/327.html

3.4 Cmdi

This weakness, classified as CWE-78, occurs when

> "The software constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component."[7]

Since there is no possibility of defining a priori which are the secure OS commands that can be run without the need of a sanitized input, any attempt of running an OS command using a tainted input are reported as a potential vulnerability. It is possible to observe from the results above that this assumption had an impact on almost the 12% of the false positive classification.

4 Conclusions

This project is publicly available on GitHub[8] and can be exploited in order to assess Infer Quandary even further. With our configuration, Quandary obtained a good sensitivity (0.8) with a low specificity (0.4). This depends on a precise choice, as it is always preferable to deal with false positives rather than false negatives. At the best of our knowledge, then, Quandary seems not to be a widely usable tool for web applications security at the moment being.
However, the obtained results are still promising, as they came without any refinement of the Quandary code, but only providing a configuration file. Providing the users with the possibility to define a more fine grained configuration may lead to better results, probably competing with those of best in class static analysis tools.

[7] https://cwe.mitre.org/data/definitions/78.html
[8] https://github.com/laurab1/Infer-no