Security Analysis of the OWASP Benchmark with Julia

Elisa Burato[1], Pietro Ferrara[1], and Fausto Spoto[1,2]

[1] Julia Srl, Verona, Italy
{elisa.burato, pietro.ferrara, fausto.spoto}@juliasoft.com
[2] Università di Verona, Italy

Abstract

Among the various facets of cybersecurity, software security plays a crucial role. This requires the assessment of the security of programs and web applications exposed to the external world and consequently potential targets of attacks like SQL-injections, cross-site scripting, boundary violations, and command injections. The OWASP Benchmark Project developed a Java benchmark that contains thousands of test programs, featuring such security breaches. Its goal is to measure the ability of an analysis tool to identify vulnerabilities and its precision. We present how the Julia static analyzer, a sound tool based on abstract interpretation, performs on this benchmark in terms of soundness and precision. We discuss the details of its security analysis over a taint analysis of data, implemented through binary decision diagrams.

1 The OWASP Cybersecurity Benchmark

Among the many aspects of cybersecurity, software security stands for the resilience of software applications against external attacks. In particular, web applications are exposed to the external world, and therefore they are a preferred point of attack by hackers. Despite the high risk of attacks against software applications, organizations do not always recognize the importance of assessing that software does not contain security vulnerabilities. The problem is exacerbated by the lack of sound automatic tools, able to find software vulnerabilities. Consequently, organizations either ignore software security threats or apply unsound tools, with no guarantee of finding *all* threats of a given category. Consequently, it is not surprising that many cybersecurity attacks keep being reported in the news, whose origin is in the software of the application. Examples are SQL-injections, that allow hackers to access a database, extract restricted data or corrupt information; cross-site scripting (XSS) attacks, that let hackers include malicious code inside web pages; trust boundary violations, when hackers can insert restricted or dangerous data inside untrusted session objects; command injection, if hackers can lead the application to execute dangerous system commands. For a larger, yet incomplete list of threats, see [4].

The relevance of such software security attacks justifies the interest of researchers and companies in finding ways of assessing that software does not contain such threats. In general, such attacks are due to unexpected flows of data (*injections*), from input *sources* to specific *sinks*, *i.e.*, program points that might execute dangerous or restricted operations. For instance, SQL-injection is the consequence of an unrestricted information flow from a user input or web service entry point into a routine that performs a database query. Tracking all possible information flows is however quite challenging: flows can be direct (e.g., through local variable assignments), but also by side-effect (e.g., by calling a method that assigns the field of an object in a data structure). Data might also flow through arrays and object fields, rather than just local variables. This complexity hinders both *soundness* or *coverage* (finding *all* threats of a given category) and *precision* or *accuracy* (limited number of false alarms) of an analysis tool. Consequently, existing tools are typically unsound (missing threats) and/or imprecise (issuing too many false alarms). For instance, penetration testers try to generate probable attack patterns,

with no guarantee of completeness but full guarantee of precision. In this scenario, establishing a metric to compare different tools is problematic since there is no general agreement about what factors (*e.g.*, soundness or precision) one should take into account in the evaluation, and why. In addition, it is not always clear what constitutes a threat, and how some attacks should be formally defined.

As far as we know, the OWASP Benchmark Project represents the most relevant attempt to establish a universal security benchmark, *i.e.*, a suite of thousands of small Java programs containing security threats. According to their web page [4]:

> *The OWASP Benchmark for Security Automation is a free and open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services. Without the ability to measure these tools, it is difficult to understand their strengths and weaknesses, and compare them to each other.*

The benchmark has benefited from the critical contribution of many organizations, so that it has also served as a way of clarifying the actual nature of the threats. During the years, it has emerged as *the* reference for the comparison of security analysis tools, and it is nowadays a must-do for any tool that asserts to find software security vulnerabilities in Java code. Most tests of the OWASP benchmark are servlets that might allow unconstrained information flow from their inputs to dangerous routines. A few tests are not related to injections, but rather to unsafe cookie exchange or to the use of inadequate cryptographic algorithms, hash functions or random number generators. Injection attacks are however the most complex to spot and have larger scientific interest. The benchmark sets traps for tools, *i.e.*, it contains also harmless servlets that *seem* to feature security threats, at least at a superficial analysis. In this way, the benchmark measures the number of true positives (that is, real vulnerabilities reported by the tool) and false positives (that is, vulnerabilities reported by the tool that are not real issues). They represent a deep and wide stress test: a perfect analyzer should not be caught in a trap while still reporting all the real vulnerabilities. In an ideal world, a tool would get 100% true positive and 0% false positives. However, to achieve such a result one should be able to explore all possible executions of a program; therefore, existing tools make a compromise between soundness, precision and efficiency of the analysis.

An important feature of this benchmark is the automatic generation of reports to compare different tools: there are several scripts to plot coverage and accuracy of the tools, inside comparative *scorecards*. This gives an immediate graphical picture of the relative positioning of the tools. Free tools are plotted in the scorecards. Commercial tools are anonymized into their overall average [4].

2 Taint Analysis with the Julia Static Analyzer

Julia [5] is a commercial static analyzer for Java bytecode, based on abstract interpretation [2]. It performs static analysis based on denotational or constraint-based semantics. Julia currently features 45 *checkers*, including the Injection checker based on the sound taint analysis defined in [3]. Julia has also checkers Cookie, Random and Cryptography that cover the other kinds of threats considered in the OWASP benchmark.

The idea of Julia's taint analysis is to model explicit information flows through Boolean formulas. Boolean variables correspond to program variables and the models of a Boolean formula are a sound overapproximation of all taint behaviours for the variables in scope at a given program point. For instance, the abstraction of the `load k` bytecode, that pushes on the

```
1  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ... {
2          response.setContentType("text/html;charset=UTF-8");
3          String param = "";
4          if (request.getHeader("Referer") != null)
5                  param = request.getHeader("Referer");
6          param = java.net.URLDecoder.decode(param, "UTF-8");
7          String bar = param;
8          if (param != null && param.length() > 1) {
9                  StringBuilder sbxyz67327 = new StringBuilder(param);
10                 bar = sbxyz67327.replace(param.length()-"Z".length(), param.length(),"Z").toString();
11         }
12         response.setHeader("X-XSS-Protection", "0");
13         Object[] obj = { "a", "b" };
14         response.getWriter().format(java.util.Locale.US,bar,obj);
15 }
```

Figure 1: Benchmark 146: This test suffers from a real XSS attack and Julia spots it.

operand stack the value of local variable k, is the Boolean formula $(\check{l}_k \leftrightarrow \hat{s}_{top}) \wedge U$, stating that the taintedness of the topmost stack element after this instruction is equal to the taintedness of local variable k before the instruction; all other local variables and stack elements do not change (expressed by a formula U); taintedness before and after an instruction is distinguished by using distinct hats for the variables. There are such formulas for each bytecode instruction. Instructions that might have side-effects (field updates, array writes and method calls) need some approximation of the heap, to model the possible effects of the updates. The analysis of sequential instructions is merged through a sequential composition of formulas. Loops and recursion are saturated by fixpoint. The resulting analysis is a denotational, bottom-up taint analysis, that Julia implements through efficient binary decision diagrams [1]. Julia uses a dictionary of sources (for instance, servlets input and input methods) and sinks (such as SQL query methods, command execution routines, session manipulation methods) of tainted data, so that flows from sources to sinks can be established.

3 Analysis of the OWASP Benchmark with Julia

In this section, we give an overview of the results of Julia on three representative test cases producing a true positive, a true negative, and a false positive. In addition, we present the overall results of Julia on the OWASP Benchmark.

True positive: A (simplified) example of OWASP benchmark test is shown in Fig. 1. Julia warns about a possible XSS attack at the last line, since the `bar` parameter to `format()` (at line 14) is tainted. In fact, this parameter is built from the content of local variable `param` (line 9), that received (line 5) an input that the user can control (the header of the connection). Note that Julia correctly spots the information flow through the constructor of `StringBuilder` and the call to `replace()`.

True negative: Consider the test in Fig. 2 now. Julia does not issue any warning here. Actually, no XSS attack is possible this time, since variable `param` (tainted at line 6) is sanitized into `bar` by Spring method `htmlEscape()` (line 8). Julia uses a dictionary of sanitizing methods and others can be specified by the user.

False positive: Consider the test in Fig. 3 now. This time Julia falls in the trap and issues a spurious warning about a potential XSS attack at the call to `format()`, since it thinks that

```
1  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ... {
2          response.setContentType("text/html;charset=UTF-8");
3          String param = "";
4          java.util.Enumeration<String> headers = request.getHeaders("Referer");
5          if (headers != null && headers.hasMoreElements())
6                  param = headers.nextElement();
7          param = java.net.URLDecoder.decode(param, "UTF-8");
8          String bar = org.springframework.web.util.HtmlUtils.htmlEscape(param);
9          response.setHeader("X-XSS-Protection", "0");
10         response.getWriter().print(bar);
11 }
```

Figure 2: Benchmark 278: No XSS attack is possible here and Julia does not issue any warning.

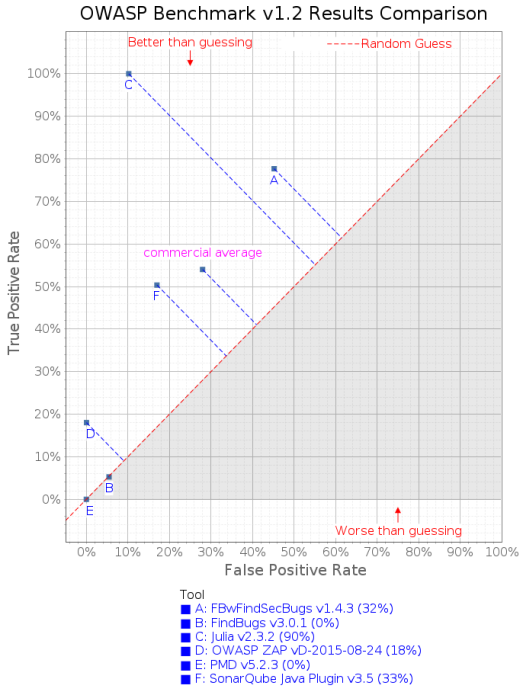
```
1  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ... {
2          response.setContentType("text/html;charset=UTF-8");
3          String param = "";
4          if (request.getHeader("Referer") != null)
5                  param = request.getHeader("Referer");
6          param = java.net.URLDecoder.decode(param, "UTF-8");
7          String bar = "alsosafe";
8          if (param != null) {
9                  java.util.List<String> valuesList = new java.util.ArrayList<String>();
10                 valuesList.add("safe");
11                 valuesList.add(param);
12                 valuesList.add("moresafe");
13                 valuesList.remove(0); // remove the 1st safe value
14                 bar = valuesList.get(1); // get the last 'safe' value
15         }
16         response.setHeader("X-XSS-Protection", "0");
17         Object[] obj = {"a", bar};
18         response.getWriter().format("Formatted like: %1$s and %2$s.",obj);
19 }
```

Figure 3: Benchmark 147: No XSS attack is possible here, but Julia issues a false alarm.

`bar` (and hence `obj`) is tainted. But this is not actually the case, since this test manipulates a `valueList` in such a way that the value finally stored into `bar` is untainted. This list manipulation is too complex for the taint analysis of Julia, that cannot distinguish each single element of the list and conservatively assumes all elements of the list to be tainted.

Overall results: Fig. 4 shows, on the left, the scorecard generated by the OWASP benchmark, that compares Julia to other free (explicitly) and commercial (anonymously) static analyzers. Scorecards report soundness on the left and precision horizontally. Hence, a perfect (*i.e.*, sound and precise) tool should stay on the top left corner of the scorecard. Fig. 4 shows that Julia is very close to that corner, much more than all free analyzers and of the anonymous average of the commercial analyzers. Fig. 4, on the right, reports the results of Julia for the eleven categories of threats considered by the OWASP benchmark. Julia is always close to the top left corner of the scorecard and always finds all threats, since it is the only sound analyzer in this comparison. Hence its results lie on the 100% line for soundness (true positive rate).

Fig. 4 reports also the number of false negatives (**FN**), true positives (**TP**) and false positives



Category	FN	TP	FP
Command Injection	0	126	20
Cross-Site Scripting	0	246	19
Insecure Cookie	0	36	0
LDAP Injection	0	27	4
Path Traversal	0	133	22
SQL Injection	0	272	36
Trust Boundary Violation	0	83	12
Weak Encryption Algorithm	0	130	0
Weak Hash Algorithm	0	129	0
Weak Random Number	0	218	0
XPath Injection	0	15	3

Figure 4: Scorecard comparing Julia with free and commercial tools and Julia's detailed results.

(**FP**) obtained by Julia on the OWASP benchmark. For all categories Julia obtained zero false negatives: this proves the (practical) soundness of the analysis, meaning that Julia is always able to spot security vulnerabilities if the programs contain them. In addition, the number of false positives is always a small percentage (below 20%) of the number of warnings produced: this proves the (practical) precision of the analysis: a developer using Julia to identify vulnerabilities will need to discard only a warning out of six, on this benchmark.

4 Conclusion

In this demo, we presented the results of Julia on the OWASP Benchmark discussing the details of three representative test cases. We showed how a sound semantic static analysis can achieve better results in terms of true and false positives than existing commercial tools. These results show the power of Julia's analysis, but this is only a first step towards commercial tools that catch security vulnerabilities in industrial software. Namely, the OWASP benchmark project has been recognized by the security community as a comprehensive benchmark, but still contains small case studies only, and industrial software is often far from these, even if the patterns of security vulnerabilities are similar. Moreover, Julia uses a sound information flow analysis that leads to a sound injection analysis only if the analyzer recognizes all sources and sinks, which is a daunting task in the presence of the many programming frameworks for Java. This is currently partially tested by the OWASP benchmark. Therefore, we plan to further industrialize the checkers that we applied to the OWASP benchmark, identify their limitations, refine our static analysis and its scalability and contribute new benchmarks to the community.

References

[1] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.

[2] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[3] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'20)*, volume 9450 of *Lecture Notes in Computer Science*, pages 130–145, Suva, Fiji, November 2015. Springer.

[4] OWASP. Benchmark. https://www.owasp.org/index.php/Benchmark. Checked on Oct 13, 2016.

[5] F. Spoto. The Julia Static Analyzer for Java. In *Proc. of Static Analysis (SAS'16)*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57, Edinburgh, UK, September 2016. Springer.

A Running Julia on the OWASP Benchmark

The following is a walk-through of the actual demonstration, that analyzes the OWASP Benchmark and generates the XML file with all warnings generated by Julia.

1. Register to Julia's online service of analysis at https://portal.juliasoft.com;
2. Install Julia's Eclipse plugin and configure it with your credential information (instructions and credentials available at login time to our service and under your user profile);
3. Set the maximum number of warnings shown to 5000;
4. Make sure you have enough credit to run the analysis (150,000 credits); otherwise please contact us;
5. Select the OWASP benchmark project in Eclipse and analyze it with Julia's Eclipse plugin: in the first screen, flag both options *Only main* and *Include .properties files*. The latter is needed since the benchmark assumes that the analyzer has access to property files, which is normally turned off for privacy;
6. In the next screen of Julia's Eclipse plugin, select only the Basic checkers Cryptography, Cookie and Random and the Advanced checker Injection;
7. Click Finish and wait until the analysis terminates. This should take a few minutes, unless there are other analyses in the queue. You can check the progress of the analysis in the console view of Eclipse;
8. Once the analysis has terminated, you can see the warnings in the Eclipse view of Julia and export the XML file of the warnings with the icon of that view that looks like a downwards arrow.

After these steps, you will be able to navigate the results of the analysis of all the test cases of the OWASP Benchmark.

At the address http://www.juliasoft.com/eng/solutions/technical-documentation one finds user manuals and short tutorials about the Eclipse plugin and the web interface.