ShiftLeft

ShiftLeft
OWASP SAST Benchmark

Table of Contents

Overview 03

Towards a new generation of static analysis products 03

Results on the OWASP benchmark 04

Ingredient #1: Our data flow tracker 05

Ingredient #2: High-level information flows 06

Analysis of false positives 08

Conclusion 10

Overview

This paper presents the results of ShiftLeft's static analysis pipeline on the OWASP 1.2 benchmark, where we achieve a true positive rate of 100% at 25% false positives. With a resulting Youden Index of 75%, this makes our Static Application Security Testing (SAST) the best in class, beating the commercial average by 45%, and being the only product capable of identifying all of the included vulnerabilities. We find that the false positives result from our decision to overtaint collections and strings - a deliberate design choice, as we leave precise collection and string tracking to our runtime agent. The success of our analyzer rests on two primary pillars: hard work on merging existing approaches from academia into an industry-grade data flow engine, and a novel approach for modeling, extracting, and evaluating high-level information flows. This work is part of our ongoing effort to develop a new generation of code analysis technology as software continues to conquer one industry after another.

As the amount of code written and deployed steadily increases each day, so does the demand for automation in vulnerability discovery. Previously existing security products employ techniques from both static program analysis ("SAST" products) and dynamic program analysis ("DAST/IAST" products), and as both approaches have their strengths and weaknesses, there is considerable interest in ways of combining the two to complement each other. While static analysis excels at comprehensively identifying vulnerabilities, it usually introduces a prohibitive amount of false positives.

Dynamic analysis, on the other hand, has trouble achieving the same level of coverage and induces a high runtime cost if employed for an entire application. It is, however, very effective at verifying whether a vulnerability is indeed triggerable. False negatives are the problem for dynamic analysis, not false positives.At ShiftLeft, we have developed a product which combines the strength of both approaches by statically identifying vulnerabilities at compile time, and leveraging these findings to dynamically mitigate their impact at runtime. As a result, the runtime cost is lower as instrumentation is performed selectively. Moreover, the negative effect of false positives on developer productivity is significantly reduced by allowing developers to focus on findings that can demonstrably be triggered at runtime.

Towards a new generation of static analysis products

The ability to filter false positives at runtime is powerful, however, as it true for all IAST products, it also requires integration of a runtime agent into a Q/A or even production environment. In many organizations, this is too intrusive, and so it comes as no surprise that SAST - being capable of working on a copy of the code - is the most widely used code analysis technique.

The quality of SAST products, however, is problematic. Not to discredit the work of those who came before us, it is evident from the OWASP benchmark that existing static analyzers leave room for improvement both in terms of true positive and false positive rate. With a top Youden-Index of 39%, much work is to be done to push static analysis to where it needs to be. At ShiftLeft, we are determined to play a key role in this push towards a new generation of static analysis products. The next generation of static analysis needs to scale to the large amount of code processed in today's organizations, integrate into new software development lifecycles, and bring developers and operators closer together. While a lot of work is ahead of us, we have already made quite some progress in the last 1.5 years, as we discuss in the following.

Results on the OWASP benchmark

The OWASP benchmark is a sample application containing thousands of vulnerabilities from 11 categories. The benchmark includes code fragments that are hard to process via static analysis, e.g., indirect calls, unreachable branches, reflection, or values that depend on configuration files. To quote the README:

> "[...]The OWASP Benchmark Project is a Java test suite designed to verify the speed and accuracy of vulnerability detection tools. The initial version is intended to support Static Analysis Security Testing Tools (SAST). A future release will support Dynamic Analysis Security Testing Tools (DAST), like OWASP ZAP, and Interactive Analysis Security Testing Tools (IAST).[...] "

As Chris Wysopal of Veracode also points out, the OWASP benchmark provides a script to trigger all test cases, and this means that the challenge of achieving coverage that purely dynamic approaches face is not accounted for by the benchmark[1]. We therefore evaluate only our static analyzer on this benchmark and leave our runtime components out of the game. Overall, we achieve a Youden index of 0.78. However, in the OWASP benchmark, overall scores are calculated as averages over the categories, in order to give each category the same weight. This is debatable, as it means that decisions in categories with fewer samples are more relevant. With this rule in place, our score is 0.75.

Category	P	N	TP	FP	TN	FN	TPR	FPR	Y
Command Injection (cmdi)	126	125	126	45	80	0	1.0	0.36	0.64
Weak Cryptography (crypto)	130	116	130	0	116	0	1.0	0.0	1.0
Weak Randomness (hash)	129	107	129	0	107	0	1.0	0.0	1.0
LDAP Injection (ldapi)	27	32	27	13	19	0	1.0	0.41	0.59
Path Traversal (pathtraver)	133	135	133	66	69	0	1.0	0.49	0.51
Secure Cookie Flag securecookie)	36	31	36	0	31	0	1.0	0.0	1.0
SQL Injection (sqli)	272	232	272	87	145	0	1.0	0.375	0.63
Trust Boundary Violation (trustbound)	83	43	83	24	19	0	1.0	0.56	0.44
Weak Randomization (weakrand)	218	275	218	0	275	0	1.0	0.0	1.0
XPATH Injection (xpathi)	15	20	15	7	13	0	1.0	0.35	0.65
Cross Site Scripting (xss)	246	209	246	48	161	0	1.0	0.23	0.77
Average	**1415**	**1325**	**1415**	**290**	**1035**	**0**	**1.0**	**0.25**	**0.75**
All	**1415**	**1325**	**1415**	**290**	**1035**	**0**	**1.0**	**0.22**	**0.78**

Table 1: OWASP Benchmark 1.2 and evaluation results. P/N is the number of positive/negative samples, TP/FP is the number of true/false positives, TN/FN is the number of true/false negatives, TPR and FPR are the true positive and false positive rates, and finally, Y is the Youden Index. We show results per category, results averaged over all categories, and results over all samples.

[1]https://www.veracode.com/blog/2015/09/no-one-technology-silver-bullet

In direct comparison to other tools (see Table 2), ShiftLeft is the only tool which achieves 100% coverage, followed by FBwFindSecBugs, which achieves 97%, however, at the cost of a 58% false positive rate. The next best commercial tool achieves a score of 33%.

Analyzer	Benchmark version	TPR	FPR	Y
ShiftLeft	1.2	1.0	0.25	0.75
FBwFindSecBugs	1.2	0.97	0.58	0.39
SonarQube	1.2	0.50	0.17	0.33
SAST-04	1.1	0.61	0.29	0.33
SAST-06	1.1	0.85	0.52	0.33
SAST-02	1.1	0.56	0.26	0.31
SAST-03	1.1	0.46	0.214	0.25
SAST-05	1.1	0.48	0.29	0.19
SAST-01	1.1	0.29	0.12	0.17
PMD	1.2	0.00	0.00	0.00

Table 2: OWASP Benchmark results for 9 static analyzers. The commercial SAST tools 01-06 are known to include Checkmarx CxSAST, HP Fortify, IBM AppScan Source, Coverity Code Advisor, Parasoft Jtest, SourceMeter, and Veracode SAST.

Ingredient #1: Our data flow tracker

The workhorse of our analysis is a state-of-the-art data-flow tracker, which we arrive at by merging many of the techniques developed and employed in academia for automated vulnerability discovery in the past decade. Our data-flow tracker is interprocedural, flow-sensitive, context-sensitive, field-sensitive, and operates on an intermediate code representation (see semantic code property graphs). The engine performs on-the-fly points-to analysis to resolve call sites and is able to benefit from the results of constant propagation, control flow graph pruning, and framework analysis passes. Framework analysis passes are able to process configuration files if present. The data flow engine provides a configurable set of heuristics to allow reporting of findings in an acceptable time frame. For example, we allow limiting the number of branches considered in strongly connected components, the maximum path length, and the total number of computation steps. The data-flow tracker caches where possible, and compromises where it must. There is still room for improvement, but even today, we are able to identify all relevant data flows in the OWASP benchmark in under 10 minutes on an Intel Xeon CPU E5-1650 v3 @ 3.50GHz.

Ingredient #2: High-level information flows

High-level information flows are the second core ingredient that contributes to the precision of our analysis. The idea is simple: for high-level programming languages such as Java, it is not sufficient to track single data-flows between APIs to understand the high-level flow of information. Instead, the information from multiple low-level flows needs to be combined. Let me illustrate this with an example. Consider the flow highlighted in bold in Listing 1: an HTTP request is passed to a post handler, a map holding parameters is extracted via the method `getParameterMap`, and the parameter `vector` is retrieved from this map via the method `get`. The parameter is then stored in the variable `param`, escaped via `htmlEscape`, renamed to `bar`, used in the initialization of an array, and finally, this array is passed to the method `printf`. On a higher level of abstraction, this is a flow of an HTTP request into the HTTP server's response, and more specifically, into the a "text/html" response. The only other relevant aspect of the flow is that data passes through an escaping routine, turning this into a sample of non-vulnerable code.

```
@Override
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html");
    java.util.Map<String,String[]>
    map = request.getParameterMap();
    String param = "";
    if (!map.isEmpty()) {
        String[] values = map.get("vector");
    if (values != null)
       param = values[0];
    }
    String bar = htmlEscape(param);
    Object[] obj = { "a", bar};
    response.getWriter()
       .printf(java.util.Locale.US,
               "Formatted like: %1$s and %2$s.",
               obj);
}
```

Listing 1: (Low-level) primary flow from servlet request to `printf`. Escaping is performed to avoid cross site scripting. The logical destination of the `printf` call is unclear unless the initialization

The high level flow of information cannot be determined using the primary data flow from the servlet request to `printf` alone. In particular, this flow alone does not tell us where data is written, that is, back into the HTTP response, and that this response is not simply a plain text response. Without knowing this, it is impossible to determine whether this is a cross-site-scripting vulnerability, regardless of whether data is escaped or not.

Instead, we need to look at multiple flows: the primary data flow, and all flows that initialize sources, sinks, and transformations. We refer to these as descriptor flows. Inspired by UNIX file descriptors, the idea is to mark parameters of sources, sinks, and transformations, that provide information on where data comes from, where it goes, or how it is transformed, similar to how file descriptors are just integers, but their initialization determines whether data is written to a file, socket or a terminal.

```
@Override
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html");
    java.util.Map<String,String[]>
    map = request.getParameterMap();
    String param = "";
    if (!map.isEmpty()) {
        String[] values = map.get("vector");
    if (values != null)
       param = values[0];
    }
    String bar = htmlEscape(param);
    Object[] obj = { "a", bar};
    response.getWriter()
       .printf(java.util.Locale.US,
               "Formatted like: %1$s and %2$s.",
               obj);
}
```

Listing 2: (Low level) descriptor flow that provides information about the initialization and configuration of the data destination. In combination with the primary data flow, and descriptor flows for the source and all transformations, we obtain a high-level information flow.

In our example, there is a single relevant descriptor flow, namely the flow that initializes the PrintWriter instance passed to the method `printf`. Looking at the primary data flow alone, all we see is that the method `java.io.PrintWriter.printf` is called. Taking into the account the descriptor flow highlighted in Listing 2, we see that the PrintWriter is initialized from the HttpServletResponse, and hence, that the data will be written into an HTTP response. We also see that the data will be returned as "text/html", since the method `setContentType` is called on the descriptor as part of its initialization.

By combining the primary data flow with its descriptor flow, we can derive high-level data flows, and formulate rules for their classification. For example, we use the following rule to describe reflected cross site scripting vulnerabilities.

CONCLUSION reflected-xss = FLOW IO (http)
-> DATA (NOT encrypted AND NOT hashed AND NOT escaped AND NOT encoded)
-> IO (print AND http-html)

This rule specifies that a data flow from http to a print operation known to write HTML to HTTP is a reflected cross-site-scripting vulnerability, if the data is not encrypted, hashed, escaped or encoded on the way.

Analysis of false positives

We analyzed false positives to better understand the capabilities and limitations of our analyzer. We found that the OWASP benchmark consists of code snippets, which are connected to form test cases, and that all false positives are caused only by the following three snippets.

1. Overtainting of collections (variant 1). **The listing below shows the first snippet. A map is populated with a tainted (`param`) and an untainted value (`"a_Value"`), and the untainted value is retrieved from the map (into `bar`). We overtaint collections, meaning that, if parts of a collection are tainted, we assume that the entire collection is tainted, resulting in a false positive as we retrieve an untainted part of the collection in this example.**

```
String bar = "safe!";
java.util.HashMap<String,Object> map65281 =
   new java.util.HashMap<String,Object>();
map65281.put("keyA-65281", "a_Value");
map65281.put("keyB-65281", param);
map65281.put("keyC", "another_Value");
bar = (String)map65281.get("keyB-65281");
bar = (String)map65281.get("keyA-65281");
```

2. Overtainting of collections (variant 2). **In this code, an untainted value(`"safe"`), a tainted value (`param`), and another untainted value (`"moresafe"`) are added to a list. The first value is subsequently removed from the start of the list, and the list element at index 1 is retrieved, that is, the untainted value `"moresafe"` is read. Again, we overtaint collection, resulting in a false positive.**

```
java.util.List<String> valuesList =
     new java.util.ArrayList<String>( );
valuesList.add("safe");
valuesList.add( param );
valuesList.add( "moresafe" );
valuesList.remove(0); // remove the 1st safe value
bar = valuesList.get(1); // get the last 'safe' value
```

3. Disabled branches. **Finally, the listing below shows the last of the three snippets that cause false positives. In this snippet, the second character is taken from the string "ABC". If this is a constant, the result is always `B`, and hence, all of the branches of a following switch statement, except for one, are unreachable. This problem could be solved by modeling the semantics of `charAt`, replacing `switchTarget` accordingly, and performing constant propagation and control flow graph pruning. As we do not model `charAt`, we receive false positives for this snippet.**

```
public String doSomething(String param)
  throws ServletException, IOException {
  String bar;
  String guess = "ABC";
  char switchTarget = guess.charAt(1);

  switch (switchTarget) {
    case 'A':
      bar = param;
    break;
    case 'B':
      bar = "bob";
      break;
    case 'C':
    case 'D':
      bar = param;
      break;
    default:
      bar = "bob's your uncle";
      break;
    }
    return bar;
}
```

Creating passes to deal with these constructs statically is possible, with the second case being the most evolved as it required modelling of lists. It would have been possible, and allowed us to achieve a false positive rate of 0%, however, it would have been intellectually dishonest, as we would never put this expensive computation into production. The amount of computation required to solve such problems in general statically is high, and the cases in which it is relevant are too few to justify it. Instead, handling these cases with a runtime agent is cheaper, simpler, and can achieve sufficient precision.

Conclusion

Increased automation is crucial in vulnerability discovery to scale to the large amount of code that need to be secured in today's industries. The ability to statically identify vulnerabilities comprehensively, efficiently, and with few false positives is an important primitive to achieve this, and at ShiftLeft, we put considerable work and energy into achieving this goal. Today, we are happy to demonstrate successes in this area, by providing the first static data-flow tracker capable of detecting 100% of the vulnerabilities in the OWASP benchmark, at a false positive rate of 25%. This false positive rate is considerably lower than that of other static analyzers and can be attributed entirely to overtainting of collections and strings, a problem we handle via runtime analysis in practice.

Acknowledgements

While many people at ShiftLeft made these numbers possible, I would like to highlight the contributions by Markus Lottmann and Niko Schmidt specifically. Thank you for all your hard work!