Interaction Platform for Improving Detection Capability of Dynamic Application Security Testing

Jonghwan Im, Jongwon Yoon and Minsik Jin
PA Division, Fasoo.com R&D Center, 396 World Cup Buk-ro Mapo-gu, Seoul, Republic of Korea

Keywords: Web Application Security Testing, SAST, DAST, IAST, XSS, SDLC.

Abstract: Dynamic application security testing detects security vulnerabilities by sending predefined strings to web applications. So if the web applications have filters which restrict input parameters, the detection capability of dynamic application security testing is degraded. To solve this problem, interactive application security testing have emerged in which dynamic application security testing interact with static application security testing. In this paper, we propose an interactive platform for storing, processing, and distributing information collected from each security test in the software development life cycle. And we use this platform to verify that we can detect cross-site script vulnerabilities that could not be detected due to web application filters. Experiments on the proposed approach for the cross-site script vulnerability test case of OWASP Benchmark show that the detection rate of the dynamic analyzer is improved by about 32.11%.

1 INTRODUCTION

Dynamic Application Security Testing (DAST), which is performed during the test and operation phases of the Software Development Life Cycle (SDLC) of web applications, is highly accurate because of detecting vulnerabilities in real-time execution of web applications. Whereas it is difficult to examine all execution paths of web applications (Ernst, 2003). And it uses a predefined attack string to check security vulnerabilities such as Cross-Site Script (XSS) that can be caused by input value (Fu et al., 2007). So, it is hard to detect security vulnerability of web application that has filter to restrict input value (Kiezun et al., 2009).

To overcome these drawbacks, an Interactive Analysis Security Testing (IAST) was proposed. IAST is a way to improve the quality of security tests by surmounting the limits of each security analysis through the interaction of Static Application Security Testing(SAST) and DAST (MacDonal, 2012).

To perform IAST on SDLC, a module is needed to store, process, and transfer the information gathered from each security testing.

Our paper makes the follow contributions:

- We propose a platform to manage information flow between security tests on SDLC.
- It uses the information provided by the platform to detect XSS that could not be detected by the filters of the web applications, thereby improving the detection capability of the dynamic analyzer.

In Section 3, we explain how to collect information from a static analyzer, how to send collected information to the platform and how to generate attack strings to bypass filter with information received by the platform. In Section 4, the effectiveness of the proposed approach is verified through experiments.

2 RELATED WORK

Several studies have been suggested to become better a dynamic analyzer by combining static and dynamic analysis.

Saner (Balzarotti et al., 2008) is a tool which extracts input value validation process of the web application source codes and checks whether the validation process during execution extracted works properly.

WebSSARI (Web application Security by Static Analysis and Runtime Inspection) (Huang et al., 2004) applies type system for inspecting input values of web application finds source codes where vulnerabilities occurs. And then it monitors and protects those codes in real time.

Im, J., Yoon, J. and Jin, M.
Interaction Platform for Improving Detection Capability of Dynamic Application Security Testing.
DOI: 10.5220/0006437104740479
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, pages 474-479
ISBN: 978-989-758-259-2

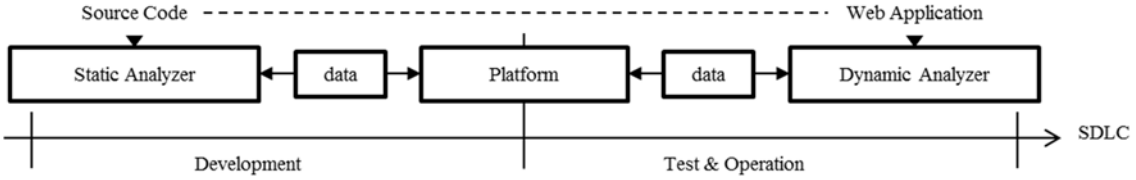


Figure 1: Platform for interaction between static and dynamic analyzer in SDLC.

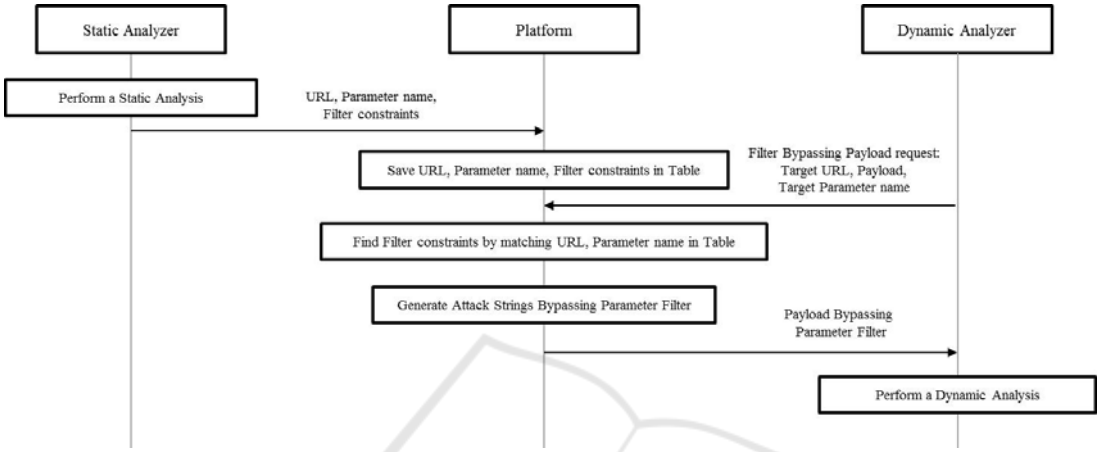


Figure 2: Interaction flow chart between static and dynamic analyzer using platform.

Some studies address dynamically, statically tracing web application input values to monitor sensitive information flow and control input values that cause vulnerabilities (Ruso and Sabelfeld, 2010).

In additional, there have been studies to circumvent filters of web applications using string constraint solver, a string generation tool that avoids constraints. During the compilation process, SAFELI (Fu et al., 2007) is a framework that uses symbolic execution to statically analyse the bytecode of the web application to find the parts where the value is input. And, it generates SQL queries using a string constraint solver and enters these into the found parts to discover SQL injection vulnerabilities. SUSHI Constraint Solver (Fu et al., 2007) blocks security holes by resolving SISE (Simple Linear String Equation), which is a constraint that is consisted of data enter conditions and attack patterns gotten by static analysis.

However, these studies have focused on including a getting source code information feature in the dynamic analyzer. It is difficult to exploit the static analyzers used on SDLC.

This paper suggests an approach that can make use of the static analyzers worked on SDLC and enhance detection capability of the dynamic analyzers at the same time.

3 PROPOSED APPROACH

Figure 1 presents a proposed interaction platform structure. This platform stores, processes and shares the information sent from each analyzer on SDLC. It helps interactions between the analyzers.

To overcome the limit, which cannot catch vulnerabilities when payloads with attack strings sent from a dynamic analyzer are restricted due to filters of web applications, attack strings bypassing the filters are required. A string constraint solver creates the attack strings to circumvent filter with input parameter constraints of the web application filters. The Input parameter constraints are obtained by the static analyzers. The platform has a string constraint solver, so this platform creates the attack strings circumventing the filter constraints and sends the strings to the dynamic analyzer. The dynamic analyzer uses these attack strings as the payloads for detecting vulnerabilities.

This section describes data flow between the static analyzer, dynamic analyzer and the platform. And it explains what information collected by the static analyzer and how to generate the attack strings that avoid web application filter constraints in the platform.

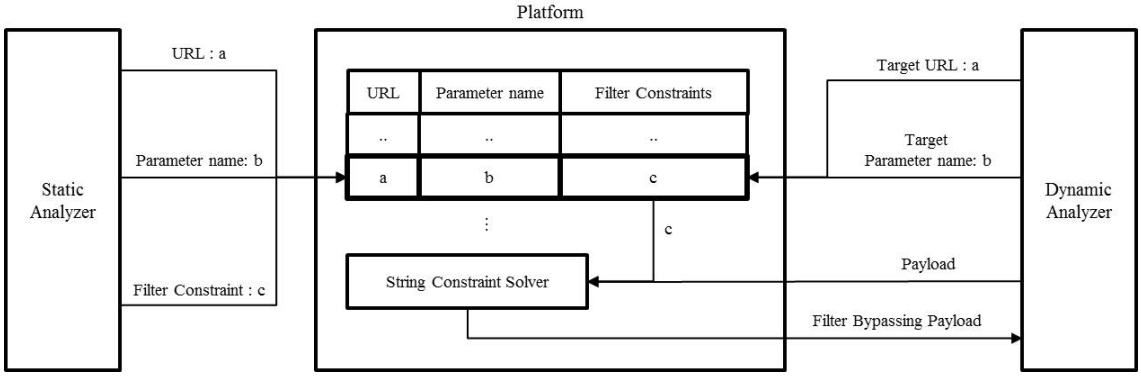


Figure 3: Data flow from static analyzer to platform, platform to dynamic analyzer.

3.1 Interaction Process

Figure 2 shows information flow between the static analyzer, dynamic analyzer and the platform. After the static analyzer performs analysis, it sends URL, parameter name, filter constraints information to the platform. These data are treated in Section 3.2. The platform saves these data. Before the dynamic analyzer attacks a target for detecting vulnerabilities, it requests attack strings circumventing filter to the platform with a target URL, payload to be used, and parameter name that is assigned the payload. Then the platform looks for filter constraints that match the data sent by dynamic analyzer form the stored data. If the filter constraints are found, the string constraint solver generates strings that avoid the filter constraints and these strings are sent to the dynamic analyzer. The dynamic analyzer uses these strings as payloads to attack web applications and detect vulnerabilities. Through such a procedure, the platform is used for interaction between each analyzer.

3.2 Gather Filter Information

Left side of Figure 3 means a data flow from the static analyzer to the platform. The following information is transmitted.

- URL: URL information that has filters for input values
- Parameter name: Parameter name information assigned the input values
- Filter Constraints: The constraints for input values in the filters.

The static analyzer gathers URL, parameter name and filter constraints information by `getFilterInfo` function.

```
FUNCTION getFilterInfo(function)

  Http request is a data which a client
sends to a server.
  Http request parameter consists of a
name and input value.
  The function should have URL
information.
  The callStack is a stack structure
which saves all caller of the function.
  The callStackList is a list of all
the function's callStack gotten by
static analyzer.

START
  parameterList ← a list of using
                  parameters in
                  function.
  WHILE parameter in parameterList
    name ← parameter.name
    WHILE callStack in callStackList
      WHILE callStack is not empty
        constraints ← find parameter
                      constraints
        callerFunction ← callStack.pop
      ENDWHILE
      IF there are constraints
        url ← URL imformation of caller
              sendToPlatform(url, name,
                             constraints)
      ENDIF
    ENDWHILE
  ENDWHILE
END
```

Figure 4: Pseudo code of getFilterInfo function.

In the Figure 4, `getFilterInfo` receives a function that invokes parameters for getting input value. Then all parameters used in the function are saved in `parameterList`. A name of a parameter

that is an element of `parameterList` is sought and assigned to `name` variable. And the constraints of the parameter are found by popping a caller in `callStack` that is the stack on which all callers of the function are piled up. Then those constraints are saved in `constraints` variable. When there are no more callers to pop from the callStack, URL information of the last extracted caller is saved in a `url` variable. Finally, the information saved in `name`, `constraints`, `url` variables are transmitted to the platform by calling `sendToPlatform` function.

3.3 Create Attack Strings Bypassing Filter

A data flow between the platform and the dynamic analyzer is shown in the right side of Figure 3. The information that the dynamic analyzer sends with a request is as follows.

- Target URL: URL information to be attacked for detecting vulnerabilities.
- Target parameter name: The name information of parameter to be assigned attack payload.
- Payload: The predefined string to be used for attack.

```
FUNCTION getBypassPayload
      (url, parameterName,
      payload)

  The filterTable stores the url,
parameter name, and filter constraints.
  The solver(payload, constraints)
creates a bypass payload.
  .
START
  constraints ←
                filterTable.getConstrain
                ts
                (url, parameterName)
   IF there are constraints
    bypass ← solver(payload,
constraints)
    RETURN bypass
   ELSE
     RETURN payload
   ENDIF
END
```

Figure 5: Pseudo code of getBypassPayload function.

When the request with the information come to the platform from the dynamic analyzer, the `getBypassPayload` function of the platform creates attack strings avoiding filter and transmits it to the dynamic analyzer.

In the `getBypassPayload` function of Figure5, filter constraints that are found by matching target URL and target parameter name in `filterTable`, which has the information sent from the static analyzer, are stored in `constraints` variable. If there are values in `constraints` variable, a string constraint solver generates strings that circumvent the constraints. If not, the platform gives back the payload sent from the dynamic analyzer.

The dynamic analyzer puts the strings obtained from the platform in http request messages as attack payloads and attacks the target for detecting vulnerabilities. In this way, the dynamic analyzer can detect vulnerabilities that could not be found because the predefined attack strings are restricted to the parameter constraints of the filters.

4 EXPERIMENT

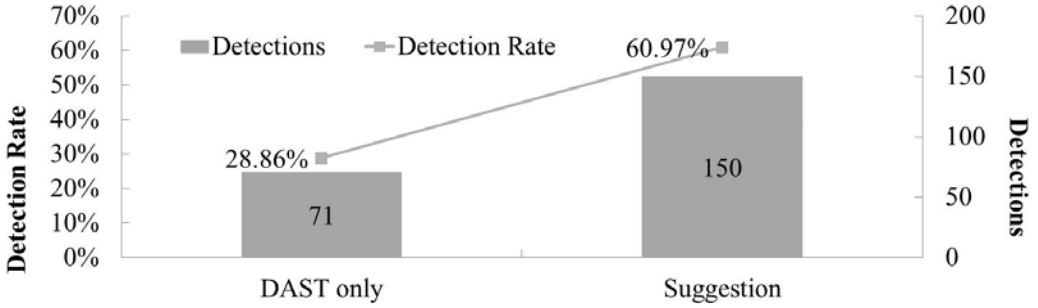
This section verifies the proposed approach actually improves the detection performance of the dynamic analyzer by conducting experiment.

4.1 Experiment Environment

An experiment were performed with XSS among the test cases of OWASP Benchmark v1.2 (OWASP, 2016). OWASP Benchmark for security automation is an open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services. The XSS test cases of OWASP Benchmark total of 455 test cases, of which 246 test cases are true vulnerability test cases and 209 test cases are false vulnerability test cases. SPARROW v4.6_46(Fasoo Inc., 2016), which has been commercialized by Fasoo Inc., was used as a static analyzer. ZAP (Zed Attack Proxy) v2.4.3 (OWASP, 2015) which is one of the OWASP open source projects was used as a dynamic analyzer. Z3str2 v1.0.0(Zheng et al., 2016) used for a string constraint solver in the platform.

4.2 Experiment Result

When DAST was performed only the dynamic analyzer, 71 vulnerabilities out of 246 true vulnerabilities were discovered and the detection rate was about 28.86%. On the other hand, the



	Detections	Detection Rate (Detections / Valid Vulnerabilities, %)
DAST only	71	28.86%
Suggestion	150	60.97%

Figure 6: OWASP Benchmark XSS vulnerabilities detection results of dynamic analyzer whether a platform is used.

Table 1: Http Request / Response to BenchmarkTest02134 whether a Platform is used.

	Http Request	Http Response
DAST Only	`vector=<SCRIPT>alert(“XSS!”);</SCRIPT>`	`vector=<SCRIPT>alert(“XSS!”);<SCRIPT`
Suggestion	`vector=<SCRIPT>alert(“XSS!”);<SCRIPT>` **`aaabbbb`**	`vector=<SCRIPT>alert(“XSS!”);<SCRIPT>` **`aaabbb`**

suggestion, which is the interaction of static and dynamic analyzer through the platform, founded 150 vulnerabilities out of the true vulnerabilities. The detection rate was about 60.87%. The suggestion was improved by about 32.11% than only using dynamic analyzer. Figure 6 presents this result as a graph and sheet.

```
1   @WebServlet("/BenchmarkTest02134")
2   public class BenchmarkTest02134 extends HttpServlet {
                                  …
7     @Override
8     public void doPost(HttpServletRequest request, …) {
9       response.setContentType("text/html");
10      String param = request.getParameter("vector");
11      if (param == null) param = "";
12      String bar = doSomething(param);
13      response.getWriter().write(bar.toCharArray()); }
14    private static String doSomething(String param) … {
15      String bar = param;
16      if (param != null && param.length() > 1)
17        bar = param.substring(0,param.length()-1);
18      return bar; }
19  }
```

Figure 7: Benchmark02134 input value processing source code containing parameter constraint.

Figure 7 is input value processing source code of BenchmarkTest02134 test case which has an input parameter constraint. In line 10, a parameter, whose name is vector, is invoked. The name is saved according to the method in Section 3.2. Likewise, a filter constraint on line 17 that the last character is removed from the input string value is stored. The static analyzer transmits a URL information, which is `/BenchmarkTest02134` in first line, the `vector` parameter name, and the constraint to the platform.

The http request messages which were sent from the dynamic analyzer for attacking target URL and the http response message to the request message are shown in Table 1. When the platform was not used, the dynamic analyzer assigned predefined attack string, `<SCRIPT>alert(“XSS!”);</SCRIPT>`, to the vector parameter of http request message and sent it to the target. But this attack string could not bring about XSS vulnerability since the filter constraint blocked this attack. So, this attack did not detect XSS vulnerability.

Using the proposed method, the string constraint solver in the platform generated `<SCRIPT>alert(“XSS!”);</SCRIPT>aaabbbb`, an attack string which avoids filter constraints, with information that had been sent from the static analyzer. This string were assigned vector parameter as a payload and sent to the target. In the web application, this string changed to `<SCRIPT>alert(“XSS!”);</SCRIPT>aaabbb` because the filter removed the last character `b`.

This modified string caused XSS so the dynamic analyzer could detect XSS vulnerability.

Experimental results showed that using the proposed approach, the dynamic analyzer could detect XSS vulnerability because attack string sent the dynamic analyzer circumvented the web application filter constraints.

5 CONCLUSION

This paper suggested the platform which stores, processes, and distributes information between each analyzer on SDLC. And we verified that proposed method improves the detection performance of the dynamic analyzer by approximate 33% through the experiment on XSS of OWASP Benchmark.

SAST has a limit that the static analyzer can bring about FP (False Positive), which it detects wrong vulnerabilities(Chess and McGraw, 2004). Like a DAST, the problem of SAST can be solved by interaction (Balzarotti et al., 2007). The platform can provide the information not only sent from the static analyzer but also sent form the dynamic analyzer.

In the future, a research is needed to improve the detection capability of the static analyzer by breaking through the problem of SAST using the information provided by the dynamic analyzer as a platform.

ACKNOWLEDGEMENTS

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R0190-15-1099, Development of an integrated management system and a security testing system that enables interaction between security vulnerability detection technologies in development and operation phases of web application).

REFERENCES

Fu, X., Li, C., 2010. A String Constraint Solver for Detecting Web Application. In *Proc. of International Conference on Software Engineering and Knowledge Engineering*, pp.535-523.

Kiezun, A., Ganech, V., Guo, P. J., Hooimeijer, P., Emst, D., 2009. Hampi: A solver for string constraints. In *Proc. of International Symposium on Testing and Analysis*, pp.105-116.

Balzarotti, M., Cova, V., Flmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G., 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proc. of the IEEE Symposium on Security and Privacy*.

OWASP, 2016. Benchmark. Available at https://www.owasp.org/index.php/Benchmark.

OWASP, 2015. Zed Attack Proxy(ZAP). Available at https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

Zheng, Y., Dolby, J., Tripp, O., Ganesh, V., Subramanian, S., Berzish, M., Zhang, X., 2016. Z3str2: An Efficient Solver for Strings, Regular Expressions, and Length Constraints. *Formal Methods in Systems Design*, invited paper at the Formal Methods in Systems Design Journal, vol.50, pp.1-40.

Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L., 2007. A Static Analysis Framework for Detecting SQL Injection Vulnerabilities. In *Proc. of 31st Annual International Computer Software and Applications Conference*, pp.519-531.

Fasoo Inc., 2016. SPARROW. Available at http://www.fasoo.com/%EC%8A%A4%ED%8C%A8%EB%A1%9C%EC%9A%B0-sparrow.

Chess, B., McGraw, G., 2003. Static analysis for security. In *Proc. of the IEEE Symposium on Security and Privacy*, pp.74-79.

Ernst, M. D., 2003. Static and dynamic analysis: synergy and duality. In *Proc. of the ICSE Workshop on Dynamic Analysis*, pp.24-27.

Russo, A., Sabelfeld, A., 2010. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, pp.186-199.

Balzarotti, D., Cova, M., Felmetsger, V., Vigna, G., 2007. Multi-module vulnerability analysis of web-based applications. In *Proc. the 14th Computer and Communications Security*, pp.24-35.

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D. T., Ku, S.-Y., 2004. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proc. of the 12th International World Wide Web Conference*, pp40-52.

Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., 2008. Dynamic Test Input Generation for Web Applications. In *Proc. of International Symposium on Software Testing and Analysis*, pp.249-260.

Emmi, M., Majumdar, R., Sen, K., 2007. Dynamic Test Input Generation for Database Applications. In *Proc. of International Symposium on Software Testing and Analysis*, pp.151-162.

MacDona, N., 2012. Interactive Application Security Testing. Available at http://blogs.gartner.com/neil_macdonald/2012/01/30/interactive-application-security-testing/