JCOMIX: a Search-based Tool to Detect XML Injection Vulnerabilities inWeb Applications A search-based tool to detect XML injection vulnerabilities in web applications

Stallenberg, Dimitri Michel; Panichella, Annibale

DOI
10.1145/3338906.3341178

Publication date
2019

Document Version
Accepted author manuscript

Published in
The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering

JCOMIX: A Search-Based Tool to Detect XML Injection Vulnerabilities in Web Applications

Dimitri Michel Stallenberg
d.m.stallenberg@student.tudelft.nl
Delft University of Technology
The Netherlands

Annibale Panichella
a.panichella@tudelft.nl
Delft University of Technology
The Netherlands

ABSTRACT

Input sanitization and validation of user inputs are well-established protection mechanisms for microservice architectures against XML injection attacks (XMLi). The effectiveness of the protection mechanisms strongly depends on the quality of the sanitization and validation rule sets (e.g., regular expressions) and, therefore, security analysts have to test them thoroughly. In this demo, we introduce JCOMIX, a penetration testing tool that generates XMLi attacks (test cases) exposing XML vulnerabilities in front-end web applications. JCOMIX implements various search algorithms, including random search (traditional fuzzing), genetic algorithms (GAs), and the more recent co-operative, co-evolutionary algorithm designed explicitly for the XMLi testing (COMIX). We also show the results of an empirical study showing the effectiveness of JCOMIX in testing an open-source front-end web application.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software testing and debugging**; • **Security and privacy → Web application security**.

KEYWORDS

XML injection, Search-based Software Engineering, Security Testing, Test Case Generation

ACM Reference Format:
Dimitri Michel Stallenberg and Annibale Panichella. 2019. JCOMIX: A Search-Based Tool to Detect XML Injection Vulnerabilities in Web Applications. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341178

1 INTRODUCTION

The typical architecture of modern web applications consists of multiple internal web services that interact with one another by exchanging data in well-defined and shared formats [5, 13], such as XML or JSON. Typical examples of modern architectures are the Service-Oriented Architectures (SOA) or microservices. On the one hand, these architectures lead to web applications that are more flexible, scalable, and maintainable than monolithic architectures [13]. On the other hand, the increasing communication complexity leads to a larger attack surface that can be exploited by attackers to compromise the system and data integrity [17].

To protect web application against security attacks, developers write validation and sanitization routines for users supplied input [19] (e.g., username and password in authentication form) to check for (validation) or remove (sanitization) malicious content. Usually, input sanitization and validation routines are part of the front-end web applications, which process users inputs and generate data messages (e.g., XML messages) for the internal web services. In XML-based web applications, user inputs need to be checked to avoid well-known yet widespread XMLi attacks, such as XML Billion Laughs (BIL) and XML External Entities (XXE) [3, 12]. However, large applications with hundreds of input forms can often have input fields that are not entirely validated [16].

Researchers have proposed various testing techniques to find vulnerabilities in front-end web applications [1, 2, 6, 8, 10, 11, 18], and their validation and sanitization routines in particular. White-box techniques (e.g., [6, 18]) have been used in the literature to detect various types of vulnerabilities, such as SQL Injection and Cross-site Scripting. These techniques require that the source code (or bytecode) of the web applications (both front-end and internal services) is available to the tester, that the internal working can be observed through instrumentation, and are language dependent. Instead, black-box techniques [1, 2, 8, 10, 11] identify vulnerabilities by inspecting the user-supplied inputs and the output generated by web applications. Note that there is no one-to-one mapping between user inputs and test output because input are processed and transformed by the SUT. Black-box techniques do not require access to the source code, are language independent, and can be used for multiple types of vulnerabilities.

Random fuzzers (e.g., ReadyAPI [1], WSFuzzer [2]) try to send some XML meta-characters (e.g., <) in a black-box fashion. Then, they inspect the SUT responses search for abnormal or malicious behaviors. However, fuzzers are not very effective for complex vulnerabilities [9], i.e., vulnerabilities that cannot be exploited by injecting few meta-characters in one single input text box. Recently, Jan et al. [8, 10, 11] applied different search-based testing algorithms suitably defined for XMLi, such as traditional genetic algorithms (GAs), many-objective GAs, local solvers and co-evolutionary algorithms. Their study showed that search-based approaches outperform random fuzzing [10] and that co-evolutionary algorithms discover more XMLi vulnerabilities than other search algorithms [11].

In this paper, we present JCOMIX, a novel tool (in Java) that implements co-evolutionary testing algorithms for XMLi attacks.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3341178

JCOMIX uses a grammar-based generator to synthesize malicious XML messages (test objectives) that the front-end web application should not generate. These test objectives are produced by leveraging an attack grammar that injects malicious content into legitimate XML messages, which can be obtained by functional tests or during software execution in the field. Then, search-based algorithms are used to search for user inputs to the front-end (i.e., strings for the web application form) that lead to the generation of the test objectives (malicious XML messages). The search is guided by the edits distance and its faster variants [10, 11]. While this paper describes our tool in the context of automated testing for XML injections, JCOMIX can be extended to detect injection vulnerabilities in front-end web application for other data formats, such as JSON.

Tool JCOMIX is publicly available on GitHub at the following link: https://github.com/SERG-Delft/JCOMIX. A walk through example of how to use JCOMIX is described in the README file of the GitHub repository. A video of the JCOMIX at work is available at: https://youtu.be/ZBqZk4qOtCk.

We also present an empirical study showing the effectiveness of JCOMIX in finding XMLi attacks in an open-source front-end application interacting with a bank card processing system.

2 THE TOOL

The goal of JCOMIX it to assess whether a malicious XML message M can be obtained by the target front-end web application (Software Under Test) by providing a specific input X. This can be expressed as a search problem: *finding the user input X for the web-application form such that the SUT generates the message M that is harmful to the internal web services.* If such inputs exist, then the validation and sensitization routines are not properly implemented, and the testing approach exposes some vulnerabilities.

Therefore, there are two main phases in the work-flow of JCOMIX as depicted in Figure 1: (1) Test Objective Generation (TOG) and Penetration Testing Generator (PTG). The first phase aims to generate malicious XML messages (Test Objectives [10], or TO for brevity) that, if generated by the front-end web application, can compromise the integrity and security of the internal web-services. In the second phase, search algorithms are used to search for user inputs that, upon validation and sanitization, results in malicious XML message (TOs). These two phases work in synergy to find security gaps in the validation and sanitization routines. The next subsections describe the two phases in details.

2.1 Test Objective Generation (TOG)

To generate TOs, JCOMIX requires two inputs. The first input is a legitimate XML message (i.e., without malicious content) that can be extracted from functional tests. The second input is the `proxy.json` file, which contains environmental parameters, such as the language of the message (i.e., XML) and the name of the text fields (tags) in the XML file that can be manipulated by user inputs. First, JCOMIX uses ANTLR[1] to parse the legitimate XML message and check the validity of TOs generated via grammar-based mutations. ANTLR is a tool for language recognition, and we selected it because it is widely-used and support multiple languages.

[1]https://www.antlr.org

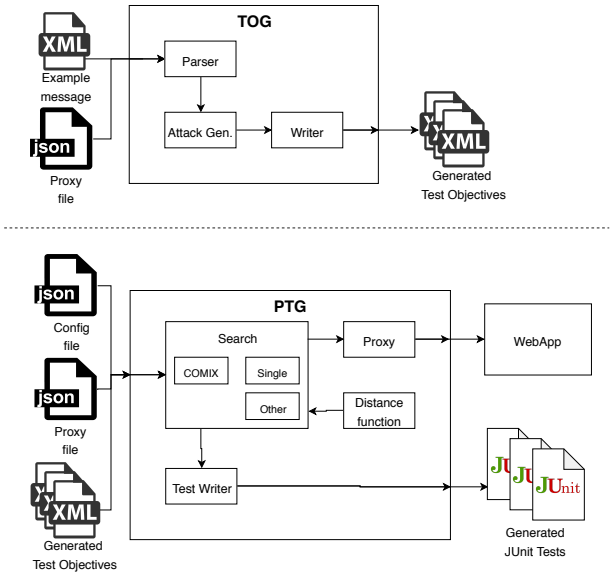

Figure 1: JCOMIX workflow

TOs (or malicious XML messages) are generated by inserting malicious content within the legitimate message using an attack grammar. The attack grammar covers four main categories of XMLi attacks [9, 10]: namely (1) *replicating*, (2) *replacing*, (3) *deforming*, and (4) *random closing tags*. *Replacing and replicating* attacks embed nested attacks (e.g., SQL injection) within XML messages. *Deforming* attacks aim to crash the XML parser in the SUT by sending malformed XML messages. Finally, *random closing tags* attacks use extra closing tags to generate malformed XML messages that reveal the hidden structure of the XML documents.

Malicious content is inserted by applying the attack grammar and injecting malicious content into the text fields specified in the file `proxy.json`. Finally, ANTLR is used to validate the generated TOs to verify that the generated malicious messages are valid XML messages. Valid TOs will be used in the second phase of JCOMIX.

While in this demo we focus on four types of XMLi attacks, JCOMIX is easily extensible to other types of attacks and other data formats besides XML. However, this requires to define an attack grammar for the new attacks/data format.

2.2 Penetration Testing Generation (PTG)

The second phase aims to generate JUnit (e.g., executable) tests that expose the XMLi vulnerabilities in the target front-end web application. More specifically, given a set of well-formed yet malicious TOs, search algorithms are used to search for inputs that lead the SUT into producing the TOs. Note that JCOMIX uses black-box search, i.e., it does not access the source code of the validation and sanitization techniques. Besides, there is no one-to-one mapping between user-supplied inputs and resulting XML messages because inputs are processed and transformed by the SUT.

A candidate solution is a list of strings $X = \langle X_1, X_2, \ldots, X_N \rangle$ to insert in the target input form, where X_i denotes the input for the i-th input in the form. A random solution X consists of N strings of variable lengths. Each string contains characters randomly taken from the set of printable ASCII characters [10]. The fitness of each solution (or individual) is measured using the edit distance (or its variants [10]) between the XML message generated by the SUT and the target TO. In general, the computation of the fitness function involves the interception of the XML messages generated by the SUT, which can be done through code instrumentation or parsing the HTTP messages. In the current implementation, JCOMIX includes a test execution engine to intercepts the produced XML messages in the body of the HTTP response messages. For the search, JCOMIX implements multiple alternative search algorithms, which are summarized in the following paragraphs. The search algorithms, as well as their internal parameters can be configured by editing the `config.json` file.

Random search. It starts with a randomly generated test X, which is executed and evaluated using the fitness function. In each subsequent iteration, a new random test X' is generated and evaluated. If X has a better fitness value than X', it is kept as the current solution; otherwise, X' replaces X in the next iterations. The search continues until a given timeout is reached or if the current solution X results into the SUT generating the target TO. The search process is repeated multiple times, once for each target TO until all TOs are considered in the search.

Traditional Genetic Algorithm. GAs are population-based search algorithms that evolve a pool of solutions iteratively. In the first iteration (also called generation), M random tests are generated, executed, and evaluated using the fitness function (i.e., the edit distance). In subsequent generations, pairs or tests (parents) are selected based on their fitness values and combined in order to create new tests (*offspring*) that inherit strings/characters from the parents. New tests are synthesized using two genetic operators, the *crossover* and *mutation* operators. The *crossover* creates new input strings by mixing the input strings of the two selected parents; the *character mutation* randomly changes, removes, or add characters in the new tests. The search terminates when a given number of iterations are performed or when a zero fitness function is obtained. A zero fitness function (edit distance) indicates that GAs found a test X that, when executed against the SUT, leads to the generation of the target TO. Similarly to random search, GAs are executed multiple times, once for each target TO until all TOs are considered in the search.

Co-evolutionary Search. JCOMIX implements the co-operative, co-evolutionary many-objective algorithm proposed by Jan et al. [11] and defined for XMLi attacks. COMIX co-evolves multiple populations rather than one single population as in traditional genetic algorithms. Different sub-populations (or islands) evolve test cases towards different TOs, one independent sub-population for each TO. Sub-populations are initialized in the first iteration of the evolutionary algorithm, similarly as done in traditional GAs. However, islands are evolved separately in each iteration using *selection*, *crossover*, and *mutation* operators. This means that the offspring is created within each island separately, and parents are selected from the same island. Besides, at the end of each iteration, the strongest individual from one island is copied into the other islands. This operation, called *migration*, is widely-used in co-evolutionary to improve genetic diversity [11]. To further reduce the likelihood that the search gets stuck in local optima, COMIX restarts the islands for which the corresponding fitness value has not improved in the latest k subsequent iterations.

```
class replace1Test {
  private WebDriver driver;

  @BeforeEach
  void setup() throws Exception {...}

  @AfterEach
  void tearDown() {...}

  @Test
  public void aTest(){
    String testObjective = "<...>";

    driver.findElement(By.xpath("//input[@name='CardNumber']")).
        ↪ sendKeys("123456789123456");
    driver.findElement(By.xpath("//input[@name='BankCode']")).
        ↪ sendKeys("0111</lu:IssuerBankCode> <!-- ");
    driver.findElement(By.xpath("//input[@name='UserName']")).
        ↪ sendKeys("wgen0001");
    driver.findElement(By.xpath("//input[@name='RequestId']")).
        ↪ sendKeys("0001User</lu:RequestId> <lu:CardNumber>
        ↪ --><lu:RequestId>0 or~/**/1");
    driver.findElement(By.xpath("//input[@name='submit']")). submit();

    String actualXML = driver.findElement(By.cssSelector("pre")).getText().
    replaceAll("\\s+", " ");

    Assertions.assertNotEquals(testObjective, actualXML);
  }
}
```

Listing 1: Example of test case generated by JCOMIX

Compared to random search and traditional GAs, COMIX targets all TOs at once. Therefore, it requires only one search process, thus, overcoming the limitations of budget allocation strategies [11, 14]. Previous studies showed that multi-targets search outperforms single-target ones for XMLi testing [11] as well as in other testing contexts [4, 14, 15]. Furthermore, we implemented COMIX in our tool since a prior study [11] showed that COMIX outperforms other multi-target solvers (i.e., MOSA [14] and MIO [4]) in the context of XMLi testing.

Post-processing. Generated tests that successfully lead to generating some TOs (malicious XML messages) when executed against the SUT are stored into the final test suite. Therefore, successful tests are post-processed and converted in JUnit test cases. An example of a test case generated by JCOMIX for the TO in Figure 2 is reported in Listing 1. The test case uses selenium to interact with the web interface, inserts the test inputs and fails because the XML generated by the SUT is equal to the target TO.

3 EVALUATION

To evaluate JCOMIX, we tested SBANK, which is a simplified and anonymized version of a front-end application interacting with a bank card processing system [10]. The version used in this study contains HTML forms and implements input processing routines. In our testing experiment, the internal web services are mocked to avoid that our tool compromises them. Besides, SBANK contains

```
...
  <lu:UserName>wgen0001</lu:UserName>
  <lu:IssuerBankCode>0111</lu:IssuerBankCode> <!--</lu:IssuerBankCode>
  <lu:RequestId>0001User</lu:RequestId>
  <lu:CardNumber>--> <lu:RequestId>0 or~/**/1</lu:RequestId>
  <lu:CardNumber>123456789123456</lu:CardNumber>
....
```

Figure 2: Example of target TO. The text highlight in red color is the malicious content added using our grammar-based TO generator.

Table 1: Number of TOs generated by JCOMIX for SBANK.

Attack Type	#Input	# Test Objectives
Replicating	1	12
	4	12
Replacing	1	0
	4	15
Deforming	1	28
	4	28
Random Closing Tag	1	4
	4	4

different forms with a different number of user inputs. In this experiment, we use the form with one and four inputs. The four inputs are: *UserName*, `IssuerBankCode`, `CardNumber`, and `RequestID`. Note that SBANK has been used in previous studies [10, 11] to assess the effectiveness of different search algorithms (including COMIX) in detecting XMLi vulnerabilities.

Test Objectives. For each form in SBANK, we use JCOMIX to generate the target TOs using our attack grammar and based on the attacks described in Section 2.1. Table 1 reports the number of TODs generated by JCOMIX for SBANK and classified by attack type. Note that attacks of type *Replacing* need at least two input fields. Hence, JCOMIX did not generate any TO for this type of attack for SBANK with one input (see Table 1). An excerpt of one TO generated by JCOMIX for SBANK is depicted in Figure 2.

Fitness Functions. In our empirical evaluation, we selected COMIX (i.e., the co-evolutionary algorithms) and compared three different variants of the fitness function [10, 11]. The first variant is the traditional *edit distance* (or Levenshtein distance) for strings, which counts the minimum number of characters to add, delete, or remove to/from a string A (XML message generated by the SUT) to obtained another string B (the target TO).

The second variant is the *real-coded* edit distance proposed in [10]. In this real-coded variant of the edit distance, the difference between characters corresponds to the relative distance between their corresponding ASCII codes. For example, let us consider the strings A=`abc` and B=`abb`. The edit distance between A and B is one, as we would need to replace one character (i.e., replace `b` with `c`) in A to obtain B. Instead, with the real-coded edit distance between A and B is $\phi(|98 - 99|)$, where $\phi(x) = x/(x + 1)$ is a normalization function [10], while 98 and 99 are the ASCII codes for the characters `b` and `c`, respectively.

However, the Levenshtein distance and the real-coded distance are particularly expensive for long strings. More specifically, their computational cost is $O(m \times n)$, where m and n are the lengths (number of characters) of the two strings to compare. Usually, XML messages contain hundreds of characters, resulting in hundreds of operations to perform to compare two strings. To reduce the computational cost of the string comparison, Jan et al. [11] proposed the *linear distance*:

$$d(A, B) = |n - m| + \sum_{i=1}^{\min\{m,n\}} \frac{|a_i - b_i|}{|a_i - b_i| + 1} \quad (1)$$

where a_i and b_i denote the ASCII codes for the characters in position i of A and B, respectively. Such a distance is less precise than the other two variants, but it has a much lower computational cost of $O(\min m, n)$. The linear distance is the third fitness function used in our evaluation

Table 2: Number of XMLi vulnerabilities detected by JCOMIX on SBANK with different fitness functions within 5 min.

#Input	Fit. Function	Mean	St. Dev.	Mean #Iterations
1	Real-Coded Dist.	44	2.05	15,458
1	Linear Dist.	45	1.30	21,649
1	Edit Dist.	44	2.49	15,953
4	Real-Coded Dist.	14	4.39	4,192
4	Linear Dist.	12	3.76	4,250
4	Edit Dist.	12	43.56	4,062

Parameter setting. In this evaluation, we use the parameter values suggested in the related literature [10, 11]. More specifically, we applied the character mutation with the probability $p_m = (1.75)/(\lambda\sqrt{L})$, where L is the length of the input string to mutate, and λ is the size of the islands. The crossover is applied within each island with a probability p_c=0.80. Parents are selected using the *tournament selection*. Instead, the size of the islands is dynamically computed and updated in COMIX [11]. We set an overall search budget of five minutes. Note that other parameter values can be used by editing the file `config.json`.

Results. We run JCOMIX five times for each fitness function variants and for each SBANK form (i.e., with one and four inputs). Table 2 reports the number of XMLi vulnerabilities detected by JCOMIX when using different fitness functions. In all cases, JCOMIX was able to generate successful XMLi attacks. For SBANK with one input, the linear distance detected 1 additional attack (on average) compared to the other fitness functions, also showing a better (smaller) standard deviation. For SBANK with four input, the real-coded distance led to discover more attacks than the alternative distances. Finally, our results confirm that the linear distance is less expensive than the alternative distances since JCOMIX could perform more iterations within five minutes. Listing 1 reports an example of successful XMLi attack generated by JCOMIX for SBANK with four inputs using the linear distance. The corresponding target TO is reported in Figure 2.

4 CONCLUSIONS

In this paper, we presented JCOMIX, a Java framework that generates successful XMLi attacks using search-based testing approaches. JCOMIX applies a two-steps approach: first, it generates malicious XML messages using an attack grammar covering different types of XMLi attacks; then, it uses a search algorithm to search for user inputs that, after being validated and sanitized, lead the front-end web application toward the generation of malicious XML messages.

We evaluate JCOMIX on a front-end application interacting with a bank card processing system. Our results show that JCOMIX can generate test cases exposing XMLi vulnerabilities. The general framework of JCOMIX can be extended to injection attacks for other data formats, such as JSON. Our future work will include other data formats, evaluate other search algorithms, and using machine learning to predict the test execution results [7].

REFERENCES

[1] [n.d.]. SmartBear ReadyAPI. http://smartbear.com/product/ready-api/overview/. Accessed: 2016-04-26.

[2] [n.d.]. WSFuzzer Tool. https://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project. Accessed: 2016-04-26.

[3] [n.d.]. XML Vulnerabilities Introduction. http://resources.infosecinstitute.com/xml-vulnerabilities/. Accessed: 2016-04-26.

[4] Andrea Arcuri. 2017. Many Independent Objective (MIO) Algorithm for Test Suite Generation. In *International Symposium on Search Based Software Engineering (SSBSE)*.

[5] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. 2002. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet computing* 6, 2 (2002), 86–93.

[6] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier. 2008. Detecting Buffer Overflow via Automatic Test Input Data Generation. *Computers & Operations Research* 35, 10 (Oct. 2008), 3125–3143. https://doi.org/10.1016/j.cor.2007.01.013

[7] Giovanni Grano, Timofey V Titov, Sebastiano Panichella, and Harald C Gall. [n.d.]. Branch coverage prediction in automated testing. *Journal of Software: Evolution and Process* ([n. d.]), e2158.

[8] Sadeeq Jan, Cu D. Nguyen, Andrea Arcuri, and Lionel Briand. 2017. A Search-based Testing Approach for XML Injection Vulnerabilities in Web Applications. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*.

[9] Sadeeq Jan, Cu D. Nguyen, and Lionel Briand. 2016. Automated and Effective Testing of Web Services for XML Injection Attacks. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA)*.

[10] S. Jan, A. Panichella, A. Arcuri, and L. Briand. 2017. Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications. *IEEE Transactions on Software Engineering* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2778711

[11] Sadeeq Jan, Annibale Panichella, Andrea Arcuri, and Lionel Briand. 2019. Search-based multi-vulnerability testing of XML injections in web applications. *Empirical Software Engineering* (13 Apr 2019). https://doi.org/10.1007/s10664-019-09707-8

[12] Meiko Jensen, Nils Gruschka, and Ralph Herkenhöner. 2009. A Survey of Attacks on Web Services. *Computer Science - Research and Development* 24, 4 (2009), 185–197. https://doi.org/10.1007/s00450-009-0092-6

[13] Sam Newman. 2015. *Building Microservices.* " O'Reilly Media, Inc.".

[14] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. 2017. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* (2017). To appear.

[15] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 1–10.

[16] James Ransome and Anmol Misra. 2013. *Core Software Security: Security at the Source.* CRC Press.

[17] S. Sharma, R. RV, and D. Gonzalez. 2017. *Microservices: Building Scalable Software.* Packt Publishing. https://books.google.lu/books?id=zU8oDwAAQBAJ

[18] Julian Thomé, Alessandra Gorla, and Andreas Zeller. 2014. Search-based Security Testing of Web Applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST 2014)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/2593833.2593835

[19] Jeff Williams and Dave Wichers. 2013. *OWASP, Top 10, The Ten Most Critical Web Application Security Risks.* Technical Report. The Open Web Application Security Project.