Application of combinatorial methods in web application security testing

Katarina Preradov, Mina Medić, Goran Sladić, Branko Milosavljević
University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia
{katarina.preradov, mina.medic, sladicg, mbranko}@uns.ac.rs

Abstract— **Web applications are one of the most vulnerable systems nowadays. Everybody uses them in everyday life and most often they handle sensitive information from users, which makes them very tempting to malicious users for crashing them or steal their data. Testing web applications against various cybersecurity attacks can lower the risk of malicious user to perform some action that can harm the system itself or its users. Adequate test data can reveal these vulnerabilities, but traditional testing methods can not determine which test sets are appropriate. It is not uncommon that testing comes down to brute force testing. Also, these techniques do not take care of the correlation between different parameters which, under certain conditions, affect the reliability of the system. In this paper, we apply the idea of a combinatorial testing technique to generate adequate test data sets for testing web applications against known cybersecurity vulnerabilities. We used ACTS for automatic test set generation. The results show that these test sets are effective and can reveal security issues. Also, the number of necessary test cases is reduced, which directly affects the time and money needed to adequately test the software.**

Keywords — **software testing, combinatorial testing, web security.**

I. INTRODUCTION

In recent decades, we have witnessed a growing presence of software in our daily lives. Errors encountered during software implementation represent one of the greatest security threats to the applications themselves, making testing a crucial step in software development. In 2003, NIST published a report [1] stating that the United States economy spends 59.5 billion dollars annually on inadequate software testing, which represents 50% of the budget provided for development. Testing all possible combinations of inputs and execution paths is impossible for most software, so testing high-risk software requires methods that involve a large number of resources, resulting in huge costs. For less critical software, budget constraints often limit the amount of testing that can be achieved, which directly increases the risk of system failure and poor security of the solution.

The software is made up of a large number of input parameters, each of which can have different values. For example, consider software that needs to work with complex numbers. In this case, the software will only take 4 different input parameters. Each of these parameters can have representative value in each test case, such as zero, negative number or a positive number. In order for the system to be fully tested, we need to try all combinations of all possible values, which are 81 different combinations

in this example. Tweaking costs can be too high, even in systems with few parameters. Real systems have much more than 4 input parameters and often these parameters can have a large number of possible values. The number of test cases is increasing exponentially as the number of parameters grows [2]. However, experience shows that many systems behave unexpectedly due to the combination of only a few parameter values. Therefore, it is not necessary to check all possible parameter combinations. This is the main motivation for applying combinatorial testing in practice, which aims to cover as many test cases as possible with as least as possible parameter combinations.

Nowadays, web applications are the most popular type of application in the world. No business company could reach maximum popularity without a web application, because the Internet became the most popular communication channel between potential customers and the business companies itself. The fact that web applications are publicly available to all and can be used by anyone directly affect the structure and user interface of the application itself. The design of such applications includes a lot of input fields through which users make interaction with the system. The average application requires a lot of user input, including sensitive information that has to be handled cautiously. Once we know all this features, the question being asked is how to develop the application to be resistant to various invalid user inputs, as well as their combinations, and how we can be sure that application is safe to use. As soon as we have a large number of different parameters and their combinations, the number of test cases becomes enormous and impossible to test all of it.

One critical and probably the most widely known security bug is known as Heartbleed bug in OpenSSL implementation of the Transport Layer Security (TLS) [3]. Message in this protocol consists of different attributes, including payload and payload_lenght. These two variables are related and the length of the actual payload must be equal to payload_length. In protocol implementation, developers assumed that this assumption is always satisfied, and therefore did not put any value checks in source code. If values that are outside the expected range are passed to these variables, malicious users would be able to read data that they should not have access to. Traditional methods of testing could not detect Heartbleed bug because it will only manifest itself if the combination of input parameters is not valid and does not cause the system to stop operating.

The purpose of this paper is to explain how to automate the process of web application security testing and lower the costs of the whole process. The primary goal to which

the greatest attention is paid in the paper is to produce realistic and valuable attack vectors who are able to detect vulnerabilities. This is achieved by using the knowledge of the experts. We focus on injection attacks, specifically XSS and SQL Injection (SQLi), which are described in details in papers [4] and [5] respectively.

This paper is structured as follows In Section II, we discuss the related work. In Section III, we introduce the concept of combinatorial methods and how they can be applied in the domain of web application security testing. In Section IV we present our solution. In Section V, we discuss the results we achieved and present limitations. The last section concludes our work and also introduce issues to be addressed as part of future work.

II. RELATED WORK

To improve the quality of the software, different testing methods have been proposed. Different methods aim to target different types of system failures. During research, we found a lot of papers with focus on problem of automating process of web application security testing, and therefore a lot of different ideas and solutions have been described. We combine different aspects to propose a solution that will be adaptable, easy to implement and reliable in this domain.

For example, mutational testing [6] is error-guided testing. Mutations represent small changes to the source code similar to the mistakes a developer would make. There are studies based on the application of mutational testing to detect security issues in web applications. For example, in paper [7], the authors introduce a solution that relies on using this method of testing for generating adequate test sets to detect XSS vulnerabilities. They also developed a tool for automatic generation. The main disadvantages of this approach are that mutation testing is time-consuming, expensive, difficult to implement and is not applicable for black-box testing as it involves a lot of source code changes.

Metamorphic testing [8] is a testing technique that generates new test cases based on those that did not find any errors. It is used in combination with other techniques that aim to generate a base set of test cases. Authors in [9] discuss how metamorphic testing can be used for negative testing and mitigate oracle problem in the testing of security-related features. A major limitation of this solution is that users can not specify their security requirements, which represent one of the future research directions as well as automatic validation.

Authors in [10] suggest formalisation of attack patterns from which test cases can be generated. They proposed model-based testing technique which uses standard UML notations in order to describe XSS and SQLi attack pattern. Their solution is automated by using tools for executing UML models.

There are various tools that use different approaches to automate software testing. One such tool is the Ardilla presented in paper [11]. The tool generates attack vectors for both XSS and SQLi. Its main disadvantages is that the tool is intended exclusively for PHP applications. To successfully generate test cases, the tool also requires the initial state of the database.

QED is another tool for automatic generation of XSS and SQLi attack vectors. It is presented in paper [12].

QED uses Goal-Directed Model Checking and can be used only for Java based web applications.

III. METHODOLOGY

Two approaches have been analysed to propose appropriate solution. In the remaining of this section, we give a brief overview for both of them along with the benefits and drawbacks.

A. "brute force" testing

The first approach would be to use "brute force" testing which is usually done by a person. In order for this approach to be successful in the field of web application security, there are several things a tester must satisfy:

- understand what attack is and based on its scenarios create valid test cases,
- analyse causes and consequences,
- execute test cases.

There are not many testers with sufficient knowledge of security flaws and different kinds of attacks. Besides this, brute force testing is very time consuming because it implies that the tester manually executes all test scenarios, which is also prone to human errors.

B. Combinatorial testing for input generation

In this part of the chapter, we will introduce some basic concepts on which combinatorial testing is relying. Suppose that the software has n parameters: c_i $(i = 1, 2, ..., n)$. The parameter c_i has ai possible discrete values from the finite set V_i, $a_i = |V_i|$. The values of these parameters or their specific combination directly affect the behavior of the software. R is a set of relations that represent the interaction between parameters. This set defines which combinations of parameters should be taken into account when testing [13]. This set is created based on the project specification or by consulting a domain expert. As a test suite, combinatorial testing uses covering arrays. A t-way Covering Array (CA) of magnitude N, with the number of factors n and strength t is a n x N matrix having the following properties:

- each column i, $(1 \leq i \leq n)$ contains only the elements of the set V_i,
- the rows of each N x t substring cover all t-tuples values from the t column at least once [13].

As already mentioned, the main advantage of combinatorial testing lies in the rule of interaction: all or almost all failures in the software are due to the interaction of only a few parameters [14]. The use of combinatorial testing for the purpose of selecting configuration parameters can make testing much more efficient, however it can also be further improved by selecting the values of input parameters.

This approach is proved to be an effective alternative of brute force testing. Because of this, it is going to be analysed in remaining chapters as part of solution.

IV. SOLUTION

The focus of our work is a combinatorial testing technique that aims to create high-coverage tests while minimizing costs. These test sets will be used for the detection of security vulnerabilities in web-based applications. Since the technique is specification-based, knowing the implementation of the source code being tested is not required. Combinatorial testing is based on

the assumption that a large number of software errors are a direct consequence of the interaction of two or more parameters. The popularity of the technique is also supported by the fact that there are several tools available to automatically generate test cases.

We used ACTS (Automated Combinatorial Testing for Software) for test set generation [15]. This tool is free and developed by the NIST organization. It helps generate t covering arrays, with t being in the interval from 1 to 6. Empirical research has shown that t = 6 level successfully covers testing of almost all applications. The case when t = 1 is a special form of testing called base-choice testing (BCT). BCT requires that each parameter value has to be used at least once and in a test in which all other values are base choices. Each parameter has one or more values that are characterized as base choices. Base choices values are those values that are "more valuable" than others, for example, baseline values or those which are most commonly used. Certain combinations of parameters can be invalid, and as such must not be part of the resultant test set. ACTS provides users to specify, via the command or GUI interface, the limitations in parameters that their combinations must comply with. The tool will take these limitations into account while generating test cases.

Combinatorial techniques were used for generating attack vectors to reveal if the web application is vulnerable on injection attacks, specifically XSS and SQLi. Structure of values in attack vectors are described through grammar. ACTS uses these grammar and its parameters to generate test set, taking care of constraints that must be satisfied. Rules for grammars are developed based on expert knowledge from people working in industry in web application security sector.

The main reason why we decided to use grammar is the fact that grammar could easily be modified and expanded. This is a key feature because today almost everything is web-based and the web is still vulnerable. There are many proven methods to protect against known cyber-attacks and security mechanisms are constantly being improved, but on the other hand, hackers are constantly coming up with new techniques to disrupt the reliability and security of a web application.

Regular testers do not have sufficient knowledge to test web applications against various cyber attacks. We used the knowledge of penetration testers and people from the industry to create the most accurate grammar, intending to spot as many vulnerabilities as possible in the applications themselves.

In the remaining of this chapter, fragments of created grammars are shown, as well as the constraints that we have been taken into account. Our grammar for XSS extends the one introduced in [16] by adding new HTML tags and executable JavaScript code. We also take into account different encoding for closing tags. The main contribution of this work is that grammar for SQLi attack is introduced, along with constraints. At the end of this chapter, it is described how our solution is verified against a real application.

A. XSS attack grammar

Part of our XSS grammar is presented in Figure 1. Even for this small grammar there is total combinations. This means that we need 3840 different test cases to cover all possible input values, which is huge number, especially since some values are not valid and their presence in the attack vector does affect revealing vulnerabilities.

Opening Tag represents the starting of HTML tag where JavaScript code could be injected. *Closing Tag* is essential in order to produce valid HTML or JavaScript code. If either of them is invalid, generated value is not relevant because it would not execute on the web page. We considered different encoding for the closing tag, for example *"/>"* and *"%2>"*. This is important because some applications have mechanisms to automatically escape *"/"* character, and the application appears to be prone to XSS attack, but *"%2"* reveals this vulnerability. *JavaScript Code* represents executable code, including some basic statements and even cookie stealing, which is a serious issue. The *Attributes* parameter is combined with *Opening Tag*. It is of great importance because malicious code does not always have to be embedded on the page exclusively over the HTML tag, but it could be executed through, for example, the *"src"* attribute. *Event Handlers* are commonly used to bypass filters while executing an XSS attack, so considering them to include in the test set is crucial. *Terminating Characters* are used to properly close attributes or event handlers, in order to produce valid values.

B. SQL Injection attack grammar

Part of our SQLi grammar is presented in Figure 2. For this grammar we have 4032 total combinations, which means that we need 4032 different test cases to cover all possible values of input. It is important to note that grammar for SQLi is not unique for all applications, as was the case with XSS grammar.

SQL parameter represents a fragment of real SQL command which can be executed by interpreter. With adding this special values in query, malicious user can perform actions on database and compromise confidentiality, integrity or availability. *Line Comments* are inserted in queries and are used to ignore part of original query, for example *SELECT * FROM users WHERE name = 'admin' --AND password = 'password'*. Parameter *Value Substitute* is some dummy value, string and numeric, which will be passed in query. This substitute values are usually used combined with other statements when the value does not affect query execution. *Special Character ";"* allows query chaining which will allow malicious user to append one or more queries to the end of the actual one, such as dropping table or entire database. The parameter named *Special Statements* has the values of some predefined SQL commands which can be combined with an existing ones. *Terminating Characters* are used to properly close values of *SQL* parameter, to produce valid instruction that can be executed by SQL interpreter. Parameter *Entities* is application specific. Values of this parameter represent names of tables in database. To properly define possible values, this approach requires some prior knowledge about entities in system. This parameter could be omitted, but if used it significantly raises the probability of detecting vulnerability to an SQLi attack.

Opening Tag	<script>, <img, <a, empty, ...
Closing Tag	</script>, <%2fscript>, />, %2f>, empty, ...
JavaScript Code	alert('XSS'), console.log(document.cookie), javascript:alert('XSS'), window.location('someUrl?cookie=' + document.cookie), empty, ...
Attributes	src=', href=', tooltip=', empty, ...
Event Handlers	onLoad='(, onClick='(, onKeyPress='(, empty, ...
Terminating Characters	',)', empty

Figure 1 - XSS attack grammar

SQL	OR 1=1, ' OR '1'='1, ' OR ''=', empty
Line Comments	#, --, empty
Special Characters	;, empty
Value Substitute	SomeString, 1, empty
Special Statements	UNION SELECT * FROM, DROP TABLE, DELETE FROM, empty
Terminating Characters	', empty
Entities	employees, user_data, user_system_data, users, articles, access_logs, empty

Figure 2 -SQLi attack grammar

C. Constraints

Constraints are considered during test set generation to prevent creating invalid HTML and JavaScript code or SQLi strings. Invalid values are usually noise in data that does not reveal much and they are excluded from the resulting test set. Without them, test sets are smaller and more effective. Constraint sets for XSS and SQLi grammar are presented bellow on Figure 3 and Figure 4 respectively. Constraints are written in ACTS constraint notation, using a form of first-order logical formulas. For example, following constraint *OpeningTag=<script> => ClosingTag=</script> || ClosingTag=<%2fscript>* means that if parameter *OpeningTag* has value *<script>* then value of *ClosingTag* parameter must be either *<script>* or *<%2fscript>*. ACTS supports three types of operators: (1) Boolean operators, including *&&, ||, =>;* (2) Relational operators, including *=, !=, >, <, >=, <=;* and (3) Arithmetic operators, including *+, -, *, /, %* which can be used in constraint modeling.

D. Generating attack vector

Each grammar and constraints defined for it in ACTS belong to one system which is described in XML format. We define two systems: one for XSS and one for SQLi. The tool uses parameters and values described through grammar and combines them with each other taking into account constraints to build a test set. It is possible to configure build parameters, such as strength *t* and combinatorial algorithms. Parameter *t* can have value in range $1 \leq t \leq 6$. Supported algorithms are IPOG, IPOG-F, IPOG-F2, IPOG-D, and Base Choice. We used the IPOG algorithm because our system have less that 20 parameters and 10 values per parameter. Result test sets can be exported to CSV, TXT or XLS format.

E. Verification

A concrete application of attack vectors generated by ACTS was demonstrated on WebGoat web application. WebGoat is an open-source web application, maintained by Open Web Application Security Project (OWASP), which is insecure by design [17]. The main purpose of this application is to teach about the most common security issues. We decided to use this application because it has some known security issues, such as XSS or SQL injection, and to prove that such test cases that are generated using combinatorial techniques can reveal

these vulnerabilities. Generated test cases are inserted into the application manually and we succeeded to detect specified injection vulnerabilities.

V. RESULTS

In this section we present our results. Resulting test sets are described for both grammars defined in Section IV. We discuss limitations at the end of this chapter.

The final test set contains 37 results for XSS attack vector and 28 results for SQLi attack vector (*strength = 2*). This means that for the XSS attack vector we need only 37 different test cases to cover all 156 possible 2-way combinations. For SQL, we need 28 test cases for covering all 248 combinations with strength 2. The graphical representation of our results is shown in Figure 5 and Figure 6 for XSS and SQLi respectively. The x-axis (Tests) shows the number of test cases required, and the y-axis (Coverage ratio) illustrates the percentage of covered combinations. As the parameter *t* increases, so does the number of test cases required to achieve 100% coverage.

The main limitation of our solution is that grammars are separated. Using ACTS, attack vector for each grammar is produced and these values are entered in application manually. With this solution, we automate only a part of whole web application security testing process, and that is input generation, which was primary goal of this research. Automating passing inputs in software is still an open issue.

VI. CONCLUSION AND FUTURE WORK

The main contribution of this research is that required number of test cases are reduced while maintaining the percentage of code coverage of tests and preserving its quality. It was decided to use combinatorial testing, mostly because it does not require to be familiar with source-code and there are a series of studies done by NIST for a couple of years that support the effectiveness of combinatorial testing. We focused on web-based applications and detecting security vulnerabilities in them. This method could also reveal faults that are rare and only occur under special conditions. Heartbleed bug is an example of such fault, as it manifests only if related parameters have values that are not in relation. It took almost 2 years for developers to notice this flow. During this time, 2/3 of all applications on the Internet used OpenSSL. If combinatorial testing was used, this critical bug could be revealed much earlier.

(OpeningTag="<script>") => (ClosingTag="</script>" || ClosingTag="<%2fscript>")
(OpeningTag="<img" || OpeningTag="<a") => (ClosingTag="/>" || ClosingTag="%2f>")
(OpeningTag=" <script>") => (EventHandlers="empty" && Attributes="empty" && TerminatingCharacters="empty")
(OpeningTag="empty") => (ClosingTag="empty" && EventHandlers="empty" && Attributes="empty" && TerminatingCharacters="empty" && JavaScriptCode="javascript:alert('XSS')")
(Attributes!="empty") => (EventHandlers="empty" && TerminatingCharacters="'")
(EventHandlers!="empty") => (Attributes="empty" && TerminatingCharacters=")'")
(Attributes="href=") => (OpeningTag="<a")
(Attributes="empty" && EventHandlers="empty") => TerminatingCharacters="empty"

Figure 3 - Constraints for XSS grammar

(SpecialCharacters=";") => (SpecialStatements != "empty" && Entities != "empty")
(TerminatingCharacters="'") => (LineComment!="empty")
(SpecialStatements != "empty") => (SpecialCharacters=";" && Entities != "empty")
(TerminatingCharacters == "'") => (LineComment = "#" || LineComment = "--")
(ValueSubstitute="empty") => (SQL!="' OR '1'='1')
(SQL="' OR '1'='1) => (TerminatingCharacters="empty")

Figure 4 - Constraints for SQLi grammar

# Of Tests:	37
# Of Covered Combinations:	156 (2-way: 156)
Execution Time:	0.012 sec

Figure 5 – XSS Statistics

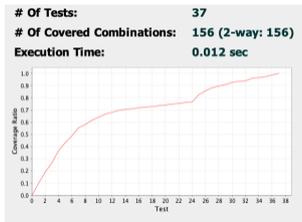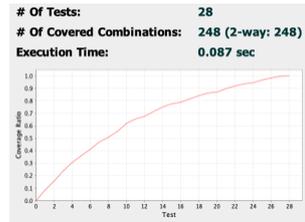| # Of Tests: | 28 |
| # Of Covered Combinations: | 248 (2-way: 248) |
| Execution Time: | 0.087 sec |

Figure 6 – SQLi Statistics

The grammars created for the purposes of this research aims to demonstrate that the use of combinatorial techniques can automate process of web application security testing by generating adequate attack vectors which are able to detects serious flaws. Our solution is verified against WebGoat application.

As part of our future work, attention will be paid to full automation of web application security testing process. In the current solution, generated test sets can be exported to CSV, TXT or XLS format. In the future, these exported test sets can be passed to some software that will merge all inputs into concrete strings. After that, they can be passed to automated software testing tools and thus provide a solution that will be adaptive to any possible changes and automate the testing process completely.

Another part of future work would be an improvement of designed grammars and constraints. To achieve this, we should focus on addressing the following statements:

- consult experts for more real examples from industry in order to expand given grammars and improve constraints,

- define a grammar for other injection attacks, such as XPath, code injection, or command injection,

- consider excluding some constraints and analyzing the impact of invalid values in detecting injection vulnerabilities, because in some special cases, these invalid values can reveal hidden flaws in the application which valid ones cannot.

REFERENCES

[1] Tassey G. "The economic impacts of inadequate infrastructure for software testing." National Institute of Standards and Technology, 2002. RTI Project 7007.011 (2002): 429-489.

[2] Zhang J., Zhang Z., Ma F "Automatic generation of combinatorial test data" Springer Berlin Heidelberg, 2014.

[3] Vassilev A., Christopher C. "Avoiding cyberspace catastrophes through smarter testing.", 2014. Computer 47.10 (2014): 102-106.

[4] Fogie S., Grossman J., Hansen R., Rager A., Petkov P. D. "XSS Attacks: Cross Site Scripting Exploits and Defense" Syngress, 2007.

[5] Clarke J., Fowler K., Oftedal E., Alvarez R. M., Hartley D., Kornbrust A., O'Leary-Steele G., Revelli A., Siddharth S., Slaviero M., "SQL Injection Attacks and Defense", Second Edition. Syngress, 2012.

[6] Yue J., Harman M. "An analysis and survey of the development of mutation testing." IEEE transactions on software engineering, 2010. 37.5 (2010): 649-678.

[7] Hossain S., Zulkernine M. "Mutec: Mutationbased testing of cross site scripting." Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems. IEEE Computer Society, 2009.

[8] Zhi Quan Z., et al. "Metamorphic testing and its applications." Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004). Xian, China: Software Engineers Association, 2004.

[9] Tsong Yueh C., et al. "Metamorphic testing for cybersecurity." Computer 49.6 (2016): 48-55.

[10] Bozic J., Wotawa F. "Security Testing Based on Attack Patterns", IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, 2014.

[11] Kieyzun A., et al. "Automatic creation of SQL injection and cross-site scripting attacks.", 2009 IEEE 31st international conference on software engineering. IEEE, 2009.

[12] Martin M. C., Lam M. S. "Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking." USENIX Security symposium, 2008.

[13] Nie C., Leung H.. "A survey of combinatorial testing". ACM Computing Surveys (CSUR), 2011.

[14] Kuhn D. R., Raghu N., Kacker Y. L.. "Introduction to Combinatorial Testing". CRC Press, 2013.

[15] User guide for ACTS, URL: https://csrc.nist.gov/CSRC/media/ Projects/Automated-Combinatorial-Testing-for-Software/ documents/ acts_user_guide_2_92.pdf. Retrieved: April 2020.

[16] Bozic J., Garn B., Kapsalis I., Simos D., Winkler S., and Wotawa F. "Attack Pattern-Based Combinatorial Testing with Constraints for Web Security Testing". 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015.

[17] Web Goat. URL: https://owasp.org/www-project-webgoat/, Retrieved: April 2020.