On the Applicability of Combinatorial Testing to Web Application Security Testing: A Case Study

Bernhard Garn
SBA Research
A-1040 Vienna, Austria
bgarn@sba-research.org

Ioannis Kapsalis
SBA Research
A-1040 Vienna, Austria
ikapsalis@sba-research.org

Dimitris E. Simos
SBA Research
A-1040 Vienna, Austria
dsimos@sba-research.org

Severin Winkler*
Security Research
A-1040 Vienna, Austria
swinkler@securityresearch.at

ABSTRACT

Case studies for evaluating tools in security testing are powerful. Although they cannot achieve the scientific rigor of formal experiments, the results can provide sufficient information to help professionals judge if a specific technology being evaluated will benefit their organization. This paper reports on a case study done for evaluating and revisiting a recently introduced combinatorial testing methodology used for web application security purposes. It further reports on undertaken practical experiments thus strengthening the applicability of combinatorial testing to web application security testing.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Testing and Debugging; G.2 [**Discrete Mathematics**]: Combinatorics; K.6 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Theory, Security

Keywords

Combinatorial testing, security testing, penetration testing tools, web application security

1. INTRODUCTION

Covering security issues still remains a big task of the testing community in academic circles as well as industrial ones. Ensuring safety and security of today's software and systems is one of the big challenges of industry but also of society especially when considering infrastructure and the still increasing demand of connecting services and systems via the internet and other means of communication. Whereas safety concerns have been considered as important for a longer time, this is not always the case for security concerns. Since security threats often are caused because of bugs in software or design flaws, quality experiments are highly required.

In this paper we provide a framework for testing and detection of both reflected and stored XSS in web applications. The framework partly builds upon previous work [1] and is comprised of two parts: an (automated) combinatorial testing method for generating the input data to be submitted to the SUT, and a semi-manual one for executing the attack vectors in penetration testing tools. The underlying principle is to use combinatorial testing for concretizing the input and penetration testing tools to execute the various attack vectors.

In our case study we use the combinatorial testing tool ACTS [8] for generation of input strings by specifying parameters and constraints in order to structure the inputs for the particular application domain. Once specified, these inputs are used by the attack pattern model in order to submit them to a web application. The goal of our approach is to cover standard XSS exploitation attempts by checking whether certain parts of the SUT are vulnerable to potentially malicious scripts. The main differences to other techniques and tools rely in the generation, structure and execution of test cases.

A detailed overview of XSS can be found in [7]. Execution of attack vectors with other similar methods, like model-based testing can be found in [2]. We discuss briefly related work concerning attack grammars for fuzz testing of XSS attack vectors. In particular, such grammars were employed in [9] and [5, 6] where learning and evolutionary approaches to detect vulnerabilities has been employed, respectively.

The paper is structured as follows: Section 2 gives a brief explanation of combinatorial testing and its potential contribution to security testing. Afterwards, Sections 3 and 4 demonstrate our case study and the penetration testing tools we have used. Then Section 5 discusses the testing results of our approach against other web applications in the different tools and finally, Section 6 concludes the work.

2. COMBINATORIAL SECURITY TESTING

Testing a SUT requires the existence of test cases and in par-

*Authors are listed in alphabetical order.

ticular a method capable of generating such test cases. For developing our testing framework we can also use methods that arise from the field of combinatorial testing, which is an effective testing technique to reveal errors in a given SUT, based on input combinations coverage. Combinatorial testing of strength t (where $t \geq 2$) requires that each t-wise tuple of values of the different system parameters is covered by at least one test case. Recently, some researchers [3, 10] suggested that some faults or errors in SUTs are a combination of a few actions when compared to the total number of parameters of the system under investigation. In that sense, combinatorial testing is motivated by the selection of a few test cases in such a manner that good coverage is still achievable. The collection of test cases comprises of a test suite and the execution of such a suite is called a test run. A detailed description of combinatorial security testing can be found in [1].

In our case study, we have considered a general structure for XSS attack vectors (test inputs) where each one of them is comprised of 11 discrete parameters (types) briefly discussed below. This structure builds upon a combinatorial grammar given in [1] by relaxing constraints and modelling white spaces that appear in an XSS attack vector. In particular, we consider an attack vector to have the following form:

AV := (JSO,WS1,INT,WS2,EVH,WS3,PAY,WS4,PAS,WS5,JSE).

We briefly describe each one of the types listed in AV:

- The JSO (JavaScript Opening Tags) type represent tags or that open a JavaScript code block. They also contain values that use common techniques to bypass certain XSS filters, like <script> or <img.
- The WS (white space) type family represents the white space but also variations of it like the tab character in order to circumvent certain filters, like tab or space.
- The INT (input termination) type represents values that terminate the original valid tags (HTML or others) in order to be able to insert and execute the payload, like "'> or ">.
- The EVH (event handler) type contains values for JavaScript event handlers. The usage of JavaScript event handlers is a common approach to bypass XSS filters that filter out the typicall JavaScript opening tag like <script> or filters that remove brackets (especially < and >) like onLoad(or onError(.
- The PAY (payload) type contains executable JavaScript. This type contains different types of executable JavaScript in order to bypass certain XSS filters like alert("XSS") or ONLOAD=alert('XSS')>.
- The PAS (payload suffix) type contains different values that should terminate the executable JavaScript payload (PAY state value). The PAS is necessary to produce valid JavaScript code that is interpreted by a browser like ') or '>.
- The JSE (JavaScript end tag) type contains different forms of JavaScript end tags in order to produce valid JavaScript code like </script> or >.

The full description of possible type values per type will appear elsewhere as this is not the topic of the current case study.

The next step in the combinatorial test design process employs the notion of mixed-level covering arrays (a specific class of combinatorial designs). For the sake of completeness we provide below the definition of mixed-level covering arrays taken from [4] since this is the underlying generated structure in the ACTS tool:

DEFINITION 1. *A mixed-level covering array which we will denote as $MCA(t, k, (g_1, \ldots, g_k))$ is an $k \times N$ array in which the entries of the i-th row arise from an alphabet of size g_i. Let $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots, k\}$ and consider the subarray of size $t \times N$ by selecting rows of the MCA. There are $\prod_{i=1}^{t} g_i$ possible t-tuples that could appear as columns, and an MCA requires that each appears at least once. The parameter t is also called the strength of the MCA.*

For all cases we shall consider in this paper, the parameters of the MCA are derived from the types that form an XSS attack vector according to the following formulation. The number of rows of the MCA equals to the number of types in the presented form of the attack vector while the size of alphabets g_i of the MCA equals to the number of different values per type. For example, a sample of XSS attack vectors when we want to test for pairwise interactions ($t = 2$) derived from an $MCA(2, 11, (3, 3, 3, 3, 3, 9, 11, 14, 15, 23))$ are given below:

```
<img    ">>onError(␣<XSS>')
<IMG␣␣␣␣'>>␣onLoad(␣<XSS>␣;␣</script>
␣␣␣␣//';␣onError(␣␣␣<XSS>;//␣␣␣␣\>
<<';onLoad(␣HREF="//ha.ckers.org/.j">    ;
```

Sample of XSS Attack Vectors

3. CASE STUDY

The SUTs used in our case study were comprised of a set of web applications that are included in the open Web Application Security Project (OWASP) Broken Application Project[1]:

- Training Applications
  - Webgoat - version 5.4
  - Mutillidae II - version 2.6.3.1
  - Damn Vulnerable Web Application (DVWA) - version 1.8
- Realistic, Intentionally vulnerable Applications
  - Gruyere - version 201-07-15
  - Bodgeit - version 1.3

All five web applications that we tested, were deployed locally. Depending on the type of each tested input, we examined both for reflected cross-site scripting (RXSS) and stored cross-site scripting (SXSS) attacks. These applications are either training or realistic, intentionally vulnerable applications. They include different kind of vulnerabilities but for the needs of our case study we focused only on XSS vulnerabilities. Each application contains one or more XSS vulnerabilities, both reflected and stored, for testing purposes which were tested using attack vectors generated by the combinatorial security testing methodology depicted in Section 2. Apart from offering the desired vulnerabilities, these web applications are ideal for our approach since some of them include several levels of security or depending on the input field they implement different input validation mechanisms. Higher security settings result in stricter filters that are harder to bypass. For example, DVWA and Mutillidae have 3 different security levels and in each one of them there are different filters applied. Moreover, in Bodgeit the

[1] https://www.owasp.org/index.php/OWASP_Broken_Web_Applications_Project

input filed in the contact page has a basic input validation by escaping some special character when the input field in the search page has no input validation. Similarly, DVWA depending on the security level used, the parameter `csrf-token` has different value resulting in different security requirements. This differentiation on the security levels benefits our case study since we can measure and identify how well the produced vectors are performing when different security requirements are taken into account.

The procedure followed in order to identify weaknesses in the web applications is as following: First we browsed throughout the applications with the Proxy module enabled. We can use a number of penetration testing tools in this step, like Burp or ZAP (c.f. Section 4), however in our case we used only the Burp Proxy. Browsing through the applications allows us to capture all web traffic between the browser and the internet. We especially focused on requests and responses linked to a page that includes a user side input field, since such fields are used as injection point for an XSS attack. By examining the requests and responses we are able to manually identify potential vulnerabilities. Following this manual procedure allowed us to gather all the vulnerable input fields that were used in our evaluation. All the vulnerable input fields were tested manually before deploying our automated attacks in order to verify that such an attack is feasible and the specific input field is appropriate for our testing. We opted not to use any other automated tool, like the Scanner offered by Burp, because the testing applications have limited number of vulnerable inputs which can be easily identified.

4. PENETRATION TESTING TOOLS

In this section we provide details about the two penetration testing tools we have used in our case study. We give a detailed description of their mechanics, functionality and test oracles, applicable when testing for XSS attacks.

4.1 Burp Suite

The Burp Suite[2] is an integrated platform for performing security testing of web applications. It was designed to fulfill many penetration testing tasks and assist security professionals in each step of a test. In order to test the vector the Intruder module may be used. This module allows automating customized attacks against web applications, to identify and exploit all kinds of security vulnerabilities including XSS attacks. Moreover, it allows to pick from various pre-created lists or create custom lists. Multiple values can be examined with each request in any desired combination. The Intruder module may also be used with a custom list, which was the case in our experiments.

One test run was executed using the following process. We browsed the desired application in order to gather sample requests on the targeted pages that include a vulnerable input field. In this way we can easily store cookies and recreate a "natural ". This is especially useful, when POST requests are employed to send the data to the server. Next, we store the HTTP request to the page containing a potentially vulnerable parameter in the Burp Suite Intruder module. In case, authentication is needed beforehand, the cookie is passed along as well. Afterwards, we mark the location of the potentially vulnerable parameter within the HTTP request and configure the attack payload (i.e. the produced XSS attack vectors from Section 2) and performance options. The performance options specifically include the number of parallel requests to be sent. We set the number of threads to be used equal to five. This has to be determined empirically, as it depends both on the connection speed as well as on the performance of the test system. Finally we execute the attack and decide whether an attack payload is flagged as an XSS using the test oracle described below. Burp offers the feature to store the results into a CSV file for easier post-processing.

The oracle within Burp Suite was implemented using the "Search responses for payload strings" configuration option within the intruder. This option flags all results where the payloads were exactly reflected to the output page. The rationale behind this decision is that if the vector was not blocked or potential dangerous characters were not stripped out, we assume an XSS vulnerability was triggered.

4.2 OWASP ZAP Tool

The second penetration tool used in our case study was the open source tool OWASP Zed Attack Proxy (ZAP)[3]. Our goal is to obtain an automated testing process where it is required the minimum interaction from the tester. Similarly to the intruder from ZAP, the fuzzer tool in ZAP is offering this automation and it can be used for our attacking procedure which includes exploitation of vulnerabilities using XSS attacks. In our case the fuzzer is not used as it would be in a usual fuzzing process, which involves injections of invalid, unexpected or random data to the tested application through different inputs. In particular, the fuzzer in ZAP, apart from the provided fuzz files, gives the ability to add your custom fuzz file which is then used as a source in the fuzz process. Therefore, we added our custom files that included the XSS attack vectors produced based on our combinatorial testing approach (see Section 2). An additional feature that this fuzzer offers, is the number of threads to be used while fuzzing. To keep it aligned with Burp we used five threats.

The fuzzing process in ZAP is trivial and requires only a few actions from the user. Firstly, we have to access the targeted web application and send some random string as a parameter using a vulnerable input field. ZAP while acting as a proxy and intercepting the request, will provide us with all necessary details. In order to start the fuzzing process we have to highlight the potentially vulnerable parameter and select the source file with the attack vectors. Subsequently, the fuzzer will produce new requests to the web application and in each request it will serially parse the custom source file and replace the highlighted set of characters with one of our XSS attack vectors. As is the case in Burp, this procedure allowed us to reproduce a "natural" request and also pass the original cookie along with this kind of request. The test oracle in ZAP is based on a simple procedure where if the injected vector is included as a whole in the response from the web application, then it is determined that this specific vector was reflected, and therefore we had a successful XSS attack.

5. EVALUATION

This section describes the evaluation of our case study test results given in Table 1. Coverage is expressed as a percentage of the positive attack vectors out of the total we used for testing each application and respective input parameter ID and difficulty level for each combinatorial strength. Moreover, we further analyze some practical experiences learned by comparing our findings on both used penetration testing tools and also give a flavor of a combinatorial testing evaluation in terms of penetration testing.

5.1 Test Results Evaluation with Burp Suite

For the evaluation of the 5 applications the Burp Suite was used as mentioned in section 4.1. The obtained results are depicted in

[2] `http://portswigger.net/burp/`

[3] `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`

Table 1: Evaluation Results

Strength	Application	Parameter ID	Type of vulnerability	Difficulty level	ZAP: # positive vectors	Burp: # positive vectors	Total # of XSS attack vectors	Coverage (%) in ZAP	Coverage (%) in Burp	Execution time (secs) in ZAP	Execution time (secs) in Burp
2	Mutillidae	1	RXSS	0	345	345	345	100,000	100,000	57	109,4
2	Mutillidae	2	RXSS	0	74	8	345	21,449	2,319	41	289,3
2	Mutillidae	3	RXSS	0	345	38	345	100,000	11,014	45	339,3
3	Mutillidae	1	RXSS	0	4876	4876	4876	100,000	100,000	1562	599,3
3	Mutillidae	2	RXSS	0	1101	122	4876	22,580	2,502	1794	3640,4
3	Mutillidae	3	RXSS	0	4876	593	4876	100,000	12,162	1834	3739,0
4	Mutillidae	1	RXSS	0	53707	53707	53707	100,000	100,000	7463	1016,8
4	Mutillidae	2	RXSS	0	16165	1300	53707	30,098	2,421	8878	733,2
4	Mutillidae	3	RXSS	0	53707	6330	53707	100,000	11,786	7851	722,6
2	Mutillidae	1	RXSS	1	345	345	345	100,000	100,000	46	55,7
2	Mutillidae	2	RXSS	1	74	8	345	21,449	2,319	41	96,6
2	Mutillidae	3	RXSS	1	345	38	345	100,000	11,014	52	58,6
3	Mutillidae	1	RXSS	1	4876	4876	4876	100,000	100,000	1826	875,9
3	Mutillidae	2	RXSS	1	1101	122	4876	22,580	2,502	1722	681,5
3	Mutillidae	3	RXSS	1	4876	593	4876	100,000	12,162	1642	565,0
4	Mutillidae	1	RXSS	1	53707	53707	53707	100,000	100,000	8456	1087,8
4	Mutillidae	2	RXSS	1	16165	1300	53707	30,098	2,421	8649	880,6
4	Mutillidae	3	RXSS	1	53707	6330	53707	100,000	11,786	8014	815,3
2	Mutillidae	1	RXSS	5	0	0	345	0,000	0,000	100	59,0
2	Mutillidae	2	RXSS	5	0	0	345	0,000	0,000	48	54,9
2	Mutillidae	3	RXSS	5	345	38	345	100,000	11,014	44	120,8
3	Mutillidae	1	RXSS	5	0	0	4876	0,000	0,000	1624	1565,6
3	Mutillidae	2	RXSS	5	0	0	4876	0,000	0,000	1663	1543,2
3	Mutillidae	3	RXSS	5	4876	593	4876	100,000	12,162	1561	1375,2
4	Mutillidae	1	RXSS	5	0	0	53707	0,000	0,000	7958	1431,7
4	Mutillidae	2	RXSS	5	0	0	53707	0,000	0,000	7768	2550,1
4	Mutillidae	3	RXSS	5	53707	6320	53707	100,000	11,768	7861	2177,0
2	Bodgeit	1	RXSS	0	345	38	345	100,000	11,014	34	0,6
2	Bodgeit	2	RXSS	0	1	9	345	0,290	2,609	42	0,6
2	Bodgeit	3	SXSS	0	66	57	345	19,130	16,522	34	0,6
2	Bodgeit	4	SXSS	0	0	0	345	0,000	0,000	33	0,5
3	Bodgeit	1	RXSS	0	4876	593	4876	100,000	12,162	491	7,5
3	Bodgeit	2	RXSS	0	12	117	4876	0,246	2,400	497	7,6
3	Bodgeit	3	SXSS	0	880	831	4876	18,048	17,043	497	7,4
3	Bodgeit	4	SXSS	0	0	0	4876	0,000	0,000	488	7,1
4	Bodgeit	1	RXSS	0	53707	6330	53707	100,000	11,786	2934	158,6
4	Bodgeit	2	RXSS	0	10	1279	53707	0,019	2,381	7606	273,3
4	Bodgeit	3	SXSS	0	8877	8996	53707	16,529	16,750	6540	298,6
4	Bodgeit	4	SXSS	0	0	8997	53707	0,000	16,752	6479	168,6
2	Gruyere	1	RXSS	0	1	38	345	0,290	11,014	37	4,1
2	Gruyere	2	SXSS	0	54	6	345	15,652	1,739	61	5,4
3	Gruyere	1	RXSS	0	33	593	4876	0,677	12,162	507	66,2
3	Gruyere	2	SXSS	0	661	88	4876	13,556	1,805	7053	475,9
2	WebGoat	2	RXSS	0	181	305	345	52,464	88,406	99	1,4
3	WebGoat	2	RXSS	0	1158	4409	4876	23,749	90,422	1052	11,2
4	WebGoat	2	RXSS	0	16109	48691	53707	29,994	90,660	5169	105,8
2	DVWA	1	RXSS	0	345	38	345	100,000	11,014	41	1,4
2	DVWA	2	SXSS	0	150	150	345	43,478	43,478	246	1,2
3	DVWA	1	RXSS	0	4876	593	4876	100,000	12,162	512	10,3
3	DVWA	2	SXSS	0	2147	2149	4876	44,032	44,073	1041	36,0
4	DVWA	1	RXSS	0	53707	6330	53707	100,000	11,786	7711	92,6
4	DVWA	2	SXSS	0	22732	23402	53707	42,326	43,573	12788	1958,7
2	DVWA	1	RXSS	1	230	28	345	66,667	8,116	45	1,4
2	DVWA	2	SXSS	1	0	0	345	0,000	0,000	35	1,1
3	DVWA	1	RXSS	1	3255	415	4876	66,756	8,511	503	9,4
3	DVWA	2	SXSS	1	0	0	4876	0,000	0,000	3665	34,2
4	DVWA	1	RXSS	1	29445	4220	53707	54,825	7,857	6334	386,2
4	DVWA	2	SXSS	1	0	2	53707	0,000	0,004	8224	2271,0

Table 1. The tested HTML input fields were of type text and text area. The performance of the tool in general was acceptable as there were not errors or application crashes. Of the 5 applications, DVWA and Mutillidae support setting different security levels that affect the filters that are applied to the input fields. Higher security settings result in stricter filters that are harder to bypass. Therefore, the parameters were treated as different test runs depending on these security settings.

The Burp Suite performed well in finding XSS vulnerabilities using the vectors generated by our combinatorial testing approach. Out of all 58 test runs only 10 were not flagged as XSS. When taking into account only strength 2, Burp Suite did not detect 4 parameters yielding a 80,0% success rate. With strength 3 the result was the same (80,0%) and with strength 4 the result was better with 88,9% because only 2 parameters were not detected. The comparison between the three different results can be seen in Figure 1. The XSS detection rate was measured as a fraction of the successful exploits occurred per total input field parameters of the tested web

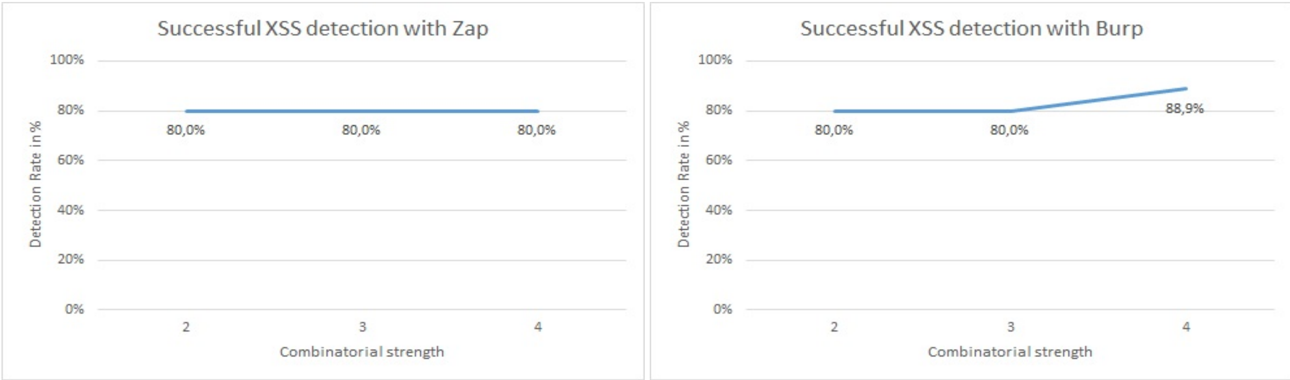

Figure 1: Comparison of XSS detection rate with different combinatorial strength

applications.

Only in 6 of the 58 test runs all vectors were reflected directly to the web pages resulting in a XSS detection. These 6 cases were carried out on the same parameter in the Mutillidae application on low and medium security level. This means that with all other test runs some filter mechanisms were in place that stripped some of the inputs. From a practical point of view this seems to be caused by filters on the one hand or due to some parameter processing issues like output context. As we employed a general grammar to generate our XSS attack vectors we did not take into account different output contexts like attributes, JavaScript or CSS.

The DVWA and Mutillidae applications support different security settings that change the behavior of the application to reflect better filters and validators. Setting the security to the second level only 2 parameters of the DVWA applications where not flagged as XSS and 4 were flagged. This means we yielded a 66,6% success rate compared to 100% on low security level. This is a big drop of the success rate and further research is needed to evaluate better grammars to bypass filters. Within the Mutillidae application the rising of the security level to the second level did not affect the test results after all. All 9 test runs with this security setting resulted in a successful XSS detection meaning that our grammar was able to bypass the employed filter mechanisms. Mutillidae supports setting the security level to a third very strict mode. Using this security mode only Mutillidae parameter #3 was detected as vulnerable to XSS and parameter #1 and #2 where no more detected due to the strict filters that are applied in the "high security" mode. This was an expected behavior. The most interesting test runs are those where only a few vectors resulted in a XSS detection. Our experiments contain 5 test runs where less than 10 vectors bypassed a filter. These vectors aid towards understanding the different filter mechanisms of the applications under test. They show which characters are filtered and thus they can used as a pattern basis to produce better grammars.

5.2 Test Results Evaluation with ZAP Tool

The evaluation of the five web applications using ZAP is also presented in Table 1. The overall performance of the tool was acceptable regarding the detection rate, but problematic regarding the execution time and the system errors produced while testing. Section 5.3 includes detailed information regarding the problems with execution time and the errors caused by ZAP.

While using our combinatorial testing approach and the attacking vectors produced by this method, we managed to achieve very good results with ZAP, similar to those achieved with Burp. In total we executed 58 test runs which included XSS vulnerabilities, and ZAP failed to find a XSS vulnerability in only 12 of them. The detection rate for strength 2 was 80% since in only 4 out of the 20 test runs our vectors were unsuccessful in detecting a XSS vulnerability. When testing with strength 3 and strength 4 the detection rate was not improved, but remained exactly the same (80%).

Out of the 58 test runs that we had when testing with ZAP, in 21 of them all of our vectors penetrated the web applications. This includes all the different security levels and the 3 different levels of strength in our vectors. In 9 runs out of the 21, a security mechanism was present but did not manage to block our vectors. On the other hand, we had 23 runs were a portion of our vectors successfully went through the security filter mechanisms. This is translated to 40% successful test runs were our vectors managed to overpass the security filters applied by the web applications. When analyzing a specific web application, we can see that in DVWA the increase of the security level from low to medium not only achieved to block more of our vectors, but also managed to totally secure the application against SXSS attacks, independently of the strength of our vectors. Likewise, Mutillidae offers three different levels of security. The filters in the first two levels were unsuccessful to protect the web application since we did not have a test run where at least one of our vectors did not went through. However, in the highest security level, the strict security filters that take place, allowed us to penetrate only parameter #3.

5.3 Practical Experiences from Testing Tools

5.3.1 *Comparison based on Results*

When analyzing the obtained results we can reckon that in the most of our test runs, we get different numbers of successful vectors from the two tools used. In an overall analysis, ZAP detected in average 25% more successful vectors than Burp. This does not automatically mean that ZAP performed better than Burp. In our analysis we do not have a formal explanation of this behavior but we have some indications leading us to the following assumptions. First we believe that there are small differences in the test oracles, despite that the same principle is followed in both tools. Our second assumption is that ZAP is using different encoding when creating the requests compared to the encoding of Burp. This leads to handling the request by the web application in a different manner and thus obtaining different results.

Regarding the performance of our vectors, we observed that does not depend on the strength of our vectors, as the percentage of the successful ones remains the same. In only two cases while using Burp we had increased performance when testing with strength 4. For example, in DVWA where Burp detected two successful vectors, one of them was

```
';onLoad( )
```

This vector was not actually successful because the response included the following string:

```
\';onLoad( )
```

when Burp considers these vectors successful. In this case, the filter used by the application was to encode " ' " with a " \ ". Burp oracle is detecting a successful vector based on the fact that it was reflected, but without taking into account if there is an escaping character before the string. But again, in total there is not a significant increase in the percentage of positive vectors when the combinatorial strength is increased.

5.3.2 Comparison on the Performance Side

ZAP performed relatively well when testing with combinatorial strength 2, in terms of execution time since it required less than a minute in almost all test runs, to complete the experiment. However, when testing with combinatorial strength 3 or 4, where the attack vectors are 4876 and 53707 respectively, the execution time rises significantly. Depending on the web application, the execution time for strength 3 ranged from six minutes to one hour for one run on DVWA. Nevertheless, the time required for the test runs with combinatorial strength 3 was usually not more than half hour. This performance can already be considered problematic, especially when compared with Burp, but the major problem on our test runs arose when executing the attack vectors produced with combinatorial strength 4. The execution time varied from one hour up to two hours and a half. This was a big obstacle in completing our test runs. When ZAP was running a test run for such a long time it was lagging and becoming unresponsive in most of the test runs. Additionally, the five web applications forming the basis of our case study were also becoming unresponsive when tested for a long period, which led us in restarting them several times on our local server.

In order to identify the reason for this behavior, we monitored the memory allocation as well as the computational power required by ZAP while doing the test runs with strength 4. Again, we do not have a formal explanation for this behavior but we elaborate on some interesting remarks. Our server has an eight core processor and 16 Gigabytes of RAM and it is running OpenSUSE 13.1. While ZAP was running we noticed that usually one processor is near to its full capacity, when the remaining 7 are either in idle mode or a maximum 20% of their power was used. This is an indication that ZAP has a poor performance in multi-threading. The memory allocation was usually between 300 MB and 400 MB which is far less than our system's max capacity. These observations could not justify the behavior of ZAP in our system since it seems that our server had handle pretty well ZAP's needs. However, ZAP led our server to crash several times especially when running tests for combinatorial strength 4. We also tested ZAP on another Linux machine running Kali distribution, but with limited hardware capabilities. The behavior of ZAP was similar to what we already described on OpenSUSE. We observed the following interesting remark when performed the same test runs with strength 4, on a Windows environment on the same machine with the limited capabilities. Termwise, the performance of ZAP was the same, which encourages our previous assumption that ZAP failed to take advantage of our server's higher capabilities. The surprising point is that ZAP did not crash once when running on Windows. As a results we assume that when ZAP is running on Linux, it is producing an exception which is not handled properly by the operating system and leads our system to crash.

6. CONCLUSION

In this paper, we presented a case study that strengthens the applicability of combinatorial testing to web application security testing, and in particular when testing for XSS vulnerabilities. The applications that were tested are standard web application that use mainly traditional concepts for data transport. From a practical point of view it would be interesting to test web applications that use new approaches like client side JavaScript to process variables, mobile concepts or server side JavaScript frameworks. However, regardless of the transport mechanism the filter mechanisms (output encoding and input validation) must still be in place to prevent XSS.

7. ACKNOWLEDGEMENTS

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice) and ITEA-2 (DIAMONDS) and the Austrian COMET Program (FFG). In addition, the work of the third author was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The research leading to these results has received funding from the *European Union* Seventh Framework Programme (FP7/2007-2013) under *grant agreement* no. 246016.

8. REFERENCES

[1] J. Bozic, D. E. Simos, and F. Wotawa. Attack pattern-based combinatorial testing. In *Proceedings of the 9th International Workshop on Automation of Software Test (AST)*, pages 1–7, 2014.

[2] J. Bozic and F. Wotawa. Xss pattern for attack modeling in testing. In *Proceedings of the 8th International Workshop on Automation of Software Test (AST)*, 2013.

[3] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, 1997.

[4] C. J. Colbourn. Covering arrays. In C. J. Colbourn and J. H. Dinitz, editors, *Handbook of Combinatorial Designs*, Discrete Mathematics and Its Applications, pages 361–365. CRC Press, Boca Raton, Fla., 2nd edition, 2006.

[5] F. Duchene, R. Groz, S. Rawat, and J.-L. Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 815–817, Washington, DC, USA, 2012. IEEE Computer Society.

[6] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection. In *CODASPY*. ACM, 2014.

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

[8] NIST. *User Guide for ACTS*.

[9] O. Tripp, O. Weisman, and L. Guy. Finding your way in the testing jungle: A learning approach to web security testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 347–357, New York, NY, USA, 2013. ACM.

[10] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.