Interrogating Virtual Agents: In Quest of Security Vulnerabilities

Josip Bozic[0000−0001−6086−8846] and Franz Wotawa✉[0000−0002−0462−2283]

Graz University of Technology
Institute of Software Technology
A-8010 Graz, Austria
{jbozic,wotawa}@ist.tugraz.at

Abstract. Chatbots, i.e., systems that communicate in natural language, have been of increasing importance over the last few years. These virtual agents provide specific services or products to clients on a 24/7 basis. Chatbots provide a simple and intuitive interface, i.e., natural language processing, which makes them increasingly attractive for various applications. In fact, chatbots are used as substitutes for repetitive tasks or user inquiries that can be automated. However, these advantages always are accompanied with concerns, e.g., whether security and privacy can be assured. These concerns become more and more important, because in contrast to simple requests, more sophisticated chatbots are able to utilize personalized services to users. In such cases, sensitive user data are processed and exchanged. Hence, such systems become natural targets for cyber-attacks with unforeseen consequences. For this reason, assuring information security of chatbots is an important challenge in practice. In this paper, we contribute to this challenge and introduce an automated security testing approach for chatbots. The presented framework is able to generate and run tests in order to detect intrinsic software weaknesses leading to the XSS vulnerability. We assume a vulnerability to be triggered when obtaining critical information from or crashing the virtual agent, regardless of its purpose. We discuss the underlying basic foundations and demonstrate the testing approach using several real-world chatbots.

Keywords: Security testing · model-based testing · chatbots · web applications.

1 Introduction

In 1966, Joseph Weizenbaum invented the very first program that communicates with users in natural language [39]. Such systems, called chatbots [29], usually provide information about services and goods from a specific domain. However, since such systems offer many opportunities [25], they are becoming increasingly popular on the global market [4,13]. Chatbots are usually deployed in form of virtual assistants, either as stand-alone applications or are integrated into websites in form of chat widgets. In such way, they are easily accessible, and

also easy to interact with. Since they provide consistent answers in real-time, they save time and effort for clients to obtain requested information. In fact, due to such advantages, they might become even more popular than classical web applications [17].

Chatbots are developed further to respond to more specific customers' demands. Virtual assistants are considered in more sensitive domains like medicine [21,26], fintech [16], and banking [22]. Besides the usual natural language processing (NLP) layer, such chatbots apply more complex techniques from AI. They rely on a knowledge base and collect private data from user interactions and also learn from them. For such systems, ensuring information security becomes of uttermost importance. Requirements like confidentiality of user data, however, are challenged by the fact that chatbots rely on the common web infrastructure. In fact, since chatbots often come in form of web applications, they inherit their vulnerabilities as well [18]. Subsequently, cyber-attacks that target web vulnerabilities like cross-site scripting (XSS) [34] can be also executed against chatbots. Even more, this vulnerability motivates further malicious attempts, like denial-of-service (DDoS) attacks [6] or content spoofing [5].

Until now, proposed approaches usually test functional correctness of NLP systems. Such works apply different dialogue strategies by generating either valid [38,20] or invalid [35] language inputs. Subsequently, correctness functions are applied in order to evaluate the chatbot's behavior with regard to correct language output. Unfortunately, there is little work that puts focus on security issues.

Lots of research has been conducted for testing of web applications. Such works focus either on strengthening the detection mechanisms against attacks [28,30] or take the role of the attacker [27,19,31]. The latter case is covered, among others, by approaches from the area of model-based testing [24]. In this technique, test cases are automatically generated from a model of the system under test (SUT) or the attack itself. In addition to that, security testing can be combined with other techniques, like combinatorial testing (CT) [36], fuzzing [23] or model-checking [33].

This paper builds upon our previous work in [18], which contains an initial discussion about a security testing problem for chatbots. The motivation behind this work is to address the previously mentioned issues. Therefore, we introduce a framework that tests chatbot implementations for reflected XSS vulnerabilities in an automated manner. Subsequently, the approach is evaluated against several real-world applications, thereby discussing the obtained empirical results. We also want to note that, to our knowledge, this is the first paper where chatbots are successfully tested for security vulnerabilities.

The paper is structured as follows. Section 2 introduces the testing approach for chatbots. Section 2.1 and Section 2.2 discuss the underlying test generation and execution techniques, respectively. Then, Section 3 discusses the results from several real-world applications. Section 4 discusses related work and Section 5 concludes the work.

2 Approach Overview

NLP systems are usually implemented to fulfill a specific purpose. This means that they expect user inputs to fit pre-defined communication patterns. However, the question arises how the system behaves when confronted with unexpected, even malicious inputs. Since user communication is difficult to predict, the chatbot must withstand a broad scope of possible inputs. In order to function correctly, the chatbot must be resistant at least against common cyber-attacks. Therefore, a testing framework must be able to successfully test chatbots, regardless of their purpose.

Cyber-attacks against web applications represent an issue since the dawn of these systems. In fact, persistent vulnerabilities in web applications [10] motivate malicious users to abuse their weaknesses. XSS, for example, is triggered by injecting malicious JavaScript code into HTML elements. This attack usually targets user input fields of a website, where a user interacts with a website in a textual manner. Subsequently, in case that the attack was successful, a malicious code is executed at the side of the user.

In this paper, we introduce a testing approach for the detection of XSS in chatbots. This testing problem can be divided into three separate tasks, which represent integral parts of a testing framework implementation:

1. Test case generation
2. Test oracle definition
3. Test case execution

In the following sections, we will elaborate every task in detail and explain their role in the overall testing framework.

2.1 Test Case Generation

In order to trigger a vulnerability, user inputs must be defined in a way so that they contain executable JavaScript code. Each of such inputs, called attack vectors, represents a concrete test case. Unfortunately, the problem with XSS represents the fact that no standardized structure exists for such inputs. Actually, this can be considered the main reason for the difficulty to effectively defend against it. However, some mandatory information is always needed in order to execute the XSS code. In this paper, the test generation resembles the technique from our previous paper [19]. We define a small formal grammar that contains information about XSS, which is used to construct executable attack vectors. For this case, we relied on the official HTML specification [8], our experience and external sources (e.g. [14,15]). The attack grammar is built from finite sets of terminal and nonterminal symbols in the standard BNF. Every row in the grammar consists of a rule, which includes a left-hand side (LHS) and right-hand side (RHS). Each LHS consists of a single symbol, whereas the RHS contains an indefinite number of symbols. As common in BNF grammars, each rule defines when the LHS can be rewritten to its RHS. The resulting attack grammar in BNF is defined in the following way.

```
<pre>::= >
<opening>::= < <html> <content> >
<html>::= input | IFRAME | SCRIPT | A | img
<content>::= _ <attribute> = <value>
<attribute>::= type | value | <div/onmouseover | SRC |
   a | HREF | _ | title
<value>::= j_a_v_a_s_c_r_i_p_t_:a_l_e_r_t_%28_1_%29 |
     "text" | ‘‘ | ’alert(1)’ | "javascript:alert(’XSS
   ’);" | ">" | "http://www.google.com" |
   j_a_v_a_s_c_r_i_p_t_:a_l_e_r_t_%28_1_%29 | "/" |
   _=" | "onerror=’prompt(1)’"
<middle>::= alert(1) | XSS
<closing>::= </A> | XSS | </IFRAME> | </SCRIPT> | _
<post>::= ’’
```

Attack grammar for test generation

As can be seen, JavaScript code is put into a formal representation of terminal and nonterminal symbols. The symbols themselves are defined so that they act as building blocks for attack vectors. Whereas concrete code is defined as terminals, the following elements are defined as nonterminals.

- `pre`: Sometimes symbols can be put in front of the actual script. This can lead to a filter bypass where the following script is executed in the aftermath.
- `opening`: This placeholder contains a HTML opening tag, which contains a set of HTML tags, attributes and values.
- `html`: This HTML tag contains statements or point to an external code. These can embed client-side scripts, images, inline frames, hyperlinks and input fields in websites.
- `content`: This element contains an attribute-value pair in HTML and an equal symbol in between.
- `attribute`: HTML attributes include, among others, the type of input elements, initial values, location, title about an element.
- `value`: The placeholder for the actual payload that is meant to be executed. It can have multiple forms, depending on the intention of the attack.
- `middle`: The content of this element is eventually placed between the opening and closing tags. It contains either a window object or simple text.
- `closing`: Closing HTML tags are usually placed at the end of the input, thereby making it a valid JavaScript code. However, this element can be omitted altogether, thereby confusing the target system.
- `post`: In rear of the code one or multiple symbols can be inserted. SUTs might behave differently when encountering these symbols.

As already mentioned, XSS lacks a standard specification, which means that its attack vectors can come in different flavors. For this reason, we want to generate a test set that covers a wide scope of possible XSS appearances. In fact,

the attack grammar provides enough information for the generation of such test cases. For example, let's consider five different cases of possible XSS structures.

```
<ex1>::=<<html><content><content><content>><closing>
<ex2>::=<<html><content>><closing>␣<<html><content>
    <content>><closing>␣<<html><content>␣<post>
<ex3>::=<<html><content>><middle><closing>
<ex4>::=<pre><middle>␣<opening><closing>
<ex5>::=<<html><content><content><content>>
```

Every case can be defined as an expression `<ex>` that consists of a set of attack grammar symbols. In fact, these expressions act as a guideline for the generation of attack vectors. We define them according to our experience and available information from the aforementioned sources. Every sequence of elements in an expression defines the order of nonterminals, which will be converted into corresponding terminals. The subsequently generated concatenation of terminals in one sequence represents an attack vector. Test cases are generated with a modified version of Grammar-Solver [7]. Grammar-Solver reads an expression and searches for corresponding nonterminals in the attack grammar. Every occurrence of a LHS symbol is rewritten by symbols of the RHS of the grammar. In fact, in every run, only one terminal is picked from the RHS for an element of an expression. Then the solver switches to the next element and repeats the process. Basically, the implementation generates attack vectors in a pairwise manner. It produces a cross product of unique combinations of terminals for every attack vector. However, even with a small grammar such as ours, the number of combinations would be too exhaustive for some of the `<ex>`'s. In order to avoid a combinatorial explosion, we restrict the number of terminal symbols on the RHS for certain expressions (like `<ex2>`). Even with a subset of the grammar, we generate a sufficient number of attack vectors. Figure 1 depicts the overall test case generation approach.

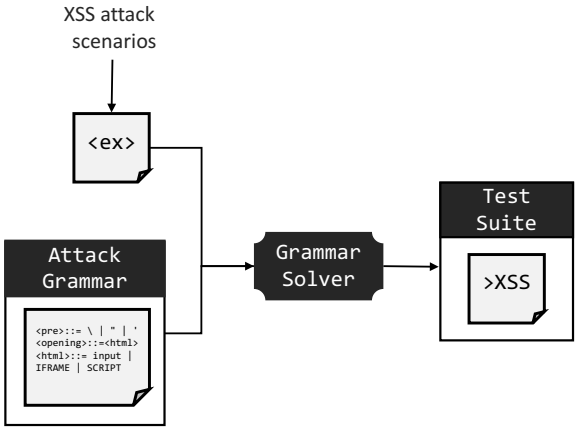

Fig. 1. Attack grammar-driven test case generation

Let's demonstrate the approach on an example. We want to generate attack vectors from the fourth expression, `<ex4>`. This expression consists of four nonterminals at the RHS and one space terminal in between. Now Grammar-Solver recursively starts at the first element, `<pre>`, and searches further in the parse tree of the nonterminal. The next element, `<middle>`, contains two terminals. After the space symbol, `<opening>` is traversed further, thereby encountering additional nonterminals `<html>` and `<content>`. The final element represents a HTML closing tag, `<closing>`. In the first run, the first terminals from the bottom of the parse tree of every nonterminal are selected, respectively. After the selection process is done, the concatenated terminals comprise the attack vector:

```
>alert(1) <input type=j a v a s c r i p t :a l e r t %28 1 %29></A>
```

The generated output represents one instance of possible values from the attack grammar. Subsequently, a different combination of values results in different attack vectors. The test generation process terminates once all combinations of terminals from the grammar are exhausted. The final output of this technique represents a unique test set for every `<ex>`.

2.2 Test Oracle & Execution

After the test set is generated from the grammar, the test execution process can be initiated. The generated attack vectors are submitted against a virtual agent in an automated manner. In this approach, a vulnerability is triggered by obtaining critical information from or breaking down the virtual agent. The shape of the expected information from a chatbot is defined in the test oracle in [19]. The typical attack vector for reflected XSS contains code that is meant to be executed on side of the client. In case of a secured application, intern security mechanisms will prevent this from happening. However, since XSS requires that its script is processed unaffected, potential input filters must be bypassed. For this reason, the attack grammar generates attack vectors with diverse input elements. By doing so, we hope to increase the likelihood that some attack vector avoids filtering mechanisms.

At the beginning of the execution process of a test case, the initial state of the SUT is processed and memorized. Then, individual attack vectors are submitted and the corresponding response is recorded. The behavior of the SUT is compared to its initial state by checking its output against the test oracle. Usually the reflected code from the SUT resembles the code from the attack vector. Actually, if the response code matches the input code, we conclude that a XSS vulnerability is triggered. On the other hand, if the submitted attack vector is filtered, the HTTP response contains an encoded input. In this case, no code will be executed, thus the attack was ineffective. Figure 2 depicts the communication flow with a filtered attack vector.

The virtual agent itself is either set up locally or accessed online over common HTTP. For every attack vector, a HTTP request is generated and sent to a

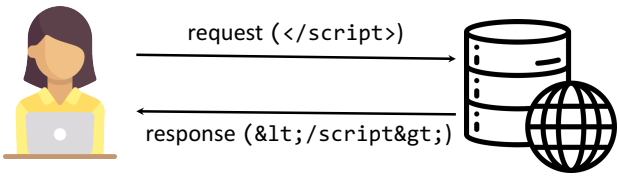

Fig. 2. Attack scenario for XSS over HTTP[1]

SUT. As already mentioned in Section 2, the main targets for XSS represent user input fields. In case that the chatbot comprises static HTML content, we rely on HttpClient [1] for interaction purposes. On the other hand, Selenium WebDriver [12] is used as the API for testing of dynamic web elements. For this reason, we set up the standalone Selenium server v.3.141.59. The attacked chatbot replies with a HTTP response, which eventually contains an executable code. This response is parsed and its extracted content is automatically checked against the test oracle. Finally, a test verdict is given. Figure 3 depicts the entire test execution process.

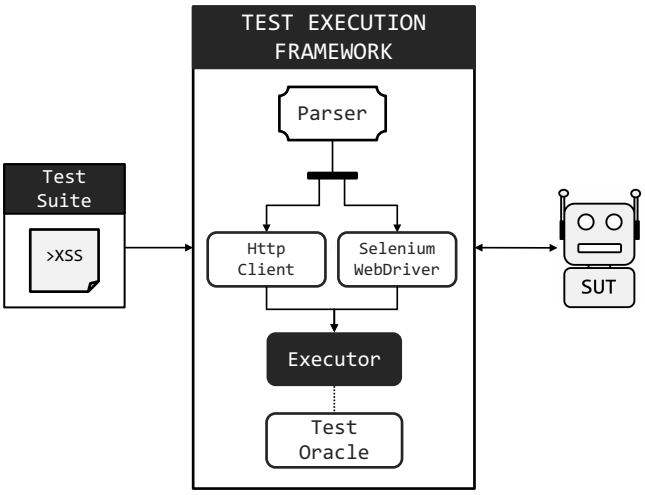

Fig. 3. Test case execution for chatbots

If a vulnerability is triggered, we conclude that the test case was successful. Otherwise, the attack vector was ineffective, thus returning a failing verdict. In fact, a test case consists of one such execution between the testing framework and chatbot. Afterwards, the execution switches to the next attack vector. The entire process is repeated until the very last test case has not been executed.

[1] Icons made by Freepik and Smashicons from www.flaticon.com, respectively.

3 Evaluation

Chatbots are programs that simulate human-like interaction based on a set of NLP rules. In this paper, we do not test its ability to engage in complex communication or its understanding capabilities of natural language. Also, we don't test its memory functions with regard to correctness of stored information. In our quest, we confront the virtual agent aggressively with malicious inputs in order to extract useful information. In this case, useful information represents reflected code that is retrieved from the SUT. As demanded by the test oracle in Section 2.2, this information indicates that a security vulnerability is encountered in the application. For every expression, a different number of test cases is generated. In total, we generated an amount of 21355 attack vectors. For `<ex1>` we obtained 1458 inputs, 10368 for `<ex2>`, 4400 for `<ex3>`, again 4400 for `<ex4>` and 729 for `<ex5>`. The testing framework generates test cases on-the-fly by assigning attack vectors to HTTP requests. We tested a set of four different SUTs for reflected XSS vulnerabilities in an automated manner. The tested chatbots include the following ones:

- Aztekium Bot [2]: This educational chatbot provides information from different topics, including technical issues and lexical items. Therefore, it differs from other chatbots since its purpose is not to pretend to be human. It supports multiple languages and does not use a database.
- Jeeney [9]: This virtual agent adapts a more private approach to every user. By doing so, it learns from their interactions. It relies on the N.R. Research Engine[2], which allows the chatbot to endeavor in more complex communication. Also, it conducts a more complex analysis of provided user information.
- *SUT3*: Primarily, this chatbot is meant for entertaining purposes. However, this interactive agent can be used to practice writing skills in English as well. It should be noted that this chatbot is no longer being updated since 2002.
- *SUT4*: This system represents an open source chatbot platform. Chatbots can be implemented and customized in this platform for multiple purposes. Each system encompasses an object database, which can be reused and manipulated further. The supplement AI engine enables the chatbot to remember information from interaction with a user. In contrast to the other tested SUTs that were tested online, this chatbot was set up locally at the Apache Tomcat server, v.9.0.10.

Table 1 depicts all testing results. Every chatbot (`SUT`) is tested with test suites from individual expressions (`TS`). The total number of successful test cases (`#success`) counts attack vectors that were successful in triggering XSS. The number of failed test cases (`#fail`) depicts the number when no vulnerability was triggered. In general, chatbots behave differently when confronted with attack vectors from individual test suites. However, in certain cases they react in a similar manner. We will elaborate our observations and interpret the outcome of every SUT separately.

[2] Neural Reliquary. `http://www.jeeney.com/nr.html`, accessed: 28.08.2020

Table 1. Test results for XSS in chatbots

SUT	TS	#success	#fail
Aztekium Bot	<ex1>	1,458	0
	<ex2>	10,368	0
	<ex3>	4,312	88
	<ex4>	4,342	58
	<ex5>	729	0
Jeeney	<ex1>	1458	0
	<ex2>	10,368	0
	<ex3>	4,350	50
	<ex4>	4,350	50
	<ex5>	672	57
SUT3	<ex1>	0	1,458
	<ex2>	0	10,368
	<ex3>	2,383	2,017
	<ex4>	0	4,400
	<ex5>	674	55
SUT4	<ex1>	1,458	0
	<ex2>	10,368	0
	<ex3>	4,350	50
	<ex4>	4,350	50
	<ex5>	672	57

Aztekium Bot: This chatbot was very receptive to XSS detection attempts. In fact, all attack vectors from `<ex1>` and `<ex2>` were successful. However, a sanitation mechanism was encountered for `<ex3>` when a textual non-HTML input was at the end of the attack vector. The same can be said with `<ex4>` but only in cases when no closing tag exists. Without the last HTML element, the attack vector succeeds. Unfortunately, we encountered a possibly false positive issue for `img` elements: The input should be rejected when a textual data was present between HTML elements. However, this was not the case. The rest of the image elements were successful for `<ex5>`. Basically, this makes this chatbot receptive to all HTML tags.

Jeeney: This virtual agent did behave in the most curious way when being security tested. It did not filter attack vectors even in case that a closing HTML element was present, as in `<ex1>`. However, we encountered a discrepancy when we manually tested successfully flagged `script` elements with a browser. To our surprise, none of these HTML tags was triggered in the browser. But this filter is mitigated in `<ex2>` by injecting a `script` inside an `iframe` element. The hidden tag is not detected and is subsequently executed. We encountered a similar scenario in `<ex3>` with `img` elements. A aforegoing text in attack vectors seems to prevent the `img` from triggering. Although flagged as successful by the testing framework, no image tag was executed in the browser. For `<ex3>`, `<ex4>` and `<ex5>` all malformed attack vectors are rejected by the chatbot. However, plane `img` elements were triggered in the chatbot for `<ex5>`. In general, we were able to trigger `iframe` and `input` elements in all cases where they occurred.

SUT3: In contrast to other SUTs, this bot demonstrated the most distinctive behavior. Relative simple attack vectors resulted in XSS, whereas more complex

inputs didn't. It resisted all attempts from `<ex1>` and `<ex2>`. The reason therefore is that the chatbot filters the last `<closing>` element in case that it comprises a non-HTML text. For `<ex3>` we were able to trigger `input` and `iframe` but just in cases where the input was properly closed by a HTML element. However, `script` was filtered even in properly closed inputs. Attack vectors from `<ex4>` were utterly rejected due to aforegoing `<pre>` element. On the other hand, the `img` element generally triggered a defect for `<ex5>` but was dismissed in cases of bad structure (like missing attributes). In general, this chatbot is the most equipped with serious input validation mechanisms.

SUT4: The bots from this multi-purpose chat platform are very receptive to XSS attempts. In fact, all `input`, `iframe` and `script` elements from `<ex1>` and `<ex2>` are subject to XSS. There were some cases in `<ex3>`, where attack vectors were rejected due to malformed inputs. This was only the case when an attribute was missing inside HTML tags. Even more interesting is the fact that aforegoing `pre` element from `<ex4>` succeeds to cover the XSS attempt. However, we did detect some false positives with the test oracle with `<ex5>`: A malformed `img` element is flagged as successful although it might not be triggered by the system.

The proposed testing framework succeeded to trigger security leaks in every chatbot. In general, a security leak indicates an implementation flaw or oversight. Unfortunately, the tested SUTs lack the ability to defend themselves against relatively simple XSS attempts. We assume that such behavior is caused either by a lack of sufficient security awareness or expertise. By triggering XSS from test cases from different expressions, we also get some insight about causes of the issue. For example, the omission or insertion of certain symbols in attack vectors can change the reaction of a SUT. The post-analysis of the results reveals that XSS can be triggered when specific elements occur in an attack vector. In fact, this observation affirms the claim of combinatorial test generation, where vulnerabilities are triggered by some critical combination of its components (e.g. [36]). The observed chatbot behavior also indicates that XSS inputs, in order to be triggered, must have a structure. On the one hand, some SUTs reject inputs due to an unexpected element in the attack vector. On the other hand, these chatbots process the XSS code in case of its absence nevertheless. Since the chatbot does not expect cyber-attacks, it fails to recognize the malicious XSS content of the received input. This means that the attack vector is treated like harmless JavaScript code. Although reflected XSS does not necessarily represent a harmful vulnerability, the inability to cover it can still lead to more devastating attacks, like unauthorized server access, etc.

However, despite positive results we must consider the occurrence of some false positives. Test oracles represent a distinct problem in software testing and XSS is no exception. Also, the discrepancy between results from methods that test static and dynamic web content must be considered, respectively. HttpClient offers the advantage to bypass a web browser by relying on its *headless* approach. On the other hand, Selenium WebDriver emulates a browser and therefore relies

on its infrastructure and content filters. Therefore, this matter affects the results to a certain degree as well.

Regardless of this open issue, we want to emphasize the positive sides of the approach. The testing framework successfully triggers security vulnerabilities in chatbots and provides some clues about its root causes. Because of these facts, we consider the presented approach as a good starting point for future endeavors.

4 Related Work

To our knowledge, almost no works exist that focus on security issues in chatbots. However, vulnerabilities do represent a real issue in these systems, which need to be addressed. Until then, the current state-of-the-art leaves them vulnerable to exploitation attempts in the future. The current research focus in chatbots lies either in testing of functional aspects and usability [38,35,20] or non-functional properties [32]. In the former case, understanding of language and context in NLP systems is tested. On the other hand, the latter work measures a chatbot's NLP capabilities by applying load testing. In both cases, input parameters are evaluated by relying on correctness and performance functions. However, these functions stand in stark contrast to test oracles from the domain of security testing.

On the other hand, several approaches exist that test for XSS in web applications. These include, among others, the following works.

The authors of [27] introduce a mutation-based XSS testing approach that exploits intrinsic security leaks in web browsers. In this approach, malicious attack vectors are stored in their "harmless form" in HTML markups. In fact, the attack vector is mutated by the browser during the generic rendering of a website. This happens because the browser accesses the markup and decodes the content in order to parse it into a DOM structure. By doing so, the attack vector is unintentionally mutated into an executable form. The mXSS attack vector is assigned to a innerHTML property, thereby executing the malicious code. The attack is demonstrated on several scenarios against web applications and mitigation mechanisms. This attack is so destructive because the attack evolves during a pre-processing stage. In such way, the attack vector escapes potential detection mechanisms. In our approach, we don't primarily target the browser, since HttpClient bypasses the browser altogether. Also, we don't apply mutations on a specific attack vector but generate them in a combinatorial manner.

A security testing methodology for online business services is presented in [37]. The work focuses on authentication protocols for Multi-Party Web Applications (MPWAs) and subsequent web vulnerabilities. The authors analyze attack strategies for known vulnerabilities, including XSS, and subsequently abbreviate reproducible, i.e. application-independent, representations. These attack patterns represent general attack scenarios for testing against specific attacks. We share this work's intention in defining reproducible, black-box testing guidelines for XSS. However, our work generates concrete test cases with an intern test generation technique.

The approach in [36] introduces a combinatorial testing approach for analyzing XSS vulnerabilities. The authors define a combinatorial input model for test case generation, which are subsequently executed. Then, an automated fault-localization technique analyses the structure of every successful attack vector. Eventually, suspicious XSS-inducing combinations of parameter values are detected. Afterwards, such a combination is added to a new input model. From this model, new attack vectors are generated in a combinatorial manner. In fact, these constitute refined test cases with regard to the initial model. Similar to our approach, this work applies combinatorial test generation. However, the structure and values of attack vectors differs from this approach.

The authors of [31] introduce a unit testing approach for testing against XSS. Unit tests are generated automatically, i.e. each unit test represents a XSS test case. In order to generate attack vectors, they define an attack grammar. The grammar is subdivided into several input types, like URI resources, CSS specifications, HTML events and JavaScript code. Subsequently, attack vectors are generated by relying on sentences for each sub-grammar. Similar to our testing approach, this paper also defines a structure for XSS attack vectors. The resulting attack grammar is used in combination with sentences, that resemble our grammar and expressions, respectively. However, in contrast to their approach, we discuss a black-box testing technique, which does not have insights into the source code of a SUT.

A general overview about XSS is given in [34], whereas popular security tools for XSS testing include OWASP ZAP [11] and Burp Suite [3]. Whereas the former relies on fuzzing for automated testing, the latter represents a manual testing tool.

5 Conclusion and Future Work

In this paper, we address security issues in the increasingly important field of NLP systems. For this sake, we introduce a security testing approach for the detection of a harmful vulnerability in chatbots, namely XSS. A grammar-based test case generation technique is presented that generates malicious inputs for this purpose. Subsequently, these attack vectors are automatically executed by a testing framework. The presented approach is evaluated on four real-world chatbots with promising empirical results. The approach confirms that XSS is encountered in every of the tested chatbots. Basically, this observation reaffirms our claim that vulnerabilities do present a real issue in chatbots.

In order to test chatbots, we relied on just a small attack grammar. Additional elements can be added to the grammar easily. In such way, different test suites will be generated. Also, custom expressions will contribute to the test case diversity as well. However, in order to define such grammar, some manual effort and expert knowledge is needed. Also, it should be noted that our approach is meant for testing purposes only. A real security breach exploitation, for example a data theft, has yet to be proven.

With this paper, we hope to raise awareness about the importance of security testing for chatbots. In fact, we claim that these issues will represent an important topic in the future. For this reason, security testing should be incorporated into the chatbot development cycle and used for regression testing as well. In the future, we plan to extend the testing framework in order to detect additional vulnerability types in chatbots [10].

Acknowledgements

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 865248 (Securing Web Technologies with Combinatorial Interaction Testing - SecWIT). We want to express our gratitude to the owners of the tested chatbots for giving us the opportunity and permission to use their systems for research purposes. Also, we want to thank the anonymous reviewers for their constructive feedback, which was addressed in the paper.

References

1. Apache HttpComponents - HttpClient. https://hc.apache.org/httpcomponents-client-ga/, accessed: 2018-09-06
2. Aztekium Bot. http://aztekium.pl/bot, accessed: 2020-08-27
3. Burp Suite. http://portswigger.net/burp/, accessed: 2020-08-27
4. Chatbot Report 2019: Global Trends and Analysis. https://chatbotsmagazine.com/chatbot-report-2019-global-trends-and-analysis-a487afec05b, accessed: 2020-08-05
5. Content Spoofing Software Attack. https://owasp.org/www-community/attacks/Content_Spoofing, accessed: 2020-08-08
6. DDoS attacks through XSS. https://www.incibe-cert.es/en/blog/ddos-attacks-through-xss, accessed: 2020-08-05
7. Grammar-solver. https://github.com/bd21/Grammar-Solver, accessed: 2018-07-13
8. HTML Tutorial. https://www.w3schools.com/html/, accessed: 2018-07-13
9. Jeeney AI. http://www.jeeney.com, accessed: 2020-08-27
10. OWASP Top Ten Web Application Security Risks. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, accessed: 2020-08-10
11. OWASP ZAP Zed Attack Proxy. https://owasp.org/www-project-zap/, accessed: 2020-08-27
12. Selenium. https://www.selenium.dev, accessed: 2020-08-10
13. Top 12 Chatbots Trends and Statistics to Follow in 2020. https://aalavai.com/post/top-12-chatbots-trends-and-statistics-to-follow-in-2020, accessed: 2020-08-05
14. XSS Filter Bypass List. https://gist.github.com/rvrsh3ll/09a8b933291f9f98e8ec, accessed: 2020-08-11
15. XSS Filter Evasion Cheat Sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, accessed: 2018-07-13

16. Altinok, D.: An Ontology-Based Dialogue Management System for Banking and Finance Dialogue Systems. In: Proceedings of the the First Financial Narrative Processing Workshop (FNP'18)@LREC'18 (2018)
17. Beriault-Poirier, A., Tep, S.P., Sénécal, S.: Putting Chatbots to the Test: Does the User Experience Score Higher with Chatbots than with Websites? In: International Conference on Human Systems Engineering and Design (IHSED'18) (2018)
18. Bozic, J., Wotawa, F.: Security Testing for Chatbots. In: Proceedings of the 30th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS'18) (October 2018)
19. Bozic, J., Wotawa, F.: Planning-based security testing of web applications with attack grammars. In: Software Quality Journal (2020) (2020)
20. Bravo-Santos, S., Guerra, E., de Lara, J.: Testing chatbots with Charm. In: Proceedings of the 13th International Conference on the Quality of Information and Communications Technology (QUATIC'20) (forthcoming) (2020)
21. Chung, K., Park, R.C.: Chatbot-based heathcare service with a knowledge base for cloud computing. In: Cluster Computing (2018)
22. Doherty, D., Curran, K.: Chatbots for online banking services. In: Web Intelligence, Vol. 17, Issue 4 (2019)
23. Duchene, F., Rawat, S., Richier, J.L., Groz, R.: KameleonFuzz : The day Darwin drove my XSS Fuzzer! In: Proceedings of the 1st European workshop on Web Application Security Research (WASR'13) (2013)
24. Felderer, M., Zech, P., Breu, R., Büchler, M., Pretschner, A.: Model-based security testing: a taxonomy and systematic classification. In: Software Testing, Verification & Reliability, Vol. 26, No. 2 (2016)
25. Følstad, A., Brandtzæg, P.B.: Chatbots and the New World of HCI. In: ACM Interactions, Vol. 24, No. 4 (2017)
26. Gabarron, E., Larbi, D., Denecke, K., Årsand, E.: What Do We Know About the Use of Chatbots for Public Health? In: Studies in Health Technology and Informatics (2020)
27. Heiderich, M., Schwenk, J., Frosch, T., Magazinius, J., Yang, E.Z.: mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS'13) (2013)
28. Lin, A.W., Barceló, P.: String Solving with Word Equations and Transducers: Towards a Logic for Analysing Mutation XSS. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16) (2016)
29. Mauldin, M.L.: ChatterBots, TinyMuds and the Turing Test: Entering the Loebner Prize Competition. In: AAAI '94 Proceedings of the twelfth national conference on Artificial intelligence (Vol. 1). pp. 16–21 (1994)
30. Mereani, F.A., Howe, J.M.: Detecting Cross-Site Scripting Attacks Using Machine Learning. In: Proceedings of the International Conference on Advanced Machine Learning Technologies and Applications (AMLTA'18) (2018)
31. Mohammadi, M., Chu, B., Lipford, H.R.: Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing. In: Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS'17). pp. 364–373 (2017)
32. Okanović, D., Beck, S., Merz, L., Zorn, C., Merino, L., van Hoorn, A., Beck, F.: Can a Chatbot Support Software Engineers with Load Testing? Approach and Experiences. In: Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE 2020) (2020)

33. Peroli, M., De Meo, F., Viganò, L., Guardini, D.: MobSTer: A model-based security testing framework for web applications. In: Software Testing, Verification & Reliability, Vol. 28, No. 8 (2018)
34. Rodríguez, G.E., G.Torres, J., Flores, P., Benavides, D.E.: Cross-site scripting (XSS) attacks and mitigation: A survey. In: Computer Networks, Vol. 166 (2020)
35. Ruane, E., Faure, T., Smith, R., Bean, D., Carson-Berndsen, J., Ventresque, A.: BoTest: a Framework to Test the Quality of Conversational Agents Using Divergent Input Examples. In: Proceedings of the 23rd International Conference on Intelligent User Interfaces Companion (IUI'18 Companion) (2018)
36. Simos, D., Kleine, K., Ghandehari, L., Garn, B., Lei, Y.: A Combinatorial Approach to Analyzing Cross-Site Scripting (XSS) Vulnerabilities in Web Application Security Testing. In: Proceedings of the 28th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS'16) (2016)
37. Sudhodanan, A., Armando, A., Carbone, R., Compagna, L.: Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications. In: Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS'16) (2016)
38. Vasconcelos, M., Candello, H., Pinhanez, C., dos Santos, T.: Bottester: Testing Conversational Systems with Simulated Users. In: IHC 2017: Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems (2017)
39. Weizenbaum, J.: ELIZA–A Computer Program For the Study of Natural Language Communication Between Man and Machine. In: Communications of the ACM Volume 9, Number 1 (January 1966) (1966)