ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/309740460

Effectiveness of Web Application Security Scanners at Detecting Vulnerabilities behind AJAX/JSON

Article · January 2015

CITATION
1

READS
166

4 authors, including:

Faustin Kagorora
Kigali Independent University (ULK)
3 PUBLICATIONS 12 CITATIONS
SEE PROFILE

Damien Hanyurwimfura
University of Rwanda
22 PUBLICATIONS 169 CITATIONS
SEE PROFILE

Lancine Camara
University of Bamako
4 PUBLICATIONS 20 CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

public policies View project

Analysis and Detection of polymorphic malware View project

All content following this page was uploaded by Lancine Camara on 10 June 2018.

The user has requested enhancement of the downloaded file.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

Effectiveness of Web Application Security Scanners at Detecting Vulnerabilities behind AJAX/JSON

Faustin Kagorora[1,3], Junyi Li[2,3], Damien Hanyurwimfura[1,3], Lancine Camara[1,3]

P.G. Student, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China[1]

Associate Professor, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China[2]

Key Laboratory for Embedded and Network Computing of Hunan Province, Hunan University, Changsha, China[3]

ABSTRACT: Web applications are used by almost all organizations in all sectors and are accessed by a large number of anonymous users, including malicious users. This wide visibility makes them susceptible to various attacks, such as SQL Injection (SQLI). Web application vulnerability scanners (WAVS) are automated black-box testing tools that examine web applications for security vulnerabilities. Evaluations of WAVSs have shown that executing client-side code is a major challenge to many scanners. However, despite the popularity of AJAX (Asynchronous JavaScript and XML) and JSON (JavaScript Object Notation) in modern web applications, no evaluation implemented test cases for the support for both AJAX and JSON technologies. This paper presents a test application and an assessment of the capability of 5 state-of-the-art black-box scanners to detect vulnerabilities hidden behind AJAX requests and JSON data. The test suite contains many vulnerability instances, with different levels of exploitation difficulty. Our experimental results show that executing AJAX code and analyzing JSON parameters are still challenges to many tools. We provide recommendations for assessing complete capability of WAVSs as evaluations did not cover all the main features.

KEYWORDS: Black-box testing, vulnerability detection, web application security, Rich Internet Application, web application vulnerability scanner.

I. INTRODUCTION

Web applications are exposed to the general public and accessed by a large number of anonymous users, including malicious users. This wide visibility makes them susceptible to various attacks, such as SQL Injection attacks. Furthermore, web applications are used by almost all organizations in all sectors including banking, health care, education, and manufacturing, among others. Thus, the security of web applications becomes an issue of critical importance.

Web application vulnerabilities account for the majority of the vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database [1]. According to Open Web Application Security Project (OWASP) Top 10 [2], input validation weaknesses cause most web application vulnerabilities. The complexity of modern web applications, along with the many different technologies used in various abstraction layers, are also the main causes of vulnerabilities in web applications.

Two main approaches to test software applications for the presence of bugs and vulnerabilities are white-box testing and black-box testing. In white-box testing, the source code of the application is analyzed to find defective or vulnerable lines of code. In black-box testing, the source code is not examined directly. Instead, special input test cases are generated and sent to the application. Then, the results returned by the application are analyzed for unexpected behavior that indicates errors or vulnerabilities.

Despite the use of many different testing techniques, we still find numerous exploitation reports in different publicly available databases, such as CVE [1] and Open Source Vulnerability Databases (OSVDB) [3]. As published in Symantec Internet Security Threat Report 2014 [4], eight of the breaches in 2013 exposed more than 10 million identities each.

Web application vulnerability scanners (WAVS) are automated black-box testing tools that examine web applications for security vulnerabilities. They are more popular due to the ease of use, automation, and independence from the specific web application’s technologies. However, WAVSs suffer from a number of limitations. They are not capable of detecting all the vulnerabilities and input vectors that exist. Several reports have shown that web application vulnerability scanners fail to detect a significant number of vulnerabilities in test applications [5]-[9]. Studies have also shown that executing client-side code and crawling multi-step processes are major challenges to many black-box scanners.

The emergence of richer and more advanced technologies, such as AJAX (Asynchronous JavaScript and XML), has made web applications more responsive, interactive and user friendly. These applications, often called Rich Internet Applications (RIAs), improved traditional web applications in two ways: First, they add more processing capability, i.e. dynamic manipulation of client-side state, on the client-side. Second, RIAs can initiate asynchronous communication with the server. However, at the same time, such techniques introduced new challenges. One important challenge is the difficulty of automatically crawling these new applications [10]-[13].

AJAX is a group of interrelated web development techniques used on the client-side to create asynchronous web applications. It is the most viable Rich Internet Application (RIA) technology so far. Client-scripts can generate requests to the server without blocking the user interaction and the responses are processed when they arrive. This is accomplished by using XMLHttpRequest object and JavaScript to make asynchronous requests to the web server, parsing the responses and then updating the page DOM HTML and CSS. Despite the name, the use of XML is not required; JSON is often used instead. JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript programming language, standard ECMA-262 3rd Edition. JSON is primarily used to transmit data between a server and web application, as an alternative to XML. JSON has also become another popular way to store data in a structured way after XML. With JSON, you use JavaScript's array and object literals syntax to define data inside a text file in a way that can be returned as a JavaScript object.

There are several studies that evaluated WAVSs but none of them implemented vulnerabilities behind both AJAX and JSON. Since JSON is the response content of AJAX calls, a scanner must be capable of executing AJAX scripts before accessing JSON code. The application language is one of the factors when choosing a WAVS. For example, if the application uses extensive AJAX code, a scanner that cannot crawl AJAX will not be a good choice. To detect a vulnerability hidden behind AJAX/JSON code, a scanner needs to execute AJAX code and audit JSON parameters.
There are many web scanners available on the market [9],[14], commercial and open source, and each scanner may be different in the way and number of vulnerabilities it can detect. However, a good scanner should find as many true vulnerabilities as possible. It should also be able to build a report that shows the time, action performed, effects, and the area of the application where it detected the vulnerability. Additionally, it should report a minimum number of false positives (spurious vulnerabilities) in the report.

This paper presents a test application and an assessment of 5 state-of-the-art web application vulnerability scanners on their capabilities to detect vulnerabilities accessible after executing AJAX and JSON code. The test suite is implemented in a realistic application (online store) and contains many vulnerability instances, with different levels of exploitation difficulty. Implemented vulnerabilities include reflected and stored XSS (also known as persistent or second-order XSS), and first-order and stored SQL Injection. Experimental results show that executing AJAX code is still a challenge to many scanners. Among the 5 tested tools, only one executed AJAX requests and audited JSON parameters successfully. Unfortunately, no scanner detected stored vulnerabilities.

The remainder of this paper is organized as follows: Section II describes the principle of web application vulnerability scanners and their support for input delivery methods and web application technologies. Section III briefly discusses related work. Section IV describes the test application. Experimental results are presented and discussed in Section V. We conclude with recommendations to assess complete capability of web application vulnerability scanners and future work.

II. WEB APPLICATION VULNERABILITY SCANNERS

Web Application Vulnerability Scanners (WAVS), also known as web application security scanners, are automated black-box testing tools [15] that examine web applications for security vulnerabilities. They crawl through a web application's pages and search the application for vulnerabilities by simulating attacks on it. This involves generation of malicious inputs that are submitted to the application and subsequent evaluation of application's response. A large number of both commercial and open source tools are available and all these tools have their own strengths and weaknesses. Web application developers use them to verify the security of their products and to preserve the integrity, confidentiality, and availability of developed applications for their clients.

2.1 Mechanics of a scanner

Most of the WAVSs use fuzz testing or fault injection. Oehlert [16] defines fuzzing as a highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The mechanics of a scanner is a three-step process:

1. **Crawling:** the scanner crawls the web application to find the web pages that are part of that application and associated input vectors (data entry points), such as forms, request parameters and cookies, and creates an indexed list of all visited pages. The task of finding pages can be fulfilled automatically (using web crawlers), manually (recorded by a proxy) or semi-automatically (crawler asks for operator's assistance). If the crawling engine of a web application scanner is poor and cannot reach the vulnerability, then the scanner will surely miss the vulnerability. WAVSs have problems indexing web pages that use JavaScript and AJAX. AJAX-based applications rely on Stateful Asynchronous client/server communication, and client-side runtime manipulation of the Document Object Model (DOM) tree. This makes them fundamentally different from traditional web applications and harder to test.
2. **Simulation of attacks (fuzzing):** the scanner sends potential attack patterns to the previously identified inputs. For each input and for each vulnerability type for which the web vulnerability scanner tests, the attacker module generates values that are likely to trigger a vulnerability. For example, the attacker module would try to inject JavaScript code when testing for XSS vulnerabilities.
3. **Response analysis**: In this phase the result of the fuzzing phase is analyzed to check if the web application is vulnerable and to provide feedback to the other modules. The result of every fuzz packet is monitored; it is vital to know what packet caused what result. How this monitoring takes place depends on the target and the type of vulnerability that is tested. For instance, monitoring for XSS vulnerabilities will consist of crawling the web application again and searching for pages that contain the input that was injected during the execution of the fuzzed data [17].

2.2. Input Delivery Methods Supported by Web Application Vulnerability Scanners

Modern web applications use various methods for delivering inputs from the browser to the server. These methods include standard input delivery methods, such as HTTP query string parameters (GET) and HTTP body parameters (POST), modern delivery methods, such as XML and JSON, and binary delivery methods for technology specific objects, such as Java serialized objects, AMF, and WCF.

According to a recent research [9], which gathered and summarized scanners' documentations, commonly supported input vectors are GET (supported by all the scanners), and POST. Cookies and Headers are supported by almost a half of scanners tested. Some input vectors, such as JSON parameters, XML Elements Content, and HTTP Path, are supported by a small number of scanners. Delivery methods for technology specific objects, namely Java Serialized Objects, .NET PostBack Encoded Parameters, .NET Serialized Objects, .NET Binary WCF Objects, and HTML 5 Web Sockets are not supported by any scanner.

Supporting the input delivery method of the tested application is a necessity as the vast majority of active scan plugins (plugins that attempt to find potential vulnerabilities by using known attacks against the selected targets) rely on input that is meant to be injected into client originating parameters. A tool cannot detect a vulnerability in a given parameter if it cannot mimic an application’s method of delivering input. The more vectors of input delivery that the scanner supports, the more versatile it is in scanning different technologies and applications.

2.3. Supported Application Technologies

WAVSs are not language dependent. A WAVS is able to scan JAVA/JSP, PHP or any other engine driven web application. However, a scanner may perform differently on applications written in different languages. Evaluations [6], [7], and [9] have shown that almost all scanners can automatically crawl HTML Links and Forms, but many of them cannot fill fields with default values while automatically crawling the application. These studies have also shown that the major challenge about crawling is the execution of client-side code. Doupé [7] and Ertaul [18] added that crawling multi-step processes is another major challenge.

Application technologies overlooked by most web application scanners include JSON, REST, Java Applets, Flash, Silverlight, and anti-XSRF/CSRF tokens. The application language is one of the factors when choosing a web scanner. For example, if the application uses extensive AJAX code, a scanner that cannot crawl AJAX will not be a good choice. Dynamic technologies, such AJAX, generally challenge WAVSs. WAVSs send requests to the application and analyze responses. Since AJAX is used on the client-side to create asynchronous web applications, a WAVS have to execute AJAX scripts before it can send a request. In classic web application, when a vulnerability is discovered, WAVSs would reference the page and parameter where the issue was found. This cannot be applied to AJAX application because everything is often presented as a single page with many possible user events. The vulnerability may rely on a certain combination of steps to occur before it exists, which makes automated scanning a challenge. To detect a vulnerability hidden behind AJAX/JSON code, a scanner needs to execute AJAX code and process and analyze JSON parameters.

III. RELATED WORK

Our work is related to two main research areas: the design of web applications for assessing web application vulnerability scanners and the evaluations of these tools.

Testing WAVSs requires web applications with vulnerabilities. Unfortunately, no standard test suite is currently available. There are several existing vulnerable web applications, such as and Mutillidae [19], Damn Vulnerable Web Application (DVWA) [20], and WebGoat Project [21], with vulnerabilities of different types, but they are mainly designed for teaching web application security rather than realistic benchmark for web vulnerability scanners. OWASP SiteGenerator Project [22] is a tool to generate dynamic websites based on XML files and predefined vulnerabilities. Wackopicko [23] and Web Application Vulnerability Scanner Evaluation Project (WAVSEP) [24], test applications used in evaluations [7] and [9], respectively, are also publicly available. However, none of these test suites includes test cases for vulnerabilities hidden behind AJAX and JSON, despite the popularity of these technologies in rich internet applications.

There are several studies that evaluated WAVSs. Some tested scanners for one particular vulnerability and others implemented many vulnerability types in their test applications. Bau et al. [6] performed a comparative study on 8 commercial scanners and tested them on well-known web applications and on a custom application. They found that most scanners tended to perform well on reflected XSS and first-order SQL injection vulnerabilities but very poorly on stored vulnerabilities. Additionally, results show that scanners have poor understanding of active content, such as Silverlight and Flash, and scripting languages, such as Java Applets. The custom application included a vulnerability in which usernames are disclosed via AJAX requests and CSRF-like JSON hijacking vulnerabilities but authors did not discuss detection results for these particular vulnerabilities. Results were presented for information disclosure and CSRF vulnerabilities in general.

Doupé et al. [7] evaluated eleven black-box web vulnerability scanners against a custom application containing different types of vulnerabilities and different crawling challenges. Executing client-side code challenged many scanners. The test application included a vulnerability behind JavaScript but no AJAX or JOSN test cases implemented. The authors used Web Input Vector Extractor Teaser (WIVET) [25] to test the scanners' support for the client-side code.

Ferreira et al. [8] built a vulnerable application and ran scanners against it. Results indicated that scanners performed decently well on reflected XSS and first-order SQL injection, and poorly on stored XSS and Cross-Site Request Forgery (CSRF). No scanner detected unrestricted URL access vulnerability.
Chen [9] compared price and features of WAVSs. Detection results show that many scanners performed fairly well in detecting reflected XSS, and first-order SQL injections, and a number of scanners found Path Traversal/ Local File Inclusion. Scanners, as a group, performed very poorly in detecting old, backup and unreferenced files, and unvalidated redirect. Chen also summarized, based on documentations, scanners audit features, scan barrier and input vector support, and authentication features.

Ertaul et al. [18] implemented OWASP Top 10 vulnerabilities in a test application and tested two WAVSs against the application. Detection results indicate that the biggest challenge for these two WAVSs is to exploit stored and multi-step vulnerabilities. Khoury et al. [26] tested 3 black-box scanners for persistent SQL injection vulnerabilities by running them on testbeds used by Bau [6] and Doupé [7] and a custom application containing one vulnerability instance. Test results show that the scanners are very poor at detecting persistent SQL injections even when they taught how to exploit the vulnerabilities. Alassmi et al. [27] extended the analysis of Bau and Doupé on detection of stored XSS and confirmed the weaknesses and limitations they discussed, mainly in scanners' analysis phase.

IV. Test Application

Testing web application vulnerability scanners requires web applications with vulnerabilities. Unfortunately, no standard test suite is currently available. As discussed in section III, none of existing test suites includes vulnerabilities hidden behind AJAX and JSON, despite the popularity of these technologies in rich internet applications. Therefore, we decided to create our own test application.

4.1 Design

The test application is a realistic and fully functional online shopping site. A user is able to browse product catalog, purchase products, write reviews on products, and leave a feedback on guest book. When logged in, a user can buy, check order history, return a product, save cart content and wishlist, and edit personal information. The system is managed via a special area for administrators only.
Authentication: the test application has a user registration system. After logging in, the user can access restricted features.
Search: The test application provides a search toolbar at the top of every client page to allow users to easily search for products.
Purchase Products: The purchase is a multi-step process in which a shopping cart is filled with the items to be bought. After adding items to the cart, the total price is calculated, discount coupons may be and the order is placed. The checkout process uses extensive AJAX scripts. At the time of checking out, if the user is logged in, his address is fetched and presented to the user as his shipping address, with an option to enter a new address. If the user is not logged in, a login form, register option, and an option to checkout as a guest are provided. After specifying shipping address,

the application fetches and displays shipping methods (for simplicity, we used flat shipping rate: charging a fixed fee per item). Afterwards, the user chooses a payment method (e.g. cash on delivery) and a *comment* field is provided for any additional information. The entire checkout process takes place without leaving the checkout page; AJAX requests are used to fetch and send content depending on the user's choice. If a scanner cannot execute AJAX and JSON code, it cannot complete the checkout process.

Review on products: on product information page, a user can write a review on that product. The review content is submitted in AJAX request, with JSON as response dataType.
Guestbook: the guestbook page provides a way to receive feedback from visitors. The form used to submit feedback contains a name, email, and message fields.
Administrator's Area: The test application has a special area for administrators only. Their tasks include managing products and user accounts, performing order transactions, among others.
The test application is a PHP-based application and is deployed on Apache server. It uses a database on MySQL server to store the data for the website. These technologies are chosen as the underlying architecture of our application because of their widespread and popularity. The test application also uses Hypertext Markup Language (HTML), Cascading Style Sheet (CSS), JavaScript, AJAX, and JSON technologies.

4.2. Vulnerabilities

The test application contains different vulnerability instances of the same vulnerability types, from easily detectable to instances harder to detect. We implemented Cross-Site Scripting and SQL injection as they are the most popular vulnerabilities in the wild.

First-order SQL Injection: In this category, we implemented 5 vulnerability instances: (1) there is SQL vulnerability on feedback page as the name of the author is inserted into the SQL query without validation. (2) To navigate to the next page of guestbook messages, the page number is sent in the GET request and used in SQL query without sanitization. (3) From product page, the review author name is inserted into the database without proper sanitization. The form data is submitted in AJAX request, and the response data is a JSON object. (4) Before checking out, a customer can view the details of his shopping cart. The coupon may be applied to see new prices. The coupon number is immediately used in SQL command without validation. The coupon value is sent into AJAX Request and the response data is a JSON object. (5) At the end of checkout process, before confirming order, the customer is given a *comment* field to add any additional information about his order. This comment is inserted into SQL query without validation. For a scanner to detect this vulnerability, it must execute all previous AJAX requests required to reach the end of the checkout process.

Second-order SQL Injection: There are two second-order SQLI vulnerability instances. (1) On guestbook, the author's name, which is inserted without sanitization, is used to fetch all messages written by this person. (2) On the product page, the name of the reviewer, which is inserted without validation, is used to retrieve reviews posted by this customer. Since reviews are submitted using AJAX script and JSON data as the response, the scanner must execute AJAX and JSON, and revisit the page to detect this vulnerability.

Reflected XSS: We implemented 2 instances of reflected XSS: (1) On Contact us page, the user is required to enter his name, email, and the message body. When the form is submitted, the name is reflected back to the user (in a JSON object), without validation, telling them that their message was successfully sent. (2) When a user submits search keywords, the search keyword is displayed back to the user, along with search results, without proper escaping.

Stored XSS: The test application contains 4 stored XSS. (1) On product page, the name of review author is not escaped, allowing an attacker to write a name containing malicious code. Whenever the product page loads, which is loaded along with the product reviews, the attack will be triggered and the code will be executed. The review form data are sent using AJAX request, without leaving the product page. (2) The review message is not properly escaped. For a scanner to detect the above two stored XSS it must have the ability to execute AJAX scripts and audit JSON response.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Vol. 4, Issue 6, June 2015

(3) On guestbook page, a customer can write a message. The name of the customer is not validated, and is later displayed on feedback page. (4) The feedback body is not properly sanitized.
In total, the test application contains 13 vulnerabilities (2 reflected XSS, 4 persistent XSS, 5 first-order SQLI, and 2 second-order SQLI).

V. EXPERIMENTAL EVALUATION

We tested 5 web application vulnerability scanners by running them on our test application. We chose scanners that claim the ability to execute AJAX or both AJAX and JSON code. These are Acunetix [28], Arachni [29], IronWASP (Iron Web Application Advanced Security testing Platform) [30], Vega [31], and ZAP (Zed Attack Proxy) [32]. ZAP also offers scanning profile for persistent XSS vulnerabilities. All these scanners claim to have the ability to audit JavaScript and AJAX code. Arachni, Acunetix, and ZAP add that they are able to audit JSON data. Characteristics of the scanners we evaluated are summarized in Table I.

Table I Characteristics of Scanners Evaluated

Name	Version	License
Acunetix	9.5 Build 20140602	Commercial
Arachni	1.0.6-0.5.6	Apache License v2.0
IronWASP	2015 beta	GPL3
Vega	1.0	EPL1
ZAP	2.3.1	Apache License v2.0

5.1. Setup

The initial state, before every test run, included 14 products (in different product categories), 2 user reviews (on different products), 1 customer account, 1 order, and 1 customer feedback. In addition to scanning the whole application, vulnerable pages were individually scanned multiple times using various configurations, and using a scan policy that only included the relevant plugins. Each scanner was also put into a proxy mode and then the user browsed to each vulnerable page. A review was written, a feedback was posted, a coupon was applied, and a new order was placed. The scanner was then instructed to scan the test application.
The test procedure consists of the following steps:

- Initialize the database
- Configure the test application, including deleting all cookies
- Configure and run a selected web vulnerability scanner
- Save scanner results or report
- Dump the database to a backup directory
- Check the scanner’s report and detection results
- Analyze database records
- Count and classify vulnerabilities in the scanner output

We repeated these steps for every scanner. After the test results are generated, they are reviewed and organized in groups, namely Detected, Missed (false negatives), and false positives.

5.2 Vulnerability Detection Results and Discussion

Overall Results: Table II presents the number of vulnerabilities detected by each scanner. Our test results show that WAVSs perform decently well on reflected XSS and first-order SQL injection on pages that do not have client-side code, and poorly on executing AJAX scripts. Among 7 vulnerabilities accessible after executing AJAX and JSON (1 instance of reflected XSS, 2 persistent XSS, 3 first-order SQL injections, and 1 second-order SQL injection), only 1 instance of first-order SQL injection was found and it was detected by one scanner which executed AJAX and JSON successfully. This suggests that AJAX and JSON support have to be improved as these technologies are popular in modern web applications.

Table II Vulnerability Detection Results

		Acunetix	Arachni	IronWASP	Vega	ZAP
First-order SQLI	Detected	3	2	2	2	2
	Missed	2	3	3	3	3
Second-order SQLI	Detected	0	0	0	0	0
	Missed	2	2	2	2	2
Reflected XSS	Detected	2	1	1	1	1
	Missed	0	1	1	1	1
Persistent XSS	Detected	0	0	0	0	0
	Missed	4	4	4	4	4

First-order SQL Injection: All scanners detected two easily detectable SQL injection vulnerabilities. These two vulnerabilities are on forms that do not involve client-side code for processing and there is no form of validation applied on user input. In addition, SQL errors are reflected back to the client immediately after the attack. The tool only needs to send a request and analyze its response immediately, without any additional step. Acunetix is the only scanner that executed AJAX requests and audited JSON data in the response after writing a product review, hence detecting the SQLI vulnerability on product page.

No scanner found SQL injection vulnerabilities present when applying a coupon on the shopping cart and on comment field at the end of checkout process. For a scanner to detect a vulnerability when using coupon, it has to maintain shopping cart session cookie and customer authentication session cookie, implemented using custom session cookies, which complicated all the scanners. The vulnerability on the comment field was not found by any scanner as no scanner reached the end of checkout process. The failure to reach this stage is due to the inability of tested scanners to understand our custom session cookies and execute AJAX/JSON code.

Second-order SQL Injection: scanners did not detect any of the two persistent SQLI vulnerabilities. One vulnerability is on the feedback page, which does not contain any client-side code, and the other one is on product page (name of review author) which requires execution of AJAX and JSON code before it can be detected. All the scanners scanned the feedback page and succeeded to store attack codes in the database, but none of them reported a vulnerability. This indicates that the failure to detect persistent vulnerabilities is not due to the failure to execute client-side code but because of weaknesses in attack and analysis modules of the scanners. Most of the attack codes were meant to exploit reflected or blind SQL injection, where an immediate answer was expected. Unlike first-order SQLI, receiving an error while trying to store an attack code could mean that the first step has failed, and the code was not stored. Detecting a blind SQL injection could mean that the first step of stored SQLI has succeeded. The scanner should know if the attack code was successfully stored and keep its state. It also should remember which form was used to store that attack code, and what other data were submitted along with the code. Later, the scanner should crawl again to find related data and try to execute its own code. No scanner followed this logic. Having a Stateful scanner [33] would be more efficient for the scanner to relate the application's inputs and outputs, and thus understand the state-based transactions that take place.

Reflected XSS: All the scanners found the XSS vulnerability on the search field but only one tool detected the vulnerability on contact us page as this vulnerability requires the execution of AJAX script and analyzing JSON parameters. To test for XSS vulnerabilities, a tool needs to identify input vectors and verify whether an application or web server responds to requests containing scripts with an HTTP response that could be executed by a browser. All scanners successfully injected attack scripts in the search field and linked the response to the attack as the injected scripted is immediately executed. Except Acunetix, executing AJAX script challenged the tools and consequently they did not find the vulnerability on contact us page.

Persistent XSS: All the scanners failed to detect stored XSS vulnerabilities despite using various configurations, including relevant scanning profiles. When scanning the feedback page, all the tools successfully saved attack codes in the database but failed to detect stored vulnerabilities. The failure may be due to the fact that the impact of stored XSS is not immediate, as all the scanners detected reflected XSS on search toolbar, and due to the weaknesses of analysis phase of web scanners cycle.

To detect stored XSS vulnerabilities, the scanner must be able to mimic persistent XSS attacks in order to detect XSS vulnerabilities. A persistent attack requires two or more steps to complete the attack. The tool has to store the attack script in the database. Later, the attack code should be retrieved from the database and executed. Scanners have difficulties with following these steps, and consequently miss these vulnerabilities. We believe that scanners failed to confirm that an attack code was successfully stored and consequently did not revisit the injected page to verify if the injected code was rendered or not. The scanners may also have trouble linking the later observation with the earlier attack event.

On product page, the scanner must execute AJAX and JSON, and revisit the page to find the vulnerabilities. Although Acunetix successfully inserted scripts, it failed to make required analysis. It did not realize that it was executing its own attack code. Failure to detect stored XSS vulnerabilities while the attack code is stored indicates that increasing observability needs further research.

False positive: Some scanners reported genuine false positives. The number of false positives measures the accuracy of the scanner. If the false positives are high, the tool will be less useful to the user as he/she has to figure out which of the vulnerabilities reported are actual flaws and which are spurious results of the analysis. And this requires a security-conscious user to evaluate the reports. The total number of false positives for each scanner is shown in Table III.

Table III False Positives

Scanner	False Positive
Acunetix	0
Arachni	1
IronWASP	1
Vega	3
ZAP	4

ZAP reported four false XSS vulnerabilities and Vega reported three false SQL injection vulnerabilities. Arachni and IronWASP reported one SQL injection each.

VI. Conclusion

This paper presented a test application and assessed 5 web application vulnerability scanners on their effectiveness at detecting vulnerabilities hidden behind AJAX/JSON. Experimental results show that executing AJAX code is a challenge to many tools. Among five scanners we tested, only one tool detected a vulnerability behind AJAX/JSON. Since JSON is the response content of AJAX requests, a tool must be capable of executing AJAX scripts before accessing JSON code.

As many tools send the same attack code into hundreds of requests or use the same attack codes to test for different vulnerability types, web scanner developers should improve on increasing coverage of attack vectors and different variations of attacks to be injected. In order to increase stored vulnerabilities detection ability, a scanner should crawl the application again after injecting attack codes and verify if its own code is rendered or not.
As WAVSs' evaluations did not cover all main features, they do not provide a complete assessment and comparison of WAVSs. A complete benchmark should implement multiple vulnerability instances, from easily exploitable and detectable to the unbreakable and virtually undetectable, for each vulnerability type. Features to be tested include entry point coverage, input vector coverage, audit features, and accuracy level of each plugin supported by the scanner.

Crawling modern web technologies, automated detection of complex vulnerabilities, such as stored XSS, stored SQL injection, and custom session management ability require further research.

VII. ACKNOWLEDGEMENT

This project is sponsored by "the Scientific Research Foundation of the Overseas Chinese Scholars".

REFERENCES

[1] MITRE, Common Vulnerabilities and Exposures. http://cve.mitre.org/
[2] OWASP, Top Ten Project. https://www.owasp.org/index.php/OWASP_Top_Ten_Project
[3] Open Source Vulnerability Database. http://osvdb.org/
[4] P. Wood, B. Nahorney, K. Chandrasekar, S. Wallace, and K. Haley, "Internet Security Threat Report 2014", Semantec Corporation, Vol.19, 2014.
[5] L. Suto, "Analyzing the Accuracy and Time Costs of Web Application Security Scanners", Beyond Trust, 2010. http://www.beyondtrust.com/Content/whitepapers/Analyzing-the-Accuracy-and-Time-Costs-of-Web-Application-Security-Scanners.pdf
[6] J. Bau, , E. Bursztein, D. Gupta, and J. Mitchell, "State of the Art: Automated Black-Box Web Vulnerability Testing", in Proc. 2010 IEEE Symposium on Security and Privacy, pp. 32-345, 2010.
[7] A. Doupé, M. Cova, and G. Vigna, "Why Johnny Can't Pentest: An Analysis of Black- Web Vulnerability Scanners", in Proc. 7th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 111-131, 2010.
[8] A. M. Ferreira, and H. Kleppe, "Effectiveness of Automated Application Penetration Testing Tools", Master dissertation., Master Education SNE/OS3, University of Amsterdam, Netherlands, 2011.
[9] S. Chen, Price and Feature Comparison of Web Application Scanners, February 2014. http://sectoolmarket.com
[10] S. Choudhary, M. E. Dincturk, S. M. Mirtaheri et al. "Crawling Rich Internet Applications: The State of the Art", Software Security Research Group, University of Ottawa. 2012. http://ssrg.eecs.uottawa.ca/docs/CASCON2012.pdf
[11] S. M. Mirtaheri, D. Zou, G. V. Bochmann G., G. V. Jourdan, and I. V. Onut, "Dist-RIA Crawler: A Distributed Crawler for Rich Internet Applications", in Proc. 2013 Int. Conf. on P2P, Grid, Cloud, and Internet Computing, pp. 105-112, 2013.
[12] K. Benjamin, G. V. Bochmann, G. V. Jourdan, and I. V. Onut, "Some Modeling Challenges when Testing Internet Rich Applications for Security", Third International Conf. on Software Testing, Verification, and Validation Workshops, pp. 403-409, 2010.
[13] A. Mesbah, A. V. Deursen, and S. Lenselink, "Crawling AJAX-Based Web Applications through Dynamic Analysis of User Interface State Changes", ACM Transactions on the Web, Vol. 6, No.1, article 3, 2012.
[14] OWASP, Vulnerability Scanning Tools. https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools
[15] E. Fong, and V. Okun, "Wed Application Scanners: Definitions and functions". in Proc. 40th Annual Hawaii Int. Conf. on System Sciences, 280b, 2007.
[16] P. Oehlert, "Violating Assumptions with Fuzzing", IEEE Computer Security and Privacy, vol. 3, No.2, pp. 58-62, 2005.
[17] S. McAllister, E. Kirda, and C. Kruegel, "Leveraging User Interactions for In-Depth Testing of Web Applications", in Proc. 11th international symposium on Recent Advances in Intrusion Detection, pp. 191-210, 2008.
[18] L. Ertaul, and Y. Martirosyan, "Implementation of a Web Application for Evaluation of Web Application Security Scanners", in Proc. 2012 Int. Conf. on Security and Management, pp. 82-89, 2012.
[19] J. Druin, "Mutillidae 2 Project", Open Web Application Security Project. https://www.owasp.org/index.php/OWASP_Mutillidae_2_Project
[20] Damn Vulnerable Web Application (DVWA), RandomStorm, http://www.dvwa.co.uk
[21] B. Mayhew, "OWASP WebGoat Project". https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project
[22] OWASP SiteGenerator Project. https://www.owasp.org/index.php/ OWASP_SiteGenerator
[23] A. Doupé, GitHub Inc.. https://github.com/adamdoupe/WackoPicko
[24] S. Chen, Web Application Vulnerability Evaluation Project (WAVSEP). http://code.google.com/p/wavsep
[25] B. Urgun, Web Input Vector Extractor Teaser. https://github.com/bedirhan/wivet
[26] N. Khoury, P. Zavarsky, D. Lindskog, and R. Ruhl, "Testing and Assessing Web Vulnerability Scanners for Persistent SQL Injection Attacks", in Proc. 1st Int. Workshop on Security and Privacy Preserving in e-Societies, pp. 12-18, 2011.
[27] S. Alassmi, P. Zavarsky, D. Lindskog, R. Ruhl, A. Alasiri, and M. Alzaidi, "An analysis of the Effectiveness of Black-box Web Application Scanners in Detection of Stored XSSI Vulnerabilities", International Journal of Information Technology and Computer Science, vol. 4, No. 1, 2012.
[28] Acunetix Web Vulnerability Scanner. http://www.acunetix.com/vulnerability-scanner
[29] T. Laskos, Arachni Scanner. http://www.arachni-scanner.com
[30] L. Kuppan, IronWASP. http://ironwasp.org
[31] Vega Vulnerability Scanner, Subgraph. https://subgraph.com/vega
[32] S. Bennetts, OWASP Zed Attack Proxy (ZAP). https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[33] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner", in Proc. 21st USENIX Security Symposium, pp. 523-538, 2012.

View publication stats