Int. J. Inf. Secur.
DOI 10.1007/s10207-016-0359-4

REGULAR CONTRIBUTION

Black-box detection of XQuery injection and parameter tampering vulnerabilities in web applications

G. Deepa[1] · P. Santhi Thilagam[1] · Furqan Ahmed Khan[1] · Amit Praseed[1] · Alwyn R. Pais[1] · Nushafreen Palsetia[1]

Abstract As web applications become the most popular way to deliver essential services to customers, they also become attractive targets for attackers. The attackers craft injection attacks in database-driven applications through the user-input fields intended for interacting with the applications. Even though precautionary measures such as user-input sanitization is employed at the client side of the application, the attackers can disable the JavaScript at client side and still inject attacks through HTTP parameters. The injected parameters result in attacks due to improper server-side validation of user input. The injected parameters may either contain malicious SQL/XML commands leading to SQL/XPath/XQuery injection or be invalid input that intend to violate the expected behavior of the web application. The former is known as an injection attack, while the latter is called a parameter tampering attack. While SQL injection has been intensively examined by the research community, limited work has been done so far for identifying XML injection and parameter tampering vulnerabilities. Database-driven web applications today rely on XML databases, as XML has gained rapid acceptance due to the fact that it favors integration of data with other applications and handles diverse information. Hence, this work proposes a black-box fuzzing approach to detect XQuery injection and parameter tampering vulnerabilities in web applications driven by native XML databases. A prototype XiParam is developed and tested on vulnerable applications developed with a native XML database, BaseX, as the backend. The experimental evaluation clearly demonstrates that the prototype is effective against detection of both XQuery injection and parameter tampering vulnerabilities.

Keywords Web application security · Vulnerability scanner · Injection attacks · Fuzz testing · Logic vulnerabilities · XML injection

1 Introduction

Over the years, web applications have become a primary source for information dissemination and transaction handling. Organizations such as banks, governments, and business corporations utilize web applications to extend their services to the general public. As the web applications handle sensitive personal information and details of financial transactions, security has become a primary concern. However, security has not kept pace with the rapid growth of web applications.

According to the Internet Security Threat Report by Symantec Corporation [1], one in eight web applications has critical unpatched vulnerabilities. A vulnerability is a coding flaw in the application. New types of attacks emerge from time to time to exploit the vulnerabilities in web applications. The vulnerabilities in the application may be exploited by users during interaction with the application (i.e., via user input). Hence, improper user input is one of the fundamental

✉ G. Deepa
gdeepabalu@gmail.com

P. Santhi Thilagam
santhi@nitk.ac.in

Furqan Ahmed Khan
furqankhan08@gmail.com

Amit Praseed
amitpraseed@gmail.com

Alwyn R. Pais
alwyn@nitk.ac.in

Nushafreen Palsetia
nushafreen@gmail.com

[1] National Institute of Technology Karnataka, Mangaluru, India

Published online: 11 January 2017

security concerns in any web application. Since almost all web applications have form fields for submitting information to the server, a malicious user can submit crafted input to interfere with the normal working of the applications. The crafted input can pose serious security risks in the absence of proper server-side validation. This kind of vulnerability is called an injection vulnerability. The security consortiums, OWASP [2], SANS [3], and WASC [4] list injection vulnerabilities as the most prevalent flaw in web applications. They allow an attacker to compromise the security of the application and permit them to steal confidential information or inject malicious data into the application.

Injection attacks can be of two types depending on the input given. (i) An attacker can craft malicious input designed to reveal sensitive information of the web application, or can corrupt the database of the web application. These attacks attempt to modify the structure of the query submitted to the database by providing malicious input to the web application server. Depending on the database server used, the attack can be referred to as an SQL injection attack or an XQuery injection attack. (ii) Malicious users can also inject invalid input easily into the web application through HTTP parameters after disabling the JavaScript execution at the client side, which bypass the precautionary measures such as user-input validation employed at the client side. These inputs will not change or affect the structure of the query in any way, but they intend to break the expected business specification of the application due to lack of server-side validation. Such attacks are typically called parameter tampering attacks [5–7]. As a simple example, in an online shopping application an attacker can tamper the value of a parameter, say *price* of an item to zero or a negative value to reduce the payment amount.

Traditionally relational databases have been in use in all major web applications. However in recent years, Extensible Markup Language (XML) is widely used due to the ease of interoperability and the ability to represent a variety of data. XML is a data representation and is used for exchange of information between heterogeneous applications in the form of XML documents. These documents are stored in either an extended relational DBMS or a native XML database system for efficient processing of the information available in the documents [8,9]. XQuery / XPath can be used as a query language for accessing the data from the XML documents. A Native XML database (NXD) has an XML document as its fundamental unit of storage and defines a logical model based on the content in the XML document [10]. NXDs are utilized in cases where the data involved fit into an XML data model rather than a relational data model. NXDs are generally preferred by applications that hold highly diverse information, handle rapidly evolving schemas, and involve integration of information from different sets of applications. The application domains such as financial applications, document management systems, healthcare systems, catalog data, business-to-business transaction records, and corporate information portals prefer NXDs [11].

As XML databases have only started gaining importance in the recent years, the literature related to XML injection is relatively small. Existing literature has gone primarily into detecting or preventing XML injection attacks on web services and restricts itself with detection of only certain types of injection attacks. Therefore, there is a need for an approach that deals with detecting or preventing injection attacks on native XML databases effectively. Since extensive precautionary measures are available for addressing injection attacks, the focus of the attackers is shifting toward abusement of the functionality of the web application, which is possible by polluting the HTTP parameters with values that violate the restrictions imposed on them. The restrictions imposed on HTTP parameters can be disabled by inactivating the execution of JavaScript at the client end, and the violated input succeeds due to lack of server-side validation. Exploitation of the HTTP parameters leads to consequences such as heavy financial loss, quality of service degradation, and denial of service. Hence, this paper focuses on development of a prototype for the detection of both XQuery injection and parameter tampering vulnerabilities in web applications.

Considering the pros and cons of the existing approaches, a prototype to identify XQuery injection vulnerabilities is developed using a query model-based approach. Existing query model-based approaches use static analysis and also require code instrumentation for detecting SQL injection. The proposed approach on the other hand is a fully automated black-box fuzzer, which does not require access to the source code of the application, and concentrates on identifying injection vulnerabilities that tamper the XQueries as well as the intended logic of the application. The prototype makes use of a context-aware crawler which can identify client-side constraints imposed on form fields while populating them with valid values. Such a crawler makes sure that fewer web pages are not missed out from getting identified, which in turn increases the effectiveness of the prototype. To the best of our knowledge, this is the first work that provides a single framework for detection of injection vulnerabilities, leading to both XQuery injection attacks and parameter tampering attacks in native XML database-driven web applications. The prototype is capable of detecting different types of XQuery injection vulnerabilities specified in OWASP guidelines [12] and in [13]. The major contributions of this work are as follows:

- We propose an automated black-box approach for identifying injection vulnerabilities that either alter the query models submitted to the databases or break the intended behavior of the application in native XML database-driven web applications.
- We implement a prototype system called XiParam based on the proposed approach.

– We develop an HTML/JavaScript analyzer that extracts constraints imposed on HTTP parameters after analyzing the HTTP response.
– We evaluate the effectiveness of the proposed approach using web applications that are customized to use native XML database for storing data.

The remainder of this paper is organized as follows: Section 2 provides a background on XQuery injection and parameter tampering vulnerabilities. Section 3 presents the related work. The proposed approach and implementation details are described in Sect. 4. Section 5 provides information on the test applications and discusses the results. The paper is concluded in Sect. 6.

2 Background

This section provides a background on injection attack, the possible mechanisms through which malicious input can be injected into the application followed by a simple example on XQuery injection and parameter tampering attack.

2.1 Injection attacks

A vulnerability is a programming flaw in the application that can be exploited by the attackers for injecting malicious code in order to compromise the security of the application. The malicious code can contain SQL/XML commands, JavaScript code, etc., which on execution results in unexpected behavior of the application. The traditional ways of injecting malicious code into the application are through the user input, cookies, server variables, and second-order injection [14]. The malicious input injected through any one of the avenues, when not validated properly by the application, results in injection attacks. Therefore, for extending support to prevent injection attacks, validation of user input has become the de facto standard for web applications. If the validation of the input is present only at the client side and not enforced at the server side, then the attacker can easily bypass the check by disabling the JavaScript. Thus, lack of server-side validation has also become one of the possible avenues for injection of malicious input.

Web applications can be exploited in various ways based on the user input. If the input consists of SQL/XML commands, it results in SQL/XML injection attack. There is a special case where the input supplied by the attacker does not contain malicious code, but affects the intended business specification of the application [5–7]. Such input results in logic attacks where the invalid input is supplied by the user through either the form fields or parameters in the HTTP request. Hence, these attacks are also referred to as parameter tampering attacks.

This work aims at detecting the input fields and HTTP parameters that are vulnerable to XQuery injection and parameter tampering attacks. The following subsections illustrate XQuery injection and parameter tampering attacks with examples.

2.1.1 XQuery injection attack

A simple example of an XQuery injection attack is described with the following XML document, *employee.xml* [15]:

```
<?xml version="1.0" encoding="ISO
    -8859-1"?>
<employeelist>
    <employee level="level1">
        <userid> 10001 </userid>
        <fname> John </fname>
        <grading> A </grading>
    </employee>
    <employee level="manager">
        <userid> 10003 </userid>
        <fname> Andrew </fname>
        <grading> A+ </grading>
    </employee>
    <employee level="level1">
        <userid> 10041 </userid>
        <fname> Ken </fname>
        <grading> C </grading>
    </employee>
</employeelist>
```

The source code for retrieving the detail of a user from the XML document is given below.

```
String strName = (String)request.
    getParameter("id");
String strQuery = "xquery for $x in doc
    ('employee.xml')/employeelist/
    employee where $x/userid='" +
    strName + "' return $x";
```

Assuming that the value for the variable *strName* is obtained from the user input, the XQuery would return the following, when the input provided by the user is *"10001"*.

```
<employee level="level1">
    <userid> 10001 </userid>
    <fname> John </fname>
    <grading> A </grading>
</employee>
```

As the input string from the user is not validated properly, an attacker can provide input in such a way that the query is manipulated for retrieving the complete set of users. By providing the input string *XXX' or '1'='1* for the variable *strName*, the attacker can make the XQuery to return a node set of all the users. The malformed XQuery is:

```
for $x in doc('employee.xml')/
employeelist/employee where $x/
userid='XXX' or '1'='1' return $x
```

Different types of XML injection attacks suggested by OWASP [12] and Palsetia et al. [13] are listed as follows:

- Tautology attack
- Meta Character injection attack
- Comment injection attack
- CDATA section injection attack
- Tag injection attack
- External entity injection attack
- Alternate encoding attack and
- Injection via evaluation function

2.1.2 Parameter tampering attack

Injection attacks can also be used to exploit a class of logic vulnerabilities called parameter tampering vulnerabilities [5–7]. In parameter tampering, instead of substituting the input with malicious code, an attacker substitutes it with an invalid input. The difference between the two is subtle. A malicious input in the case of injection attacks could potentially affect the database or reveal sensitive data. On the other hand, parameter tampering does not, for the most part, cause information leakage. It attempts to break the intended logic flow of the web application and gain some benefits. For example, consider an online shopping site. Suppose a user has selected two items $item1$ and $item2$. The application has a field called quantity where a user can enter the quantity of the items he wants. Ideally, the quantity values should be positive integers, as enforced commonly by client-side scripts. However, client-side scripts can be easily bypassed, and a malicious user can insert a negative value for a quantity. Unless this is checked at the server side, this can result in some serious monetary damage for the company. If the attacker enters a positive quantity for $item1$ and a negative quantity say -4 for $item2$, then he effectively secures himself a discount. This is a simple example of parameter tampering. This weakness is referred using *CWE-472*[1] (External Control of Assumed-Immutable Web Parameter). More sophisticated attacks may be launched by capturing all parameters and tampering with their values.

3 Related work

This section describes the work done so far for identifying injection and parameter tampering vulnerabilities/attacks that target the database and server side of the web application.

3.1 XML injection

Extensive literature is available on detection and prevention of SQL injection vulnerabilities/attacks [16–26]. Relatively less amount of research has been done on XML injection vulnerability detection and attack prevention. A detailed classification of the existing approaches can be found in the literature [14,27–31].

3.1.1 Secure programming

Works such as [32–34] emphasize on following secure programming practices to prevent injection attacks. The secure coding standards include pre-compilation of queries, parameterizing queries, escaping the special characters in the input, validation of user-supplied input, and embedding grammar of the query language into that of the host language (e.g., Java, C#, and VB). However, this approach imparts overhead on programmers as they have to learn and annotate the security paradigms during construction.

3.1.2 Static analysis

Mitropoulos et al. [35,36] developed a knowledge-based framework to prevent a broad class of injection attacks. During training phase, insecure code statements are identified and stored using unique signatures for differentiating normal and abnormal executions. During runtime, the framework checks and blocks the code statements that do not conform with the standards of the learnt model. However, the disadvantage with this technique is that it requires a new training to reassign the identifiers if there is any change in source code. Rosa et al. [37] presented an XML injection strategy-based detection system for mitigating zero-day attacks using static analysis approach.

3.1.3 Dynamic analysis and penetration testing

A signature-based framework Sign-WS was developed by Antunes and Vieira [38] to identify injection vulnerabilities in web services. The framework employs penetration testing and interface monitoring for detection of the attack signatures. Signature-based approach cannot detect new unknown attacks, even if they have only small variations from a known payload [37].

3.1.4 Query model-based approach

The proposed prototype follows a query model-based approach, and hence, this subsection emphasizes on the literature that employs query model-based approach for protecting web applications from SQL/XML injection. A number of query model-based approaches exist for preventing SQL injection attacks in web applications [17,18,20,22,24,25]. Laranjeiro et al. [39] formulated an approach to detect and prevent SQL and XPath injection attacks in web services by combining static code analysis and penetration testing. Antunes et al. [40] utilized aspect-oriented programming

[1] https://cwe.mitre.org/data/definitions/472.html

(AOP) for identifying injection vulnerabilities in web services environment. AOP intercepts all the calls to API methods executing SQL commands. The structure of SQL/X-Path commands issued in the presence of attacks is compared against the commands that are previously learnt when running the application in the absence of attacks for discovering vulnerabilities. An approach similar to the methodology developed by Antunes et al. [40] was proposed by Asmawi et al. [41] for preventing XPath injection attacks in a web services environment.

The query model-based approach fails to detect attacks in the case of dynamic variation in the structure of the query based on conditional input [25]. Most of the existing literature that adopts query model-based approach focuses on SQL injection attack prevention and requires customization of the source code which needs to be executed for prevention of attacks. SQLProb [24] employs a proxy server for intercepting the queries which is a overhead at the server side of the application. The proposed work is relevant to Sania [22]. Sania requires the developer to generate valid HTTP requests manually, generates predefined set of attack requests, and employs two proxies: an HTTP proxy to intercept HTTP requests/responses, and a SQL proxy to capture SQL queries. Utilization of two proxies is a overhead in Sania and is capable of detecting SQL injection vulnerabilities alone. In contrast, our approach is fully automated, generates attack strings based on regular expressions and set of parameter constraints, does not make use of any proxy, does not involve any manual intervention, and detects injection vulnerabilities leading to both XQuery injection and parameter tampering attacks. This article follows a query model-based approach entirely based on black-box fuzzing for detection of injection vulnerabilities. Unlike the work by Antunes et al. [40] which focuses on web services, this work focuses on the identification of vulnerabilities in web applications using native XML databases.

Tools that are available in the commercial market and in open source for identification of XML injection vulnerabilities cover only certain type of XML injection attacks. XCat [42] identifies blind XPath injection vulnerabilities in web applications, and WebCruiser [43] detects only tautology injection attacks and cannot detect other types of XML injection attacks. XMLMao [44] is another tool used for exploiting XML injection vulnerabilities in web applications. The limitation of the tool is that the user has to manually enter attack strings in input fields. The vulnerability scanners such as Acunetix [45], Arachni [46], Wapiti [47], and W3af [48] identify XPath injection vulnerabilities in web applications that involve XML documents [49]. Each of the existing tools covers only a specific type of vulnerability, and none of the existing tools is capable of detecting XML vulnerabilities in web applications driven by native XML databases. Hence, there is a demand for a system that concentrates on identification of different types of XQuery injection vulnerabilities in web applications driven by native XML databases.

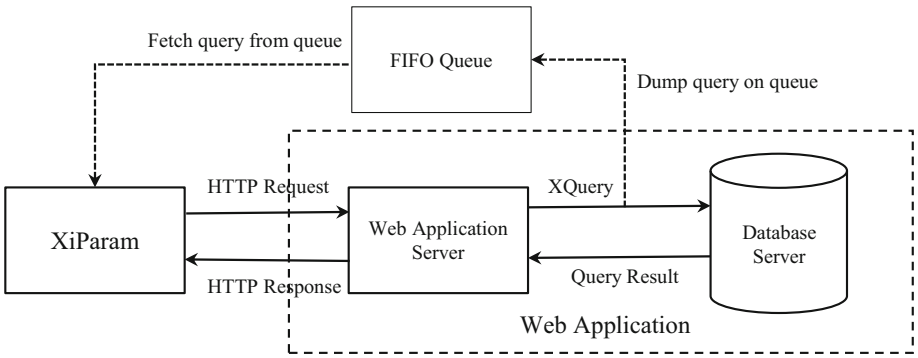
3.2 Parameter tampering

This subsection discusses the approaches available in the literature for detection of parameter tampering vulnerabilities/attacks. Bisht et al. [5,6] devised two tools, namely NoTamper [5] and WAPTEC [6], for identification of parameter tampering vulnerabilities. NoTamper uses a black-box approach, while WAPTEC uses a white-box approach for detection. Both the prototypes examine the source code available at the client side of the application to determine the anticipated behavior of the application. The prototypes extract the restraints placed on user-supplied input from the client-side code, which are used as a reference for the expected behavior of the application at the server side. The restraints imposed on the user input are violated, and the response obtained is observed for identifying vulnerabilities. Compared to NoTamper, WAPTEC [6] takes into consideration the server-side code together with the database schema expressed in MySQL for detecting vulnerabilities. A black-box approach is proposed by Mouelhi et al. [50] for safeguarding the web application against malicious users bypassing the sanitization functions available at the client side. Alkhalaf et al. [51] implemented a prototype named ViewPoints, which follows an approach similar to WAPTEC. ViewPoints employs differential string analysis to identify the inconsistencies existing between the input validation functions employed at the client side and server side of the application. TamperProof [7] can be used to protect web applications from parameter tampering attacks. It is an online defense deployed between the client and server. TamperProof imparts an identifier known as patchID to each web form rendered from the server, which is used for verifying whether the requests submitted by the application are legitimate requests. The aforementioned works identify vulnerabilities that arise due to restrictions on user input which exist at client side but missed at server side of the application. Balduzzi et al. [52] proposed a prototype PAPAS for identification of HTTP parameter pollution (HPP) vulnerabilities. HPP vulnerabilities allow a malicious user to inject a parameter with a value that masks the existing value of the parameter.

4 Proposed approach

The primary objective of this work is to propose an automated approach and develop a prototype system XiParam for detecting XQuery injection vulnerabilities and parameter tampering vulnerabilities in native XML database-driven web applications. The prototype XiParam uses a black-box fuzzer for discovering vulnerabilities. Black-box fuzzing

Fig. 1 System architecture



ensures that the prototype does not require the availability of source code of the application. The prototype XiParam operates in two phases: training and testing phases.

Training phase: In training phase, the application under test is crawled starting from the seed URL provided by the tester. The forms and form fields through which the user interacts with the application are identified and stored in the database. These form fields are considered as the points of injection. The HTTP response obtained is also captured and stored. An HTML/JavaScript analyzer is employed to analyze the HTTP response for extraction of constraints involved if any on the parameters flowing in the HTTP request. The extracted constraints are taken into account for populating injectable points with valid input, and the corresponding web form is submitted to the web application. The HTTP requests submitted to the application are captured using a proxy server, and the queries submitted to the database server are intercepted and are used for constructing legitimate query models.

Testing phase: The injectable points identified during training phase are populated with attack strings that are generated based on the regular expressions confined to different types of XQuery injection attacks and the extracted constraints, and the corresponding web form is submitted to the application. The queries that are processed successfully by the server are intercepted and are used for constructing illegitimate query models. The illegitimate query models are compared against their respective legitimate query models for identifying vulnerabilities. Vulnerabilities are reported when there is a mismatch between the generated query models. In case the query models obtained are the same, then the HTTP response obtained is compared with the learnt response for identification of parameter tampering vulnerabilities.

Seed URLs taken as input identify the starting points from which the web application under test needs to be crawled for finding the points of injection. Additionally, authentication information is taken as user input to enable the crawler to visit pages accessible only to authenticated users. The output is the list of forms that are vulnerable to XQuery injection vulnerabilities and parameter tampering vulnerabilities, the type of attack which can be executed by exploiting this vulnerability, and the location of the vulnerability existing in the web application under test. The vulnerable fields identified would help web application developers to identify and fix flaws in the existing web application so that it can be made immune to XML injection and parameter tampering attacks.

4.1 System architecture

Figure 1 represents the modification made to the basic architecture of the web application to implement the proposed approach as in [13]. The proposed approach is based on the query model and hence requires access to the queries passed between the server and the database. As part of implementation, the database driver files specific to the native XML database on the test application are modified to capture successful queries. The captured queries are placed on a FIFO queue using Redis [53] server, which is a data structure server. The queries are later used for construction of query models. The implementation details are explained in detail in the following subsections.

4.2 Training phase

The training phase consists of three major components for learning the behavior of the application: (i) crawler, (ii) query model constructor, and (iii) HTML/JavaScript analyzer. The prototype takes seed URL and authentication information of a valid user as inputs and produces a list of form fields, set of parameter constraints, and legitimate query models and HTTP responses as outputs. The outputs of the training phase are used as input by the prototype in the testing phase. The crawler component takes the seed URL and authentication information as input and submits HTTP requests to the web application by following the hyperlinks available in web pages. The HTTP response obtained is stored and analyzed for extraction of the form fields available in the web pages. The HTTP response is also analyzed by the HTML/JavaScript analyzer for extraction of constraints that are placed on the parameters flowing in the HTTP request. The

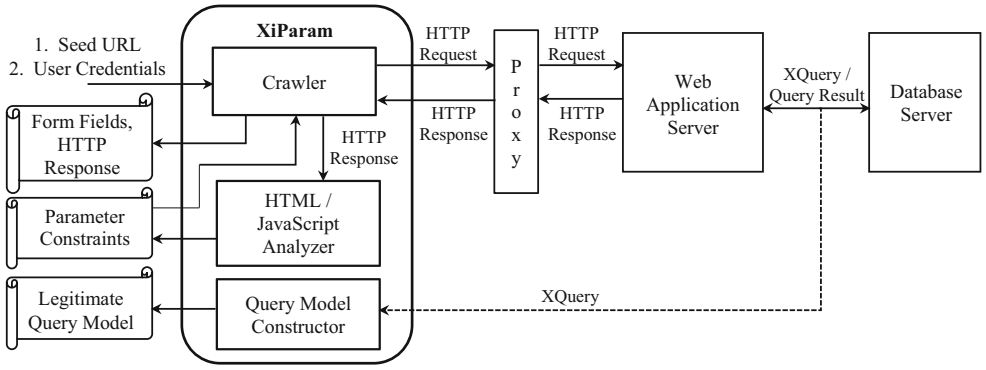


Fig. 2 XiParam: training phase

XQueries that are executed by the database on submission of the HTTP requests are intercepted and are used for construction of legitimate query models which are later used for comparison with the query models generated during attacks. Figure 2 represents the operation of the prototype in training phase.

4.2.1 Crawler

A context-aware crawler is employed to navigate through the application. Apart from following the hyperlinks on each web page to browse through the entire application, the crawler also records the web forms encountered on each web page. Each form is filled with appropriate values and submitted. These form fields are the vulnerable fields, which are used as points of attack string injection. Crawling starts from the seed URL and only proceeds to those links which are in the same domain as the seed URL. The crawler crawls the web application twice, once without session authentication and once with session authentication, i.e., as a registered user.

The crawler should cover all the URLs ideally so as to not miss out on any forms which are potentially vulnerable. The crawler makes use of the standard HTML page processing libraries for extracting hyperlinks in a web page. The crawler looks for the attributes such as *href, src,* and *action* in the HTML content, and JavaScript events such as *window.open, window.location, .load, .location.assign, .href, .action,* and *.src* as well for identifying the URLs in a web page. The developed web crawler does not index web pages like other existing open-source crawlers WebSPHINX [54] and JSpider [55] because indexing is a performance intensive task and is not a feature required for our work. Hence, we developed a crawler which solely performs the task of identifying forms in the web application. A few points to be noted regarding the coverage property of the crawler are:

- The crawler supports type enhanced form submission, i.e., the restrictions imposed on input fields in the web application are taken care by the current crawler. For example, an application may have an input field of type integer, say *Marks* that can hold values between 0 and 100. The crawler takes care of these restrictions and hence prevents internal server errors caused due to erroneous submission of values.
- The major objective of the crawler is to identify form fields that are injectable, and hence, this crawler crawls only hyperlinks in HTML, but not other resources such as documents and images in HTML.
- The crawler does not consider AJAX-based form submissions for identifying injectable points.
- A self-regulating or politeness feature can be added to the crawler so that it can be utilized on real-world web applications.

The forms captured during crawling phase are populated with valid input strings and submitted to the web application. The constraints extracted from the web page using the HTML/JavaScript analyzer helps the crawler in populating the form fields with valid input. This helps to reduce the number of pages missed out while crawling and hence decreases the number of false negatives. The queries executed on this form submission are intercepted and modeled. This model is stored as the learnt model for each query.

4.2.2 Query model constructor

The query models are constructed by replacing the keywords and special characters associated with an XQuery with the following strings.

XQuery keywords are replaced with *KEYWORD*, tags (i.e., <,>) are replaced with *OPEN_TAG* and *CLOSE_TAG*, *doc(...)* is replaced with *DOCPATH*, special characters such as round parentheses (i.e., (,)), comma, and semicolon are replaced with *OPEN_BRACE* and *CLOSE_BRACE*, *COMMA* and *SEMI_COLON*, respectively, *xx/xx/$xxx* with *VARPATH*, constants with *CONSTANT*, *$xxx* with *VARIABLE*, literals with *LITERAL*, operators with operator description such as = with *EQUALS*, and so on.

For example, consider a vulnerable web application using non-parameterized query:

```
"insert node <title>'" + title + "'</
    title> as last into doc('postdb')"
```

Assuming the input submitted by the user for the variable *title* is *Title1*, then the XQuery submitted to the XML database is

```
insert node <title>'Title1'</title> as
    last into doc('postdb')
```

The query model constructed for the above XQuery is KEYWORD KEYWORD OPEN_TAG LITERAL CLOSE _TAG KEYWORD KEYWORD KEYWORD DOCPATH.

The learning phase models the expected behavior of the application. Query models are formed for all the queries that are executed successfully in the XML database on submission of forms identified in the crawling phase. In the discovery phase, the legitimate query models obtained in the learning phase are validated against the query models formed on submission of the same form with different attack strings generated from the attack grammar. Algorithm 1 presents the pseudocode for training phase involving query model construction and constraint extraction.

Algorithm 1 Algorithm for training phase

Input: forms_list, response
1: //Case 1: Constraint Extraction
2: constraints = HTML_JSANALYZER(response);
3: constraints_list.add(constraints);
4: **for** each f in forms_list **do**
5: $data \leftarrow \{\}$;
6: $input \leftarrow extractInputFields(f.content)$;
7: **for** each i in input **do**
8: $data[i] \leftarrow generate\_valid\_value(constraints[i])$;
9: **end for**
10: $payload \leftarrow data$;
11: $response \leftarrow submitHTTPRequest(URL, payload)$;
12: //Case 2: Query Model Construction
13: $learnt\_models \leftarrow get\_query\_model()$;
14: **end for**

4.2.3 HTML/JavaScript analyzer

In the learning phase for parameter manipulation, we observe the parameters flowing between web pages by analyzing the HTTP requests and responses. Each web request expects a certain number and type of parameters that get passed with it. Each parameter has a name and a value, and the value may have some specific constraints it must satisfy. For example, the value of the field *quantity* in an e-commerce portal must be nonnegative. To identify the constraints imposed on parameters in the web application, we leverage the idea of developing an HTML/JavaScript analyzer from NoTamper [5].

Algorithm 2 Algorithm for HTML/JavaScript analyzer

1: **function** HTML_JSANALYZER(HTTP_Response)
2: $ScriptTags[] \leftarrow$ Script tags from the HTTP Response;
3: $InputFields[] \leftarrow$ HTML form input elements from the Response;
4: $constraints \leftarrow [\ ]$;
5: **for** $function$ in $ScriptTags[]$ **do**
6: $function\_list.add(function)$;
7: **end for**
8: //Mapping parameter constraints against each input field;
9: **for** $input$ in $InputFields[]$ **do**
10: **for** $function$ in $function\_list[]$ **do**
11: $statements[] \leftarrow$ function.split("\n");
12: **for** $statement$ in $statements[]$ **do**
13: **if** $statement$ contains $input$ **then**
14: **if** $statement$ contains 'if(' **then**
15: $return\_type$ = return type of function;
16: $Constraint[input] =$ PROCESSIFCONDITION($return\_type$);
17: $constraints.add(Constraint[input])$;
18: **else if** $statement$ contains '==' or '<=' or '>=' or '<' or '>' or '! =' **then**
19: $return\_type$ = return type of function;
20: $Constraint[input] =$ PROCESSMULTIIFCONDITION($return\_type$);
21: $constraints.add(Constraint[input])$;
22: **else if** $statement$ contains '=' **then**
23: Var_name = Local variable to which input parameter is assigned;
24: //Map the input parameters and local JS Variables;
25: $Constraint[input] =$ PROCESSVARIABLE(Var_name);
26: $constraints.add(Constraint[input])$;
27: **end if**
28: **end if**
29: **end for**
30: **end for**
31: **end for**
32: **end function**

HTML/JavaScript analyzer is developed for analyzing the web page to extract constraints involved (if any) in each field of the form. The term 'constraints' gives information about the possible values that an input field in the form can hold, form fields which are mandatory, etc. The validation code in the JavaScript is also taken into account for building the constraints. The validation routine gets executed (1) on submission of a web form and (2) in event handlers each time when the user enters or modifies data on the form. If any web application relies only on client-side validation of input parameters and does not enforce the same constraints at server code level, then a malicious user can bypass the validation check by disabling JavaScript and thus violate the business logic. To ensure that the constraint on any parameter is imposed in the client side as well as the server side, JavaScript routines are considered for building the constraints.

The analyzer takes the URL of the web page, web page content, and inline JavaScript code as input, analyzes the

source code, and displays the input parameters with the set of constraints (if any) as output. The JavaScript analyzer processes a given web page to produce a list of input field elements (textboxes, radio buttons, checkboxes, etc.). It also filters out JavaScript code from the given web page and places them in a list. The JavaScript code is scanned for *if-conditions* for extracting constraints on the parameters. The component, input element locator, identifies the input element that is being processed by the current JavaScript function. *Multiple-if* statements, and return type of the function are also considered so as to avoid any errors in the extraction of constraints. This information would be further needed to carry out parameter violation attacks. Algorithm 2 gives the algorithm for HTML/JavaScript analyzer.

Constraints extracted from HTML/JavaScript analyzer: The JavaScript analyzer extracts constraints on the parameters flowing between the web pages from the client-side code. The different types of constraints extracted are as follows: (i) data type constraint, (ii) value constraint, and (iii) length constraint. Data type constraint on a parameter states the type of value that should be provided as input by the user. For example, an input field may prompt the user to enter only integer values. Value constraint refers to the value that an input field can hold. For example, an input field such as password in a registration page may require a user to enter value that contains a minimum of one uppercase alphabet, one lowercase alphabet, one special character, and one integer. This constraint is extracted and represented in the form of a regular expression against the input field. Length constraint refers to the length of the field that an input can hold. For example, an input field that takes the name of the user may have a constraint stating that the length of the name should be less than 25.

4.3 Testing phase

The testing phase involves three major components for identifying vulnerabilities in the application: (i) attack generator, (ii) query model constructor, and (iii) detector. Figure 3 represents the operation of the prototype in testing phase. Algorithm 3 presents the working mechanism of the testing phase.

4.3.1 Attack generator

The attack generator module generates two kinds of attack strings: (i) Malicious strings that contain keywords related to XQueries, and (ii) Invalid input strings that do not contain keywords related to XQueries but violate the constraints extracted by the HTML/JavaScript analyzer. The former is used for identifying XQuery injection vulnerabilities, while the latter is used for identifying parameter tampering vulnerabilities.

Algorithm 3 Algorithm for testing

```
Input: learnt_models, input_fields, learnt_response, constraints_list
Output: List of Vulnerabilities
//Case 1: XQuery Injection Attack Generation and Discovery
attack_strings =getRegularExpr();
for each i in input_fields do
    for each a in attack_strings do
        payload = generate_attacks(i,attack_strings[a]);
        status_code = getStatusCode(submitHTTPRequest(URL,
payload));
        if status_code='200' then
            attack_query_model = get_query_model(q);
            learnt_model = learnt_models[query_cid];
            if learnt_model ≠ attack_query_model then
                report_vulnerability();
            end if
        else if (status_code[0]='5') or (status_code[0]='4') then
            report_internal_server_error();
        end if
    end for
end for
//Case 2: Parameter Tampering Attack Generation and Discovery
//Call for Algorithm 4
PARAMETER_ATTACK(constraints_list);
```

XQuery injection attack generation

To detect XQuery injection, malicious input is provided in the form fields identified in the crawling phase. Malicious strings are generated using regular expressions developed for detecting XQuery injection. The attack strings are constructed by taking into account the different types of XML injections identified by the security consortium OWASP, and the ones stated by Palsetia et al. [13].

The attack language of strings is framed based on regular expressions representing each type of injection attack as specified in Sect. 2.1. In this work, a regular expression has been framed to generate malicious strings representing each of these types of attacks. By combining these regular expressions, the attack language has been formed. Hence, the proposed set of regular expressions enables the identification of all the types of XML injection attacks listed in Sect. 2.1. For example, the regular expression for generating tautology attack string is: $(A\text{-}Za\text{-}z0\text{-}9)^{+}$*' or 'a'= 'a' or* '$(A\text{-}Za\text{-}z0\text{-}9)^{+}$. A sample string generated by this regular expression is *xyz' or 'a'= 'a' or 'aaa*. The regular expression for comment injection attack string is: $(A\text{-}Za\text{-}z0\text{-}9)^{+}$ ⟨!--. A sample string generated by this regular expression is *xyz678* ⟨!--. Similarly, regular expressions can be formed for each type of attack string. A combination of these regular expressions describes the attack language.

The vulnerable form fields identified in the web pages are populated with these attack strings, and a request is submitted to the web server. After receiving a successful response from

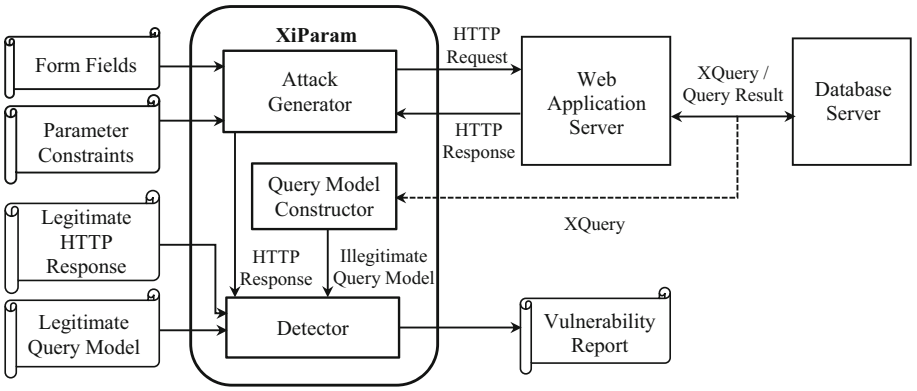


Fig. 3 XiParam: testing phase

the web server, the query generated is captured and a query model is constructed.

Parameter tampering attack generation

The constraints extracted from the HTML and JavaScript code are stored in a database. The next step is the generation of attack strings. It uses the learnt set of constraints for generating the attack vectors for each type of vulnerability. During this phase, the attacks are generated by submitting the form fields either with input violating the constraints imposed on the HTTP request parameters, or including parameters that are not supposed to be present in the web request. This helps us in identifying the attacks related to parameter tampering problems. The response obtained for the invalid input is gathered. Using the HTML response comparator, the valid HTTP response stored in the database is validated against the invalid web response. If similarity exists between the responses, then vulnerability is reported for the form fields submitted with invalid values. The attack vectors generated for identifying parameter tampering vulnerabilities are as follows and the details are described in Algorithm 4.

- Modifying the value of the parameters in such a way that the constraints imposed on the parameter are violated. Each type of constraints imposed on the input field such as data type constraint, value constraint, and length constraint is violated for identifying vulnerabilities. For example, consider a form field which is supposed to have a positive integer value. In such case, attack requests are submitted twice with the following malicious input: (i) a negative value ($price = -100$) and (ii) a string value ($price =$ 'abc'). Thus, each constraint extracted is violated for identifying vulnerabilities.
- Including a parameter which is not supposed to be there in the HTTP request for a web page. The constraints extracted from the JavaScript analyzer have information regarding the parameters that should flow from one page to another page, as well as the parameter which is substituted with a value only for a user with particular privilege. To violate such constraint, the parameter is substituted with a value for a user with lower privilege. For example, consider a parameter $discount = 20\%$, which is supposed to be there only for regular customers. If the application is vulnerable, then the parameter can be included in the HTTP request of a guest user of the application consequently making him avail the discount.

Algorithm 4 Algorithm for parameter tampering attack generation

Input: Set of Input Constraints

```
for webpage in webpage_list do
    parameter_values ="";
    new_parameter ="";
    for form in webpage do
        //Case 1: Modifying the values of parameters
        for input in form do
            parameter_value = negateCostraint(constraint[input]);
            attack_Response = SubmitHTTPRequest(url, parameter_value);
            learnt_response = getValidResponse(url);
            if learnt_response = attack_Response then
                report_vulnerability();
            end if
        end for
        //Case 2: Inclusion of new parameter
        new_parameter.append(parameter);
        attack_Response = SubmitHTTPRequest(url, new_parameter);
        learnt_response = getValidResponse(url);
        if learnt_response = attack_Response then
            report_vulnerability();
        end if
    end for
end for
```

The response obtained during attack generation is stored and compared with the response obtained during normal execution of the application. When a success response is received for attack vector, then vulnerability is reported. The identified vulnerabilities are recorded in the report which gives information about the web page that is vulnerable, the vulnerability that is exploited for launching the attack and the type of attack launched. This report aids the developers to eliminate the vulnerabilities existing in the application before the application goes into production.

4.3.2 Query model constructor

The forms captured during crawling phase are populated with attack strings and submitted to the web application. The queries executed on this form submission are modeled by using the same method as described in the training phase. This model is compared against the legitimate query model for each query.

4.3.3 Detector

The detector identifies the type of vulnerability exploited for unveiling the injection attack. The detector first compares the query model generated during the attack against the appropriate query model generated during training phase. If the query models are dissimilar, then an XQuery injection vulnerability is reported. If the query models are similar, then the HTTP responses obtained during testing and training phases are compared. If the responses are dissimilar, then a parameter tampering vulnerability is reported.

XQuery injection vulnerability detection

The illegitimate query model is validated against the corresponding legitimate query model for the identification of vulnerabilities. If the query models are dissimilar, then the field in the form is a vulnerable field and the type of vulnerability is reported. Considering the example discussed in Sect. 4.2.2, the leaf nodes of the node_structure clause for legitimate and illegitimate input strings are *OEPN_TAG LITERAL CLOSE_TAG* and *OEPN_TAG LITERAL META CLOSE_TAG*, respectively. The leaf nodes of the learnt and discovered query models obtained are dissimilar, and hence, a vulnerability is reported. In case the leaf nodes of the node_structure clause for legitimate and illegitimate input strings are similar, then no vulnerability is reported.

Parameter tampering vulnerability detection

During the testing phase, the expected HTML response for each attack request is stored. During attack generation, we attempt to violate certain constraints and reach particular states. We then compare the HTML response received during our attack against the response obtained during our training phase. If the two responses match, then a parameter tampering vulnerability is reported.

The testing phase performs rigorous testing. Accurate identification of vulnerabilities is done in this phase by attacking all the form fields identified in the crawling phase, using the regular expressions and the set of parameter constraints extracted for generating attack strings.

5 Evaluation

This section describes the applications considered for evaluating the prototype and the discusses the effectiveness of the results. The prototype is implemented using Django [56] web framework with Python and uses PostgreSQL [57] as the database. Redis [53], a data structure server, is used for storing the intercepted queries. The prototype requires modification to BaseX client files based on the web technology used by the web application under test for capturing the queries and is designed to work only for web applications that use BaseX [58] as the XML database. The prototype is designed to run on both Linux and Windows operating systems.

5.1 TestBed applications

The proposed approach is evaluated using the test suite provided by Halfond and Orso [17]. The applications in the test suite use a relational database at the backend. For evaluating our approach, the backend of these applications is modified to a popular native XML database, BaseX. All the SQL queries are subsequently replaced with XQueries. The applications used for test include BookStore, Classifieds, Employee Directory, and Events. To identify parameter tampering vulnerabilities, these applications are instrumented with JavaScript code to enforce constraints on various parameters flowing between the web pages. The details of the test applications are given in Table 1. Columns 1–3 give the details about the test application. Column 4 states the number of locations that issue XQueries (hotspots) in the application.

The prototype intended for detection of XQuery injection vulnerabilities works only for web applications using BaseX as the native XML database, since database driver file modification is required for web applications driven by other native XML databases.

5.2 Results

The prototype is evaluated on the test applications listed in Table 1 to measure the effectiveness of the prototype against detection of XQuery injection and parameter tampering vulnerabilities. The evaluation results are discussed below.

Tables 2 and 3 summarize the experimental findings. The number of attack requests submitted to the application, number of attacks that were successful, number of vulnerabilities existing in the application, and number of vulnerabilities detected by the prototype for both XQuery injection and parameter tampering are listed in Table 2. Table 3 presents the effectiveness of the prototype against detection of both the types of vulnerabilities. It lists the number of vulnerable forms existing in the application, number of vulnerable

Table 1 Test application details

Application	Description	Web technology	# Hotspots
BookStore	Online bookstore	JSP	71
Classifieds	Online management system for classifieds	JSP	34
Employee directory	Online employee directory	JSP	23
Events	Event tracking system	JSP	31

Table 2 Number of attack requests and vulnerabilities reported by XiParam

Application	#Attack requests		#Successful attacks		#Vulnerabilities (E)		#Vulnerabilities (D)	
	#XQI	#PT	#XQI	#PT	#XQI	#PT	#XQI	#PT
BookStore	726	113	235	64	156	64	153	64
Classifieds	528	86	125	86	106	86	89	86
Employee directory	286	42	111	42	60	42	68	42
Events	396	39	157	39	87	39	87	39
Total	1936	280	628	231	409	231	397	231

E existing, D detected, XQI XQuery injection, PT parameter tampering

Table 3 Effectiveness of XiParam

Application	#Forms	#Vulnerable forms (E)		#Vulnerable forms (D)		#False positives		#False negatives	
		#XQI	#PT	#XQI	#PT	#XQI	#PT	#XQI	#PT
BookStore	32	19	5	17	5	1	0	2	0
Classifieds	20	14	16	12	16	0	0	2	0
Employee directory	9	7	8	7	8	1	0	0	0
Events	12	10	11	10	11	0	0	0	0
Total	73	50	40	46	40	2	0	4	0

E existing, D detected, XQI XQuery injection, PT parameter tampering

forms detected by the prototype, and the number of false positives and false negatives reported by the prototype. Table 4 gives the different types of constraints extracted by the HTML/JavaScript analyzer for the test applications, number of attack requests submitted to the application, and the number of parameter tampering vulnerabilities identified in the application.

Table 3 shows that for applications BookStore and Classifieds, two forms that are vulnerable to XQuery injection are missed from being identified. The forms are missed because of the validation placed on few parameters on the server side but missed at the client side. The developed context-aware crawler submits values for the parameters based on the client-side restrictions on it. If the constraints are placed only at the server side and missed at the client side, then the constraints extracted from the HTML/JavaScript analyzer will not contain this restriction, because of which both legitimate and illegitimate requests fail for both the applications. In the case of applications Events and Employee Directory, all the existing vulnerable forms are identified by the prototype.

The results show that the prototype detects maximum number of vulnerabilities present in the test applications with few false positives. The vulnerabilities detected by the prototype in applications like BookStore and Employee Directory are greater than the existing number of vulnerabilities due to the presence of the false positives. It can be inferred from the table that in case of parameter tampering vulnerabilities, the prototype did not report any false positives and false negatives. The detection rate of the prototype with respect to XQuery injection and parameter tampering is 92% and 100%, respectively. On the whole, the detection rate of the prototype is found to be 95.5%. The accuracy of the results obtained is 93.5%.

A vulnerability report is generated that specifies the number of URLs processed and the number of vulnerabilities existing in the application. The number of vulnerabilities found in the application and the related details are reported under each category of vulnerability. The vulnerability details include the URL of the web page and the specific form where the vulnerability is found along with the attack

Table 4 Number of constraints extracted and number of parameter tampering vulnerabilities reported

Application	# Forms	# Constraints extracted				# Vulnerabilities (Detected)
		# Length constraints	# Data constraints	# Value constraints	# Total constraints	
BookStore	32	16	48	49	113	64
Classifieds	20	12	18	14	44	86
Employee directory	9	0	8	16	24	42
Events	12	0	12	8	20	39

Table 5 Details of a vulnerability provided in vulnerability report

S. no.	Vulnerability details	
1.	Vulnerable web page:	http://localhost:8080/BookStore/Registration.jsp
	Vulnerable form name:	Registration
	Target web page:	http://localhost:8080/BookStore/Registration.jsp
	Attack injected:	Injection via alternate encoding
	Attack string:	{convert:binary-to-string(xs:hexBinary("48656c6c6f576f726c64"))}
	Malicious query:	for $res in doc('bookstorexmldb_scan')//bookstore/members/member where $res/member_login='{convert:binary-to-string(xs:hexBinary("48656c6c6f576f726c64"))}' return $res
2.	Vulnerable web page:	http://localhost:8080/bookstore/ShoppingCart.jsp
	Vulnerable form name:	Purchase
	Vulnerable form field:	Quantity
	Target web page:	http://localhost:8080/bookstore/ShoppingCart.jsp
	Attack injected:	Parameter tampering attack
	Constraint violated:	Quantity$>$ 10
	Type of constraint:	Value constraint
	Malicious String:	-10
3.	Vulnerable web page:	http://localhost:8080/bookstore/Registration.jsp
	Vulnerable form name:	Registration
	Vulnerable form field:	Name
	Target web page:	http://localhost:8080/bookstore/Registration.jsp
	Attack injected:	Parameter tampering attack
	Constraint violated:	Length of the name field $\leq$15
	Type of constraint:	Value constraint
	Malicious String:	'abcdefghijklmnopqr' (Length $>$ 15)

string used to detect it as given in Table 5. This information will be useful for the web application programmer to identify the point of injection and the type of vulnerability that needs to be addressed.

5.3 Effectiveness of the prototype

The positive aspects of the developed prototype as compared to the existing approaches are as follows:

- A context-aware crawler is employed for learning through the application. This improves the coverage property of the crawler and hence reduces the number of false negatives.
- A client-side code analyzer is developed for extraction of restrictions imposed on the input fields available in the web forms of the application. The extracted restrictions were used effectively by the crawler for generating valid values for the input fields, which in turn has prevented missing of few web pages during crawling.
- The prototype is fully automated and does not require any manual intervention for detection of vulnerabilities.

– The prototype is capable of detecting both XQuery injection and parameter tampering vulnerabilities in web applications by using a single framework.

The proposed approach reports false negatives in some cases. One of the cases is when a vulnerable web page is missed out by the crawler. Consider a web page named *"cart items"*. Supposing this web page is visited by the crawler before adding items to the cart, there would not be any hyperlinks leading to the subsequent pages, and hence, the subsequent pages will not be visited by the crawler. In case of the subsequent pages being vulnerable, those web pages would be missed from being identified as they are unvisited by the crawler. In some situations, the crawler may not be able to generate valid query models for a web page. This can occur because the input supplied does not match the server specifications in some way. In such a case, if an attack string is able to generate an invalid query model, then the prototype will not report this vulnerability.

Our approach results in false positives in the following scenario: When query models of two different queries are compared. For example, consider a scenario where an HTTP request with valid values for the HTTP parameters executes two queries. When proper sanitization techniques are employed by the application, a malicious input does not cause an attack but rather redirects the user to a different page which also by chance executes two queries. In this case, the queries executed during valid request submission and invalid request submission are totally different, resulting in different query models being compared. The prototype reports vulnerabilities as inappropriate query models are compared, resulting in false positives. Two false positives are reported by the developed prototype on the considered test applications for the above-mentioned scenario.

The prototype resulted in two false positives and four false negatives. Two false positives are reported for the applications BookStore and Employee Directory. The reason behind this is the legitimate and illegitimate query models were not compared appropriately as the illegitimate request was not processed successfully. False negatives are reported in applications Bookstore and Classifieds. In case of application BookStore, the web page *ShoppingCart* did not have any products that provide hyperlinks to ShoppingCartRecord page, and hence, the web page *ShoppingCartRecord* is missed by the crawler. Vulnerabilities in three other pages, namely *BookDetail.jsp, BookMaint.jsp* and *OrdersRecord.jsp*, are missed from being identified since they have restrictions on one of the text fields at the server side alone. Similarly, application Classifieds has one page *MyAdRecord.jsp* which has a restriction on one of the text fields and hence is missed from being identified. The false negatives can be eliminated by considering the constraints employed at server side as well. False positives can be eliminated by comparing the HTTP response of the application. Figures 4 and 5 depict the effectiveness of the prototype against detection of XQuery injection and parameter tampering vulnerabilities.

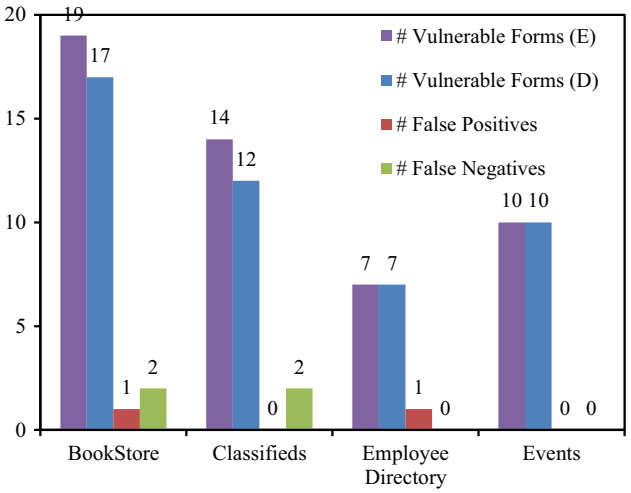


Fig. 4 Effectiveness of the prototype with respect to XQuery injection

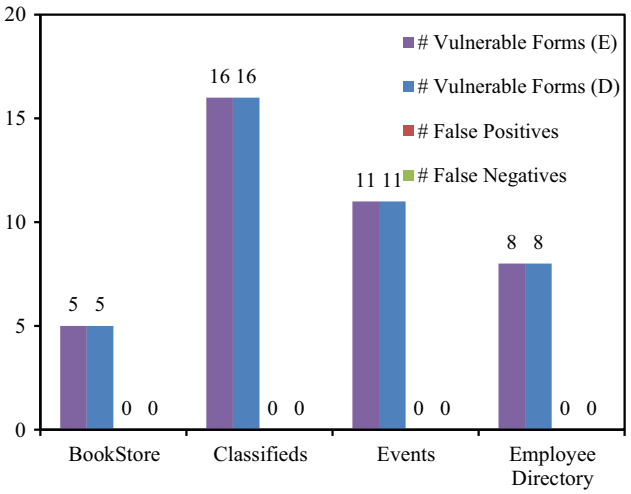


Fig. 5 Effectiveness of the prototype with respect to parameter tampering

The results are related to the specific test applications considered and cannot be generalized to other real-world web applications. Even though the proposed approach does not require source code of the application for identifying vulnerabilities, it is technology dependent as it requires modification to the database driver files for capturing the queries. In addition, in case of parameter tampering vulnerabilities, constraints are extracted from HTML and JavaScript code alone and hence does not work for applications that involve other client-side scripting languages.

6 Conclusion

A prototype for identification of XQuery injection vulnerabilities is developed as XML injection has become a critical vulnerability with the increased use of XML databases by web applications. The prototype employs a context-aware crawler for identifying vulnerable injection points and an HTML/JavaScript analyzer for extraction of constraints imposed on the points of injection. For identifying XQuery injection vulnerabilities, the points of injection are automatically filled with illegitimate strings generated using regular expressions, and the requests are submitted to the web server. The XQueries submitted to the XML database on submission of the illegitimate requests are captured, and query models are constructed. The query models generated during legitimate and illegitimate input submissions are compared for identifying injection points, and vulnerable points of injection are finally reported. In case of parameter tampering vulnerabilities, the set of constraints extracted is violated for generating attacks. The HTTP responses obtained during normal and attack executions are compared to identify vulnerabilities. The prototype has been exhaustively tested on customized benchmark web applications and is found to work effectively with minimum number of false positives and false negatives. It is completely automated and can become an essential tool for ensuring the security of applications driven by native XML databases against injection and parameter tampering attacks.

Acknowledgements This work was supported by the Ministry of Communications and Information Technology, Government of India and is part of the R&D project entitled "Development of Tool for detection of XML-based injection vulnerabilities in web applications," 2014–2016.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Symantec Corporation: Symantec internet security threat report: vol. 19. Symantec Corporation. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us (2014)
2. Foundation, O.: Top 10 2013-top 10. https://www.owasp.org/index.php/Top_10_2013-Top_10 (2013)
3. CWE/SANS top 25 most dangerous software errors. http://www.sans.org/top25-software-errors/ (2011)
4. Gordeychik, S.: Web application security statistics. The Web Application Security Consortium. http://projects.webappsec.org/w/page/13246989/WebApplicationSecurityStatistics (2008)
5. Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., Venkatakrishnan, V.N.: Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10, pp. 607–618. ACM, New York (2010)
6. Bisht, P., Hinrichs, T., Skrupsky, N., Venkatakrishnan, V.N.: Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pp. 575–586. ACM, New York (2011)
7. Skrupsky, N., Bisht, P., Hinrichs, T., Venkatakrishnan, V.N., Zuck, L.: Tamperproof: A server-agnostic defense for parameter tampering attacks on web applications. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy, CODASPY '13, pp. 129–140. ACM, New York (2013)
8. Chaudhri, A., Zicari, R., Rashid, A.: XML Data Management: Native XML and XML Enabled DataBase Systems. Addison-Wesley Longman Publishing Co. Inc, Boston (2003)
9. Liu, Z.H., Murthy, R.: A decade of XML data management: An industrial experience report from oracle. In: IEEE 25th International Conference on Data Engineering, 2009. ICDE '09, pp. 1351–1362 (2009). doi:10.1109/ICDE.2009.18
10. Pavlovic-Lazetic, G.: Native XML databases vs. relational databases in dealing with XML documents. Kragujevac J. Math. **30**, 181–199 (2007)
11. Staken, K.: Introduction to native XML databases. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html (2001)
12. Foundation, O.: Testing for XML injection. https://www.owasp.org/index.php/Testing_for_XML_Injection_OTG-INPVAL-008 (2014)
13. Palsetia, N., Deepa, G., Khan, F.A., Thilagam, P.S., Pais, A.R.: Securing native XML database-driven web applications from XQuery injection vulnerabilities. J. Syst. Softw.**122**, 93–109 (2016). doi:10.1016/j.jss.2016.08.094. http://www.sciencedirect.com/science/article/pii/S0164121216301571
14. Halfond, W., Viegas, J., Orso, A.: A classification of SQL-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, pp. 65–81 (2006)
15. WASC: XQuery injection. http://projects.webappsec.org/w/page/13247006/XQueryInjection (2009)
16. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th International Conference on World Wide Web, pp. 40–52. ACM (2004)
17. Halfond, W.G., Orso, A.: Amnesia: analysis and monitoring for neutralizing SQL-injection attacks. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 174–183. ACM (2005)
18. Buehrer, G., Weide, B.W., Sivilotti, P.A.: Using parse tree validation to prevent SQL injection attacks. In: Proceedings of the 5th International Workshop on Software Engineering and Middleware, pp. 106–113. ACM (2005)
19. Huang, Y.W., Tsai, C.H., Lin, T.P., Huang, S.K., Lee, D., Kuo, S.Y.: A testing framework for web application security assessment. Comput. Netw. **48**(5), 739–761 (2005). Web Security
20. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06, pp. 372–382. ACM, New York (2006)
21. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. USENIX Secur. **6**, 179–192 (2006)
22. Kosuga, Y., Kernel, K., Hanaoka, M., Hishiyama, M., Takahama, Y.: Sania: Syntactic and semantic analysis for automated testing against SQL injection. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, pp. 107–117. IEEE (2007)
23. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pp. 32–41. ACM, New York (2007)

24. Liu, A., Yuan, Y., Wijesekera, D., Stavrou, A.: SQLProb: A proxy-based architecture towards preventing SQL injection attacks. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, pp. 2054–2061. ACM, New York (2009)
25. Bisht, P., Madhusudan, P., Venkatakrishnan, V.: Candid: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Trans. Inf. Syst. Secur. (TISSEC) **13**(2), 14 (2010)
26. Jang, Y.S., Choi, J.Y.: Detecting SQL injection attacks using query result size. Comput. Secur. **44**, 104–118 (2014)
27. Shahriar, H., Zulkernine, M.: Taxonomy and classification of automatic monitoring of program security vulnerability exploitations. J. Syst. Softw. **84**(2), 250–269 (2011)
28. Shahriar, H., Zulkernine, M.: Mitigating program security vulnerabilities: Approaches and challenges. ACM Comput. Surv. **44**(3), 11:1–11:46 (2012)
29. Li, X., Xue, Y.: A survey on server-side approaches to securing web applications. ACM Comput. Surv. **46**(4), 54:1–54:29 (2014)
30. Deepa, G., Thilagam, P.S.: Securing web applications from injection and logic vulnerabilities: approaches and challenges. Inf. Softw. Technol. **74**, 160–180 (2016). doi:10.1016/j.infsof.2016.02.005. http://www.sciencedirect.com/science/article/pii/S0950584916300234
31. Chandrashekhar, R., Mardithaya, M., Thilagam, P.S., Saha, D.: SQL injection attack mechanisms and prevention techniques. In: Advanced Computing, Networking and Security, pp. 524–533. Springer, Berlin (2012)
32. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 3–12. ACM (2007)
33. OWASP: XPath injection. https://www.owasp.org/index.php/XPATH_Injection (2015)
34. Truelove, J., Svoboda, D.: Ids09-j. prevent XPath injection. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=61407250 (2011)
35. Mitropoulos, D., Karakoidas, V., Spinellis, D.: Fortifying applications against XPath injection attacks. In: Proceedings of the 4th Mediterranean Conference on Information Systems (MCIS'09), Athens, Greece, pp. 1169–1179 (2009)
36. Mitropoulos, D., Karakoidas, V., Louridas, P., Spinellis, D.: Countering code injection attacks: a unified approach. Inf. Manag. Comput. Secur. **19**(3), 177–194 (2011)
37. Rosa, T.M., Santin, A.O., Malucelli, A.: Mitigating XML injection 0-day attacks through strategy-based detection systems. IEEE Secur. Priv. **11**(4), 46–53 (2013). doi:10.1109/MSP.2012.83
38. Antunes, N., Vieira, M.: Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. In: IEEE International Conference on Services Computing (SCC), pp. 104–111. IEEE (2011)
39. Laranjeiro, N., Vieira, M., Madeira, H.: Protecting database centric web services against SQL/XPath injection attacks. In: Database and Expert Systems Applications, pp. 271–278. Springer, Berlin (2009)
40. Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H.: Effective detection of SQL/XPath injection vulnerabilities in web services. In: IEEE International Conference on Services Computing, pp. 260–267. IEEE (2009). doi:10.1109/SCC.2009.23
41. Asmawi, A., Affendey, L.S., Udzir, N.I., Mahmod, R.: Model-based system architecture for preventing XPath injection in database-centric web services environment. In: 7th International Computing and Convergence Technology (ICCCT), pp. 621–625. IEEE (2012)
42. Forbes, T.: Exploiting XPath injection vulnerabilities with xcat. http://tomforb.es/exploiting-xpath-injection-vulnerabilities-with-xcat-1 (2014)
43. WebCruiser: Webcruiser-web vulnerability scanner. http://www.ehacking.net/2011/07/webcruiser-web-vulnerability-scanner.html (2011)
44. XMLMao: XMLMao. https://www.soldierx.com/tools/XMLmao (2012)
45. Acunetix: Acunetix. http://www.acunetix.com/ (2014)
46. Laskos, T.: Web application vulnerability scanning framework. http://www.arachni-scanner.com/
47. Wapiti: The web-application vulnerability scanner. http://wapiti.sourceforge.net/ (2013)
48. Riancho, A.: w3af. http://w3af.sourceforge.net (2011)
49. van der Loo, F.: Comparison of penetration testing tools for web applications. Ph.D. thesis, Master thesis, Radboud University Nijmegen, 2011. http://www.ru.nl/publish/pages/578936/frank_van_der_loo_scriptie.pdf (2011)
50. Mouelhi, T., Le Traon, Y., Abgrall, E., Baudry, B., Gombault, S.: Tailored shielding and bypass testing of web applications. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), pp. 210–219 (2011)
51. Alkhalaf, M., Choudhary, S.R., Fazzini, M., Bultan, T., Orso, A., Kruegel, C.: Viewpoints: Differential string analysis for discovering client- and server-side input validation inconsistencies. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pp. 56–66. ACM, New York (2012)
52. Balduzzi, M., Gimenez, C.T., Balzarotti, D., Kirda, E.: Automated discovery of parameter pollution vulnerabilities in web applications. In: Proceedings of the 18th Network and Distributed System Security Symposium, NDSS'11. San Diego (2011)
53. Redis: Redis. http://redis.io/
54. WebSPHINX: WebSPHINX: A personal, customizable web crawler. http://www.cs.cmu.edu/~rcm/websphinx/ (2002)
55. JSpider: Jspider. http://j-spider.sourceforge.net/ (2013)
56. Django: Django-the web framework for perfectionists with deadlines. https://www.djangoproject.com/
57. PostgreSQL: PostgreSQL-the world's most advanced open source database. http://www.postgresql.org/
58. BaseX: Basex-the XML database. http://basex.org/