The Journal of Systems and Software 122 (2016) 93–109

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Securing native XML database-driven web applications from XQuery injection vulnerabilities

Nushafreen Palsetia, G. Deepa*, Furqan Ahmed Khan, P. Santhi Thilagam, Alwyn R. Pais

Department of Computer Science and Engineering, National Institute of Technology Karnataka, Mangalore, India

ARTICLE INFO

Article history:
Received 9 April 2016
Revised 27 June 2016
Accepted 29 August 2016
Available online 30 August 2016

Keywords:
Web application security
Vulnerability scanner
Injection attacks
Fuzz testing
XML injection
XPath injection

ABSTRACT

Database-driven web applications today are XML-based as they handle highly diverse information and favor integration of data with other applications. Web applications have become the most popular way to deliver essential services to customers, and the increasing dependency of individuals on web applications makes them an attractive target for adversaries. The adversaries exploit vulnerabilities in the database-driven applications to craft injection attacks which include SQL, XQuery and XPath injections. A large amount of work has been done on identification of SQL injection vulnerabilities resulting in several tools available for the purpose. However, a limited work has been done so far for the identification of XML injection vulnerabilities and the existing tools only identify XML injection vulnerabilities which could lead to a specific type of attack. Hence, this work proposes a black-box fuzzing approach to detect different types of XQuery injection vulnerabilities in web applications driven by native XML databases. A prototype XQueryFuzzer is developed and tested on various vulnerable applications developed with BaseX as the native XML database. An experimental evaluation demonstrates that the prototype is effective against detection of XQuery injection vulnerabilities. Three new categories of attacks specific to XQuery, but not listed in OWASP are identified during testing.

1. Introduction

Extensible Markup Language (XML) is a data representation that favors integration and interoperability between heterogeneous web applications. The information exchanged between the applications in the form of XML documents can be processed efficiently when they are stored appropriately. These documents are stored in either an extended relational DBMS or a native XML database system (Chaudhri et al., 2003; Liu and Murthy, 2009). XQuery / XPath can be used as a query language for retrieving the data from XML documents.

A Native XML database (NXD) has XML document as its fundamental unit of storage, and defines a logical model based on the content in the XML document (Pavlovic-Lazetic, 2007). NXDs are employed in cases where the data involved do not fit the relational data model, but fit the XML data model. NXDs are generally preferred for applications that hold highly diverse information, involve integration of information from different set of applications, and handle rapidly evolving schemas. NXDs are also preferred for applications that work with a huge set of documents or large-sized documents (e.g., books, web pages, marketing brochures), and involve management of long-running transactions like finance, pharmaceuticals, etc. (Bourret, 2009). Use of relational databases and flat file systems for building such applications results in issues such as scalability and lack of structured queries. These issues can be overcome by using NXDs with XQuery/XPath as the query language for processing. Some of the popular NXDs are BaseX, eXistDB, and MarkLogic.

NXDs find applications in a wide variety of domains such as document management systems, healthcare systems, financial applications, business-to-business transaction records, catalog data, and corporate information portals (Staken, 2001). Real-world business applications that employ NXDs to manage their content are Elsevier Science publishers, Las Vegas Sun publishers (Bourret, 2009), the Tibetan Buddhist Resource Center (TBRC) (Siegel and Retter, 2014), etc. The Tasmanian government websites use NXD for helping users to track legislation. NXDs are used to store various other types of documents such as drug information sheets, contracts, case law, and insurance claims. Commerzbank and Hewlett Packard use NXD for integration of information from a variety of sources to handle financial and business transaction data (Bourret, 2009). Healthcare applications prefer to store electronic health records (EHR) in NXDs for efficient storage and retrieval of

* Corresponding author.
E-mail addresses: nushafreen@gmail.com (N. Palsetia), gdeepabalu@gmail.com (G. Deepa), furqankhan08@gmail.com (F. Ahmed Khan), santhi@nitk.ac.in (P.S. Thilagam), alwyn@nitk.ac.in (A. R. Pais).

http://dx.doi.org/10.1016/j.jss.2016.08.094
0164-1212/

information from the available medical records (i.e., scan reports, prescriptions, etc.) (de la Torre et al., 2011; Lee et al., 2013; Al-Hamdani, 2010).

The existing literature reveals that there is a growing demand towards usage of NXDs in web applications. Even though various XML security standards (W3C; Hirsch) such as XML Encryption, XML Digital Signature, and XML access-control markup language are defined for preserving confidentiality, integrity and access-control mechanisms of XML documents, when NXDs are used at the backend, any vulnerability in the source code of the application may allow an adversary to perform unwanted actions resulting in extraction/modification of information from/in the documents. As the content of highly sensitive applications like finance, healthcare, etc. are driven by NXDs, security of NXDs is vitally important to ensure the integrity, privacy and confidentiality (Baviskar and Thilagam, 2011), and to make sure that information is used appropriately (Huang, 2003).

According to the Internet Security Threat Report by Symantec Corporation (Symantec, 2014), one in eight web applications has critical unpatched vulnerabilities. A vulnerability is a coding flaw in the application and new types of attacks emerge from time to time to exploit these vulnerabilities. The security consortiums, OWASP (Top10, 2013), SANS (2011), and WASC (Gordeychik, 2010) list injection vulnerabilities as the most prevalent flaw in web applications. Injection vulnerabilities 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 such as SQL injection and XML injection exploit the vulnerabilities for extraction/insertion of data from/into the database of the application. These attacks attempt to modify the query submitted to the database by providing malicious input to the web application server. If the web application submits SQL queries to a relational database, then the attack is referred to as an SQL injection attack. If the web application submits XQuery to an XML database, it is referred to as an XQuery injection attack. XQuery contains a superset of an XPath expression syntax. XQuery 1.0 includes XPath 2.0 as a sublanguage. According to the Payment Card Industry Data Security Standard (PCI DSS) and Common Vulnerability Scoring System (CVSS), XQuery and XPath injections are high risk threats, and hence detection of the vulnerabilities that could lead to these injection attacks is of critical importance (Gordeychik, 2010).

Even though a large body of literature exists for preventing SQL injection (Huang et al., 2004, 2005; Halfond and Orso, 2005; Buehrer et al., 2005; Xie and Aiken, 2006; Su and Wassermann, 2006; Kosuga et al., 2007; Thomas and Williams, 2007; Wassermann and Su, 2007; Liu et al., 2009; Bisht et al., 2010; Scholte et al., 2012; Lee et al., 2012; Jang and Choi, 2014), only a limited number of articles exist for identifying XML injection vulnerabilities in web applications (Huang, 2003; Antunes et al., 2009; Antunes and Vieira, 2011; Asmawi et al., 2012). The existing works on SQL injection focus on detection/prevention of known patterns of SQL injection attacks. The proposed solutions for detecting SQL injection have their own pros and cons, and they are described as follows. Secure programming imparts overhead on developers for implementing the security guidelines during development (Bravenboer et al., 2007; XPath-Injection, 2015; Trulove and Svoboda, 2011). Signature-based approach does not prevent zero-day attacks, and suffers from false negatives when attack query matches the structure of a legitimate query (Huang, 2003; Mitropoulos et al., 2011; Antunes and Vieira, 2011). The knowledge-based approach requires new training whenever modifications are made to the source code of the application (Huang et al., 2005; Mitropoulos et al., 2009; Rosa et al., 2013). The accuracy of machine learning approach is dependent on appropriate selection of the training set (Scholte et al., 2012; Valeur et al., 2005; Menahem et al., 2012; Chan et al., 2013). Existing approaches for addressing XML injection concentrate on detection/prevention of vulnerabilities/attacks in web services only, and cover a certain types of XML injection attacks only. Therefore, there is a demand for a system that is capable of detecting different kinds of XQuery injection vulnerabilities in native XML database-driven applications. Hence, this paper focuses on development of a prototype for the detection of XQuery injection vulnerabilities in web applications.

Taking into account the limitations of existing approaches, a prototype is developed following a query model-based approach for identifying XQuery injection vulnerabilities. The existing query model-based approaches employ static analysis for preventing SQL injection attacks and involve code instrumentation for identifying and preventing the attacks, whereas the proposed approach is a fully-automated black-box fuzzer which does not require access to the source code of the application and concentrates on identifying XQuery injection vulnerabilities. To the best of our knowledge, this is the first work that focuses on identifying XQuery injection vulnerabilities in web applications driven by native XML databases. The prototype detects different types of XQuery injection vulnerabilities as specified in OWASP (2014) guidelines. While the existing approaches detect different kinds of SQL injection attacks that are predefined, we have identified three new categories of attack vectors which are not listed in OWASP.

The major contributions of this work are as follows:

- We propose an automated black-box approach for identifying XQuery injection vulnerabilities in native XML database-driven web applications.
- We implement a prototype system called XQueryFuzzer based on the proposed approach.
- We propose an attack grammar to generate different types of XQuery attack strings for identifying vulnerabilities existing in the application and prove the consistency and completeness of the grammar.
- We identify three new categories of attacks namely, alternate encoding, evaluation function and XQuery comment injection attack that are specific to XQuery, but not listed in OWASP.[1]
- 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 vulnerabilities. Section 3 presents the related work. The proposed approach and implementation details are described in Section 4. Section 5 discusses the new category of attacks identified with examples. Section 6 provides information on the experimental setup and discusses the results. The paper is concluded in the last section.

2. Preliminaries

This section provides a simple example of an XQuery injection attack and discusses its various types.

2.1. Injection attacks

A vulnerability is a coding flaw in the application that can be exploited by the attackers for injecting malicious code to compromise the security of the application. The malicious code can be injected through the user input, which are used as part of a query. If an adversary injects a malformed input into an XQuery for extracting/inserting information from/into an XML document resulting in

[1] The attacks identified are communicated to the OWASP Testing Guide Project.

undesirable actions, it is referred to as an XQuery injection attack. Injection attacks are mainly caused due to lack of validation or insufficient validation of input supplied by the user.

2.1.1. XQuery injection attack

A simple example of an XQuery injection attack is described with the following XML document, *users.xml* (Auger, 2010):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<userlist>
  <user category="group1">
    <fname>john</fname>
    <status>good</status>
  </user>
  <user category="admin">
    <fname>john</fname>
    <status>good</status>
  </user>
  <user category="group2">
    <fname>mary</fname>
    <status>good</status>
  </user>
</userlist>
```

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

```
String strName = (String) request.getParameter("name");
String strQuery = "xquery for $x in doc('users.xml')/userlist/user
where $x/fname='" + 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 *"john"*.

```
<user category="group1">
  <fname>john</fname>
  <status>good</status>
</user>
<user category="admin">
  <fname>john</fname>
  <status>good</status>
</user>
```

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('users.xml')/userlist/user
where $x/fname='XXX'  or '1'='1'  return $x
```

The XQuery when executed retrieves the whole XML document and presents it to the attacker. It can be observed that absence of proper validation mechanism for the user input has resulted in an attack. Each security weakness in web applications is assigned a unique number called Common Weakness Enumeration (CWE)[2] to enable easier tracking and understanding. CWE is a community-developed dictionary of software weakness types, which provides a unified, measurable set of software weaknesses. *CWE-652*[3] (Improper Neutralization of Data within XQuery Expressions ('XQuery injection')) refers to the aforementioned weakness.

Different types of XML injection attacks suggested by OWASP (2014) are listed as follows:

- *Tautology attack:* This attack is carried out by appending an expression that always returns 'True' to the user input. E.g., *or '1'= '1, or 'a'= 'a*. The example discussed above is a tautology attack.
- *Meta Character injection attack:* This attack is carried out by inserting an XML meta character such as *',"*, ⟨, ⟩, or *&* in the user input. The CWE identifier assigned for this type of attack is *CWE-150*.
- *Comment injection attack:* This attack is performed by inserting a comment character sequence ⟨!– in the user input *(CWE 151)*.
- *CDATA section injection attack:* This attack is carried out using the CDATA section *(CWE 146)*. CDATA sections are generally used for escaping blocks of text which would otherwise be recognized as markup.
- *Tag injection attack:* This attack is carried out by injecting a tag in the user input causing the structure of the XML to be modified and hence corrupting the database.
- *External entity injection attack:* This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser (XXE, 2016). The set of valid entities that can be processed by the XML parser can be extended by defining new entities. If the definition of an entity is a URI, it is called an external entity. If it is not configured properly, then external entities force the XML parser to access confidential resource such as a file placed in a remote system *(CWE-611)*.

3. Related work

This section provides insight about the work done so far for identifying/preventing injection vulnerabilities/attacks that target the database of the web application. With respect to SQL injection, works such as Huang et al. (2005, 2004); Xie and Aiken (2006); Kosuga et al. (2007); Wassermann and Su (2007) belong to the category of vulnerability detection. Works focusing on SQL injection attack prevention include Halfond and Orso (2005); Buehrer et al. (2005); Bisht et al. (2010); Jang and Choi (2014). Relatively less amount of research has been done on XML injection vulnerability detection and attack prevention. The drawback with attack prevention systems is that false positives drop a valid query from being submitted to the application. Hence, we prefer to design a vulnerability detection approach in which the detected vulnerabilities can be analyzed and fixed by the developer appropriately.

A detailed classification of the existing approaches can be found in the literature (Halfond et al., 2006; Shahriar and Zulkernine, 2011, 2012; Li and Xue, 2014; Deepa and Thilagam, 2016; Chandrashekhar et al., 2012). Based on the methodology (i.e., signature-based approach, knowledge-based approach, etc.) adopted for detecting /preventing vulnerabilities or attacks, the existing literature is categorized as follows:

(a) Secure programming: Secure coding practices to prevent injection attacks include validation of user-supplied input, parameterizing queries, pre-compilation of queries, escaping the special characters in the input, and embedding grammar of the query language into that of the host language (e.g., Java, C#, VB, etc.) (Bravenboer et al., 2007; XPath-Injection, 2015; Truelove and Svoboda, 2011). The drawback with this approach is the overhead imparted on developers to learn and implement the security paradigms during construction.

(b) Signature-based approach: Huang (2003) discussed detection of intrusions in database systems through fingerprinting

[2] https://cwe.mitre.org/.
[3] https://cwe.mitre.org/data/definitions/652.html.

Table 1
Comparison of XQueryFuzzer with other query model-based approaches.

Research article	Area of focus	Type of analysis	Remarks
AMNESIA (Halfond and Orso, 2005)	SQL Injection Attack Prevention	Static analysis & Runtime protection	Involves code instrumentation
SQLGuard (Buehrer et al., 2005)	SQL Injection Attack Prevention	Static analysis & Runtime protection	Involves manual code annotation
SQLCHECK (Su and Wassermann, 2006)	SQL Injection Attack Prevention	Runtime Protection	Involves code instrumentation
Sania (Kosuga et al., 2007)	SQL Injection Vulnerability Detection	Dynamic analysis	Uses HTTP proxy to intercept HTTP packets and SQL proxy to intercept SQL queries
SQLProb (Liu et al., 2009)	SQL Injection Attack Prevention	Black-box approach	Uses proxy for capturing queries
CANDID (Bisht et al., 2010)	SQL Injection Attack Prevention	Static analysis & Runtime protection	Involves code instrumentation
XQueryFuzzer	XQuery Injection Vulnerability Detection	Black-box fuzzer	Fully automated approach

transactions. A broad class of injection attacks is prevented using a signature-based approach in Mitropoulos et al. (2011). During training phase, vulnerable code statements are identified and registered using unique signatures for differentiating normal and abnormal executions. During runtime, the framework checks the compliance of all statements with the learnt model and blocks code statements containing maliciously injected elements. Sign-WS (Antunes and Vieira, 2011) employs penetration testing and interface monitoring for detection of the attack signatures to identify injection vulnerabilities in web services. New unknown attacks cannot be detected using the signature-based approach, even if they have only small variations from a known payload (Rosa et al., 2013).

(c) Schema-based approach: Groppe and Groppe (2008) identified XPath queries that do not satisfy the constraints defined in the schema, and forbid the queries from being executed for preventing attacks. Lampesberger (2013) detected anomalies in XML documents from grammatical-inference of the documents by constructing a visibly pushdown automaton. The automaton provides the syntactic structure and data types of the parameters used in the XML document, which helps in identifying the anomalies.

(d) Query model-based approach: A number of query model-based approaches exist for preventing SQL injection attacks in web applications (Halfond and Orso, 2005; Buehrer et al., 2005; Su and Wassermann, 2006; Kosuga et al., 2007; Liu et al., 2009; Bisht et al., 2010). Laranjeiro et al. (2009) proposed an approach that combines penetration testing and static code analysis to detect and prevent SQL and XPath injection attacks in web services. Antunes et al. (2009) identified injection vulnerabilities in web services using Aspect Oriented Programming that intercept all the calls to API methods executing SQL commands. The structure of SQL/XPath commands issued in the presence of attacks to the ones previously learnt when running the workload in the absence of attacks are compared for identifying vulnerabilities. Asmawi et al. (2012) proposed an approach similar to Antunes et al. (2009) for preventing XPath injection attacks in a web services environment.

The problem with query model-based approach is that it fails to detect attacks wherein the structure of the query varies dynamically based on conditional input (Bisht et al., 2010). Table 1 provides a comparison of the proposed approach with existing query model-based approaches. Table 1 shows that the existing approaches focus on SQL injection attack prevention, and involve instrumentation of the source code which needs to be executed for preventing attacks. SQLProb (Liu et al., 2009) uses a proxy for capturing the queries which is a overhead at the server-side of the application. The proposed work is relevant to Sania developed by Kosuga et al. (2007). Sania requires the developer to generate valid HTTP requests, and employs two proxies: HTTP proxy to capture HTTP packets, and SQL proxy to capture SQL queries. Usage of two proxies is a overhead in Sania. In contrast, our approach is fully automated, does not make use of any proxy, and does not involve any manual intervention. In this paper, we follow a query model-based approach entirely based on black-box fuzzing. Unlike the work by Antunes et al. (2009) which focuses on web services, this work focuses on the identification of vulnerabilities in web applications using native XML databases.

(e) Knowledge-based approach: Huang et al. (2005) developed a framework called WAVES for detecting SQL injection vulnerabilities. The framework crawls through the application to identify entry points in the application that can propagate attacks. It constructs a knowledge-base for injecting the entry points with appropriate inputs to launch successful attacks. Mitropoulos et al. (2009) developed a methodology that constructs a knowledge-base during the training phase. During the testing phase, when an XPath query is encountered, an identifier is generated. If the identifier does not exist in the knowledge-base, the query is considered as malicious. The major drawback of this technique is that any change in source code requires a new training to reassign the identifiers. Rosa et al. (2013) presented an XML injection strategy-based detection system for mitigating zero-day attacks.

(f) Machine learning approach: A machine learning based approach proposed by Scholte et al. (2012) prevents SQL injection vulnerabilities in web applications. Valeur et al. (2005) calculated anomaly scores for each query being executed, and detected SQL injection attacks when the anomaly score exceeds the maximum anomaly score obtained during attack free execution. Anomalies in XML transactions are detected using a framework called XML-AD (Menahem et al., 2012). The features of the XML documents are extracted and transformed into attribute vectors, and anomalies are detected using a novel multi-univariate anomaly detection algorithm, ADIFA. A predictive fuzzy associative rule model (FARM) (Chan et al., 2013) is developed for identifying attack patterns and anomalies to counter both signature and anomaly-based XML related attacks. The problem with the machine learning approach is that the number of false positives and false negatives generated depends on appropriate selection of the training set.

(g) Instrumenting sanitization code: WebSSARI (Huang et al., 2004) analyzes the source code to identify vulnerable injection points and instruments the source code with appropriate sanitization mechanisms for preventing SQL injection attacks. Thomas and Williams (2007) identified and replaced vulnerable SQL statements in the code with secure SQL statements. The existing approaches for SQL and XML injection vulnerability detection/prevention are summarized in Table 2.

Existing tools for identification of XML injection vulnerabilities cover only certain type of XML injection attacks. XCat (Forbes, 2014) is a tool used to identify blind XPath injection vulnerabilities in web applications. XMLMao (2012) is another tool used for exploiting XML injection vulnerabilities in web applications. The limitation of the tool is that the user has to enter attack strings in user input fields. WebCruiser (2011) is designed to detect XML injection, but it detects only tautology injection attacks and cannot detect other types of XML injection attacks. The vulnerability scanners Acunetix (2016), Wapiti (2013), w3af, and Arachni (2016) identify XPath injection vulnerabilities in web applications that

Table 2
Summary of SQL/XML injection prevention/detection approaches.

Research Article	Area of focus				Type of analysis			Approach				
	Vulnerability detection	Vulnerability prevention	Attack detection	Attack prevention	Static analysis	Dynamic analysis	Penetration testing	Signature-based	Schema-based	Query model-based	Knowledge-based	Machine learning
SQL Injection												
Huang et al. (2004)	✓				✓							
Buehrer et al. (2005)				✓	✓					✓		
Halfond and Orso (2005)			✓	✓	✓	✓				✓		
Huang et al. (2005)	✓					✓	✓				✓	
Su and Wassermann (2006)				✓	✓					✓		
Kosuga et al. (2007)	✓					✓	✓			✓		
Thomas and Williams (2007)		✓			✓			✓				
Wassermann and Su (2007)	✓				✓							
Liu et al. (2009)				✓		✓				✓		
Bisht et al. (2010)				✓		✓				✓		
Scholte et al. (2012)		✓			✓							✓
Lee et al. (2012)			✓		✓	✓						
Jang and Choi (2014)				✓		✓						
XML Injection												
Huang (2003)				✓				✓				
Groppe and Groppe (2008)				✓					✓			
Antunes et al. (2009)	✓					✓	✓			✓		
Laranjeiro et al. (2009)	✓				✓		✓			✓		
Mitropoulos et al. (2009)				✓	✓						✓	
Antunes and Vieira (2011)	✓						✓	✓				
Mitropoulos et al. (2011)				✓				✓				
Asmawi et al. (2012)				✓	✓					✓		
Menahem et al. (2012)				✓								✓
Chan et al. (2013)				✓								✓
Lampesberger (2013)				✓					✓			
Rosa et al. (2013)				✓	✓			✓			✓	

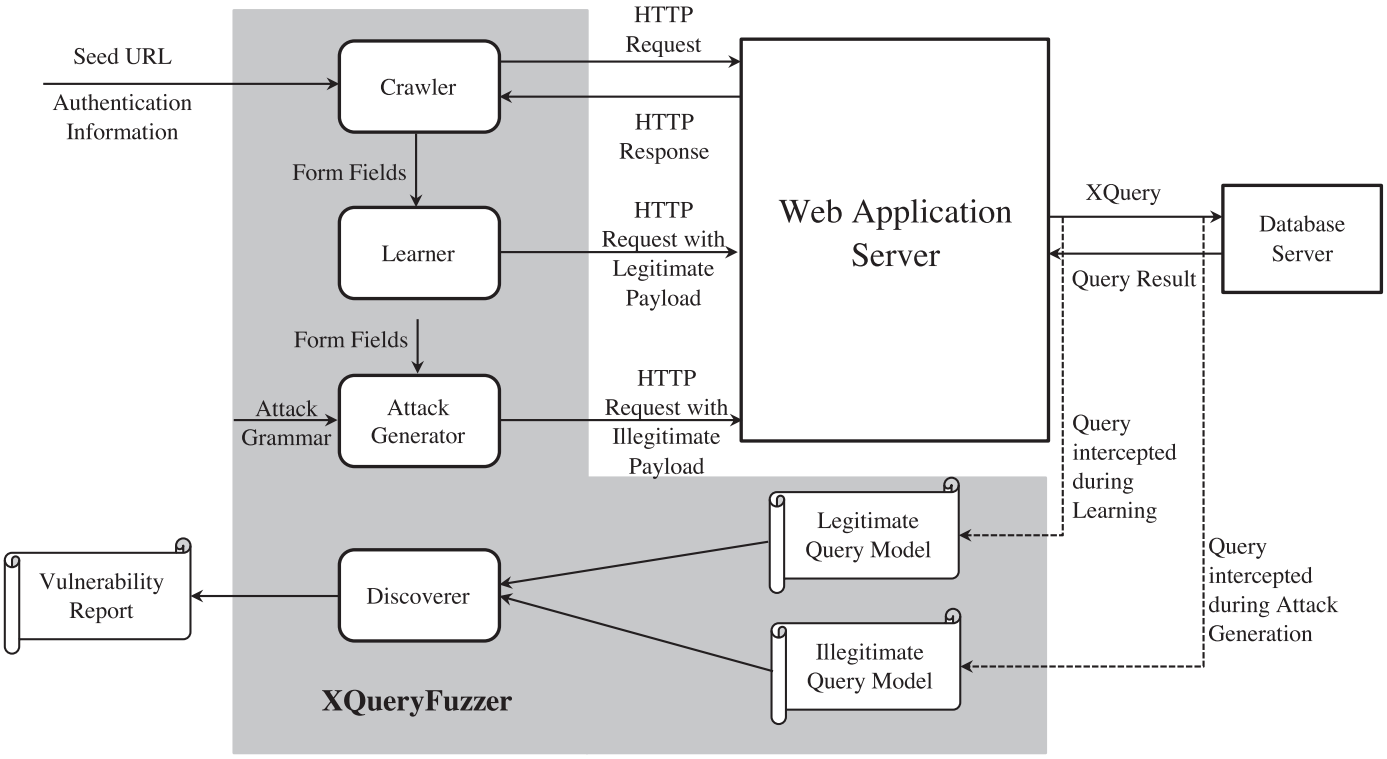

Fig. 1. Design of the prototype: XQueryFuzzer.

involve XML documents (van der Loo, 2011). Each of the existing tools covers only a specific type of vulnerability, such as tautology attacks, or focuses on XPath injection. In addition, none of the existing tools detect XML vulnerabilities in web applications using native XML databases. Hence, there is a need for a prototype that focuses on detection of various types of XQuery injection vulnerabilities in web applications using native XML databases.

4. Prototype design

The primary objective of this work is to propose an automated approach and develop a prototype system XQueryFuzzer for detecting XQuery injection vulnerabilities based on OWASP (2014) guidelines in native XML database-driven web applications. The prototype uses a black-box fuzzer for discovering vulnerabilities. Black-box fuzzing ensures that the prototype does not require the availability of source code of the application. A summary of the processing steps is explained below.

Crawling: 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.

Learning: The injectable points are populated with valid inputs and the corresponding web form is submitted to the web application. The queries submitted to the database server are intercepted, and are used for constructing legitimate query models.

Attack generation: The injectable points are populated with attack strings that are generated based on the attack grammar, and the corresponding web form is submitted to the application. The queries that are processed by the server are intercepted and used for constructing illegitimate query models.

Discovery: 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.

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, 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 based on the attack grammar 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 attacks. Fig. 1 represents the design of the prototype. The different phases involved in the proposed approach are explained in detail in the following subsections.

4.1. Crawling phase

A crawler is employed for navigating through the application. In addition to following hyperlinks as in a conventional web crawler, it also identifies the forms in each page and stores information related to the form fields in a database. These form fields are the vulnerable fields which are used as points of attack string injection. The crawling process starts from the seed URL supplied by the user as input. The crawler only crawls links which belong to 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 provides extensive coverage of the URLs and forms in the web application as none of the points of injection should be missed out to ensure that all existing vulnerable fields are reported. 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,*

Algorithm 1 Algorithm for Crawler.

```
1: function START_CRAWL(seed_URLs, logout_URL, authentication_
   info, authenticated=False)
2:     yet_to_visit_URLs ← [ ]
3:     visited_URLs ← [ ]
4:     if authenticated=True then
5:         session_info ← perform_login(authentication_info)
6:     end if
7:     for seed_URL in seed_URLs do
8:         if seed_URL not in visited_URLs then
9:             yet_to_visit_URLs.add(seed_URL)
10:            forms_list ← [ ]
11:            while count(yet_to_visit_URLs) ≠ 0 do
12:                u ← getURL(yet_to_visit_URLs)
13:                r ← getHTMLresponse(submitToWebApp(u,
                   session_info))
14:                l ← searchLinkTags(r)
15:                for each v in l do
16:                    yet_to_visit_URLs.add(v)
17:                end for
18:                forms_list ← ExtractFormTags(r)
19:                START_LEARNING(forms_list,auth,session_info)
20:                yet_to_visit_URLs.delete(u)
21:                visited_URLs.add(u)
22:                forms_list ← [ ]
23:            end while
24:        end if
25:    end for
26: end function
```

window.location, .load, .location.assign, .href, .action, and .src as well for identifying the URLs in a web page. The working principle of the crawler is shown in Algorithm 1. Existing open source crawlers (WebSPHINX, 2002; JSpider, 2003) index the web pages found, a performance intensive task, which 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:

- Unlike other crawlers, this crawler crawls only links such as anchor tags in HTML but not other resources such as documents and images in HTML as the purpose of the crawl is only to identify injectable points, i.e., form fields.
- The crawler is tested against benchmark web application WIVET (Urgun) for testing the coverage. The coverage score obtained by the crawler is 83.9%. The test cases which are not passed by the developed crawler are links created using AJAX or external JavaScript files, links attached to Flash buttons and links mentioned in HTML comments.
- The crawler does not support type enhanced form submission, i.e., the restrictions imposed on input fields in the web application are not 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 does not take care of these restrictions, and as a result, the forms submitted with a value 1000 for *Marks* may fail to execute resulting in an error page and prevents the subsequent pages from being crawled.
- The crawler does not extract information about forms submitted using XMLHTTPRequest (i.e., AJAX).
- A self-regulating or politeness feature can be added to the crawler so that it can be applied to a real-world web application.

4.2. Learning phase

The forms captured during crawling phase are populated with valid input strings and submitted to the web application. The queries executed on this form submission are intercepted and modeled. This model is stored as the learnt model for each query. Algorithm 2 explains the working mechanism of learner.

Algorithm 2 Algorithm for Learner.

```
1: function START_LEARNING(forms_list, auth, session_info=NULL)
2:     for each form f in forms_list do
3:         data ← {}
4:         T ← findInputFieldTags(f.content)
5:         for each t in T do
6:             data[t] ← generate_valid_value(t)
7:         end for
8:         f.payload ← data
9:         queryq.clear()
10:        submitFormToWebApp(f, session_info)
11:        learnt_models ← {}
12:        query_id ← 0
13:        q ← queryq.dequeue()
14:        while q exists do
15:            query_id ← query_id + 1
16:            learnt_model ← get_query_model(q)
17:            learnt_models[query_id] ← learnt_model
18:            q ← queryq.dequeue()
19:        end while
20:        START_DISCOVERING(f,queryid,learnt_models,T,session_info)
21:        learnt_models ← {}
22:    end for
23: end function
```

4.2.1. Query model construction

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(...)* are replaced with *DOCPATH*
- Round parenthesis (i.e., (,)) with *OPEN_BRACE* and *CLOSE_BRACE*
- *xx/xx/$xxx* with *VARPATH*
- *$xxx* with *VARIABLE*
- separators such as commas and semicolons with *COMMA* and *SEMI_COLON*
- constants with *CONSTANT*
- literals with *LITERAL*
- operators with operator description such as = with *EQUALS*
- literals containing meta characters are replaced with *META*

For example, consider the following scenarios:

- **Case I:** A vulnerable web application (coded in Python) using non-parameterized query:

  "for $emp in doc('empdb')/employees/employee
  where $emp/employee_id='" + emp_id +
  "' return $emp"

  Assuming the input submitted by the user for the variable *emp_id* is *E123*, then the XQuery submitted to the XML database is

  for $emp in doc('empdb')/employees/employee

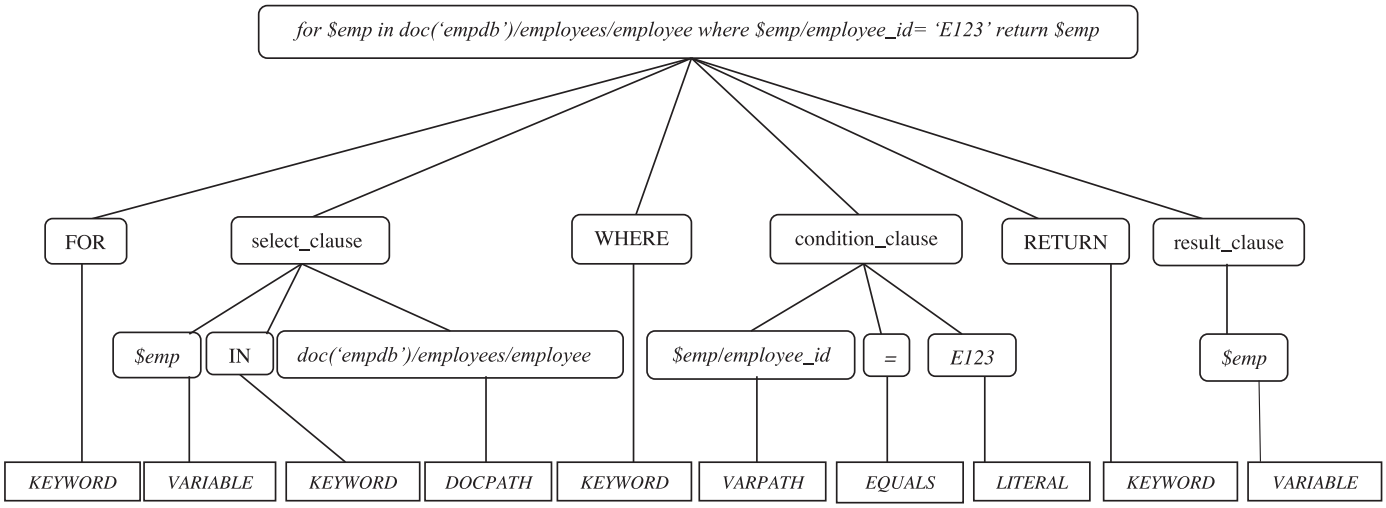

Fig. 2. Query model for a retrieval query with valid input.

where $emp/employee_id='E123' return $emp

The query model constructed for the above XQuery is as shown in Fig. 2.

- **Case II:** Application using parameterized query:

"for $emp in doc('empdb')/employees/employee
where $emp/employee_id='%s'
return $emp" %(emp_id)

XQuery submitted to the XML database with valid input is:

for $emp in doc('empdb')/employees/employee
where $emp/employee_id='E123'
return $emp

Query model for valid input query is as shown in Fig. 2.

Thus, the query models formed for both parameterized and non-parameterized queries are the same.

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.

4.3. Attack generation phase

To detect XQuery injection, malicious input is provided in the form fields identified in the crawling phase. Malicious strings are generated using the attack grammar developed for detecting XQuery injection. The attack grammar is constructed by taking into account the different types of XML injections identified by the security consortium, OWASP, and the new category of attack vectors listed in Section 5.

4.3.1. Attack grammar

The attack grammar is framed based on regular expressions representing each type of injection attack as detailed in Sections 2.1 and 5. 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 grammar has been formed. Hence, the proposed attack grammar enables the identification of all the above mentioned types of XML injection attacks. For example, the regular expression for generating tautology attack strings is: *(A-Za-z0-9)$^{+}$' or '1'= '1' or '(A-Za-z0-9)$^{+}$*. A sample string generated by this regular expression is *abc' or '1'= '1' or 'xyz*. The regular expression for comment injection attack string is: *(A-Za-z0-9)$^{+}$ ⟨!--*. A sample string generated by this regular expression is *abc123 ⟨!--*. Similarly, regular expressions can be formed for each type of attack string. A combination of these regular expressions is used to form a context free grammar.

The attack grammar is a context free grammar *G* defined by *(N, T, P, S)* where *N* is the set of non-terminals, *T* is the set of terminals, *P* is the set of productions, and *S* is the start symbol.

N={S, TAUT, METACHAR, COMMENT, CDATA_END, CDATA_CHAR, TAG, EXT_ENTITY, W_CDATA, ALT_ENCODING, EVAL, STRING_SC, SPECIAL_CHAR, STRING}

$T = \{', ", \langle, \rangle, \&, !, [, /, ?, ., -, =, ;, ,, :, @, \{, \}, [,], (,), \_, a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9\}$

Productions *P* of the grammar *G* are formed as follows:

S → TAUT | METACHAR | COMMENT | CDATA_END | CDATA_CHAR | TAG | EXT_ENTITY | ALT_ENCODING | EVAL

TAUT → STRING' or '1'='1' or 'STRING | STRING' or 'a'='a' or 'STRING | STRING' or 1=1 or 'STRING | STRING" or '1'='1' or *"STRING | STRING" or 'a'='a' or "STRING | STRING" or 1=1 or "STRING*

METACHAR → STRING" STRING | STRING" STRING | STRING' STRING | STRING' STRING | STRING ⟨ STRING | STRING ⟩ STRING | & STRING

COMMENT → STRING⟨!-- | (: STRING :)

CDATA_END → STRING]]⟩

STRING → STRING | a | b | ...| z | A | B | ...| Z | 0 | 1 | ...| 9 |_ | ϵ

Similarly, productions are framed for the remaining types of injection attacks; *CDATA_CHAR, TAG, EXT_ENTITY, ALT_ENCODING*, and *EVAL* attacks.

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 the web

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

Proof of Consistency and Completeness of Attack Grammar: The attack grammar framed is a context free grammar *G*. Generally the consistency and completeness of a context free grammar is proved using mathematical induction (Gopalakrishnan, 2006). However, this technique is most suitable for grammars containing recursive productions, especially in the start production. As the attack grammar framed does not contain recursive productions, except to generate alphanumeric strings, the proof of consistency and completeness for the grammar is trivial. Consistency ensures that all strings generated by the grammar belong to the language for which the grammar is intended to be framed. Completeness ensures that all strings that belong to the language are generated by the grammar.

Proof of consistency: Let *L* be the language of XQuery attack strings. The start production for the grammar *G* is as follows:

$S \rightarrow TAUT \mid METACHAR \mid COMMENT \mid CDATA\_END \mid CDATA\_CHAR \mid TAG \mid EXT\_ENTITY \mid ALT\_ENCODING \mid EVAL$

Let *W* be a word generated by *G*. *W* is generated by any of the non-terminals on the right-hand-side of the start production. Say, *TAUT* generates *W*. In this case, *W* contains any one of the following expressions *'1'= '1', "1"="1", 1=1, 'a'= 'a'*, or *"a"="a"* as a substring. If a string contains any of these expressions, the string is an attack string under the category of tautology attacks. Hence, *W* belongs to *L*. Similarly, consistency can be proved for a word *W* generated by any of the other non-terminals on the right-hand-side of *S*.

Proof of completeness: Let *W* be a word which belongs to the language *L*. Say, *W* belongs to the category of tautology attack strings. In this case, *W* contains *'1'='1', "1"="1", 1=1, 'a'='a'*, or *"a"="a"* as a substring. All strings containing these expressions as substrings can be generated by the production $S \rightarrow TAUT$. Hence, all tautology attack strings in the language can be generated by the grammar *G*. Similarly, completeness can be proved for all strings that belong to any category of attacks in *L*, all strings of which are generated by the corresponding start production.

4.3.2. Query model construction

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 learning phase. This model is compared against the learnt model for each query.

Consider the same example specified in the learning phase.

- **Case I:** Non-parameterized query: Assuming the input submitted by the user for the variable *emp_id* is *E123' or '1'='1*, then the XQuery submitted to the XML database is

  for $emp in doc('empdb')/employees/employee
  where $emp/employee_id='E123' or '1'='1'
  return $emp

  The condition clause of the query model generated with invalid input string is as shown in Fig. 3.
- **Case II:** Parameterized query:
  XQuery submitted to the XML database formed with invalid input:

  for $emp in doc('empdb')/employees/employee
  where $emp/employee_id= 'E123\' or \'1\'= \'1'
  return $emp

  Since the query is parameterized, this invalid input query returns no results as there is no employee found in the database having emp_id = *"E123' or '1'= '1"*. The injection attack is unsuccessful as the employee id form field is not vulnerable.
  The query model for invalid input query has the condition clause as shown in Fig. 4.

4.4. Discovery phase

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 Case I, the leaf nodes of the condition clause for legitimate and illegitimate input strings are *VARPATH EQUALS LITERAL* and *VARPATH EQUALS LITERAL KEYWORD LITERAL EQUALS LITERAL* respectively. The leaf nodes of the learnt and discovered query models obtained for Case I are dissimilar and hence a vulnerability is reported. For Case II, the leaf nodes of the condition clause for legitimate and illegitimate input strings are *VARPATH EQUALS LITERAL* and *VARPATH EQUALS LITERAL* respectively. The leaf nodes of the learnt and discovered query models obtained are similar, and hence no vulnerability is reported. Algorithm 3 provides the pseudocode of the discovery phase.

Algorithm 3 Algorithm for Discoverer.

```
1: function START_DISCOVERING(f, lqlength, learnt_models, T, session_info=NULL)
2:     i ← 1
3:     while i ≤ len(list_vectors) do
4:         data ← {}
5:         for each t in T do
6:             data[t] ← generate_attack_value(t, list_vectors[i])
7:         end for
8:         f.payload ← data
9:         queryq.clear()
10:        status_code ← getStatusCode(submitFormToWebApp(f, session_info))
11:        if status_code = '200' then
12:            dqlength ← len(queryq)
13:            if dqlength=lqlength then
14:                query_cid ← 1
15:                q ← queryq.dequeue()
16:                while q exists do
17:                    discovered_model ← get_query_model(q)
18:                    learnt_model ← learnt_models[query_cid]
19:                    if learnt_model ≠ discovered_model then
20:                        report_vulnerability()
21:                    end if
22:                    q ← queryq.dequeue()
23:                    query_cid ← query_cid + 1
24:                end while
25:            end if
26:        else if (status_code[0]= '5') or (status_code[0]= '4') then
27:            report_internal_server_error()
28:        end if
29:        i ← i + 1
30:    end while
31: end function
```

The discovery 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 attack grammar for generating attack strings. To ensure that these strings are indeed attack vectors representing one of the types of XQuery injection attacks, we prove the consistency of the attack grammar. To ensure that the application is attacked by different types of

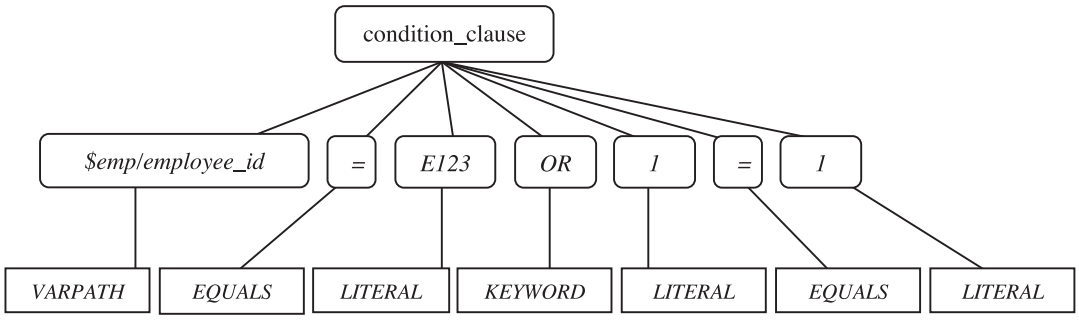

Fig. 3. Query model (condition clause) for non-parameterized retrieve query with invalid input string.

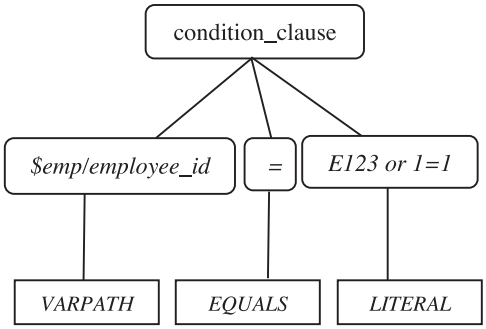

Fig. 4. Query model (condition clause) for parameterized retrieve query with invalid input string.

XQuery injection attack strings specified in the OWASP guidelines, we prove the completeness of the grammar.

5. New category of XQuery injection attacks

Apart from the attacks specified in OWASP (2014), new kinds of attacks are always possible and an adversary can make use of the advancements offered by the query language to manipulate the database request for launching attacks. We exploited the vulnerabilities in the test applications to launch different kinds of XQuery injection attacks that are not specified in OWASP (2014). The three new categories of XQuery injection attacks identified during our testing are discussed in the following subsections.

5.1. Alternate encoding attack

In this attack, the malicious code is injected in encoded form (e.g., ASCII, hexadecimal, etc.) into the query. The malicious code is encoded by the attacker to defeat the defensive coding practices employed for blocking special characters such as ampersand (&), single quote (‘), and less than symbol (<) in the user input. A conversion function such as convert (2016) is included in the user input to decode the encoded attack string. This attack is generally used in conjunction with other types of attacks. In other words, alternate encodings do not provide any unique way to attack an application; it is a technique that allows attackers to evade detection and prevention techniques and exploit vulnerabilities that might not otherwise be exploitable (Halfond et al., 2006).

Example 1. *“xquery let $up: =” + parameter + “return insert node (<post><from>User1</from><data>{$up}</data></post>) as last into doc (‘postsdb/posts.xml’)/posts_all”*

Attack String:
parameter=convert : binary−to−string(xs : hexBinary(‘3c212d2d’))

The query generated by the application is:

xquery let $up: = convert:binary-to-string(xs:hexBinary(‘3c212d2d’))
return insert node
(<post><from>User1</from><data>{$up}</data></post>)
as last into doc(‘postsdb/posts.xml’)/posts_all

The string equivalent of the hexadecimal code *‘3c212d2d’* is *<!--*, a comment tag in HTML. Therefore, the query will be evaluated as:

insert node (<post><from>User1</from>
*<data>**<!--**</data></post>)*
as last into doc (‘postsdb/posts.xml’)/posts_all

When the query is executed, a new record is inserted into the database with the comment tag stored as a string ‘<!--’. Whenever a web page makes use of this injected record, the HTML comment tag gets inserted into the HTML code of the web page, and the rest of the HTML code is commented out from the point of insertion, and thus an attack is made successfully.

Example 2. Assuming that the application blocks special characters like <, >, etc. in the user input, the application can still be exploited by using an encoded version of the characters. For the same query discussed above, a stored XSS attack is possible with the following attack string.

Attack String:
parameter = convert : binary − to − string(
xs: hexBinary(‘3c7363726970743e616c65727428277873
7327293c2f7363726970743e’))

The query generated by the application is:

xquery let $up: = convert : binary − to − string(xs :
hexBinary(‘3c7363726970743e616c657274282778737327
293c2f7363726970743e’))
return insert node (<post><from>User1</from>
<data>{$up}</data></post>) as last into doc
(‘postsdb/posts.xml’)/posts_all

The query will be evaluated as:

insert node (<post><from>User1</from><data>
<script>alert(‘xss’)</script> *< /data >*
< /post >)as last into doc (‘postsdb/posts.xml’)/posts_all

Consequently, a new child node gets inserted into the XML document and whenever a web page makes use of this injected record, a JavaScript alert will be thrown on the web page.

Thus, encoded version of the malicious characters can be used along with *convert* function of XQuery for causing attacks.

5.2. Injection via evaluation function

The absence of proper sanitization mechanisms for validating the user input can be exploited by an attacker for creating attacks that affect the intended behavior of the application. The expected behavior can be modified due to execution of malicious expressions by using *eval (2016)* function.

Consider a web page in an application that allows a user to enter a string against the price of a product. As the input field accepts string values, an adversary can provide input with an arithmetic expression embedded within *eval* function of XQuery. Suppose, the application fails to validate the value of price (i.e., fails to check if the value of price is greater than zero) at the client-side, then the adversary can provide the expression in such a way that it evaluates to a negative value thus affecting the intended behavior of the application. In case, client-side validation is done but lacks server-side validation, then still there is a possibility to inject the *eval* function for parameter price via the HTTP request.

Example 3. *"xquery let $up: =" + parameter + "return insert node (<product><name> Item 1 </name> <price>{$up}</price> </product>) as last into doc ('products.xml')/products_all"*

Attack String:
parameter = *eval*('100 * −1')

The query generated by the application is:

xquery let $up: = eval('100*-1') return insert node (<product><name> Item 1 </name><price>{$up}</price> </product>) as last into doc ('products.xml')/products_all

The function *eval* evaluates the given expression to -100 and the query becomes as follows:

*insert node (<product><name>Item 1</name> <price>**-100**</price></product>) as last into doc ('products.xml')/products_all*

When the query is executed, a new product gets inserted into the database with the price of the product as **−100** and thus an injection is made successfully.

The *eval* function can also be used to evaluate an XQuery expression.

5.3. XQuery comment injection attack

In this attack, an attacker injects XQuery comment symbol ('(:') into the query so as to ignore a part of the XQuery expression from being executed which in turn affects the intended behavior of the application.

Example 4. *"xquery for $x in doc('users.xml')/users/user where $x/fname='" + strName + "' and $x/password ='" + strPassword + "' and $x/year=" + iYear + " return $x"*

Attack String:
fname = *admin'*(:
password =*abc*
year = 1975 :)

The query generated by the application is:

xquery for $x in doc('users.xml')/users/user where $x/fname = 'admin' (:' and $x/password = 'abc' and $x/year = 1975 :) return $x

The query will be evaluated as:

xquery for $res in doc('users.xml')/users/user

where $x/fname = 'admin' return $x

When the query is executed, the attacker gains access to the application with the privileges of an administrator without submitting valid credentials. Thus, an attacker can bypass the authentication page using XQuery comment injection.

Example 5. *"xquery for $res in doc('students.xml')/students/student where $res/total <" + ulimit + "and $res/total >" + llimit + " return (replace value of node with 'First Class')"*

Attack String:
ulimit = 20 (:
llimit = :)

The query generated by the application is:

xquery for $res in doc('students.xml')/students/student where $res/total <20 (: and $res/total >:) return (replace value of node with 'First Class')

The query will be evaluated as:

xquery for $res in doc('students.xml')/students/student where $res/total <20 return (replace value of node with 'First Class')

When the query is executed, the students who have scored marks less than 20 will be assigned to "First Class". Thus, an untrusted user can violate the intended behavior of the application.

6. Implementation and evaluation

This section describes the changes made to the web application architecture for capturing the XQueries, the applications considered for testing the prototype and the evaluation results. The prototype XQueryFuzzer is built using Django web framework with Python, and uses PostgreSQL as the database. Redis, a data structure server, is used for storing the queries on a FIFO queue. The prototype is designed to work only for web applications that use BaseX as the XML database, and requires modification to BaseX client files based on the web technology used by the web application under test. The prototype is designed to run on both Linux and Windows operating systems.

6.1. Query capturing mechanism

Fig. 5 represents the modification made to the basic architecture of the web application to implement the proposed approach. The proposed approach requires access to the queries exchanged between the web application server and the database for constructing query models. The queries executed successfully are captured by modifying the database driver files specific to the native XML database used by the web application under test. The captured queries are placed on a FIFO queue using Redis server, which is a data structure server. Thus, the proposed approach provides a layer of abstraction between the application server and database server for capturing the queries.

The following steps implement the proposed approach:

(i) The prototype developed based on the proposed approach takes as primary input, the seed URLs which include the index page of the web application under test. Using each seed URL as a starting point, the prototype scans all other web pages of the application to identify forms. The forms contain input fields which are the injectable points of the application. These forms are stored in the database of the prototype.

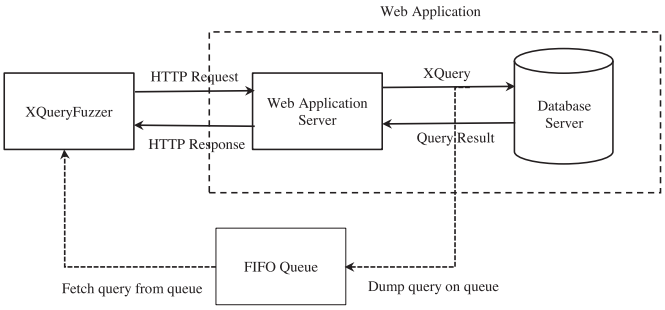

Fig. 5. Query capturing mechanism.

(ii) The details of the forms stored in the database are retrieved, populated with valid inputs, and submitted to the web application server using an HTTP request. The XQueries generated by the web application server on processing the request are sent to the XML database server for execution. These queries are intercepted and placed in a FIFO queue for constructing query models. The queries are fetched from the queue sequentially and assigned a query ID which is a simple iterator. For instance, four queries are found on a queue after form submission, and are assigned IDs 1,2,3,4. These queries are modeled and the models along with the query ID are stored in the database of the prototype. After the XML database server returns the results to the web application server, the server builds the HTTP response and sends it to the prototype.

(iii) The queue is cleared and the same form is then submitted with attack strings as input in the form fields. XQueries generated are captured and placed on the FIFO queue. The results returned by the database server are processed by the web application server to form the HTTP response which is forwarded to the prototype. If the response is successful and the length of the FIFO queue is the same as that of the length of the queue obtained when the form was submitted with valid inputs, then the queries are fetched from the queue one after the other. When the first query is fetched from the queue, it is modeled and is compared against the learnt model having query ID as '1' fetched from the prototype's database. A vulnerability is reported, if the models are dissimilar. Similarly, each of the four queries in the queue is fetched and compared. This step is repeated with each category of attack strings generated from the attack grammar.

(iv) Steps (ii) and (iii) are repeated for each form stored in the database.

(v) A vulnerability report is generated when all the web forms are processed, and the discovered vulnerable forms are listed.

6.2. Testbed applications

The test suite developed by Halfond and Orso (2005) is used for testing the proposed approach. Applications such as BookStore, Classifieds, Employee Directory, and Events are used for evaluation. These applications use a relational database at the backend for storing the data. For evaluation of our approach, the database of these applications is modified to BaseX (native XML database), and SQL queries in the application are replaced with XQueries. The details of test applications gathered from Halfond and Orso (2005) are given in Table 3. Table 3 gives the name of the application, description, technology involved, and the number of locations that issue XQueries (hotspots).

Table 3
Test application details.

Test 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

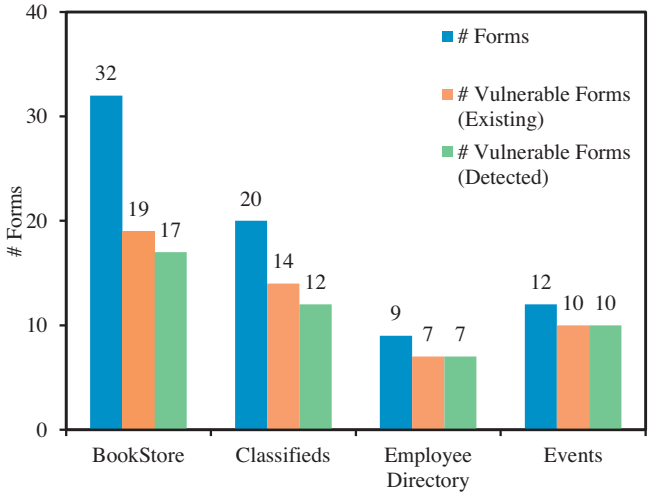

Fig. 6. Detection rate of the prototype.

The prototype 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.

6.3. Results and discussions

The prototype is evaluated along two dimensions: (1) experimental effectiveness, and (2) comparison with Zed Attack Proxy (ZAP) (ZAP, 2016), an open-source vulnerability scanner. The evaluation results are discussed below.

Table 4 describes the number of attack requests submitted to the application, number of successful attacks, number of forms existing in the application, number of vulnerable forms existing in the application, number of vulnerable forms detected by the prototype, and the detection rate of the prototype. The table shows that for applications BookStore and Classifieds, two forms are missed from being identified as vulnerable. The forms are missed because of the validation placed on few parameters on the server-side. As the developed crawler is not aware of the context of the application, valid values could not be placed for the parameters that have restrictions on it, and hence both legitimate and illegitimate requests fail for both the cases. All the existing vulnerable forms are identified by the prototype for applications Events and Employee Directory. On the whole, the detection rate of the prototype is found to be 93.8%. Fig. 6 depicts the results obtained.

Table 5 lists the number of vulnerabilities existing in the test application, the number of vulnerabilities detected by the prototype, and the number of false positives and false negatives. On analyzing the results, it is found that the prototype detects maximum number of vulnerabilities present in the test applications with few false positives.

A vulnerability report is generated that specifies the number of URLs processed and the number of vulnerabilities found in the application. Under each category of attack, it reports the number of vulnerabilities found and the related details. The details include the URL and the specific form where the vulnerability is found along with the attack string used to detect it as given in Table 6. This will be useful for the web application developer to identify the injection point as well as the type of vulnerability that needs to be addressed. For example, if a tautology vulnerability is reported for a form on a given URL, it can be fixed by the web application developer by parameterizing the query.

6.3.1. False positives and false negatives

The proposed approach results in false negatives in two cases. The first case is when the learner fails to generate valid query models. For example, consider an input field which takes a numeric value within the range of 0–100. As the developed crawler does not take into account the constraints on the input field, the request submitted to the application will not be executed successfully, and hence legitimate query models will not be generated. On the other hand, if the attack string substituted on the same page succeeds, which would be very rare, then illegitimate query models would be generated. The proposed approach reports vulnerabilities by comparing the length of the queue generated during the learning and discovery phase, and this attack would be missed from being identified, since the number of queries generated during learning is zero and during discovery is one. Thus, failing to generate legitimate query models results in false negatives.

The second case is where a vulnerable web page is missed from being identified during crawling phase. Consider a web page named *"items purchased"*. If this web page is visited by the crawler before adding items to the cart, then there would not be any hyperlinks 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.

Our approach results in false positives in the following scenario: the number of query models generated during the learning and discovery phase is the same, but the queries executed are actually different. For example, consider a case where an HTTP request executes three queries for a valid payload submission. When proper sanitization mechanisms are employed by the application, substitution of malicious parameters in the HTTP request fails to execute the illegitimate queries, and hence the application redirects the HTTP response to some other web page that executes three queries as well. Here, the number of queries executed during valid payload submission and invalid payload submission are the same, but the queries are actually different resulting in different query models being compared. The prototype reports vulnerabilities as the query models generated are different resulting in false positives. The developed prototype reported two false positives on the considered test applications for the above-mentioned scenario.

Table 7 gives the number of false positives and false negatives reported by the prototype, and the same is depicted in Fig. 7.

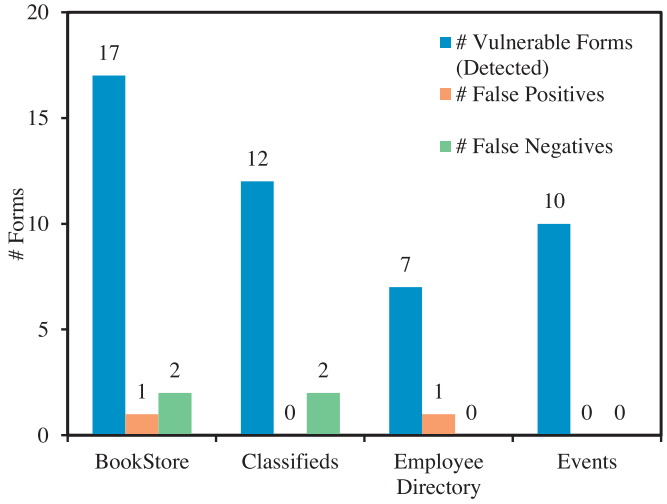

Fig. 7. Effectiveness of the prototype.

The prototype resulted in two false positives and four false negatives. The prototype reported two false positives: one in application BookStore, and the other in Employee Directory. The reason behind this is the number of queries executed during legitimate and illegitimate requests accidentally matches, even though the illegitimate request was not processed successfully. False positives can be eliminated by comparing the HTTP responses obtained for legitimate and illegitimate requests of the application. The reason behind false negatives is explained below. In case of application BookStore, web page *ShoppingCartRecord.jsp* is not visited by the crawler as the parent page *ShoppingCart.jsp* did not have any items that provide hyperlinks to ShoppingCartRecord page during the visit. Vulnerabilities in three other pages namely, *OrdersRecord.jsp, BookMaint.jsp* and *BookDetail.jsp*, are missed from being identified since they have constraints on one of the text fields. Similarly, application Classifieds has one page *MyAdRecord.jsp* which has a constraint on one of the text fields, and hence is missed from being identified. The false negatives can be eliminated by using a context-aware crawler, and false positives can be eliminated by comparing the HTTP response of the application.

The results are related to the specific test applications considered and cannot be generalized to other real-world web applications. Even though more experimentation is required before drawing definitive conclusions on the effectiveness of the approach, the results we obtained so far are promising.

6.3.2. Comparison with ZAP

Table 8 gives the results of the proposed approach as compared to the open-source penetration testing tool ZAP (2016) of OWASP. The table shows the number of vulnerable parameters identified by the prototype and the penetration testing tool, ZAP. Column 3 lists the number of parameters that are vulnerable to XQuery injection,

Table 4
Evaluation results.

Test Application	#Attack requests	#Successful attacks	#Forms	#Vulnerable forms (Existing)	#Vulnerable forms (Detected)	Detection rate (%)
BookStore	726	235	32	19	17	89.47
Classifieds	528	125	20	14	12	85.71
Employee directory	286	111	9	7	7	100
Events	396	157	12	10	10	100
Total	1936	628	73	50	46	93.80

Table 5
Number of vulnerabilities existing in test web applications and detected by the prototype.

Type of injection attack	BookStore				Classifieds				Emp. directory				Events			
	E	D	FP	FN	E	D	FP	FN	E	D	FP	FN	E	D	FP	FN
Tautology injection	21	20	1	2	15	13	0	2	7	7	0	0	10	10	0	0
Comment injection	14	15	2	1	9	8	0	1	3	5	2	0	3	3	0	0
Meta character ⟩ injection	21	19	0	2	15	13	0	2	7	7	0	0	10	10	0	0
Meta character & injection	0	3	3	0	0	0	0	0	0	2	2	0	0	0	0	0
Meta character ⟨ injection	14	15	2	1	9	8	0	1	3	5	2	0	3	3	0	0
Tag injection with CDATA	5	5	1	1	2	1	0	1	4	4	0	0	7	7	0	0
CDATA end character sequence injection]]⟩	16	14	0	2	12	10	0	2	7	7	0	0	10	10	0	0
Tag injection	21	19	0	2	15	13	0	2	7	7	0	0	10	10	0	0
Special character (single/double quotes) injection	12	11	1	2	7	5	0	2	8	8	0	0	14	14	0	0
Alternate encoding injection	16	14	0	2	12	10	0	2	7	7	0	0	10	10	0	0
Evaluation function injection	16	15	1	2	10	8	0	2	7	7	0	0	10	10	0	0
External entity injection	0	3	3	0	0	0	0	0	0	2	2	0	0	0	0	0

E – Vulnerabilities Existing, D – Vulnerabilities Detected, FP – False Positives, FN – False Negatives.

Table 6
Details of a vulnerability provided in vulnerability report.

Parameter	Details
Description:	Tautology Attack
URL:	http://127.0.0.1:8000/login/
Form name:	login
Method:	POST
Action:	http://127.0.0.1:8000/login/
Malicious Query:	for $res in doc('bookstorexmldb')/bookstore/members/member where $res/member_login = '1WR3Cn' or 1 = 1 or 'd3S1tmCKAsH' return $res

Table 7
Effectiveness of the prototype.

Test application	# Vulnerable forms (Detected)	# FP	# FN
BookStore	17	1	2
Classifieds	12	0	2
Employee Directory	7	1	0
Events	10	0	0

FP – False Positives, FN – False Negatives.

Table 8
Comparison with ZAP.

Test application	#Vulnerable forms		
	Existing	Detected by Prototype	Detected by ZAP
BookStore	19	17	11
Classifieds	14	12	9
Employee Directory	7	7	6
Events	10	10	6

whereas column 4 lists the number of parameters that are vulnerable to SQL injection. Even though ZAP identifies SQL injection vulnerabilities, the parameters that are placed in the SQL query and XQuery are the same. From the table, it can be inferred that our prototype identifies more vulnerabilities compared to ZAP.

6.3.3. Advantages and extensions

The advantages of the proposed approach are as follows:

- Identifies various types of XQuery injection attacks using a fully automated methodology.
- Uses a black-box approach, and hence web application source code is not required.
- Identifies known attacks, but in case a new attack is to be considered, a regular expression can be framed representing the attack and added as a production to the attack grammar. Hence, this approach is flexible and easily extendable for new attack vectors.

However, few limitations exist which can be overcome as part of further work. The extensions that can be made to the proposed approach are as follows:

- XQuery model construction needs to be updated as and when new versions of the language emerge.
- The attack grammar needs to be updated with additional productions, when new types of vulnerabilities are discovered.
- A context-aware crawler can be utilized to eliminate false negatives.
- The HTTP response obtained for both valid and invalid payload submission can be compared to eliminate false positives.

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. Other types of vulnerabilities such as those leading to blind XPath injection attack are not addressed by the proposed approach. Different from other types of XML injection attacks, blind XPath injection is a special kind of injection which focuses on extraction of information about data stored in the XML document by asking a series of true/false questions (i.e., booleanized queries) (Klein, 2005). The types of injection addressed by the proposed approach are straightforward, and either extract a piece of information or inject untrusted information into the database, whereas blind XPath injection attack employs two methods such as XPath crawling and booleanization for inferring information from the XML document. Each injection query extracts a single bit of information from the XML document, and the whole of the XML document is inferred from unlimited number of queries, which is conceptually different from the proposed approach, and hence it is not addressed as part of this work.

7. Conclusion and further work

XML injection has become a critical vulnerability with the increased use of XML databases by web applications. XML injection vulnerabilities need to be detected and corrected so that the web application is secure against various types of XQuery injection attacks. Hence, a prototype for identification of XQuery injection vulnerabilities has been developed. The prototype employs a crawler for identifying all possible points of injection, which are filled with malicious strings generated using an attack grammar. The attack grammar is capable of generating various types of XML injection strings to detect the vulnerabilities. The queries submitted to the XML database on submission of the web forms are intercepted, and

query models are constructed. The query models generated during normal and attack executions are compared for identifying possible points of injection, and vulnerable points of injection are finally reported. 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 attacks.

Future work in this area could be on improvements in the performance of the prototype by parallelizing the task of submission of the existing forms with valid/invalid inputs, and by building appropriate query models while scanning the application under test so as to make it scalable for larger web applications. Also, the proposed approach for identifying XQuery injection vulnerabilities can be extended from web applications using native XML databases to web services.

Acknowledgments

This work is 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.

References

Acunetix, 2016. Acunetix. http://www.acunetix.com/.
Al-Hamdani, W.A., 2010. XML security in healthcare web systems. In: 2010 Information Security Curriculum Development Conference. ACM, New York, NY, USA, pp. 80–93. doi:10.1145/1940941.1940961.
Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H., 2009. Effective detection of SQL/XPath injection vulnerabilities in web services. In: IEEE International Conference on Services Computing. IEEE, pp. 260–267. doi:10.1109/SCC.2009.23.
Antunes, N., Vieira, M., 2011. Enhancing penetration testing with attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. In: Services Computing (SCC), 2011 IEEE International Conference on. IEEE, pp. 104–111.
Arachni, 2016. Arachni - web application security scanner framework. http://www.arachni-scanner.com/.
Asmawi, A., Affendey, L.S., Udzir, N.I., Mahmod, R., 2012. Model-based system architecture for preventing XPath injection in database-centric web services environment. In: 7th International Computing and Convergence Technology (ICCCT). IEEE, pp. 621–625.
Auger, R., 2010. XQuery injection. http://projects.webappsec.org/w/page/13247006/XQueryInjection.
BaseX. BaseX-the XML database. http://basex.org/.
Baviskar, S., Thilagam, P.S., 2011. Protection of web users privacy by securing browser from web privacy attacks. Int. J. Comput. Technol. Appl. 2, 1051–1057.
Bisht, P., Madhusudan, P., Venkatakrishnan, V., 2010. Candid: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. ACM Trans. Inf. Syst. Secur. (TISSEC) 13 (2), 14:1–14:39.
Bourret, R., 2009. Going native: Use cases for native XML databases. http://www.rpbourret.com/xml/UseCases.htm.
Bravenboer, M., Dolstra, E., Visser, E., 2007. Preventing injection attacks with syntax embeddings. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering. ACM, pp. 3–12.
Buehrer, G., Weide, B.W., Sivilotti, P.A., 2005. Using parse tree validation to prevent SQL injection attacks. In: Proceedings of the 5th International Workshop on Software Engineering and Middleware. ACM, pp. 106–113.
Chan, G.-Y., Lee, C.-S., Heng, S.-H., 2013. Discovering fuzzy association rule patterns and increasing sensitivity analysis of XML-related attacks. J. Netw. Comput. Appl. 36 (2), 829–842.
Chandrashekhar, R., Mardithaya, M., Thilagam, P.S., Saha, D., 2012. SQL injection attack mechanisms and prevention techniques. In: Advanced Computing, Networking and Security. Springer, pp. 524–533.
Chaudhri, A., Zicari, R., Rashid, A., 2003. XML Data Management: Native XML and XML Enabled DataBase Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
convert, 2016. Conversion module. http://docs.basex.org/wiki/Conversion_Module.
Deepa, G., Thilagam, P.S., 2016. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. Inf. Softw. Technol. 74, 160–180. http://dx.doi.org/10.1016/j.infsof.2016.02.005.
Django. Django-the web framework for perfectionists with deadlines. https://www.djangoproject.com/.
eval, 2016. Xquery module. http://docs.basex.org/wiki/XQuery_Module.
eXistDB. eXistDB - the open source native XML database. http://exist-db.org/exist/apps/homepage/index.html.
Forbes, T., 2014. Exploiting XPath injection vulnerabilities with XCat. http://tomforb.es/exploiting-xpath-injection-vulnerabilities-with-xcat-1.
Gopalakrishnan, G., 2006. Computation Engineering-Applied Automata Theory and Logic. Springer.
Gordeychik, S., 2010. Web application security statistics. The Web Application Security Consortium. http://projects.webappsec.org/w/page/13246989/Web Application Security Statistics.
Groppe, J., Groppe, S., 2008. Filtering unsatisfiable XPath queries. Data & Knowledge Engineering 64 (1), 134–169.
Halfond, W., Viegas, J., Orso, A., 2006. A classification of SQL-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, pp. 65–81.
Halfond, W.G., Orso, A., 2005. Amnesia: analysis and monitoring for neutralizing SQL-injection attacks. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, pp. 174–183.
Hirsch, F., 2002. Getting started with XML security. http://www.sitepoint.com/getting-started-xml-security/.
Huang, Y., 2003. Safeguarding a native XML database system. https://www.cs.auckland.ac.nz/courses/compsci725s2c/archive/termpapers/725huang.pdf.
Huang, Y.-W., Tsai, C.-H., Lin, T.-P., Huang, S.-K., Lee, D., Kuo, S.-Y., 2005. A Testing framework for web application security assessment. Comput. Netw. 48 (5), 739–761.
Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., Kuo, S.-Y., 2004. Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th international conference on World Wide Web. ACM, pp. 40–52.
Jang, Y.-S., Choi, J.-Y., 2014. Detecting SQL injection attacks using query result size. Comput. Secur. 44, 104–118.
JSpider, 2003. Jspider. http://j-spider.sourceforge.net/.
Klein, A., 2005. Blind XPath Injection. Technical Report. Watchfire Corporation. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.2076&rep=rep1&type=pdf
Kosuga, Y., Kernel, K., Hanaoka, M., Hishiyama, M., Takahama, Y., 2007. Sania: Syntactic and semantic analysis for automated testing against SQL injection. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007. IEEE, pp. 107–117.
Lampesberger, H., 2013. A grammatical inference approach to language-based anomaly detection in XML. In: Eighth International Conference on Availability, Reliability and Security (ARES). IEEE, pp. 685–693.
Laranjeiro, N., Vieira, M., Madeira, H., 2009. Protecting database centric web services against SQL/XPath injection attacks. In: Database and Expert Systems Applications. Springer, pp. 271–278.
Lee, I., Jeong, S., Yeo, S., Moon, J., 2012. A novel method for SQL injection attack detection based on removing SQL query attribute values. Math. Comput. Model. 55 (1–2), 58–68. Advanced Theory and Practice for Cryptography and Future Security
Lee, K.K.-Y., Tang, W.-C., Choi, K.-S., 2013. Alternatives to relational database: Comparison of NoSQL and XML approaches for clinical data storage. Comput. Methods and Programs in Biomed. 110 (1), 99–109.
Li, X., Xue, Y., 2014. A survey on server-side approaches to securing web applications. ACM Comput. Surveys 46 (4), 54:1–54:29.
Liu, A., Yuan, Y., Wijesekera, D., Stavrou, A., 2009. SQLProb: A proxy-based architecture towards preventing SQL injection attacks. In: Proceedings of the 2009 ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 2054–2061.
Liu, Z.H., Murthy, R., 2009. 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. doi:10.1109/ICDE.2009.18.
van der Loo, F., 2011. 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.
MarkLogic. MarkLogic: Enterprise NoSQL database. http://www.marklogic.com/.
Menahem, E., Schclar, A., Rokach, L., Elovici, Y., 2012. Securing your transactions: Detecting anomalous patterns in XML documents. arXiv preprint arXiv:1209.1797.
Mitropoulos, D., Karakoidas, V., Louridas, P., Spinellis, D., 2011. Countering code injection attacks: a unified approach. Inf. Manage. Comput. Secur. 19 (3), 177–194.
Mitropoulos, D., Karakoidas, V., Spinellis, D., 2009. Fortifying applications against XPath injection attacks. In: Proceedings of the 4th Mediterranean Conference on Information Systems, pp. 1169–1179.
OWASP, 2014. Testing for XML injection. Technical Report. https://www.owasp.org/index.php/Testing_for_XML_Injection_OTG-INPVAL-008.
Pavlovic-Lazetic, G., 2007. Native XML databases vs. relational databases in dealing with XML documents. Kragujevac J. Math 30, 181–199.
PostgreSQL. PostgreSQL-the world’s most advanced open source database. http://www.postgresql.org/.
Redis. Redis. http://redis.io/.
Rosa, T.M., Santin, A.O., Malucelli, A., 2013. Mitigating XML injection 0-day attacks through strategy-based detection systems. Secur. Privacy IEEE 11 (4), 46–53. doi:10.1109/MSP.2012.83.
SANS, 2011. CWE/SANS TOP 25 Most Dangerous Software Errors. Technical Report.
Scholte, T., Robertson, W., Balzarotti, D., Kirda, E., 2012. Preventing input validation vulnerabilities in web applications through automated type analysis. In: 2012 IEEE 36th Annual Computer Software and Applications Conference (COMPSAC), pp. 233–243.
Shahriar, H., Zulkernine, M., 2011. Taxonomy and classification of automatic monitoring of program security vulnerability exploitations. J. Syst. Softw. 84 (2), 250–269.

Shahriar, H., Zulkernine, M., 2012. Mitigating program security vulnerabilities: approaches and challenges. ACM Comput. Surveys 44 (3), 11:1–11:46.
Siegel, E., Retter, A., 2014. eXist. O'Reilly Media. https://www.safaribooksonline.com/library/view/exist/9781449337094/ch01.html.
Staken, K., 2001. Introduction to native XML databases. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html.
Su, Z., Wassermann, G., 2006. The essence of command injection attacks in web applications. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 372–382.
Symantec, 2014. Symantec Internet Security Threat Report:Vol. 19. Technical Report. Symantec Corporation. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf.
Thomas, S., Williams, L., 2007. Using automated fix generation to secure SQL statements. In: Third International Workshop on Software Engineering for Secure Systems. SESS '07: ICSE Workshops 2007., p. 9.
Top10, 2013. Top 10 2013-top 10. https://www.owasp.org/index.php/Top_10_2013-Top_10.
de la Torre, I., Díaz, F.J., Antón, M., Díez, J.F., Sainz, B., López, M., Hornero, R., López, M.I., 2011. Choosing the most efficient database for a web-based system to store and exchange ophthalmologic health records. J. Med. Syst. 35 (6), 1455–1464.
Truelove, J., Svoboda, D., 2011. IDS09-J. Prevent XPath injection. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=61407250.
Urgun, B. Wivet: Web input vector extractor teaser. https://github.com/bedirhan/wivet.
Valeur, F., Mutz, D., Vigna, G., 2005. A learning-based approach to the detection of SQL attacks. In: Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer-Verlag, Berlin, Heidelberg, pp. 123–140.
w3af. w3af. http://w3af.sourceforge.net.
W3C. XML security. https://www.w3.org/standards/xml/security.
Wapiti, 2013. The web-application vulnerability scanner. http://wapiti.sourceforge.net/.
Wassermann, G., Su, Z., 2007. Sound and precise analysis of web applications for injection vulnerabilities. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 32–41.
WebCruiser, 2011. Webcruiser-web vulnerability scanner. http://www.ehacking.net/2011/07/webcruiser-web-vulnerability-scanner.html.
WebSPHINX, 2002. WebSPHINX: A personal, customizable web crawler. http://www.cs.cmu.edu/~rcm/websphinx/.
Xie, Y., Aiken, A., 2006. Static detection of security vulnerabilities in scripting languages. In: USENIX Security, Vol.6, pp. 179–192.
XMLMao, 2012. XMLMao. https://www.soldierx.com/tools/XMLmao.
XPath-Injection, 2015. XPath injection. https://www.owasp.org/index.php/XPATH_Injection.
XXE, 2016. XML external entity (XXE) processing. https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing.
ZAP, 2016. OWASP zed attack proxy project. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

Nushafreen Palsetia received her B.E. degree in Computer Engineering in 2009 from University of Mumbai, India and M.Tech. degree in Computer Science and Engineering in 2015 from National Institute of Technology Karnataka, Surathkal, India. She is currently working as a Software Development Engineer with Dell R&D in the area of Storage Engineering.

G. Deepa is currently a Ph.D. Scholar with the Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal, India. She received her M.E. degree in Computer Science and Engineering from Anna University, Chennai in 2012. Her research interests include Web application security, Data mining and Software testing.

Furqan Ahmed Khan received his M.Tech. degree in Software Engineering from Galgotias University, Greater Noida, India in 2014. He is currently working as a project scientist in an R&D project supported by the Ministry of Communications and Information Technology, Government of India at National Institute of Technology Karnataka, Surathkal, India. His research interests include Web application security and Web application architecture.

P. Santhi Thilagam is currently an Associate Professor in the Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal (NITK), India. She received her Ph.D. in Computer Science and Engineering from NITK in 2008. Her research interests include Distributed data management, Graph mining and Data security. She has published widely in various database and data mining conferences. She was the recipient of the BITES best PhD thesis award for the year 2009 in Computer Science and Engineering category. More details of her research can be obtained at http://www.cse.nitk.ac.in/faculty/p-santhi-thilagam.

Alwyn R. Pais is currently an Assistant Professor in the Department of Computer Science and Engineering, National Institute of Technology Karnataka (NITK), Surathkal, India. He completed his B.Tech. (CSE) from Mangalore University, India, M.Tech. (CSE) from IIT Bombay, India and Ph.D. from NITK, Surathkal, India. His area of interests include Information Security, Image Processing, Computer Vision, Wireless Sensor Networks, and Internet of Things (IoT).