Available online at www.sciencedirect.com

ScienceDirect

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

Computers & Security

A deliberately insecure RDF-based Semantic Web application framework for teaching SPARQL/SPARUL injection attacks and defense mechanisms

Hira Asghar [a], *Zahid Anwar* [a,b,*], *Khalid Latif* [a]

[a] *School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad, Pakistan*

[b] *Department of Software and Information Systems, University of North Carolina at Charlotte, Charlotte, NC, USA*

ARTICLE INFO

Article history:
Received 12 July 2014
Received in revised form 8 November 2015
Accepted 24 November 2015
Available online 18 December 2015

Keywords:
Semantic web
SPARQL
SPARUL
Web application firewall
Vulnerable web applications

ABSTRACT

The Semantic Web uses the Resource Description Framework (RDF) and the Simple Protocol and Query/Update Languages (SPARQL/SPARUL) as standardized logical data representation and manipulation models allowing machines to directly interpret data on the Web. As Semantic Web applications grow increasingly popular, new and challenging security threats emerge. Semantic query languages owing to their flexible nature introduce new vulnerabilities if secure programming practices are not followed. This makes them prone to both existing attacks such as command injection as well as novel attacks, making it necessary for application developers to understand the security risks involved when developing and deploying semantic applications. In this research, we have analyzed and categorized the possible SPARQL/SPARUL injection attacks to which semantic applications are vulnerable. Moreover, we have developed a deliberately insecure RDF-based Semantic Web application, called SemWebGoat – inspired by the open source vulnerable web application, WebGoat – which offers a realistic teaching and learning environment for exploiting SPARQL/SPARUL oriented injection vulnerabilities. With the aim of teaching both developers and web administrators the art of protecting their Semantic Web applications, we have implemented web application firewall (WAF) rules using the popular open-source firewall – ModSecurity – and extended some penetration testing tools to detect and mitigate SPARQL/SPARUL injections. For the evaluation, we conducted a user study to determine the usability of SemWebGoat attack lessons as well as a detection rate and false alarm analysis of our proposed firewall rules based on OWASP top-ten attack dataset. The results of the user study conclude that web developers are not normally familiar with the injection vulnerabilities demonstrated. The positive test results of our ModSecurity rule set show that it a suitable defense mechanism for protecting vulnerable Semantic Web application against injection attacks.

* *Corresponding author.* Tel.: 19802330150; fax: 18554206869.
E-mail addresses: zahid.anwar@seecs.edu.pk; zanwar@uncc.edu (Z. Anwar).

http://dx.doi.org/10.1016/j.cose.2015.11.004
0167-4048/

1. Introduction

The Semantic Web, considered to be the next generation of the Web, has been expanding rapidly in recent years. After the publication and recommendation of several standards by the World Wide Web Consortium (W3C) (W3C, 2015), the Internet industry is rapidly exploring the benefits of Semantic Web technologies. Semantic Web has been broadly adopted by a large number of domains including finance (Cheng et al., 2009; Zhang et al., 2000), business (Hepp, 2008), medicine (Pisanelli, 2004), and e-learning (Isotani et al., 2013), amid raising concerns regarding the security of Semantic Web data. Semantic Web uses a standardized graph data model namely Resource Description Framework (RDF) (Manola et al., 2004), to make its data machine-readable. Semantic Web data are a collection of RDF statements known as triples in RDF terminology, each consisting of three parts: subject, predicate and object. RDF triples are stored and managed by different RDF data management systems including Jena (McBride, 2002), Sesame (Broekstra et al., 2002), Openlink Virtuoso (Erling, 2001) and 3Store (Harris, 2003). Simple Protocol and RDF Query Language (SPARQL) (Hommeaux and Seaborne, 2008) is the standard query language for RDF data recommended by W3C. SPARQL/Update (SPARUL) (Seaborne et al., 2008) is an extension to the standard SPARQL query language that is used to insert, delete and update RDF data.

A number of security incidents have taken place in the recent past that have proven that the previous query languages (Berglund et al., 2010; Database Language SQL, Part 2: Foundation (SQL/Foundation), 1999; Sermersheim, 2006) such as SQL, XPath, LDAP, etc., are highly vulnerable to attacks based on non-sanitized user inputs. In such attacks, the attacker directly concatenates the malicious query string with the input to take over control of the application to achieve desired results. The need to create awareness among the web developer community about security vulnerabilities is already established (Marsa-Maestre et al., 2013; Shaw et al., 2009). In order to educate and develop the required skill-set in application developers so that they can practice writing "bug" free code when using query languages riddled with vulnerabilities, a number of deliberately insecure web applications such as HacmeBank (OWASP, 2015c) and WebGoat (OWASP, 2015a) were created and different safeguard measures (ORACLE, 2015; OWASP, 2015b, 2015f) were proposed. Not unlike their predecessors, SPARQL/SPARUL query languages are also vulnerable to various injection attacks. It is worth mentioning that some researchers have tried to create awareness of security issues in Semantic Web applications in the past. Thuraisingham conducted an early research (Thuraisingham, 2005) on the security needs of the Semantic Web, whereby the importance of implementing security mechanisms in different layers of the application was examined. Agarwal and Sprick (Schmidt et al., 2009) identified and developed access control policies and their corresponding mechanisms for Semantic Web services. In other related research (Orduna et al., 2010; Yang et al., 2011), the authors identified basic vulnerabilities in Semantic Web query and update languages and discussed possible mitigation solutions (MORE LAB, 2010; Yang et al., 2011). Therefore, to the best of our knowledge, while researchers have expressed security concerns in the Semantic Web to a limited extent in the past, no systematic efforts exist to develop tools or frameworks that enumerate specific vulnerabilities, their variations and mitigation strategies in real applications as has been the case for previous versions of the Web. Due to this, a lack of awareness exists in the developer community regarding vulnerabilities in Semantic Web applications. Our goal in this work, therefore, focuses on providing a realistic educational environment, henceforth called SemWebGoat named after the popular WebGoat framework, where application developers can safely experiment with and exploit vulnerabilities in Semantic Web applications and learn about their associated safeguard measures.

To examine the vulnerabilities in SemWebGoat, five different scanners and penetration testing tools have been employed, and their accuracy in detecting the vulnerabilities in Semantic Web applications has been evaluated. The major contributions of our research are as follows. (i) Analysis and categorization of attacks specific to Semantic Web query and update languages. (ii) Development of a deliberately insecure Java EE based Semantic Web application so that users, developers, penetration testers and students alike, can learn and practice exploiting and patching Semantic Web application security holes in a safe and legal environment. (iii) Development of a testing platform for security professionals to test their security controls. Security professionals frequently need to measure the performance and accuracy of their test tools against a platform known to be vulnerable to ensure that they perform to a satisfactory extent. (iv) Demonstration of how attacks may be mitigated using web application firewalls (WAFs) for protecting vulnerable Semantic Web applications without having to rewrite their source. A sample customized rule set has been implemented using the popular open-source WAF-ModSecurity as way of illustration. (v) Extension of some well known penetration testing scanners to support SPARQL/SPARUL injection attacks with the aim of assisting developers in testing arbitrary Semantic Web application deployments.

The rest of the paper is structured as follows: The next section presents a background on relevant tools and Semantic Web languages that have been used in the implementation of SemWebGoat and reviews the related research work in our domain. Section 3 details the architecture and design of the proposed framework including the RDF data model. Section 4 explains the various security lessons along with examples of injection attacks on Semantic Web applications. Section 5 describes the implementation of a customized WAF rule set to prevent the attacks listed in the previous section. Section 6 presents the evaluation criteria and results. Finally, the last section concludes the entire study and outlines future work directions.

2. Background and related work

2.1. *Semantic web languages*

The term "Semantic Web" is often used to refer to the technologies and formats that enable it. These technologies are specified as W3C standards and are combined in order to provide descriptions that could enhance or replace the content of current Web documents to make the Web more intelligible. SemWebGoat has been specifically designed as a Semantic

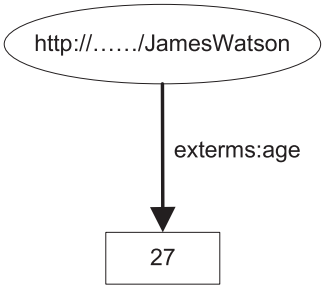

Fig. 1 – An Example of a Simple RDF Graph.

Web application, and so it employs a collection of Semantic Web technologies and formats for its implementation, which are summarized below.

RDF (Manola et al., 2004) is a language used for representing Semantic Web data. Each RDF statement consists of a *subject*, *predicate* (also called *property*) and *object* triple, where the subject and object are used to represent any two things in the world and a predicate is used to define their relationship. The following conditions apply to RDF data:

- A subject can only be a Uniform Resource Identifier (URI) or a blank node.
- A predicate/property can only be a URI.
- An object can be of any type, such as a blank node, a URI or a literal.

RDF statements are best represented in the form of RDF graphs. The arc in the RDF graph is labeled as a predicate and starts from a subject node and ends at the object node. For example, the RDF graph in Fig. 1 states "The value of James Watson's age is 27", where James Watson is a subject, age represents the predicate, and 27 represents the object value. As a complete URI description is usually rather long and cumbersome to read, the CURIE (Birbeck, 2007) form maybe used as in the case of the predicate (exterms:age) in this example. The part before the ":" represents a namespace and is known as namespace prefix. The *nsprefix:localname* form corresponds to the URI of the namespace that is concatenated with the localname.

The SPARQL Protocol and RDF Query Language (SPARQL) (Clark et al., 2008; Hommeaux and Seaborne, 2008) is a semantic query language for databases, able to retrieve and manipulate data stored in RDF format. The SPARQL Protocol uses WSDL 2.0 (Chinnici et al., 2007) to describe a means for conveying SPARQL queries to a SPARQL query processing service and returning the query results to the entity that requested them. The SPARQL Protocol has been designed for compatibility with the SPARQL query language that is used for querying the RDF graphs. SPARQL has the following four forms:

- **SELECT:** Returns the variables and their bindings in the result.
- **CONSTRUCT:** Returns an RDF graph by transforming the results.
- **ASK:** Returns the result in a Boolean form. It states whether the query matches or not.
- **DESCRIBE:** Returns an RDF graph that contains the description of the resources found.

A SPARQL query consists of two parts. The first part identifies the variables to appear in the query results and the WHERE part provides a graph pattern to match against the RDF data graph. A simple sample query is shown in Listing 1 that has been formulated to return the age of James Watson. A variable is indicated by the "?" prefix and will return the binding for ?age.

Listing 1: SPARQL Query for Figure 1

```
SELECT ?age
WHERE {
        exterms:JamesWatson exterms:age ?age .
         }
```

The standard language SPARUL (Seaborne et al., 2008) is an extension to SPARQL and is used for performing updates to RDF graphs. It can be used for performing the following tasks:

- Inserting triples in the RDF model stored in a database.
- Deleting triples from the RDF model stored in a database.
- Clearing an RDF model in a database.
- Loading an RDF model in a database.
- Moving, copying and adding the content of one RDF model to another.
- Dropping an RDF model from a database.
- Executing a number of operations in a single action.

A variety of Semantic Web tools including Jena (McBride, 2002), ARC (W3C, 2014), RDFLib (W3C, 2013) and Protege (2015) are available that assist in the development of Semantic Web applications. The Jena (McBride, 2002) framework, for instance, can be used for creating and querying a back-end RDF database. It includes an API that supports reading, processing and writing of RDF data into XML, N-triples, JSON, TriG, N-Quads and Turtle formats. The Jena framework also includes services that allow the publishing of RDF data to other applications by using various protocols including SPARQL. In Jena, an RDF graph is known as an RDF model and the model interface is used to represent it.

2.2. *Attacks in Semantic Web applications*

SQL injection attacks are considered as one the most serious threats for Web applications (OWASP, 2015e; InfoSec Institute, 2013) today. Any web application that receives unsanitized user inputs is susceptible due to inadequate validation in the

developer's code. In these crafted attacks, the user inputs are concatenated with SQL queries in such a way that they are unknowingly treated by the vulnerable application as a part of the underlying SQL code. As a result, this may allow an attacker to get complete access to the underlying database leading to possibly serious security breaches including loss of confidential data or information, identity theft and fraud. SQL injection therefore refers to a category of code-injection attacks that if utilized properly has the potential of damaging the very system that hosts the web application. "Injection through User Input" is one of the most basic types of injection attacks and there are a number of variations of this quoted in literature that allow the ability to gain unauthorized data access (Halfond et al., 2006). While researchers have proposed various solutions and defensive coding guidelines (Howard and LeBlanc, 2003) to protect web applications against these attacks, their countless variations limit the ability to reliably detect even a small subset of the possible injection attacks. The evolution of the Semantic Web and its ability to automatically manipulate huge volumes of data via new and complex tools and languages has further exacerbated both the detection accuracy and the impact mitigation of successful injection attacks. As the construction of applications using Semantic Web technologies is a relatively new trend, developers continue to employ classical query languages such as SQL and XPath with which they are familiar together with emerging ones in the same application. This causes Semantic Web applications to combine vulnerabilities of both worlds making them weaker in withstanding attacks as compared to their predecessors. Security issues of Semantic Web applications have been relatively unexplored. To a limited extent, researchers (MORE LAB, 2010; Orduna et al., 2010; SPARQL, 2012; Yang et al., 2011) have identified some of the new vulnerabilities in the past and emphasized the dearth of analysis tools (Middleton et al., 2015) in order to both protect semantic data from exposure as well as report on secure Semantic Web configurations to external trust services. However, we did not find any significant security assessment tools that address these vulnerabilities in a real world environment.

In Orduna et al. (2010), the authors presented three basic Semantic Web injection techniques: SPARQL injection, Blind SPARQL injection and SPARUL injection. In these attacks, SPARQL and SPARUL commands are injected from a web form into the database logic of an application to acquire unauthorized data access and to make unauthorized deletion and alteration of the data. Blind SPARQL injection is identical to normal SPARQL injection except that in this technique an attacker attempts to steal data by asking a series of true and false questions through SPARQL statements. Libraries for other query languages such as SQL provide tools to avoid code injection attacks. For example, the Java API supports prepared statements (ORACLE, 2015) or the use of parameterized queries (SQL, 2015a). Parameterized queries force the developer to first define the query code, and then subsequently pass in each parameter separately. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied. Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. Orduna et al. (MORE LAB, 2010) provided a solution called *ParameterizedString* for preventing SPARQL/SPARUL injection attacks. It works in the same way as prepared statements do for preventing SQL injection attacks. They also developed a patch for Pellet 1.5.1 and Jena 2.5.5 by adding support for the ParameterizedString object in QueryEngine, QueryFactory and UpdateFactory modules. The drawback of this solution is that it is only suitable when the Semantic Web applications are in the development phase because ParameterizedStrings are used in-line. In case the vulnerabilities are identified in the production phase, then the use of ParameterizedString would require code rewriting, which is not usually economical and is generally ignored if the development team is under pressure to quickly roll out the software product. Our research instead provides a solution that can externally address Semantic Web vulnerabilities and can protect applications without having to change any line of source code.

In Yang et al. (2011), the authors presented the classification and detection of possible SPARQL/SQL injection attacks for an RDBMS-based Semantic Web system. Such systems use both SPARQL and SQL as the query languages. RDBMS-based systems store data in a relational database for persistence and use a translator module to translate SPARQL queries to SQL. The injection attacks are therefore classified as either SPARQL-oriented or SQL-oriented. To detect these attacks, the authors proposed a parse tree based validation technique, which represents the syntactic structure of a string according to the grammar. The intended SQL and SPARQL queries are constructed through a programmer formulation code. The formulation code generates the hard-coded portion of the parse tree and the user supplied input portion is represented as empty leaf nodes that is later filled by the user supplied input. Consequently, if the user inputs do not store the content in the parse tree nodes as the intended queries do, then it implies that there is an injection attack in progress. In summary, this technique detects the SPARQL/SQL injection attack by comparing the intended parse tree with the resulting parse tree that is generated by the user supplied input. To validate the proposed technique, the authors developed a prototype system "SemGuard", which was used as a plug-in for a Java application. In this work, only basic SPARQL injection attacks have been discussed and no classification or solution for Blind SPARQL and SPARUL injection attacks has been provided. This motivated us to address the remaining SPARQL/SPARUL injection attacks as well and provide appropriate defense mechanisms for their mitigation.

Onofri and Napolitano (SPARQL, 2012) presented a motivating case-study depicting the dangers of basic SPARQL injection attacks in Semantic Web applications. For demonstrating the attack they developed a vulnerable Semantic login where the user was able to bypass the authentication without knowledge of the correct password. They recommended the use of parametric queries and data validation techniques as well as suggested that proper programming practices be used for authoring SPARQL code in a correct manner. They also addressed the same issues in the ARC2 library and demonstrated that ARC2 version v2011-12-01 and possibly the previous versions are vulnerable to Blind SQL Injection and Cross Site Scripting vulnerabilities. ARC (W3C, 2014) provides RDF and SPARQL support for PHP applications by storing triples into a MySQL database. It is possible to attempt Blind SQL Injection attack against RDB-based applications that use ARC by

injecting SQL commands in the SPARQL WHERE clause. These issues have been fixed by the ARC vendors in their latest version and users are recommended to update ARC2 to the latest release or manually fix the "ARC2_StoreEndpoint.php" and other files. This research did not propose any new safeguard mechanisms, which encouraged us to pursue this work on developing a deliberately vulnerable real-world Semantic application where users can test basic and advanced SPARQL injections attacks and experiment with the provided mitigation techniques.

2.3. *Deliberately insecure web applications*

WebGoat is among the most well known and widely used deliberately insecure web applications. It is maintained by OWASP and is designed to teach web application security lessons. The version WebGoat-5.3 in particular includes more than sixty lessons related to possible Web 2.0 attacks, which are split up into eighteen main categories depending on the nature of the threat. Each lesson contains a specific vulnerability that is supposed to be exploited by the user to complete the lesson plan. Some of the main lessons in this web application are cross site scripting, thread safety, SQL injection, hidden form fields, web services and weak session cookies. Users can observe cookies, parameters, and other data sent to and from the application by using a web proxy. For better understanding of security lessons users are provided with hints, source code and solutions. Some of the lessons in WebGoat-5.3 require third-party software such as WebScarab (OWASP, 2015i), Firebug (2015), IEWatch (IE, 2015) and Wireshark (2015) to effectively exploit the vulnerability demonstrated. For developing SemWebGoat the basic structure of interface design, database and lesson scenarios of WebGoat-5.3, have been followed. Users who are familiar with WebGoat's user interface have an easy transition to SemWebGoat as it provides a similar learning environment.

2.4. *Web application firewalls*

In the software development lifecycle, if vulnerabilities remain undiscovered in the design or testing phases and only manifest in the production phase, then their remediation usually becomes prohibitively expensive requiring extensive source code modification. Moreover these situations provide a sizeable time window to malicious users for exploiting the vulnerability in the unpatched web applications. Web application firewalls (WAFs) offer an attractive alternative and work by blocking attempts to exploit the vulnerabilities without having to fix the source code. WAFs can examine and selectively block everything in a typical web transaction ranging from user entry fields, URLs, request/response headers to user sessions and cookies. ModSecurity (2015) is a noteworthy and open source web application firewall (WAF) that can identify and block attempts targeted toward exploiting specific vulnerabilities in an application. ModSecurity provides a generic core rule set that is designed to provide a variety of defense mechanisms including XML protection, malicious software detection, error detection and application level attacks detection (Barnett, 2009). In 2008, Stephen Craig Evans lead an OWASP Summer of Code (SoC) Project entitled "Securing WebGoat with ModSecurity" (Evans, 2008). The purpose of this project was to create a custom ModSecurity rule set that, in addition to the core rule set, would provide protection to the various modules of WebGoat-5.2 standard release from as many of its vulnerabilities as possible without changing any line of source code. Different rules were implemented to mitigate each vulnerability, where each had its own HTML error file page that appeared in the browser when the associated attack was detected. In this SoC project, SQL injection attacks were mitigated by using whitelisting and blacklisting rules. The SoC project encouraged us to implement our own customized ModSecurity rule set; as a safeguard measure against the SPARQL/SPARUL injection attacks introduced in this paper.

2.5. *Vulnerability scanners*

Web application scanners are used for automating the detection of web application vulnerabilities. In addition to this, they generate observance reports and provide suggestions on mitigating the vulnerabilities detected. The most common vulnerabilities that the scanners investigate extensively are SQL injection (SQLI), cross-site scripting (XSS), cross-site request forgery (CSRF) and information disclosure. HacmeBank (OWASP, 2015c), WebGoat (OWASP, 2015a) and WackoPicko (GitHub, 2013) are some of the popular vulnerable applications that are often used for evaluating scanners. Extensive literature is available on the evaluation of web application scanners. For example, Doupe et al. evaluated the performance of eleven scanners against their own test application (WackoPicko) and detailed the reasons for a scanner's failure or success (Doupe et al., 2010). Suto (2007) performed a comparative analysis of the abilities of three popular scanners in detecting various web application vulnerabilities. Similar research (Suto, 2010) presented the evaluation of seven scanners on the basis of their detection capabilities and time efficiency. Peine (2006) compared WebGoat with a real-world application for evaluating the user interface of the seven scanners. Along the same lines, we have used SemWebGoat to evaluate the efficiency and accuracy of popular scanners in detecting vulnerabilities in Semantic Web applications and found them to be lacking. Therefore, this paper also describes how we extended two of the more well known vulnerability scanners to assist developers in detecting vulnerabilities in Semantic Web applications.

3. Design of proposed learning framework

3.1. *High-level architecture*

The objective of the proposed framework is to provide a realistic training environment for users to practice SPARQL and SPARUL injection attacks on Semantic Web applications as well as experiment with firewall mitigation techniques based on our contributed ModSecurity rule set. In order to achieve this, the framework uses several third-party open-source software as well as builds upon many of the components of the popular WebGoat application as can be seen in the architecture and design diagram (Fig. 2). The high-level components of the proposed framework and their interactions are detailed below:

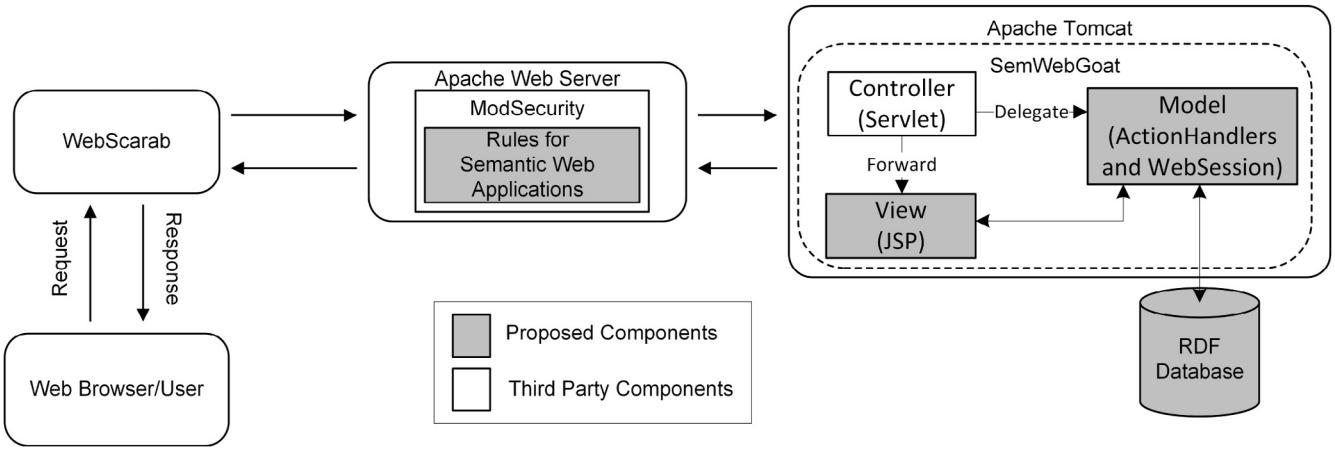

Fig. 2 – Architecture of Proposed Framework.

- **Web browser/user:** The web browser generates an HTTP request on behalf of a user to retrieve data. The request can either be valid in case of an authenticated user, or malicious whereby an attacker is trying to steal data. For sending requests directly to the web server (Apache, 2015a, in our case), the web browser is configured on localhost with port 80 but in case an intercepting proxy such as WebScarab (OWASP, 2015i) is required, the IP and port can be changed to that of the proxy so that it may modify requests routed through it.
- **HTTP proxy:** An HTTP proxy such as WebScarab is a tool that intercepts HTTP requests and responses and exploits the vulnerabilities in Semantic Web applications by allowing the attacker to examine or modify the parameters at runtime. A proxy is required for completing at least two of the lesson plans designed. It can typically be configured on the client machine by setting the HTTP proxy as local host and port as 80 so that it can forward requests to the web server.
- **WAF:** HTTP requests are intercepted by a WAF, such as ModSecurity (2015) in our case, to detect attack attempts and apply appropriate mitigation actions such as denying or redirecting the requests. A combination of blacklist rules describing illegal input strings and whitelist rules depicting the set of allowed expressions have been implemented to block injection attacks. In this architecture, ModSecurity has been configured on Apache web server as a reverse proxy listening on port 80 so that it can communicate with Apache Tomcat-a Java EE compliant server hosting SemWebGoat (Apache, 2015b).
- **SemWebGoat:** A deliberately insecure Semantic Web application has been developed. It includes attack lessons that are specific to Semantic Web languages. SemWebGoat has been deployed on Apache Tomcat. It retrieves data from the RDF database and displays it to the user.
- **RDF database:** This is the backend datastore that is used for generating dynamic content for SemWebGoat and contains data in the form of triples. In our case we have created this via the Jena API.

3.2. *Internal design of semantic WebGoat framework*

Semantic WebGoat extends the architecture of the WebGoat framework, which is modeled using the Model-View-Controller design pattern typical of J2EE applications (see blown-up SemWebGoat design on the right hand side of Fig. 2). All requests are handled by a main servlet, called the *HammerHead*, which acts as the controller and delegates it to the respective *Action Handler*. The controller functionality was borrowed without modification from WebGoat. The model encompasses *Action Handlers*, which are customized lesson objects that will execute their business logic, load data into the WebGoat WebSession object, and then turn the control over to the view component (JSP). This modular design allows developers interested in writing new lessons to identify their *Action Handler* and add all their classes involved in that lesson in that folder. Following this convention, we also added all our Semantic Web related attacks as new *Action Handlers* in the model. We also wrote a customized data access component that allows this new set of *Action Handlers* to connect to an RDF based datastore as opposed to an SQL based relational-database. Similarly, the JSP code in the presentation layer was extended to render the data retrieved from the RDF store using the new business logic.

3.3. *RDF data model and storage*

A major extension realized in SemWebGoat was the implementation of an RDF based datastore using the Apache Jena triple-store and data-access layer implemented using the SPARQL query interface. WebGoat, is an E-Commerce application that allows hackers to test their skills by breaking into an employee personal record keeping system, credit card information system and weather stations data. In order to remain compatible with the overall theme of its predecessor, the SemWebGoat datastore includes the major subjects: *Employee*, *Pins*, *Rewards* and *Weather* along with several minor subjects. Each entity further has properties such as *userid*, *name*, *salary* for the entity *Employee* and *station number*, *state name* for entity *Weather* (a comprehensive data-model design in the form

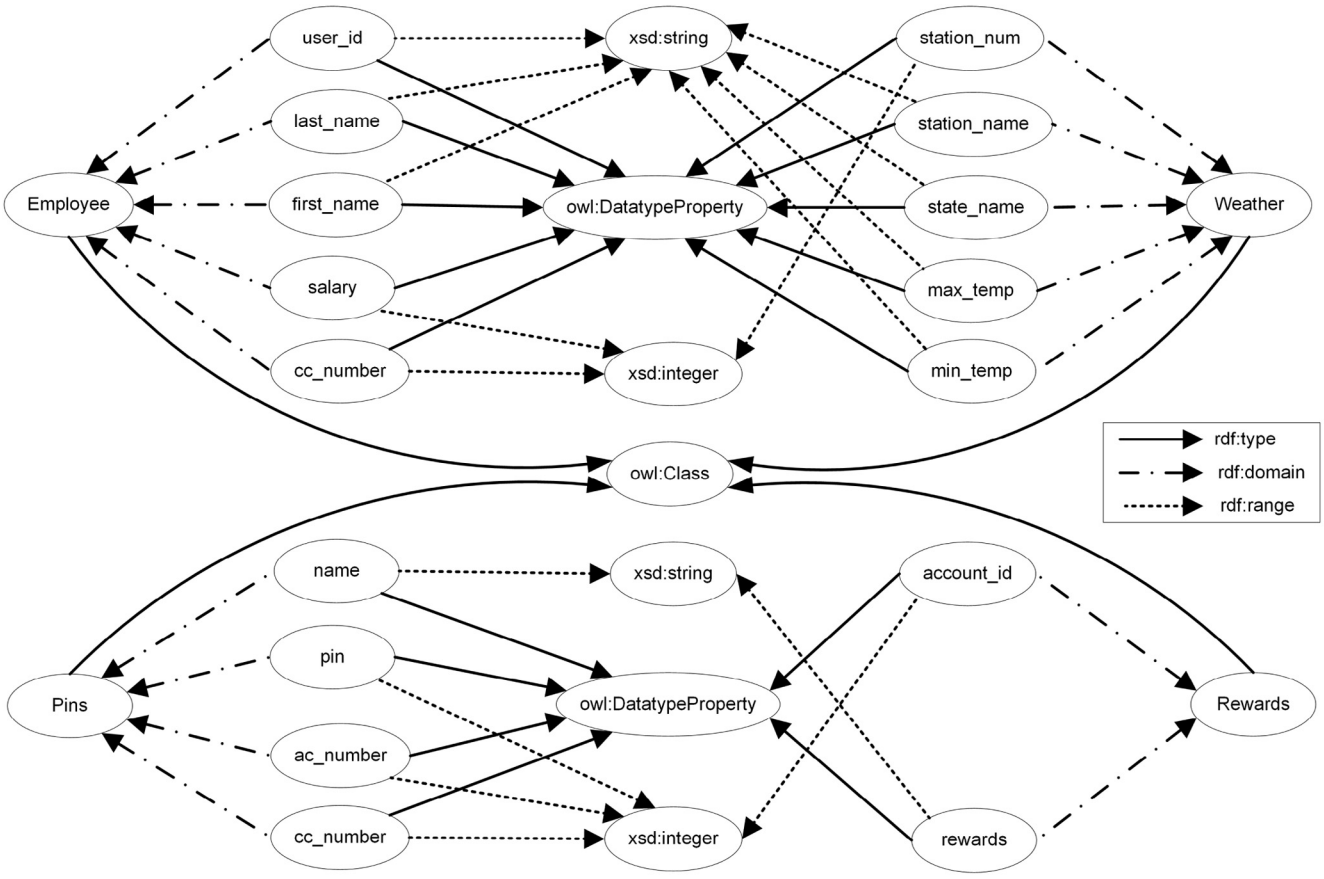

Fig. 3 – RDF Graph Model of SemWebGoat Data.

of an RDF graph is shown in Fig. 3 and was derived in part from WebGoat's relational database schema). Each lesson in SemWebGoat is based on a certain scenario and has access to the appropriate RDF domain according to the requirement of that scenario.

3.4. *Lesson flow and user interface*

SemWebGoat is designed to mirror a typical E-Commerce application where users can interact with different parts of the system by filling in the corresponding input fields and selecting elements through drop-down lists as a result of which dynamic content is generated based on underlying data in the data-store. Features such as search, update and viewing personal records are available. Data are meant to be private, and the ability to see other people's confidential records such as salaries and credit card numbers via injection attacks is considered a breach of security as can be imagined in a real commercial e-commerce web application. Lessons have been designed in a mix of difficulty levels where some of the more challenging attacks require expertise on the part of the hacker whereby he can effectively use information exploited from one vulnerable input field as a stepping stone to exploit another weakness.

Typical of dynamic web sites, the architecture of the Semantic Web application also revolves around the navigation of web pages. To represent the functionality and architecture of the web system, designers use different models and viewpoints; such as Implementation Views, Site Maps, Deployment, Analysis and Design Models. An Implementation View describes the abstraction of the web pages and the relationships between them using hyperlinks. At an abstract level, the Implementation View of a web application is similar to a Site Map of a web system. Figure 4 presents the Implementation View of SemWebGoat, where the two hyperlinks (<<link>> , <<build>>) are used to represent navigation paths through the web application and parameter values (e.g. StationNumber, LastName) are passed as arguments to be used by the server pages to build the client pages.

For the implementation of SemWebGoat, we leveraged the user interface design layout of WebGoat. WebGoat already contains a category for injection attacks. We kept the same category name, but made minor modifications for invoking our specific SPARQL/SPARUL and XML versions instead of the SQL based predecessors. The modifications to the user interface (elements shown as white in Fig. 4) include updates to: (1) the navigation links for each lesson, (2) hints for solving each lesson, (3) underlying source code for invoking the appropriate ActionHandler in the model, (4) lesson solutions and (5) configuration details. In addition, a completely new category for SPARQL-based DDoS attacks was added (elements highlighted in grey in Fig. 4), because we did not find any related attacks in the predecessor version. The highlighted portion includes the back-end business logic as well as the JSP code needed to retrieve data from the RDF database.

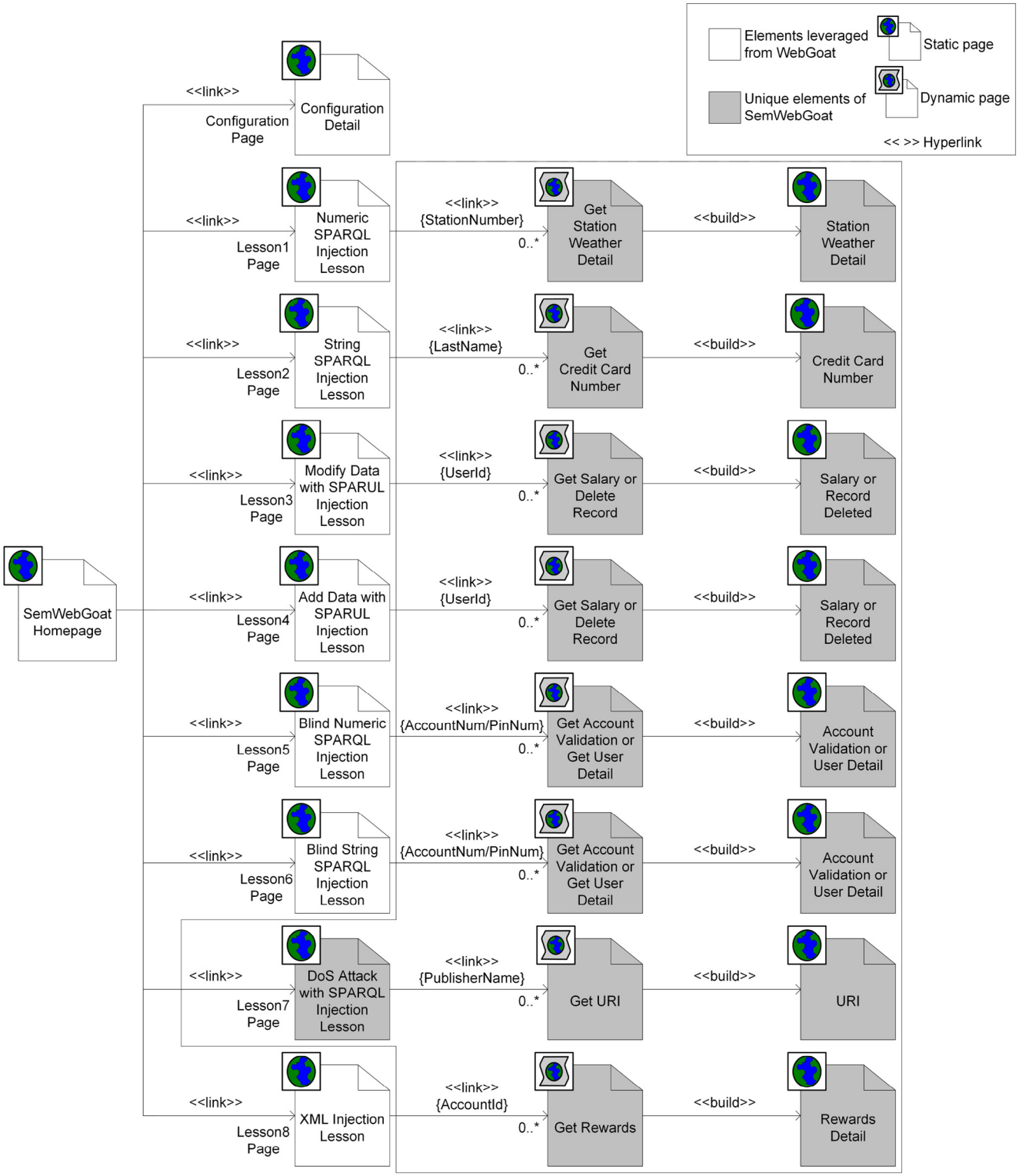

Fig. 4 – Implementation View of SemWebGoat.

4. Security lessons

WebGoat includes a number of security lessons related to a variety of SQL injection attack types under the category "Injection Flaws". Following this convention, SemWebGoat also lists all lessons related to possible SPARQL/SPAUL injection attacks under the same category. For the implementation of the lessons, we have followed the scenario naming scheme employed by WebGoat. To better understand the implementation details, a side by side comparison of SQL injection attacks from WebGoat and SPARQL/SPARUL injection attacks from SemWebGoat has been provided

Table 1 – Lessons of WebGoat and SemWebGoat.

Injection lessons in WebGoat	Injection lessons in SemWebGoat
Numeric SQL	Numeric SPARQL
String SQL	String SPARQL
Modify Data with SQL	Modify Data with SPARUL
Add Data with SQL	Add Data with SPARUL
Blind Numeric SQL	Blind Numeric SPARQL
Blind String SQL	Blind String SPARQL
N/A	DoS Attack with SPARQL
XML	XML

in this section and summarized in Table 1. A total of eight lessons have been demonstrated in SemWebGoat; two of which require the use of third-party software such as "WebScarab" to inspect or modify data entered via user input fields.

4.1. *Numeric SPARQL injection*

This lesson uses the same scenario that has been used by "Numeric SQL Injection" of WebGoat. According to the scenario, the user can view the weather data of a particular station by selecting a station number (such as 101, 102 or 103) from a drop-down listbox. The goal is to inject a SQL string that results in all the weather data being displayed. The WebGoat application receives input from the select box and inserts it at the end of a pre-formed SQL command. In this case, the user requires the use of a proxy such as WebScarab to intercept the HTTP request to concatenate the malicious string with the URL-encoded value. For instance, upon replacing the URL-encoded value with the SQL string provided in Listing 2a, WebGoat unwittingly displays weather data for all the stations since the SQL statement "101 or 1 = 1!" always resolves to true.

Listing 2a: Numeric SQL Injection

```
Pre-formed SQL query: SELECT * FROM weather_data WHERE
        station = '" + stationNum + "'
Valid URLEncoded value: 101
Malicious SQL string concatenated with the URLEncoded value:
        101 or 1=1!
```

In the equivalent SemWebGoat lesson, the user has to inject a SPARQL string instead of the SQL equivalent to display the weather data of all stations. The essential details required to build an understanding of this lesson are provided in Listing 2b. The "+stationNum+" variable in the pre-formed SPARQL query returns the data for the particular station number selected by the user from the dropdown listbox. In this scenario, it is necessary for the user to have prior knowledge of the predicates that are being used in the back-end query so that a correct malicious query can be injected. In the malicious query, the user can use any arbitrary variable (e.g. *s*, *subject*) for retrieving all the subjects except for "uri" because this variable is already bound to return the weather data for the single station case. Similarly for retrieving the object values, the user can use any variable (such as *o*, *object* etc). In the malicious query "?s", "?station_name", "state_name", "max_temp" and "min_temp" will return all data associated with the predicates specified. The purpose of the "#" is to comment out the rest of the pre-formed query.

Listing 2b: Numeric SPARQL Injection

```
Pre-formed SPARQL query: SELECT * WHERE
        { ?uri weather:station_num  '" + stationNum + " '.
        ?uri weather:station_name ?station_name .
        ?uri weather:state_name ?state_name .
        ?uri weather:max_temp ?max_temp .
        ?uri weather:min_temp ?min_temp . }
Valid URLEncoded value: 101
Malicious SPARQL string concatenated with URLEncoded value:
        101' .  ?s weather:station_name ?station_name .
        ?s weather:state_name ?state_name .
        ?s weather:max_temp ?max_temp .
        ?s weather:min_temp ?min_temp .  } #
```

4.2. *String SPARQL injection*

This lesson follows the "String SQL Injection" scenario of WebGoat. According to the scenario, the user can enter the last name of a person in an input field and view corresponding details of the credit cards that he owns. The goal is to inject a SQL string that would display all credit card numbers in the data store simultaneously. This is possible by appending a tautology such as "Smith' OR '1' = '1" to the query instead of having just "Smith" as shown in Listing 3a. This change will thereby display all the credit card numbers because the system is forced to ignore the WHERE conditional.

Listing 3a: String SQL Injection

```
Pre-formed SQL query: SELECT * FROM user_data WHERE
        last_name = '" + LastName + "'
Valid input: Smith
Malicious SQL string concatenated with the input: Smith'OR '1'='1
```

In the corresponding SemWebGoat lesson the difference in the scenario is that this time, the user is required to inject a SPARQL string equivalent of the SQL string shown earlier to obtain the desired result. Essential details are illustrated in Listing 3b. The "+LastName+" variable in the pre-formed SPARQL query is taking the input value from the input field and will return the number of the particular credit card owner whose last name has been entered. Like the previous lesson, it is necessary for the attacker to have a priori knowledge of the predicates that are being used in back-end query. In the malicious query, the attacker can choose any variable name for retrieving all the subjects except "uri" whereas for retrieving the object values the user can use any variable name. In the malicious query, "?s" and "?cc_number" will return all data associated with the predicates specified and "#" will comment out the rest of the pre-formed query. As a result, all credit card numbers in the data store will be displayed at the same time.

Listing 3b: String SPARQL Injection

```
Pre-formed SPARQL query: SELECT *
        WHERE { ?uri employee:last_name '" + LastName + "' .
        ?uri employee:cc_number ?cc_number . }
Valid input: Smith
Malicious SPARQL string concatenated with the input: Smith' .
        ?s employee:cc_number ?cc_number . } #
```

4.3. *Modify data with SPARUL injection*

This lesson follows the scenario employed by the "Modify Data with SQL Injection" lesson in WebGoat. According to the scenario, a user can view salaries associated with a particular user_id. The input field is designed to be vulnerable to SQL injection and the goal is to inject specially crafted SQL strings to modify the victim's salary, in this case with user_id "jsmith" (that belongs to John Smith). The solution and source code details are provided in Listing 4a.

Listing 4a: Modify Data with SQL Injection

```
Pre-formed SQL query: SELECT * FROM salaries WHERE userid =
        '" + UserId + "'
Valid input: jsmith
Malicious SQL string concatenated with the input: jsmith';
        UPDATE salaries SET SALARY=5000 WHERE userid='jsmith
```

In the equivalent SemWebGoat lesson users are provided with two input fields. The first field allows a user to view individual employee salaries, while the second allows deletion of entire employee records by entering corresponding user_ids. The intuition behind the use of two input fields in the implementation of this lesson is that the attack requires the use of two separate semantic query languages. The SELECT query uses the SPARQL syntax whereas the UPDATE query uses SPARUL and the semantic query language disallows the mixing of the two queries in a single statement. To inject the SPARUL string, the user requires a priori knowledge about the subject URI, predicate and object value (salary). This can be gleaned using the first input field, allowing the attacker to for instance acquire information about the URI and the salary of victim John by entering his corresponding user_id "jsmith". Subsequently, the attacker can use the second input field to update the salary information, which in SPARUL requires two actions performed together: DELETE and INSERT. This two-step procedure is important, otherwise the new record will be added with the previous record of salary, and in such a case when a user searches for John's salary, two results will be displayed. After modifying John's salary the user can use the first input field to view the updated salary record. Since the pre-formed query is a DELETE query (illustrated in Listing 4b), the attacker concatenates the SPARUL statement with the user_id of another valid user whose record maybe compromised such as *mstooge* or *lstooge* and then performs an INSERT salary for user_id "jsmith". This technique avoids having to completely delete and reconstruct the entire record of the victim John Smith and instead the consequence is that just the salary gets updated. Note that these examples illustrate just one out of multiple possible ways of achieving the desired results and the attacker can tailor his attack query based on his preferences and skill-set.

Listing 4b: Modify DATA with SPARUL Injection

```
Pre-formed SPARUL query: DELETE
        WHERE  {?uri employee:user_id '" + UserId + "'.
        ?uri employee:salary ?salary . }
Valid input: jsmith
Malicious SPARUL string concatenated with the input: lstooge'.
        employee:John employee:salary '20000'.  }
        INSERT DATA { employee:John employee:salary '5000 '.  } #
```

4.4. *Add data with SPARUL injection*

The "Add Data with SQL Injection" lesson in WebGoat forms the basis for this lesson and is conceptually very similar to the previous one. However, in this case, the attacker is required to insert new data instead of modifying the stored data. According to the scenario, the user can view salaries associated with a user_id. Since the input field is vulnerable, the goal is to inject an SQL string to add a new salary record for an arbitrary employee name of the attacker's choice. The solution and source code details are provided in Listing 5a.

Listing 5a: Add DATA with SQL Injection

```
Pre-formed SQL query: SELECT * FROM salaries WHERE userid =
        '" + UserId + "'
Valid input: jsmith
Malicious SQL string concatenated with the input: jsmith';
        insert into salaries values('nome',10000);
```

In the corresponding SemWebGoat lesson, users are provided with two input fields. The first allows a user to view salaries belonging to employees with the user_ids provided as input, while the second field allows their deletion. A SPARQL based SELECT query and a SPARUL INSERT query cannot be used together in a single statement and therefore mandates the need for two separate fields. To add a new salary record into the database the attacker can employ an INSERT query illustrated in Listing 5b. The first closing bracket has been used to close the DELETE query and then the INSERT query has been concatenated. Execution of this SPARUL statement causes a new user record with name "Nelson", user_id "nome" and salary "10000" to be added to the database. Subsequently the attacker can use the first input field to verify the newly added salary record for user_id "nome".

Listing 5b: Add DATA with SPARUL Injection

```
Pre-formed SPARUL query: DELETE
        WHERE  {?uri employee:user_id '" + UserId + "'.
        ?uri employee:salary ?salary . }
Valid input: jsmith
Malicious SPARUL string concatenated with the input: mstooge'.
        } INSERT DATA { employee:Nelson employee:user_id
        'nome' .  employee:Nelson employee:salary '10000' .  } #
```

4.5. *Blind numeric SPARQL injection*

This lesson follows the scenario used by the "Blind Numeric SQL Injection" attack in WebGoat. The scenario employs an input field that allows a user to determine the validity of an account number provided as input. Such an input field can be misused by an attacker to develop a true/false test to check other entries in the database. The goal is to determine the pin value (that is of integer type) for a particular credit card number which is "1111222233334444" in the case of this example. The SQL string provided in Listing 6a returns either valid or invalid. If it returns the former then it means that the pin value is greater than 10000 while the latter implies that the pin value is less than 10000. By executing this statement, a number of times with different pin values, the attacker can narrow down and guess the correct pin. The attacker can only pass the lesson by entering the correct pin, which is 2364 in the case associated with the specified credit card number.

Listing 6a: Blind Numeric SQL Injection

```
Pre-formed SQL query: SELECT * FROM user_data WHERE userid =
        '" + accountNo + "'
Valid input: 101
Malicious SQL string concatenated with the input:101 AND
        ((SELECT pin FROM pins WHERE cc_number=
        '1111222233334444') > 10000 );
```

In the corresponding SemWebGoat lesson users are provided with two input fields. The first field allows a user to validate account numbers and can only view the information of a credit card when the correct pin value is entered in the second field. The ASK query has been used to implement the functionality of the first as it is designed in the language to return the result as a Boolean (true/false) value. Similar to the previous lessons, it is assumed that the user will have prior knowledge of the predicates so that a correct malicious query can be concatenated with the input. The malicious SPARQL string that has been concatenated with the ASK query for comparing the pin value is provided in Listing 6b. This string uses the FILTER clause, which allows the filtering of results based on certain conditions. For instance, numbers can be filtered using equalities and inequalities. For determining the pin, the attacker can vary the pin value ("10000" in the provided SPARQL) each time and examine the Boolean result. After determining the correct pin (2364 in the case of our example system) the user can enter it into the second input field to pass the lesson.

Listing 6b: Blind Numeric SPARQL Injection

```
Pre-formed SPARQL query: ASK
        WHERE {?uri pins:ac_number '" + accountNo + "'.}
Valid input: 101
Malicious SPARQL string concatenated with the input: 101'.
        ?s pins:cc_number '1111222233334444' .
        ?s pins:pin ?pin.  FILTER (xsd:integer(?pin)>10000) } #
```

4.6. *Blind string SPARQL injection*

This lesson uses the same scenario that has been used by "Blind String SQL Injection" of WebGoat, and is conceptually very similar to the previous lesson. This time, however, the attacker has to search for a string, and not for a number. According to the scenario, the input field allows a user to enter an account number and determine if it is valid. The goal is to find the value of the account holder name for a particular credit card number (which is "4321432143214321" in the case of our example). The SQL strings provided in Listing 7a make use of the *SUBSTRING(STRING,START,LENGTH)* function to determine the letters constituting the name one character at a time. The particular snapshot in time shown in Listing 7a is trying to match the first letter with "J" as the correct name is "Jill". Once the first letter has been determined by brute force, the attacker moves on to determining the next character "i" by changing the SUBSTRING parameters as shown in the second malicious SQL string. In the second malicious string, "<u>" and "</u>" tags indicate that the regular expression is case insensitive. WebGoat assumes that the first character representing a name must be capitalized, but the same is not true for the rest of the characters. A similar procedure can be employed for the remaining letters of the name. Finally, the user can enter the correct name in the input field to clear the lesson.

Listing 7a: Blind String SQL Injection

```
Pre-formed SQL query: SELECT * FROM user_data WHERE userid =
        '" + accountNo + "'
Valid input: 101
Malicious SQL string for comparing first letter of name:101
        AND (SUBSTRING((SELECT name FROM pins
        WHERE cc_number='4321432143214321'), 1, 1) < 'J' );
Malicious SQL string for comparing second letter of name:101
        AND (SUBSTRING((SELECT name FROM pins
        WHERE cc_number'4321432143214321'),
        2, 1) <= '<u>i</u>' );
```

In the equivalent SemWebGoat lesson, users are provided with two input fields. The first input field allows a user to verify the validity of account numbers. A user can only view credit card information when the correct pin is entered in the second input field. In Listing 7b, the FILTER clauses in the malicious SPARQL strings are using the regex operand that allows the comparison of two text strings. The provided SPARQL query compares the first letter in the value of "?name" with the character "J" using the caret (^) symbol to constrain the comparison to be with the starting letter only. The third parameter "i" indicates that the regular expression search is case insensitive but in case the attacker, wants it to be case sensitive then only the first two parameters would be required. By using the first malicious SPARQL string the attacker can determine the first letter of name and then to figure out the second letter the first character of the regular expression is fixed at "J" and the second character is permuted as shown in the second malicious SPARQL string. The user can follow a similar procedure to determine the rest of the letters of the name and then type in the name in the second input field to clear the lesson.

Listing 7b: Blind String SPARQL Injection

```
Pre-formed SPARQL query: ASK
        WHERE {?uri pins:ac_number '" + accountNo + "'. }
Valid input: 101
Malicious SPARQL string for comparing first letter of name:
        101'.  ?s pins:cc_number '4321432143214321' .
        ?s pins:name ?name.  Filter regex(?name, '^J','i') } #
Malicious SPARQL string for comparing second letter of name:
        101'.  ?s pins:cc_number '4321432143214321' .
        ?s pins:name ?name.  Filter regex(?name, '^Ji', 'i') } #
```

4.7. *DoS attack with SPARQL injection*

An injection attack leading to denial-of-service (DoS) is an attack scenario unique to SemWebGoat due to some inherent properties of the SPARQL language and there is no closely related SQL equivalent in the predecessor application-WebGoat. The SPARQL syntax makes it trivial to mount DoS attacks because it allows querying of very large triple stores without requiring the need to specify table names. This property can be easily leveraged by an attacker to amplify his attack and force the

semantic application datastore to perform a very long time-consuming operation requiring a great deal of processing in response to a simple query injected into an input field. The purpose is to deprive other users of the same application or host server from the services of a resource they normally expect to have. For demonstrating a DoS attack via SPARQL injection we populated SemWebGoat's datastore with a relatively small portion of the Barton Library Dataset (MIT, 2015) so that users could query one million triples (file size: 75.6 MB). In actuality, the data set used contains over 50 million triples and it is not unusual for RDF datasets to scale to billions of triples. The motivation was to demonstrate that even a small fraction of real datasets that exist today suffice for mounting dangerous attacks, supporting our case that injection attacks can be a very serious threat to Semantic Web applications. In our experimental setup, SemWebGoat was hosted on an Apache Tomcat server running on an Intel (R) Core(TM) i3 machine with 2.10 GHz CPU, 8 GB RAM with a 64-bit Microsoft Windows 7, Service Pack 1 operating system.

According to the scenario, the URI of the specific publisher is displayed when the user enters the publisher name in the input field. However, in the case where an attacker injects a malicious SPARQL statement through the same field that requires a large amount of processing, it will cause the server to be unresponsive for all users. In the example injected string provided in Listing 8, the server became completely unresponsive for at least 10 minutes. Dissecting the example injected string, one can observe that the "?s" variable simply returns all the subjects, "?p" returns all the predicates and "?o" returns all the object values stored in the RDF dataset. No prior knowledge of the back-end dataset schema is required to prepare these statements and the attacker is free to use arbitrary variable names of his choosing such as "?x ?y ?z" or "?aa ?ab ?ac" to get the same desired behavior. These strings are generally used to fetch all the data stored in the back-end RDF dataset and when the dataset contains millions of triples then the result is typically a DoS attack.

Listing 8: DoS Attack with SPARQL Injection

```
Pre-formed SPARQL Query:SELECT *
        WHERE { ?uri mods:value '" + publisherName + "'. }
Valid input: Oxford University Press
Malicious SPARQL string concatenated with the input:
        Oxford University Press' .  ?s ?p ?o .  } #
```

4.8. *XML injection*

Typically, web services or web applications that use Ajax or similar technologies for reloading parts of a page selectively employ XML to store data and exchange messages. XML documents are usually treated as databases that include sensitive information. These XML messages and documents can be captured and altered by an attacker if the attacker has the ability to write the raw XML. According to the scenario, as the user enters the account_id, the function *getRewards()* (demonstrated in Listing 9a) sends a request with the parameter account_id to the server and the server responds with the corresponding account's rewards in XML format using a *callback* function. The goal of the attacker is to try and increase his rewards by adding more entries to the allowed list using XML injection. To add more rewards, the user requires a tool for traffic interception such as WebScarab to intercept and modify the HTTP response as illustrated in Listing 9c.

Listing 9a: Code for XML Injection Lesson in WebGoat

```
<input id= "accountID" type="TEXT" name="accountID"
value=""onkeyup="getRewards();">
function getRewards(){
 var accountIDField = document.getElementById('accountID');
 ...
 req.open('GET', url, true);
 req.onreadystatechange = callback;
 req.send(null);
}
function callback() {
 ...
 var rewards = req.responseXML.getElementsByTagName('reward');
 ...
}
```

Semantic Web applications have an even larger attack surface as they use a wide breadth of formats other than XML such as RDF/XML, N-triples, RDF/JSON and Turtle to serialize graph data and perform data exchange, all of which are vulnerable to injection attacks. This lesson has been implemented in XML as this format remains the most popular; however, implementations using other formats such as RDF can be designed similarly. Similar to the "XML Injection" lesson of WebGoat, in the corresponding semantic version as the attacker enters the account_id, a pre-formed SPARQL query (illustrated is Listing 9b) will return the list of associated rewards. Listing 9c describes the tampered response in which two new rewards has been added to the allowed list of rewards by using XML Injection.

Listing 9b: Back-end Query for XML Injection Lesson in SemWebgoat

```
Pre-formed SPARQL query: SELECT * WHERE
        {rewards:myRewards rewards:account_id '"+ accId + "'.
        rewards:myRewards rewards:rewards ?rewards . }
```

Listing 10: ModSecurity Rules for Detecting SPARQLI/SPARULI Attacks

```
Rule 1:  For Numeric SPARQL Injection
SecRule "ARGS_POST:stationNum" "!^[0-9]*$" "id:‘955555’,
t:urlDecode,log,auditlog,
redirect:http://localhost:80/errorpage.html"
Rule 2:  For String SPARQL Injection
SecRule "ARGS_POST:LastName "!^[A-Za-z]*$" "id:‘955556’,
t:urlDecode,log,auditlog,
redirect:http://localhost:80/errorpage.html"
Rule 3:  For Add/Modify Data with SPARUL Injection
SecRule "ARGS_POST:UserId "(}\s*#)$" "id:‘955557’,
t:urlDecode,log,auditlog,
redirect:http://localhost:80/errorpage.html"
Rule 4:  For Blind Numeric/String SPARQL Injection
SecRule ARGS_POST:AccountNum "!^[0-9]{3}$" "id:‘955558’,
t:urlDecode,log,auditlog,
redirect:http://localhost:80/errorpage.html"
Rule 5:  For DoS Attack with SPARQL Injection
SecRule ARGS_POST:name ""(\\?\D)" "id:‘955559’,
t:urlDecode,log,auditlog,
redirect:http://localhost:80/errorpage.html"
```

5. Attack prevention using WAF rule set

Approaches for defending Semantic Web applications against attacks include 1) adopting secure programming practices; 2) using penetration testing tools to test every aspect of the functionality prior to deployment; and 3) deploying the application behind the protection of a WAF. As already discussed, the first two approaches require a steep learning curve making WAFs as an attractive option for countering application layer threats. Unlike traditional network layer firewalls, a WAF is deployed in between the web client and server and works as a filter to monitor every HTTP conversation looking for specific attack signatures or anomalies within the transactions traffic. Rules can be classified as either blacklists defining unacceptable patterns or whitelists defining permitted characteristic values or patterns.

Unfortunately, the effectiveness of WAFs is only as good as the effectiveness of their rule configurations. Default WAF configurations only provide a template rule set that require application specific tuning and requirements specific customization. A key challenge is to have sound knowledge of the possible attacks and understanding of the application behavior being protected to be able to properly capture application and WAF rule context. Otherwise, the rule set may end up being either too restrictive such that it blocks requests both legal and illegal, or can be too permissive allowing all kinds of requests to pass through the WAF. Consequently, part or whole of the Semantic Web application no longer remains under the WAF's protection resulting in false positives and negatives manifesting as inaccessibility problems, legitimate HTTP transactions being dropped and unpatched security holes.

To demonstrate the feasibility of application-level protection, we employed the popular open-source WAF-ModSecurity to implement a customized rule set to detect and block attacks targeting the various lessons available in SemWebGoat. We configured ModSecurity-2.7 (ModSecurity, 2015) on Apache-2.4 server (Apache, 2015a) and added our rules to the modsecurity.conf file. The general rule syntax for ModSecurity can be described as in Equation 1.

Sec*Rule VARIABLES OPERATOR ACTIONS* (1)

Variables, *operator* and *actions* are three basic parts of any ModSecurity rule. Where the *variables* part specifies where to look, *operator* part specifies how to look and the *actions* specify what to do if a match occurs.

In order to detect injection attacks targeting Semantic Web applications, we analyzed the categories of attack strings individually. By placing certain checks on the unique names assigned to each input field, we developed a total of 40 ModSecurity rules for detecting these attack strings. Blacklist rules were based on detecting particular *features* of the attack strings, not normally found in valid input for instance the presence of SPARQL/SPARUL reserved words such as **SELECT**, **WHERE**, **ASK**, **INSERT**, **DELETE** and **uri**. Similarly, whitelists were developed based on semantic knowledge of the input string, for instance account numbers can only be strings comprising digits of length 3. Due to space constraints, a small sample of the rule set developed is shown in Listing 10, which includes both whitelists (rules 1, 2 and 4) and blacklists (rules 3 and 5). The SemWebGoat user input fields being protected and shown in these examples use the POST method to retrieve data, therefore the "ARGS_POST" variable has been used in order to examine the request post body to match the regular

expression. A brief description for each of the sample rule shown is as follows:

- **Rule 1:** Inspect the argument called "stationNum". If the user input string is not in the form of digits then intercept the transaction, log it and redirect it to the specific link.
- **Rule 2:** Inspect the argument called "LastName". If the user input string is not in the form of upper and lower case letters then intercept the transaction, log it and redirect it to the specific link.
- **Rule 3:** Inspect the argument called "UserId". If the user input string ends with "}#" then intercept the transaction, log it and redirect it to the specific link.
- **Rule 4:** Inspect the argument called "AccountNum". If the user input string length not exactly three digits then intercept the transaction, log it and redirect it to the specific link.
- **Rule 5:** Inspect the argument called "name". If the user input string contains a non-digit variable appended exactly after a question mark (?), then intercept the transaction, log it and redirect it to the specific link.

Listing 9c: Response Tampered by using XML Injection

```
Actual HTTP Response: <root>
        <reward>WebGoat t-shirt 20 Pts</reward>
        <reward>WebGoat Secure Kettle 50 Pts</reward>
        <reward>WebGoat Mug 30 Pts</reward>
        </root>
Tampered HTTP Response with XML Injection:<root>
        <reward>WebGoat t-shirt 20 Pts</reward>
        <reward>WebGoat Secure Kettle 50 Pts</reward>
        <reward>WebGoat Mug 30 Pts</reward>
        <reward>WebGoat Core Duo Laptop 2000 Pts</reward>
        <reward>WebGoat Hawaii Cruise 3000 Pts</reward>
        </root>
```

6. Evaluation

For the evaluation of our work, we conducted a user study to determine the comprehensiveness and difficulty level of the training as well as performed an empirical, comparative study of existing penetration testing tools to figure out the level of support provided for SPARQL vulnerability analysis. We also analyzed the detection rate and false alarm rate of our customized WAF rules by testing the system against injection attacks derived from OWASP top 10 attack dataset. For purposes of the evaluation, the SemWebGoat application was deployed on an Intel (R) Core(TM) i3 machine with 2.10 GHz CPU, 8 GB RAM and Microsoft Windows 7, Service Pack 1 64-bit operating system.

Table 2 – Summary of demographics.

	Total: 40
Total users	Undergraduate students: 24 (60%) Postgraduate students: 6 (15%) Programmers: 10 (25%)
Age	20–30 years
Working experience	2 months–5 years
Current general computer use	5–10 hours a day
Awareness of Semantic Web and web security tools	Yes: 60% Some extent: 40%
Work experience in the domain of Semantic Web	Yes: 25% No: 75%
Familiarity with ModSecurity rule language	Yes: 15% No: 85%

6.1. *User study*

On-line access to "SemWebGoat" was provided to a mix of programmers and students in a laboratory setting, the demographics summary of which is shown in Table 2. The participants were initially provided a brief overview of the experiment requirements without going into any detail of the vulnerabilities and defenses. After this step, they were asked to follow the lesson plan and try to exploit vulnerabilities in SemWebGoat and subsequently experiment with the WAF rules to protect the same system. After completing the lessons, the participants were asked to answer some survey questions. The survey results (Table 3) show that although users were aware of Semantic Web concepts, but they were mostly unfamiliar with the vulnerabilities demonstrated in SemWebGoat and overall the lessons served to improve their security concepts of this domain. Results also show that it is important for the Semantic Web developers to know about such vulnerabilities so that Semantic Web applications can be protected against attacks. In case of SemWebGoat with ModSecurity WAF rules, survey results show that use of ModSecurity negligibly affects the performance and usability of SemWebGoat and it is a suitable attack prevention mechanism against SPARQLI/SPARULI attacks.

6.2. *Evaluation of customized WAF rule set*

The customized WAF rule set that was developed in ModSecurity to detect SPARQL/SPARUL injection attacks was evaluated on the basis of detection rate and false alarm rate. For the purposes of the evaluation, the backend RDF database was populated using the MIT Barton dataset (MIT, 2015) and the WebGoat default dataset (OWASP, 2015a). Our customized rule set was categorized as "Specific" (whitelist) WAF rules that were tailored for specific form fields and scenarios (such as Rules 1, 2 and 4 mentioned in Section 5) and "Generic" (blacklist) WAF rules that were created for blocking common types

Table 3 – Summary of survey questions and results.

Survey questions	Results		
	Yes	Some extent	No
The lessons improve my concepts of vulnerabilities in Semantic Web applications.	100%	0%	0%
Every lesson has a well-designed scenario to teach each injection technique.	65%	25%	0%
Each lesson provides sufficient material, such as hints, java code, solutions for exploiting the vulnerability.	100%	0%	0%
The lesson covers the injection techniques that I would like to learn about.	90%	10%	0%
The lessons stimulate my further interest in learning other security technologies/concepts.	70%	30%	0%
I am already aware of the vulnerabilities demonstrated in SemWebGoat.	0%	25%	75%
The application is user-friendly and easy to use.	90%	10%	0%
Each lesson is related directly to the security of Semantic Web applications.	100%	0%	0%
The lessons provide a more practical understanding of security concepts than a lecture based approach.	100%	0%	0%
Semantic Web developers should be aware of such vulnerabilities.	100%	0%	0%
Estimated time spent in completing all the lessons.	2 hours		
In my view the most difficult lesson is:	Blind string SPARQLI (80%) Blind numeric SPARQI (20%)		
In my view the easiest lesson is:	XML injection(75%) DoS attack with SPARQLI (25%)		
It was difficult to use attack prevention mechanism based on ModSecurity WAF.	5%	15%	80%
It is necessary to fully understand ModSecurity Rule language before using it.	0%	0%	100%
Use of ModSecurity WAF negligibly affects the performance of the Semantic Web application.	100%	0%	0%
The customized ModSecurity rule set provides a suitable mechanism for detecting injection attacks.	75%	25%	0%

of SPARQL/SPARUL malicious queries and can be applied to multiple web form fields by just altering the associated field names (such as Rules 3 and 5 mentioned in Section 5). The system was exposed to a list of 100 test vectors that included both valid and malicious user inputs (which were variations of the example inputs mentioned earlier in Section 4). For building the collection of attack vectors we referred to various security cheat sheets including the input validation testing cheat sheet maintained by OWASP (OWASP, 2015h). The attack vectors were divided into following two categories:

- Attack traffic – The malicious SPARQL queries injected by the attacker. Since these are SPARQL specific, we developed attack strings from scratch based on 40 different variations derived from OWASPs equivalent SQL injection attacks set.
- Normal traffic – Which mirrored genuine user inputs on web application form fields by a valid user designed to retrieve dynamic content from the backend database. The backend database was populated from the WebGoat and Barton dataset, and we verified that Barton dataset contained entries with characters with and without special meaning in SPARQL syntax.

OWASP's WebScarab tool (OWASP, 2015i) was used for launching these attacks in an automated fashion. With minor variation, we have reused the criteria, which was set out by Sarasamma et al. (2005) for calculating the detection rate and false alarm rate:

- **Total normal inputs (TN):** the total number of valid input strings.
- **Total attack strings (TA)**: the total number of malicious strings.
- **False positive (FP):** the total number of valid input strings that are classified as malicious.
- **False negative (FN):** the total number of malicious strings that are classified as valid inputs.
- **Detection rate** = [(TA-FN)/TA]*100
- **False alarm rate** = [FP/TN]*100

The detection rate and false alarm rate are summarized in Table 4. This table presents a comparative analysis of the detection rate of the ModSecurity core rule set (OWASP, 2015d) and our customized rule set. In case of ModSecurity core rule set, 200 OK HTTP status code shows that none of the injection attacks were detected, thus motivating us to develop our own customized rule set. For a large majority of the attack vectors our customized rule set provided 100% detection rate against injection attacks without generating any false positives or false negatives. Whereas in some cases, a small percentage of false alarm rate was recorded against the rules created to block user inputs that contain special characters such as in case of String SPARQL Injection (discussed in Section 4.2) and DoS attacks (discussed in Section 4.7). For example, a user might enter valid person and publisher names that contain special characters such as "Kai'La", "St. John", "Galleria d'arte Medea" and "Franklin Pub. Co.", which were identified as malicious because the characters have a special meaning in SPARQL/SPARUL.

We also observed some minor degradation in detection rate against some malicious inputs such as in case of DoS attacks. In this case, an attacker may simply evade the rules for detecting SPARQL strings for fetching all data by injecting any one of a very large variation other time-consuming queries such as SELECT UNION or JOIN to successfully launch a DoS attack. Accurate detection of DDoS attack strings requires semantically identifying queries that invoke extensive server computation and not just simple syntactic checking of input queries for SPARQL characters.

Based on the results, we believe the existing rules cannot be further improved without disturbing the detection rate versus false positive tradeoff ratio. If the attack patterns are made overly permissive, then they can be evaded easily. Whereas, if the attack pattern is made too broad in scope, it improves detection, but also causes many false positives. Rules are

Table 4 – Testing results of customized WAF rules.

Injection attacks	ModSecurity core ruleset	Customized ModSecurity ruleset
Numeric SPARQL injection	200 OK	Detection rate: 100% False alarm rate: 0%
String SPARQL injection	200 OK	Detection rate: 100% False alarm rate: 10%
Add/modify data with SPARUL injection	200 OK	Detection rate: 100% False alarm rate: 0%
Blind numeric/string SPARQL Injection	200 OK	Detection rate: 100% False alarm rate: 0%
DoS attack with SPARQL injection	200 OK	Detection rate: 80% False alarm rate: 20%

generally refined based on the environment and nature of normal traffic where the WAF is deployed.

6.3. *Evaluation of scanners for detecting SemWebGoat vulnerabilities*

It was discussed in Section 2.2 that application design using Semantic Web technologies does not preclude vulnerability to classical web based attacks that target an application's business logic such as cross-site scripting (XSS), cross-site request forgery (CSRF), clickjacking, session hijacking, HTTP banner disclosure and information disclosure. Let's consider the XSS attack, which manifests when a web application uses unsanitized user input allowing an attacker to store a malicious script into a vulnerable dynamic page or datastore, which unknowingly gets executed by the victim allowing the attacker access to private information. This attack may target Semantic Web applications and legacy applications alike. In fact, a Semantic Web application such as SemWebGoat developed without following secure programming practices will typically retain these vulnerabilities in addition to introducing new ones that target the data access layer and business logic.

This section describes how we empirically tested the ability of popular web application scanners and penetration testing tools to automatically discover Semantic Web vulnerabilities in SemWebGoat and verify if it retains classical vulnerabilities. The detail of the scanners and tools is listed in Table 5 and the summary of the vulnerabilities detected is presented in Table 6.

The results of the evaluation verified that like other web applications, Semantic Web applications are equally vulnerable to attacks such as XSS, CSRF and information disclosure. Moreover, it was established that none of the scanners had support for SPARQLI/SPARULI vulnerability probing due to an entirely different syntax and database structure as compared to SQL.

The evaluation results motivated us to address the need for a scanner that could automatically detect SPARQL/SPARUL vulnerabilities. We explored various options including a process known as *Manual Request Editing* available in scanning tools whereby attack vector parameters can be manually edited (e.g. using a text editor) to supply specific values (e.g. SPARQL strings) in each POST request generated by the scanner, but found it to be a time consuming process. We then looked into the possibility of extending existing scanners to support SPARQLI/SPARULI vulnerability probing. We found a "Fuzzer Tool" in WebScarab (OWASP, 2015i) and Zed Attack Proxy (ZAP) (OWASP, 2015g) that allows a user to load a new file that can contain

Table 5 – Characteristics of the scanners used.

Scanner	Version	Type	Scanning profiles used
Acuneitx WVS	8.0 Free Edition	Standalone	XSS only
Netsparker	2.4.5 Community Edition	Standalone	Default
Websecurify	0.8	Standalone	Default
WebScarab	N/A	Proxy	Fuzzer and Manual
Zed Attack Proxy	1.4.1	Proxy	Active Scan, Fuzzer and Manual

Table 6 – Vulnerabilities detected by the scanners in SemWebGoat.

Scanner	XSS	CSRF	Clickjacking	Session hijacking	HTTP banner disclosure	Info disclosure	SPARQLI/ SPARULI
Acunetix	✗	✗	✗	✗	✗	✗	✗
Netsparker	✗	✗	✗	✓	✓	✓	✗
Websecurify	✗	✗	✗	✗	✓	✓	✗
WebScarab	✓	✓	✓	✓	✓	✓	✶
Zed attack proxy	✓	✓	✓	✓	✓	✓	✶

user-specified parameter values (attack strings), which are automatically substituted and tested against the selected HTTP POST request. We extended WebScarab and ZAP using their respective fuzzer tools for detecting SPARQLI/SPARULI vulnerabilities (marked as * in Table 6) as a consequence of which detection was made possible. The essential steps required to extend the Fuzzer Tool in WebScarab and ZAP are mentioned in Listing 11.

Listing 11: Extension of Fuzzer Tool in WebScarab and ZAP

```
1. Create a sitemap of the web application (SemWebGoat).
2. Select the HTTP request for fuzzing.
3. Using the Fuzzer tool option, upload the text file containing
the fuzz vectors (attack strings).
4. For replacive fuzzing, select the portion of the HTTP request
(e.g. user input value) for automatic substitution.

Example:
Consider String SPARQL Injection Lesson (Section 4.2),
where the user provides input values (e.g. Smith) in the
input field "LastName". In this case input value "Smith" will
be selected and automatically substituted one at a time with
the provided fuzz vector values. (Highlighted as Bold Text)

LastName == Smith
LastName == Smith' .  ?  x ?y ?z .  } #
.
.
LastName == Smith' .  ?s employee:salary ?salary .}#
```

7. Conclusion and future work

Semantic Web applications are prone to injection attacks that can allow an attacker to access or modify unauthorized data. It is important for developers to be aware of the specific vulnerabilities that can plague applications that adopt Semantic Web technologies if secure programming practices are not followed. In this work, we presented and categorized a variety of injection attack variations for the Semantic Web and detailed the design of an insecure Semantic Web application named SemWebGoat that can be used by developers, penetration testers, or students to learn about Semantic Web application vulnerabilities in a safe and legal environment. We also provided a custom set of ModSecurity WAF rules that SemWebGoat users can practice with to be able to detect the SPARQL/SPARUL injection attacks in Semantic Web applications. For the evaluation of our work we conducted a user study, analyzed the detection rate of our custom WAF rule set and performed an empirical analysis of scanners to discover vulnerabilities in SemWebGoat. The results of the user study concluded that developers should be more aware of Semantic Web application vulnerabilities and that the lessons in SebWebGoat help in easily developing an understanding of the security concepts involved. The detection rate of the customized ModSecurity rule set illustrated it as a viable defense measure for blocking SPARQL/SPARUL injection attacks. The evaluation also established that there is a need to upgrade web application scanners for automatic detection of SPARQL/SPARUL vulnerabilities. In the future, a second class of attacks will be studied, for example "Stored SPARQL injection" and a vulnerability analysis of libraries that provide SQL/SPARQL interoperability will be performed.

REFERENCES

Apache. Apache http server project. <http://httpd.apache.org/>; 2015a [accessed 24.12.15].

Apache. Apache tomcat. <http://tomcat.apache.org/>; 2015b [accessed 24.12.15].

Barnett R. WAF virtual patching challenge: securing WebGoat with ModSecurity. Tech Rep, Breach Security 2009.

Berglund A, Boag S, Chamberlin D, Fernandez MF, Kay M, Robie J, et al. XML path language (XPath) 2.0. W3c recommendation, World Wide Web consortium, 2010. <http://www.w3.org/TR/xpath20/>. [accessed 24.12.15].

Birbeck M. CURIE syntax 1.0. W3C working draft, world wide web consortium, <http://www.w3.org/TR/2007/WD-curie-20070307/>; 2007 [accessed 24.12.15].

Broekstra J, Kampman A, Van Harmelen F. Sesame: a generic architecture for storing and querying RDF and RDF schema. The Semantic Web, Vol 2342, Sardinia. Springer; 2002. p. 54–68.

Cheng H, Lu Y-C, Sheu C. An ontology-based business intelligence application in a financial knowledge management system. Exp Syst Appl 2009;36(2):3614–22.

Chinnici R, Moreau J, Ryman A, Weerawarana S. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3c recommendation, World Wide Web Consortium, June <http://www.w3.org/TR/wsdl20/>; 2007 [accessed 24.12.15].

Clark KG, Feigenbaum L, Torres E. SPARQL protocol for RDF. W3c recommendation, World Wide Web Consortium, January <http://www.w3.org/TR/rdf-sparql-protocol/>; 2008 [accessed 24.12.15].

Database Language SQL, Part 2: Foundation (SQL/Foundation). ANSI/ISO/IEC international standard (IS), September 1999.

Doupe A, Cova M, Vigna G. Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Proceedings of detection of intrusions and malware and vulnerability assessment (DIMVA). 2010. p. 111–31.

Erling O. Implementing a SPARQL compliant RDF triple store using a SQL-ORDBMS. Technical Report, OpenLink Software Virtuoso 2001.
Evans SC. Project Leader) Securing WebGoat using ModSecurity, summer of code 2008. OWASP beta level. Version 1.0. Published by OWASP Foundation; November 2008.
Firebug. Firebug web development evolved. <http://getfirebug.com/>; 2015 [accessed 24.12.15].
GitHub. WackoPicko. <https://github.com/adamdoupe/WackoPicko>; 2013 [accessed 24.12.15].
Halfond WGJ, Viegas J, Orso A. A classification of SQL injection attacks and countermeasures. Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA, March 2006.
Harris N. 3store: efficient bulk RDF storage. 1st international workshop on practical and scalable semantic systems, Sanibel Island, Florida, pp. 1–15, 2003.
Hepp M. Goodrelations: an ontology for describing products and services offers on the web. In: Gangemi A, Euzenat J, editors. Knowledge engineering: practice and patterns, volume 5268 of lecture notes in computer science. Springer; 2008. p. 329–46.
Hommeaux EP, Seaborne A. SPARQL query language for RDF. W3c recommendation, World Wide Web Consortium, January <http://www.w3.org/TR/rdf-sparql-query/>; 2008 [accessed 24.12.15].
Howard M, LeBlanc D. Writing secure code. 2nd ed. Redmond, Washington: Microsoft Press; 2003.
IE. IEWatch. <http://download.cnet.com/IEWatch/3000-12512_4-10276586.html>; 2015 [accessed 26.12.15].
InfoSec Institute. What is an SQL injection? SQL injections: an introduction. <http://resources.infosecinstitute.com/sql-injections-introduction/>; 2013 [accessed 26.12.15].
Isotani S, Mizoguchi R, Isotani S, Capeli O, Isotani N, Albuquerque A, et al. A Semantic Web-based authoring tool to facilitate the planning of collaborative learning scenarios compliant with learning theories. Comput Educ 2013;63:267–84.
Manola F, Miller E, McBride B. RDF primer. W3c recommendation, World Wide Web Consortium, February <http://www.w3.org/TR/rdf-primer/>; 2004 [accessed 26.12.15].
Marsa-Maestre I, Hoz E, Gimenez-Guzman J, Lopez-Carmona M. Design and evaluation of a learning environment to effectively provide network security skills. Comput Educ 2013;69:225–36.
McBride B. Jena: a Semantic Web toolkit. IEEE Internet Comput 2002;6(6):55–9.
Middleton B, Halbert J, Coyle FP. Security impacts on semantic technologies in the coming decade. <http://stko.geog.ucsb.edu/sw2022/sw2022_paper4.pdf>; 2015 [accessed 26.12.15].
MIT. MIT Barton catalog MODS. <http://archive.org/details/barton_catalog_mods>; 2015 [accessed 26.12.15].
ModSecurity. Open source web application firewall. <https://www.modsecurity.org/>; 2015 [accessed 26.12.15].
MORE LAB. Injection attacks. <http://www.morelab.deusto.es/code_injection/>; 2010 [accessed 26.12.15].
Orduna P, Almeida A, Aguilera U, Laiseca X, Lopez-de-Ipina D, Goiri AG. Identifying security issues in the Semantic Web: injection attacks in the semantic query languages. JSWEB, p. 4350, Valencia, Spain, September 2010.
ORACLE. Using prepared statement. <http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html>; 2015 [accessed 26.12.15].
OWASP. Category:OWASP WebGoat project. <https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project>; 2015a [accessed 26.12.15].
OWASP. Input validation cheat sheet. <https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet>; 2015b [accessed 26.12.15].
OWASP. OWASP HacmeBank. <https://www.owasp.org/index.php/HacmeBank>; 2015c [accessed 26.12.15].
OWASP. OWASP ModSecurity core rule set. <http://spiderlabs.github.io/owasp-modsecurity-crs/>; 2015d [accessed 26.12.15].
OWASP. OWASP top 10 2010. <http://www.applicure.com/blog/owasp-top-10-2010>; 2015e [accessed 26.12.15].
OWASP, Web application firewall. <https://www.owasp.org/index.php/Web_Application_Firewall>; 2015f [accessed 26.12.15].
OWASP. Zed attack proxy project. <https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project>; 2015g [accessed 26.12.15].
OWASP. Inut validation cheat sheet. <https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet>; 2015h [accessed 26.12.15].
OWASP. WebScarab getting started. <https://www.owasp.org/index.php/WebScarab_Getting_Started>; 2015i [accessed 26.12.15].
Peine H. Security test tools for web applications. IESE Report-Nr 48, 2006.
Pisanelli D. Ontologies in medicine, vol. 102. IOS Press; 2004.
Protege. A free, open-source ontology editor and framework for building intelligent systems. <http://protege.stanford.edu/>; 2015 [accessed 26.12.15].
Sarasamma ST, Zhu QA, Huff J. Hierarchical Kohonenen net for anomaly detection in network security. IEEE Trans Syst Man Cybern Part B Cybern 2005;302–12.
Schmidt M, Hornung T, Lausen G, Pinkel C. SP2Bench: a SPARQL performance benchmark. In: ICDE. 2009. p. 222–33.
Seaborne A, Manjunath G, Bizer C, Breslin J, Das S, Davis I, et al. SPARQL/update: a language for updating RDF graphs. W3c recommendation, World Wide Web Consortium, July <http://www.w3.org/Submission/SPARQL-Update/>; 2008 [accessed 26.12.15].
Sermersheim J. Lightweight directory access protocol (LDAP): the protocol. RFC 4511, June <http://tools.ietf.org/html/rfc4511>; 2006 [accessed 26.12.15].
Shaw RS, Chen C, Harris A, Huang H-J. The impact of information richness on information security awareness training effectiveness. Comput Educ 2009;52(1):92–100.
SPARQL. SPARQL injection-attacking the triple store. <https://www.owasp.org/images/0/0f/Onofri-NapolitanoOWASPDayItaly2012.pdf>; 2012 [accessed 26.12.15].
SQL. SQL injection prevention cheat sheet. <https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet>; 2015a [accessed 26.12.15].
Suto L. Analyzing the effectiveness and coverage of web application security scanners. Case Study. <https://www.beyondtrust.com/resources/white-paper/>; 2007 [accessed 26.12.15].
Suto L. Analyzing the accuracy and time costs of web application security scanners. <https://www.beyondtrust.com/resources/white-paper/>; San Francisco, February 2010 [accessed 26.12.15].
Thuraisingham B. Security standards for the Semantic Web. J Comput Stand Interf 2005;27(3):257–68.
W3C. RDFLib. <http://www.w3.org/2001/sw/wiki/RDFLib>; 2013 [accessed 26.12.15].
W3C. ARC. <http://www.w3.org/2001/sw/wiki/ARC>; 2014 [accessed 26.12.15].
W3C. W3C. <http://www.w3.org/>; 2015 [accessed 26.12.15].
Wireshark. About wireshark. <http://www.wireshark.org/>; 2015 [accessed 26.12.15].
Yang X, Chen Y, Zhang W, Zhang S. Exploring injection prevention technologies for security-aware distributed

collaborative manufacturing on the Semantic Web. Int J Adv Manuf Technol 2011;54:1167–77.
Zhang Z, Zhang C, Ong SS. Building an ontology for financial investment. In: Leung KS, Chan L-W, Meng H, editors. Intelligent Data Engineering and Automated Learning IDEAL 2000. Data mining, financial engineering, and intelligent agents, volume 1983 of lecture notes in computer science. Springer; 2000. p. 308–13.

Hira Asghar completed her MS in Computer and Communications Security from the NUST School of Electrical Engineering and Computer Science. Her research interests include Semantic Web and Information Security.

Zahid Anwar received his Ph.D. and M.S. degrees in Computer Sciences in 2008 and 2005 respectively from the University of Illinois at Urbana-Champaign, USA. Zahid has research experience working at IBM, New York, USA, Intel, Oregon, USA, Motorola, Chicago, USA, the National Center for Supercomputing Applications (NCSA), Urbana, Illinois, USA, xFlow Research and CERN, Geneva, Switzerland on various projects related to information security and data analytics. Zahid holds post-doctorate experience from Concordia University, Montreal, Canada. Currently he is an Assistant Professor at the National University of Sciences and Technology (NUST), Islamabad, Pakistan and an Associate Member of the graduate faculty at the University of North Carolina at Charlotte, USA.

Khalid Latif received his PhD degree in Computer Science from Vienna University of Technology in 2007 and Masters in Computer Science from Islamia University of Bahawalpur in 2003. He is currently serving as Assistant Professor at NUST School of Electrical Engineering and Computer Science. He also holds Project Coordination position for deployment and support of LMS throughout the University. His research interests are in the area of information retrieval, knowledge management, and advanced learning technologies with emphasis on application of Semantic Web concepts and standards.