

A Deliberately Insecure J2EE Semantic Web Application Framework: Injection Attacks and Defense Mechanisms for Web 3.0



By
Hira Asghar
2010-NUST-MS-CCS-17

Supervisor
Dr. Zahid Anwar
Department of Computing

A thesis submitted in partial fulfillment of the requirements for the degree
of Masters in Computer and Communication Security (MS CCS)

In
School of Electrical Engineering and Computer Science,
National University of Sciences and Technology (NUST),
Islamabad, Pakistan.

(April, 2013)

Approval

It is certified that the contents and form of the thesis entitled “**A Deliberately Insecure J2EE Semantic Web Application Framework: Injection Attacks and Defense Mechanisms for Web 3.0**” submitted by **Hira Asghar** have been found satisfactory for the requirement of the degree.

Advisor: **Dr. Zahid Anwar**

Signature: _____

Date: _____

Committee Member 1: **Dr. Khalid Latif**

Signature: _____

Date: _____

Committee Member 2: **Dr. Farooq Ahmed**

Signature: _____

Date: _____

Committee Member 3: **Dr. Fauzan Mirza**

Signature: _____

Date: _____

Abstract

Semantic Web is an emerging technology that is increasing being employed across application and community boundaries to make the World Wide Web more easily interpretable and improve content sharing. Semantic Web uses Resource Description Framework (RDF), as a standardized logical data model to make its data machine-readable and RDF Query/Update (SPARQL/SPARUL) as standard languages to manipulate RDF data. As Semantic Web applications grow increasingly popular, new challenges of protecting them against security threats emerge. Semantic query languages due to their flexible nature are prone to existing attacks such as command injection as well as attacks that exploit new vulnerabilities in these languages making it necessary for application developers to understand the security risks involved when deploying Semantic applications. In this research we have analyzed and categorized the possible injection attacks to which Semantic languages are vulnerable. We have developed a deliberately insecure J2EE Semantic Web application, called SemWebGoat-inspired by the open source vulnerable web application- WebGoat, that offers a realistic teaching environment for exploiting vulnerabilities in web applications. We have also implemented Web Application Firewall (WAF) protection mechanisms for mitigating SPARQL/SPAURL injection attacks. For the evaluation of SemWebGoat we conducted a user study as well as performed experimental evaluation in which we used different web application scanners and penetration testing tools to detect Semantic Web application vulnerabilities. In addition to these, we also carried out testing to evaluate the performance of SemWebGoat under various test scenarios and stress conditions.

The results of the user study concludes that regular web developers are not normally familiar with the injection vulnerabilities demonstrated in SemWebGoat. Moreover web application scanners and penetration testing tools do support SPARQL/SPARUL grammar and are unable to detect its corresponding injection vulnerabilities. The performance testing validates that the use of our WAF rules negligibly effect the performance of SemWebGoat, making it a suitable defense mechanism for vulnerable applications.

We have implemented and evaluated WAF rules using the popular open-source firewall-ModSecurity as well as extended some existing penetration testing tools to support SPARQL/SPARUL injections with the aim of assisting both developers and web administrators in protecting their Semantic Web applications.

Certificate of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis.

I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics which has been acknowledged.

Author Name: **Hira Asghar**
Signature: _____

Acknowledgment

I am thankful to Almighty Allah for granting me persistence and ability to complete MS thesis. I am greatly indebted to my university NUST for providing me with several opportunities for both academic and personal growth. The teaching staff was always available for counseling and university administration was very cooperative.

Next, I would like to express my deep gratitude to Dr. Zahid Anwar, Dr. Khalid Latif and Dr. Farooq Ahmed for their guidance, support and useful input throughout the course of this work. I wish to thank Dr. Zahid Anwar and Dr. Khalid Latif for their mentorship, support and availability. Their enthusiasm has always been a great motivation and they taught me the importance of independent learning and continuous effort. I would also like to thank Dr. Fauzan Mirza for being my committee member and reviewing this thesis.

Last, but not the least, I thank my family and friends for their love and support. They all were a great source of motivation and mental relaxation for me during all this time.

Hira Asghar

Contents

1	Introduction and Background	1
1.1	Introduction	1
1.2	Organization	4
2	Overview of Semantic Web Languages and Tools	5
2.1	RDF	5
2.2	SPARQL	6
2.3	SPARUL	6
2.4	Semantic Web Tools	7
3	Related Work	8
3.1	SPARQL and SPARUL Injection Attacks and their Mitigation	8
3.2	Injection Attacks for RDBMS Semantic Web System and a Defense Mechanism	9
3.3	Issues in ARC2	10
4	Proposed Framework	11
5	Implementation	13
5.1	Implementation of SemWebGoat	13
5.1.1	RDF Database	15
5.1.2	Interface Design	15
5.1.3	Lesson Plans	18
5.2	Implementation of ModSecurity Rule	31
6	Evaluation	36
6.1	User Study	36
6.2	Experimental Evaluation	38
6.3	Performance Testing	38
6.3.1	Evaluation Criteria	39
6.3.2	Results	40

CONTENTS

vii

7 Conclusion and Future Work	43
7.1 Conclusion	43
7.2 FutureWork	44

List of Figures

2.1	A Simple RDF Graph Example	6
4.1	Architecture and Design of Proposed Framework	12
5.1	Implementation View of SemWebGoat	14
5.2	Entity-Relationship Diagram for WebGoat	16
5.3	RDF Data Model of SemWebGoat	16
5.4	Interface of SemWebGoat	17
5.5	Numeric SPARQL Injection Lesson Interface	20
5.6	String SPARQL Injection Lesson Interface	21
5.7	Modify Data with SPARUL Injection Lesson Interface	23
5.8	Add DATA with SPARUL Injection Lesson Interface	25
5.9	Blind Numeric SPARQL Injection Lesson Interface	27
5.10	Blind String SPARQL Injection Lesson Interface	29
5.11	DoS Attack with SPARQL Injection Lesson Interface	31
5.12	XML Injection Lesson Interface	32
5.13	HTML Error Page	34
6.1	Throughput (requests/sec) VS Number of Users	40
6.2	Error Rate (%) VS Number of Users	41
6.3	Average Response Time (ms) VS Number of Users	41

List of Tables

5.1	Lessons of WebGoat and SemWebGoat	18
6.1	Summary of Demographics	37
6.2	Summary of Survey Questions and Results	37
6.3	Characteristics of Scanners and Penetration Testing Tools	39
6.4	Vulnerabilities Detected	39

Chapter 1

Introduction and Background

This chapter gives an introduction of the overall work done and it includes briefing about the technologies used.

1.1 Introduction

Semantic Web, considered to be the next generation of Web 2.0, has been expanding rapidly in recent years. After the publication and recommendation of several standards by W3C (World Wide Web Consortium), 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, business, scientific research and bioinformatics that has raised issues of security for Semantic Web data. Semantic Web uses a standardized logical data model namely Resource Description Framework (RDF)[1], to make its data machine-readable. Semantic Web data is 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[2], Sesame[3], Openlink Virtuoso[4] and 3Store[5], on a single machine. Simple Protocol and RDF Query Language (SPARQL)[6], is the standard query language for RDF data recommended by W3C. SPARQL/Update (SPARUL)[7], is an extension to the SPARQL query language standard that is used to insert, delete and update RDF data.

A number of incidents have taken place in past years which proved that the previous query languages[8], [9], [10] such as SQL, XPath and LDAP etc are highly vulnerable to attacks based on non-sanitized user inputs. In such attacks, the attacker directly concatenates the malicious query strings with the inputs to take over control of the application to achieve desired results.

To create awareness among the developers about the vulnerabilities present in these query languages a number of deliberately insecure web applications such as HacmeBank[11] and WebGoat[12] were developed and different safeguard measures[13], [14], [15] were proposed and provided for the mitigation of vulnerabilities. Like previous query languages, SPARQL/SPARUL are also vulnerable to injection attacks. Various authors have addressed the security in the Semantic Web. Thuraisingham did an early research[16] on the security needs in the Semantic Web where he examined the importance of implementing security mechanisms in different layers of the application. Agarwal and Sprick[17] identified and developed access control policies and their corresponding mechanisms for Semantic Web services. In [18], [19] authors identified the basic vulnerabilities in Semantic Web query/update languages and proposed solutions[19], [20] to prevent such vulnerabilities but so far no such Semantic Web application has been developed that can provide a realistic teaching environment for exploiting vulnerabilities in Semantic Web applications. Thus limited safeguard measures have been provided to protect Semantic Web applications against such vulnerabilities.

WebGoat is among the most well known and widely used deliberately insecure J2EE web applications that is maintained by OWASP and is designed to teach web application security lessons. WebGoat-5.3 includes lessons on possible web 2.0 attacks. Lessons in WebGoat-5.3 are split up into eighteen main categories depending on the nature of the threat and more than sixty lessons have been demonstrated. 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, Weak Session Cookies etc. 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, code and solutions. Some of the lessons in WebGoat-5.3 require third-party software such as WebScarab[21], Firebug[22], IEWatch[23] and Wireshark[24] to exploit the vulnerability demonstrated. For developing SemWebGoat the 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 follows a similar teaching style.

ModSecurity is an open source WAF that can identify and block attempts to exploit a specific vulnerability in an application. In web applications if vulnerabilities are not found early in the design or testing phases but rather in production phase then remediation of vulnerabilities becomes expensive and requires extensive source code modification. In these situations there is enough time for malicious users to exploit the vulnerability in unprotected

web applications. Web application firewalls offer an impressive and ideal platform for dealing with web application vulnerabilities by examining everything from user entry fields to URLs, and headers as well as observing user sessions and cookies, and blocking leakage of sensitive data. ModSecurity provides a set of generic core rulesets that cover areas including XML protection, malicious software detection, error detection and application level attacks detection [25]. In 2008, Stephen Craig Evans lead an OWASP Summer of Code (SoC) Project entitled as “Securing WebGoat with ModSecurity” [26]. The purpose of this project was to create custom ModSecurity rulesets that, in addition to the Core Set, will protect WebGoat-5.2 Standard Release from as many of its vulnerabilities as possible without changing any line of source code. Different rulesets were implemented to mitigate each vulnerability and each ruleset had its own HTML error file page that appeared on the browser when every 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 ModSecurity rules; as a safeguard measure against the SPARQL/SPARUL injection attacks. This paper presents the implementation of rules that externally address the vulnerabilities demonstrated in SemWebGoat.

Web application scanners are used for detecting vulnerabilities in web applications. In addition to this they generate observance reports and also suggest a method for mitigating each vulnerability that has been detected. The most well-known vulnerabilities that the scanners investigate most extensively are SQL Injection (SQLI), Cross Site Scripting (XSS), Cross-Site Request Forgery (CSRF) and Information Disclosure. HacmeBank[11], WebGoat[12] and WackoPicko[27] 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 performance of eleven scanners against their own test application (WackoPicko) and explained the reasons for a scanner’s failure or success [28]. Suto in [29] showed the comparison of three scanners on detecting web application vulnerabilities and in [30] presented the evaluation of seven scanners on the basis of their detection capabilities and time efficiency. Peine compared WebGoat with a real-world application for evaluating the user interface of the seven scanners [31]. To examine the vulnerabilities in SemWebGoat five different scanners and penetration testing tools have been used and their performance for detecting the vulnerabilities in Semantic Web applications has been evaluated.

The Major contributions of our research are:

- Analyzing and categorizing the attacks specific to Semantic Web lan-

guages.

- Developing a deliberately insecure J2EE Semantic Web application so that every user (such as programmer, penetration tester and student) could learn and practice Semantic Web application security holes in a safe and legal environment.
- Providing a testing platform to security professionals to test their security tools (security professionals frequently need to test tools against a platform known to be vulnerable to ensure that they perform well).
- Implementing ModSecurity rules for protecting the Semantic Web applications that have been already developed but are vulnerable to SPARQL/SPARUL injection attacks.
- Extending some existing penetration testing tools to support SPARQL/SPARUL injections with the aim of assisting both developers and web administrators in protecting their Semantic Web applications.

1.2 Organization

The rest of the thesis is structured as follows: Chapter 2 presents the overview of Semantic Web languages. Chapter 3 outlines the work done in our domain. In Chapter 4, we have explained the architecture and design of our proposed framework. In Chapter 5, the detail implementation of SemWebGoat and ModSecurity Rule has been discussed that includes the interface, RDF database and lessons implemented in SemWebGoat. Chapter 6, presents the evaluation criteria and the results. Chapter 7 concludes the whole study and presents future work direction.

Chapter 2

Overview of Semantic Web Languages and Tools

This chapter provides introduction to Semantic Web languages and tools that have been used for developing SemWebGoat.

2.1 RDF

RDF[1] is a language used for representing Semantic Web data. Each RDF statement consists of subject, predicate and object, where the subject and object are used to represent any two things in the world and a predicate is used to define the relationship between the two things. The main aspect of RDF is that the names for triples must be kept global, so the following RDF conditions apply to store the RDF data in the database:

- 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 predicate. Arc starts from a subject node and ends at the object node. For example the RDF graph in Figure 2.1 states “JamesWatson has age whose value is 27”, where JamesWatson is a subject, age represents the predicate and 27 represents the object value. The predicate (exterms:age) in this example is represented in the XML QName (Qualified Name) form. As the name of the URIs are rather cumbersome and long so

in diagrams they are represented in XML QName form. The part before the “:” represents a namespace and is known as namespace prefix. In RDF graphs, predicates are mostly represented in the QName format when written in RDF/XML as it is convenient. The nsprefix:localname form corresponds to the URI of the namespace that is concatenated with the localname.

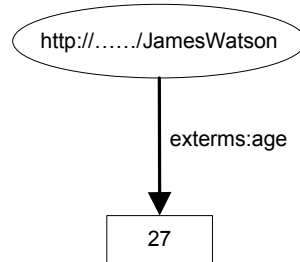


Figure 2.1: A Simple RDF Graph Example

2.2 SPARQL

For querying the RDF database SPARQL[6] language is used. SPARQL query language has four query forms:

- **SELECT:** Returns the variables and their bindings in 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 SELECT 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 Table I that will return the age of JamesWatson. A Variable is indicated by the ‘?’ prefix and will return the binding for the ?age.

2.3 SPARUL

SPARUL[7] is a standard language used for performing updates to the RDF models/graphs in a database.

SPARUL can be used to perform following tasks:

Table I: SPARQL Query for Figure 2.1

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

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

2.4 Semantic Web Tools

Number of Semantic Web tools including Jena[2], ARC[32], RDFLib[33] and Protege[34] are available that are assisting in the development of Semantic Web applications. Jena framework is used for creating and querying the back-end RDF database. Jena offers a collection of Java libraries and tools for developing Semantic Web applications, servers and tools. It includes an API that supports reading, processing and writing of RDF data into XML, N-triples and Turtle formats. Jena framework also includes the servers that allow the publishing of RDF data to other applications by using various protocols including SPARQL. In Jena, an RDF graph is known as RDF model and the Model interface is used to represent it.

Chapter 3

Related Work

This chapter gives a brief overview of the work done in domain of Semantic Web vulnerabilities and their mitigation.

Constructing applications using Semantic Web technologies is a relatively new trend and therefore previous query languages such as SQL and XPath are still employed because of developer familiarity along with new Semantic query languages and due to this reason it is more vulnerable than the previous web applications. Semantic Web application security issues have been relatively unexplored and while researchers have identified some of the Semantic Web vulnerabilities in the past but we did not find any significant security assessment tools that address these vulnerabilities in real Semantic applications.

3.1 SPARQL and SPARUL Injection Attacks and their Mitigation

In [18] authors presented the three basic Semantic Web injection techniques: SPARQL injection, Blind SPARQL injection and SPARUL injection. SPARQL/SPARUL injection is a code injection technique that exploits the security vulnerabilities in Semantic Web applications. SPARQL/SPARUL commands are injected from the web form into the database of an application to get 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 in other query languages like SQL provide tools to avoid the code injection attacks, such as Java API supports prepared statements[14] or the

use of parameterized queries[35]. Parameterized queries permit the database to differentiate between data and code, in spite of what user input is provided. In this coding method developers are forced to define all the SQL code before passing any parameters to the query. Use of prepared statements ensure that the malicious SQL commands entered by an attacker will not be able to change the intent of the query. In [20] Orduna et al. provided a solution with the name of `ParameterizedString` for preventing SPARQL/SPARUL injection attacks, that work in the same way as prepared statements do for preventing SQL injection attacks. Orduna et al. sent a patch for Pellet 1.5.1 and Jena 2.5.5, adding support for the `ParameterizedString` object in `QueryEngine`, `QueryFactory` and `UpdateFactory`. 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 recoding which is not usually economical and is generally ignored if the development team is under pressure to roll out the software product. Our research provides a suitable solution that can externally address Semantic Web vulnerabilities and can protect applications without having to change any line of source code.

3.2 Injection Attacks for RDBMS Semantic Web System and a Defense Mechanism

In [19] authors presented the classification and detection of possible SPARQL/SQL injection attacks for a RDBMS-based Semantic Web system. Such systems use SPARQL and SQL both as the query languages. RDBMS-based systems store data in a relational database for persistence and use a translator module to translate SPARQL data queries to SQL data queries. In this paper authors have classified injection attacks as either SPARQL-oriented or SQL-oriented. To detect the injection attacks they proposed a parse tree validation technique which represents the syntax structure of a string according to the grammar. In this proposed technique 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. So if the user inputs do not store the content in the parse tree nodes as the intended queries do then it means there is an injection attack. In brief 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 they developed a prototype system “SemGuard”, that was used as a plug-in in a Java application. In this paper only a basic SPARQL injection attack has been discussed and no classification or solution for Blind SPARQL and SPARUL injection attacks has been provided. This motivated us to classify the basic SPARQL/SPARUL injection attacks and provide a appropriate defense mechanism for their mitigation.

3.3 Issues in ARC2

In [36] Onofri and Napolitano presented the possibility of basic SPARQL injection attacks in Semantic applications. For demonstrating the SPARQL injection they developed a vulnerable Semantic login where the user was able to login without even entering the correct password. For the mitigation of SPARQL injection attacks they recommended the use of parametric query and data validation techniques as well as suggested proper programming techniques to write the SPARQL code in safe manner. They also addressed the same issues in ARC2 and demonstrated that ARC2 version v2011-12-01 and possibly the lower versions are vulnerable to Blind SQL Injection and Cross Site Scripting vulnerabilities. ARC[37] provides RDF and SPARQL functionalities to PHP applications and store triples into a MySQL database so it is possible to attempt Blind SQL Injection attack against the RDB-based applications by injecting SQL commands in the SPARQL WHERE clause. As these issues have been fixed by the ARC vendors so for the protection of Semantic applications users are recommended to update ARC2 to the latest release or manually fix the “ARC2.StoreEndpoint.php” and other files. In this research no new safeguard mechanism has been proposed. This encouraged us to propose a new safeguard measure as well as provide a real Semantic application where users can not only test basic SPARQL injections but also other types of possible injection attacks in Semantic applications.

Chapter 4

Proposed Framework

This chapter discusses the overall design of our proposed framework and all its technical details.

Figure 4.1 explains the architecture and design of our proposed framework. The objective of this framework is to protect Semantic Web application from as many of its vulnerabilities as possible by using ModSecurity rule. Major components of proposed framework are:

- **Web Browser/User:** User will send the request to retrieve the data. The request can either be valid or malicious. For sending requests directly to the web server (Apache in our case), the web browser is configured on localhost with port 80 but in case an intercepting proxy such as WebScarab is required, the port can be changed to WebScarab's port so that every request can route through it.
- **WebScarab:** WebScarab is a tool that intercepts HTTP requests and responses and exploits the vulnerabilities in Semantic Web applications by editing the parameters at runtime. WebScarab is required for two of the lesson plans. WebScarab is configured by setting the HTTP proxy as localhost and port as 80 so that it can forward each request to the Apache web server.
- **ModSecurity:** Every request sent by the user is intercepted by a WAF such as ModSecurity in our case to check attack attempts and block them accordingly. ModSecurity is configured on Apache web server by setting the listening port as 80. Apache web server has been configured as a reverse proxy so that it can communicate with Apache Tomcat-a J2EE compliant server.

- **SemWebGoat:** A deliberately insecure J2EE 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 the 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.

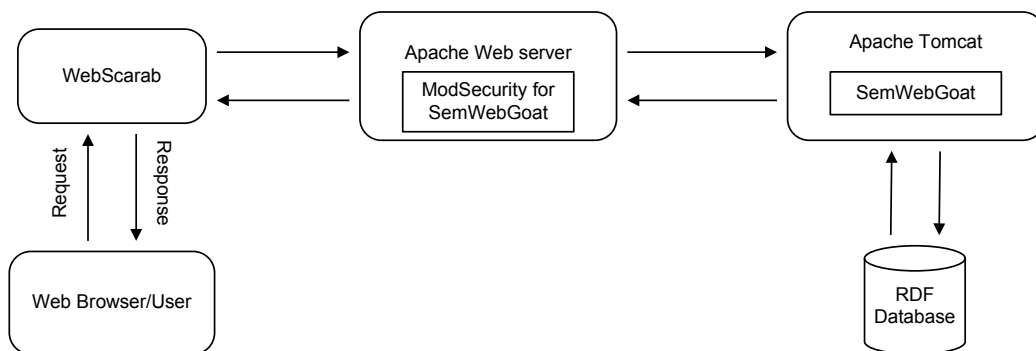


Figure 4.1: Architecture and Design of Proposed Framework

Chapter 5

Implementation

The chapter presents the the implementation details of SemWebGoat and describes the implementation of ModSecurity Rule that has been implemented for detecting SPARQL/SPARUL injection attacks.

5.1 Implementation of SemWebGoat

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 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 is 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 considering 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 at the end of the hacker whereby he/she can use information exploited from one vulnerable input field as a stepping stone to exploit another weakness.

Like a dynamic web site, the architecture of a web application also revolves around the navigation of the web pages. To represent the functionality and architecture of the web system, designers use different models and viewpoints; such as Implementation View, Site Map, Deployment Model, Analysis and Design Model. Implementation View describes the abstraction of web pages and the relationships between them. In Implementation View hyperlinks are used to represent relationships between the web pages. At abstraction level the Implementation View of a web application is just like a Site Map of a web system. Figure 5.1 presents the Implementation View

of SemWebGoat, where the two hyperlinks (`<<link>>`, `<<build>>`) are used to represent the navigation paths through the web application and the parameter values (such as `StationNumber`, `LastName`) are being passed as arguments and are used by the server pages to build the client pages.

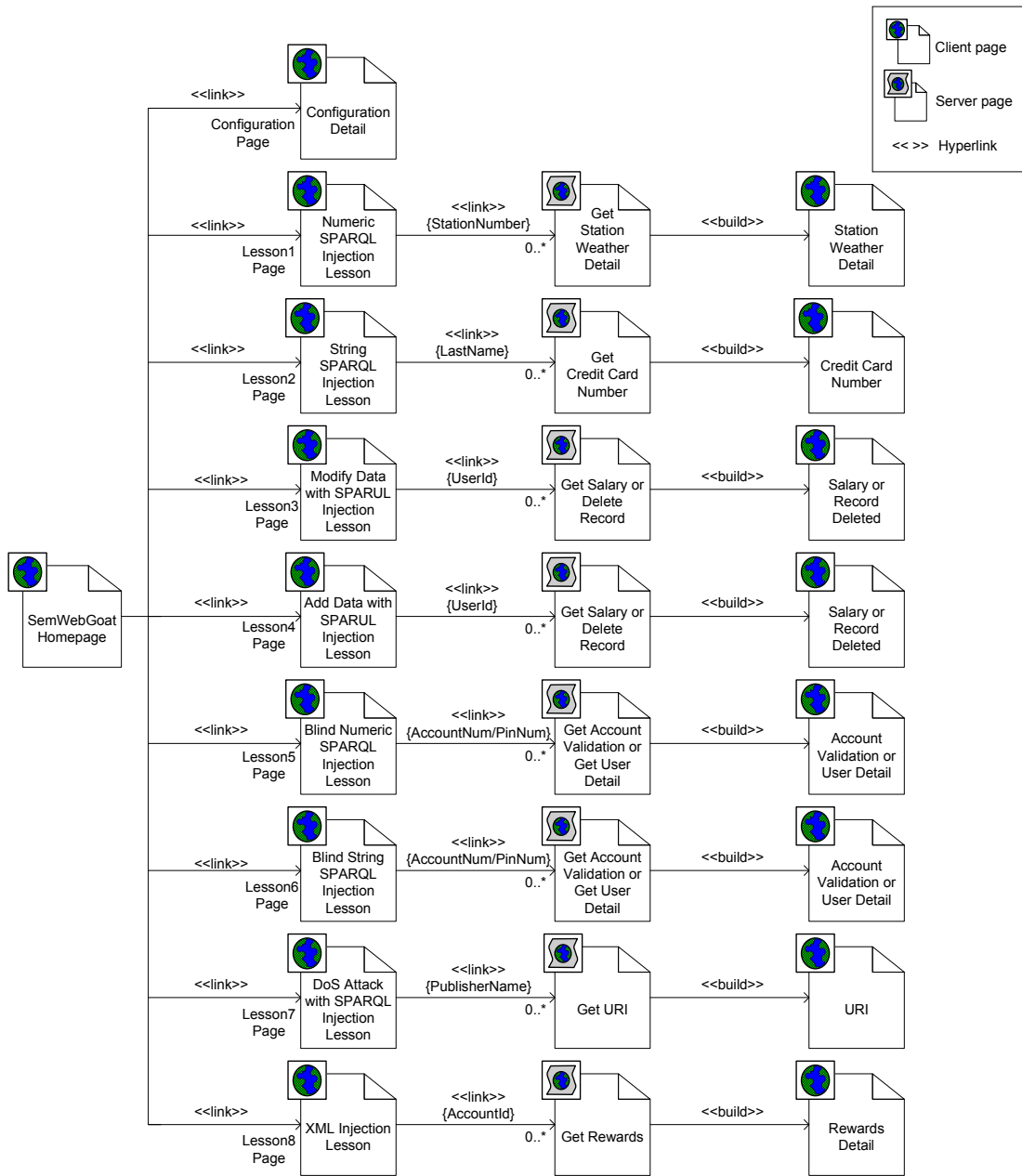


Figure 5.1: Implementation View of SemWebGoat

5.1.1 RDF Database

The source code of WebGoat has been analyzed to gather information about the data model of WebGoat. The WebGoat database contains a number of SQL tables that include information about products, weather, employees etc. Each lesson in WebGoat is based on a certain scenario and has access to a certain database table according to the requirement of that scenario. For our lessons requirements the number of SQL tables that have been used in implementing the SemWebGoat RDF database are presented in Figure 5.2. This figure explains the type of data and the fields stored in the database tables of WebGoat. The entity-relationship diagram defines the following database tables:

- **Employee:** Contains all the necessary personal and public information (such as names, credit card numbers, social security numbers, addresses, phones numbers) regarding the employees of a company.
- **Weather:** Contains weather data of various stations that includes the station number, station name, state name, maximum and minimum temperature of each station.
- **Pins:** Contains information related to credit cards that includes the credit card numbers, cardholder names, Personal Identification Numbers (PIN) and the account numbers.
- **Rewards:** Contains information about the rewards and the account IDs that are required to win the rewards.

Figure 5.3 presents the same data and fields that are presented in Figure 5.2, but in the form of RDF graph/model. RDF Data models are used for representing the Semantic Web data and are considered similar to the conceptual database modeling approaches such as class diagrams or entity-relationship diagrams. In RDF data model web resources are described in the form of triples. To have better understanding of this RDF model/graph you may refer to Section 2 or [1].

5.1.2 Interface Design

SemWebGoat interface design as shown in Figure 5.4, is similar to WebGoat and like WebGoat, SemWebGoat also provide users with hints, code and solutions. The interface elements of SemWebGoat are summarized below.

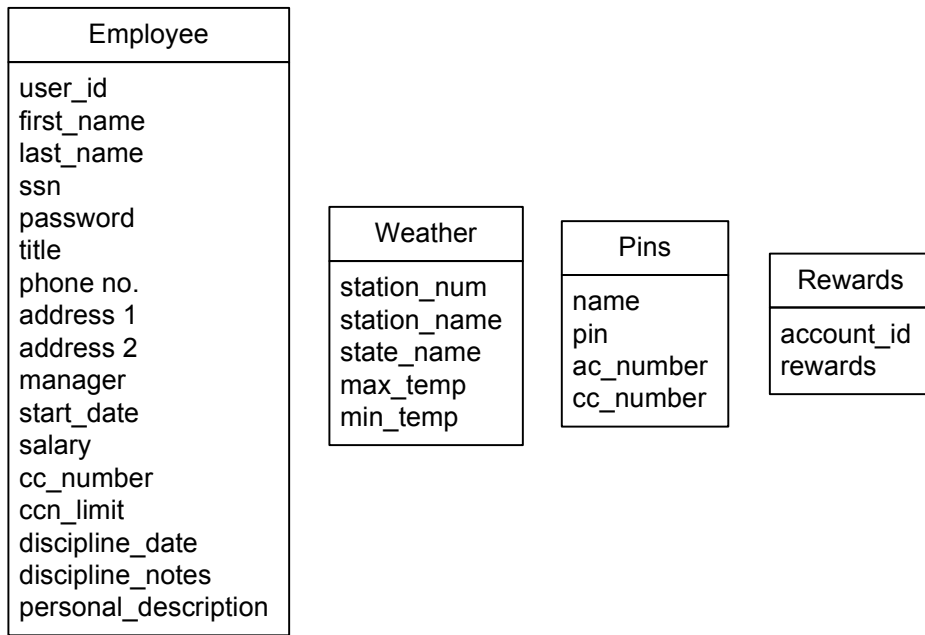


Figure 5.2: Entity-Relationship Diagram for WebGoat

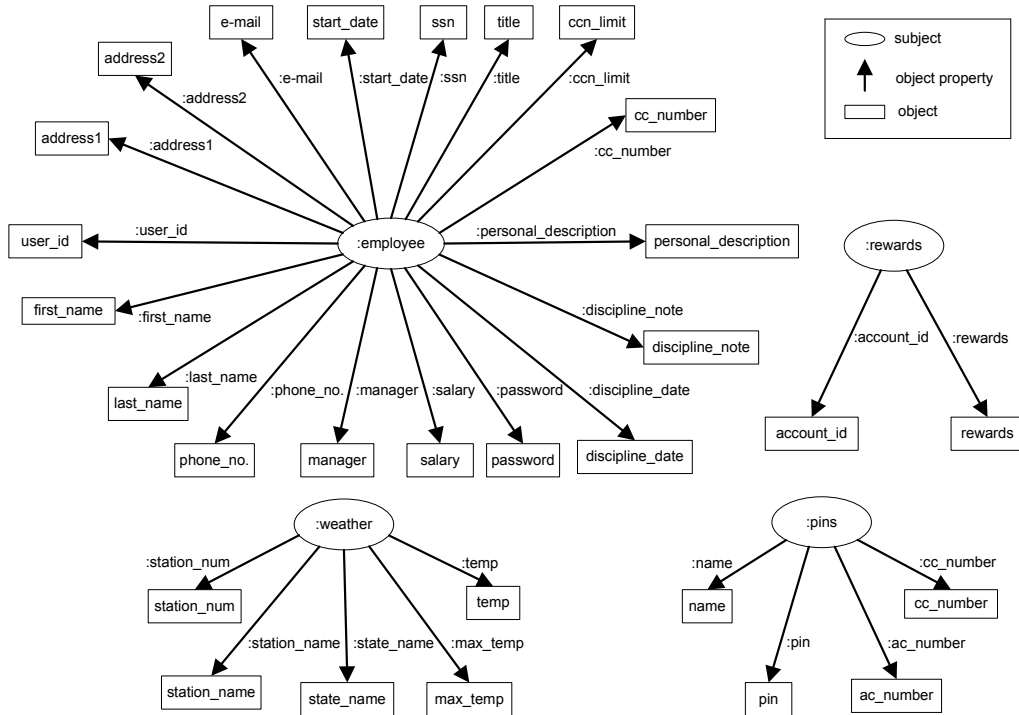


Figure 5.3: RDF Data Model of SemWebGoat

1. **Introduction and Injection Flaws Categories:** Introduction category contains short introduction of SemWebGoat and WebScarab plus provides the configuration settings required for using SemWebGoat. Injection Flaws category contains the list of lessons demonstrated in SemWebGoat.
2. **Hints:** Shows the technical hints to solve the lesson.
3. **Show Params:** Shows the HTTP Request Parameters.
4. **Show Cookies:** Shows the HTTP Request Cookies.
5. **Lesson Plan:** Shows the goals and objectives of the particular lesson.
6. **Show Java:** Shows the underlying source code.
7. **Solution:** Shows the complete solution for that particular lesson.
8. **Restart this Lesson:** This link will restart the lesson.

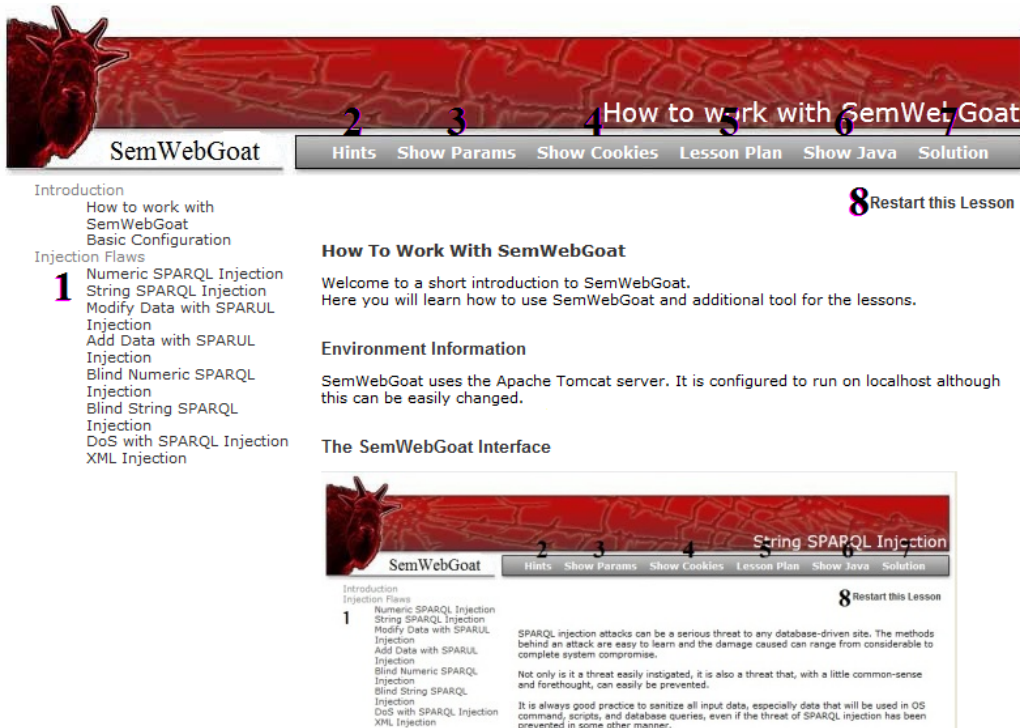


Figure 5.4: Interface of SemWebGoat

5.1.3 Lesson Plans

WebGoat includes a number of security lessons on different types of SQL injections under the category of “Injection Flaws”. In SemWebGoat under the same category (Injection Flaws) all lessons have been listed that demonstrate the possible injection attacks to which Semantic Web applications are vulnerable. For the implementation of lessons we have followed the naming schemes and scenarios of WebGoat. For better understanding of lessons and to provide a good comparison of SQL injection attacks with SPARQL/SPARUL injection attacks the SemWebGoat lessons will be discussed along with those of WebGoat. Table 5.1 contains the lessons that will be discussed in this section. A total of eight lessons have been demonstrated in SemWebGoat; two of which will require third-party software such as “WebScarab” to exploit the vulnerabilities and the remaining six lessons teach the users how to exploit the vulnerabilities through user input fields.

Table 5.1: 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

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 the drop-down list. The goal is to inject a SQL string that results in all the weather data being displayed. The WebGoat application is taking the input from the select box and inserting it at the end of a pre-formed SQL command. In such case the user would need WebScarab

to intercept the HTTP request and to concatenate the malicious string with the URL-Encoded value. When the user will replace the URL-Encoded value with the SQL string that is provided in Table IIa, it will display the weather data of all the stations since the SQL statement “101 or 1=1!” always resolves to true.

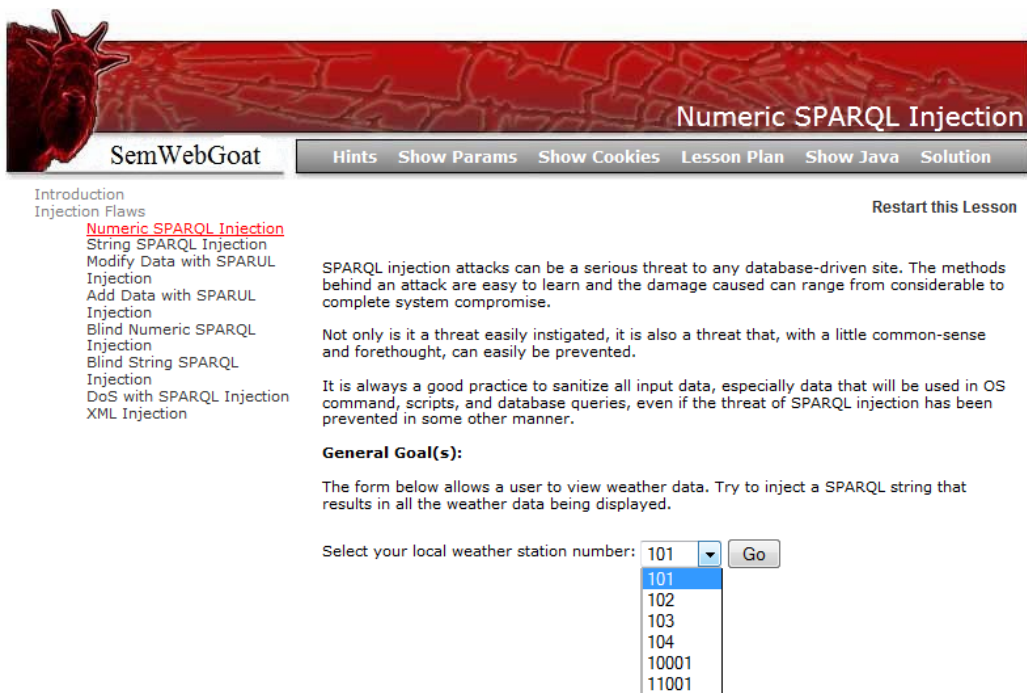
Table IIa: Numeric SQL Injection

<pre>Pre-formed SQL query: SELECT * FROM weather_data WHERE station = '"+ stationNum + "' Valid URLEncoded value: 101 Malicious SQL string: 101 or 1=1!</pre>
--

In the equivalent SemWebGoat lesson as shown in Figure 5.5, the user has to inject a SPARQL string instead of SQL string to display the weather data of all stations. Essential details required to understand this lesson are provided in Table IIb. The “ + stationNum+” in the pre-formed SPARQL query is taking the input value from the select box and will return the data of that particular station number that will be selected by the user from the drop-down list. 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 variable (such as *s*, *abc*, *subject* etc) for retrieving all the subjects except the “uri” because this variable has been already bound to return the weather data for the single station whereas 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 the data that is associated with the predicates specified in the query and “#” will comment out the rest of the pre-formed query. By injecting this malicious query the user will get the weather data of all stations together.

String SPARQL Injection

This lesson uses the same scenario that has been used by “String SQL Injection” of WebGoat. According to the scenario the user can view credit card numbers by entering the last names in the input field. The goal is to inject a SQL string that would display all the credit card numbers at the same time. When the user will enter “Smith’ OR ‘1’=‘1” instead of “Smith” as shown in Table IIIa, it will display all the credit card numbers because the SQL statement always resolves to true.



Introduction
Injection Flaws

Numeric SPARQL Injection
String SPARQL Injection
Modify Data with SPARUL Injection
Add Data with SPARUL Injection
Blind Numeric SPARQL Injection
Blind String SPARQL Injection
DoS with SPARQL Injection
XML Injection

Restart this Lesson

SPARQL injection attacks can be a serious threat to any database-driven site. The methods behind an attack are easy to learn and the damage caused can range from considerable to complete system compromise.

Not only is it a threat easily instigated, it is also a threat that, with a little common-sense and forethought, can easily be prevented.

It is always a good practice to sanitize all input data, especially data that will be used in OS command, scripts, and database queries, even if the threat of SPARQL injection has been prevented in some other manner.

General Goal(s):

The form below allows a user to view weather data. Try to inject a SPARQL string that results in all the weather data being displayed.

Select your local weather station number:

101
102
103
104
10001
11001

Figure 5.5: Numeric SPARQL Injection Lesson Interface

Table IIb: 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: 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 .
} #
```

In the corresponding SemWebGoat lesson the difference in the scenario is that this time the user has to inject a SPARQL string instead of a SQL string

Table IIIa: String SQL Injection

<pre>Pre-formed SQL query: SELECT * FROM user_data WHERE last_name = '"+ LastName + "' Valid input: Smith Malicious SQL string: Smith' OR '1'='1</pre>
--

to display all credit card numbers. Essential details required to understand this lesson are provided in Figure 5.6 and Table IIIb. The “+LastName+” in the pre-formed SPARQL query is taking the input value from the input field and will return the credit card number of that particular user whose last name has been entered. Like the previous lesson, in this scenario it is also necessary for the user to have prior knowledge of the predicates that are being used in back-end query. In the malicious query user can use any variable for retrieving all the subjects except the “uri” whereas for retrieving the object values the user can use any variable. In the malicious query “?s” and “?cc_number” will return all the data that is associated with the predicates specified in the query and “#” will comment out the rest of the pre-formed query. In the result all the credit card numbers will be displayed at the same time.

String SPARQL Injection

SemWebGoat [Hints](#) [Show Params](#) [Show Cookies](#) [Lesson Plan](#) [Show Java](#) [Solution](#)

[Introduction](#)
[Injection Flaws](#)
 Numeric SPARQL Injection
String SPARQL Injection
 Modify Data with SPARUL Injection
 Add Data with SPARUL Injection
 Blind Numeric SPARQL Injection
 Blind String SPARQL Injection
 DoS with SPARQL Injection
 XML Injection

[Restart this Lesson](#)

SPARQL injection attacks can be a serious threat to any database-driven site. The methods behind an attack are easy to learn and the damage caused can range from considerable to complete system compromise.

Not only is it a threat easily instigated, it is also a threat that, with a little common-sense and forethought, can easily be prevented.

It is always a good practice to sanitize all input data, especially data that will be used in OS command, scripts, and database queries, even if the threat of SPARQL injection has been prevented in some other manner.

General Goal(s):

The form below allow users to view their credit card numbers. Try to inject a SPARQL string that results in all the credit card numbers being displayed. Try the user name of 'Smith'.

Enter your last name:

Figure 5.6: String SPARQL Injection Lesson Interface

Table IIIb: 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: Smith' .
    ?s employee:cc_number ?cc_number .
} #

```

Modify Data with SPARUL Injection

This lesson uses the same scenario that has been used by “Modify Data with SQL Injection” of WebGoat. According to the scenario the user can view salaries associated with a user.id. As the input field is vulnerable to SQL injection so the goal is to inject SQL string to modify the salary for user_id “jsmith” (that belongs to John Smith). The solution and source code details are provided in Table IVa.

Table IVa: Modify Data with SQL Injection

```

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

```

In the equivalent SemWebGoat lesson as shown in Figure 5.7, the users are provided with two input fields. The first input field allows a user to view the salaries by entering the user_ids. The other input field allows deletion of the record by entering the user_ids. For the implementation of this lesson two input forms have been used because SELECT query uses SPARQL syntax whereas MODIFY query uses SPARUL syntax and we cannot use SELECT and MODIFY queries together in a single statement. To inject this SPARUL string the user should have prior knowledge about the subject URI, predicate and object value (salary). This can be achieved using the first input field to get information about the URI and the salary of John by entering his user_id “jsmith” and the predicate is provided in the pre-formed query. After

viewing his URI and salary the user can use the second input field to update his salary. According to the scenario the user has to update salary where user_id is “jsmith” so user should not concatenate the SPARUL statement with this user_id as this will delete the record of “jsmith” . Rather the user needs to use some other user_id such as *mstooge* or *lstooge* and concatenate the SPARUL statement with it. To update the salary the user would always need to perform two actions together: DELETE and INSERT. The preformed query provided in Table IVb illustrates that it is a DELETE query, so for updating the salary record the user needs to delete the existing salary record before inserting the new one. It is also necessary to first delete the existing record because otherwise the new salary record, will be added with his previous record of salary and in such a case when the user searches for John’s salary he/she will get two results. The malicious SPARUL string clearly illustrates that firstly John’s salary has been deleted and then the new salary record has been added. After modifying John’s salary the user can use the first input field to view the updated salary record.

SemWebGoat **Modify Data with SPARUL Injection**

[Hints](#) [Show Params](#) [Show Cookies](#) [Lesson Plan](#) [Show Java](#) [Solution](#)

[Introduction](#) [Restart this Lesson](#)
[Injection Flaws](#)
 Numeric SPARQL Injection
 String SPARQL Injection
[Modify Data with SPARUL Injection](#)
 Add Data with SPARUL Injection
 Injection
 Blind Numeric SPARQL Injection
 Injection
 Blind String SPARQL Injection
 Injection
 DoS with SPARQL Injection
 XML Injection

SPARUL Injection can change the meaning of the query. SPARUL injections can modify the back-end database by the use of INSERT, DELETE and MODIFY statements.

General Goal(s):

The forms shown below allow the users to view salaries of employees and delete the employee record by entering their user_id . The second form is vulnerable to SPARUL Injection. In order to pass this lesson, use SPARUL Injection to modify the salary for user_id **jsmith** that belongs to John Smith. After modifying the salary use the first form to check the modified salary.

To view the employee salary use the below form:

Enter user_id:

To delete the employee record use the below form. Try user_id lstooge or mstooge.

Enter user_id:

Figure 5.7: Modify Data with SPARUL Injection Lesson Interface

Add Data with SPARUL Injection

This lesson uses the same scenario that has been used by “Add Data with SQL Injection” of WebGoat. This lesson is conceptually very similar to

Table IVb: 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: lstooge' .
    employee:John employee:salary '20000'. }
INSERT DATA { employee:John employee:salary
    '5000'. } #

```

the previous lesson. This time the user has to insert new data instead of modifying the stored data. According to the scenario the user can view salaries associated with a user.id. As the input field is vulnerable to SQL injection so the goal is to inject a SQL string to add a new salary record with any arbitrary name. The solution and source code details are provided in Table Va.

Table Va: Add DATA with SQL Injection

```

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

```

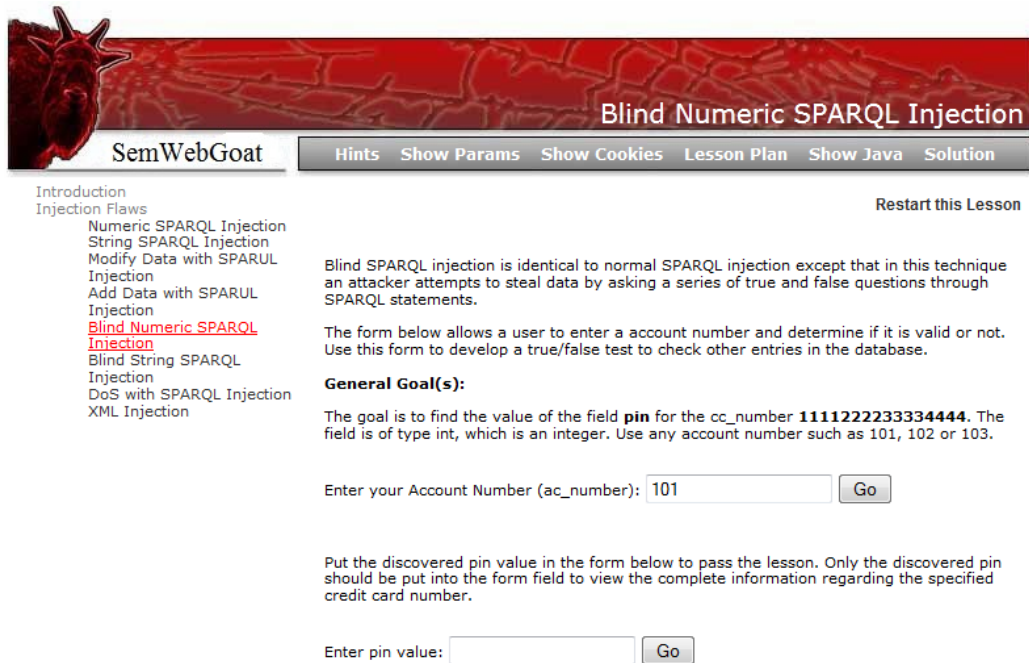
In the corresponding SemWebGoat lesson as shown in Figure 5.8, the users are provided with two input fields. The first input field allows a user to view the salaries by entering the user_ids. The other input field allows the deletion of the record by entering the user_ids. For the implementation of this lesson two input forms have been used because SELECT query uses SPARQL syntax whereas INSERT query uses SPARUL syntax and we cannot use SELECT and INSERT queries together in a single statement. To add a new record of salary to the database user can use INSERT query that is provided in Table Vb. The first closing bracket has been used to close the DELETE query and then the INSERT query has been concatenated to it. After entering this SPARUL statement a new user record with name “Nelson”, user_id “nome” and salary “10000” will be added into the database.

a user to enter an account number and determine if it is valid or not. Such input field can be used to develop a true/false test to check other entries in the database. The goal is to find the pin value (that is of type int) for the credit card number “1111222233334444”. The SQL string provided in Table VIa would either return valid or invalid. If it would return valid then it means that the pin value is greater than 10000 and if it would return invalid then it means that the pin value is less than 10000. By entering a number of similar SQL statements with different pin values the user can guess the correct pin value. The user can only pass the lesson by entering the correct pin value (which is 2364) associated with the specified credit card number.

Table VIa: Blind Numeric SQL Injection

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

In the corresponding SemWebGoat lesson as shown in Figure 5.9, the users are provided with two input fields. The first input field allows a user to check the valid and invalid account numbers and a user can only view the information of credit card when the correct pin value will be entered in the second input field. For the implementation of this lesson, ASK query has been used behind the first input form as it returns the result in a boolean (true/false) form. Like previous lessons, in this scenario it is also necessary for the user to 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 Table VIb. This string uses the FILTER clause. FILTER allows the filtering of results on certain conditions. For filtering the numbers FILTER clause uses inequalities and equalities. For determining the pin value the user can change the pin value (in the provided SPARQL string “10000” is denoting the pin value) in each statement and can look for each boolean result. Each boolean result will provide a clear clue to the user regarding the correct pin value. After determining the correct pin value (which is 2364) the user can enter it into the second input field to pass the lesson.



SemWebGoat Hints Show Params Show Cookies Lesson Plan Show Java Solution

Introduction
Injection Flaws Restart this Lesson

Numeric SPARQL Injection
String SPARQL Injection
Modify Data with SPARUL Injection
Add Data with SPARUL Injection
Blind Numeric SPARQL Injection
Blind String SPARQL Injection
DoS with SPARQL Injection
XML Injection

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.

The form below allows a user to enter an account number and determine if it is valid or not. Use this form to develop a true/false test to check other entries in the database.

General Goal(s):

The goal is to find the value of the field **pin** for the cc_number **1111222233334444**. The field is of type int, which is an integer. Use any account number such as 101, 102 or 103.

Enter your Account Number (ac_number):

Put the discovered pin value in the form below to pass the lesson. Only the discovered pin should be put into the form field to view the complete information regarding the specified credit card number.

Enter pin value:

Figure 5.9: Blind Numeric SPARQL Injection Lesson Interface

Table VIIb: Blind Numeric SPARQL Injection

```

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

```

Blind String SPARQL Injection

This lesson uses the same scenario that has been used by “Blind String SQL Injection” of WebGoat. This lesson is conceptually very similar to the previous lesson. This time the user has to search for a string, 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 or not. The goal is to find the value of the field name for the credit card number “4321432143214321”.

The SQL strings provided in Table VIIa include the SUBSTRING method because without using it the user would be trying to compare the complete string to one letter and this will not help. SUBSTRING method includes (STRING,START,LENGTH); in the first malicious SQL string it is comparing the first letter with “H”. As the correct name associated with the specified account number is “Jill” so it will return false. By using several more statements with different letters, inequality and equality operators the user will be able to determine the first letter. To determine the second letter, the user can change the SUBSTRING parameters as shown in the second malicious SQL string. After using several more statements the user will be able to determine the second letter too. In a similar way user can find the third and fourth letters of the name. Then the user can enter the correct name in the input field to clear the lesson.

Table VIIa: Blind String SQL Injection

```

Pre-formed SQL query: SELECT * FROM user_data
                      WHERE userid = " + accountNo + "
Valid input: 101
Malicious SQL string:101 AND
                      (SUBSTRING((SELECT name FROM pins
                      WHERE cc_number='4321432143214321'),
                      1, 1) < 'H' );
Malicious SQL string:101
                      AND (SUBSTRING((SELECT name FROM pins
                      WHERE cc_number '4321432143214321'),
                      <u>2</u>, 1) <= '<u>h</u>' );

```

In the equivalent SemWebGoat lesson as shown in Figure 5.10, the users are provided with two input fields. The first input field allows a user to check valid and invalid account numbers and a user can only view the credit card information when the correct pin value will be entered in the second input field. In Table VIIb, 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 value of “?name” with the letter “h”. The caret sign has been used to indicate that the string for “?name” must start with “h”, not just have it somewhere within the string. The “i” as the third parameter for the regex operand means that the regular expression is case insensitive but in case the user, want it to be case sensitive then only the first two parameters would be required. By using the first malicious SPARQL string the user can find the first letter of name and then

Table VIIb: Blind String SPARQL Injection

```

Pre-formed SPARQL query: "ASK
    WHERE {?uri pins:ac_number '"+ accountNo +''.
    }
Valid input: 101
Malicious SPARQL string: 101'.
    ?s pins:cc_number '4321432143214321' .
    ?s pins:name ?name.
    Filter regex(?name, '^h','i')
    }#
Malicious SPARQL string: 101'.
    ?s pins:cc_number '4321432143214321' .
    ?s pins:name ?name.
    Filter regex(?name, '^Jh', 'i')
    }#

```

field. In case when the malicious SPARQL statement that is provided in Table VIII will be passed through the input field, it will make the server busy for at least 10 minutes. In the provided malicious SPARQL string “?s” will return all the subjects, “?p” will return all the predicates and “?o” will return all the object values stored in the RDF dataset. The user can use any random variables in the string instead of “s”, “p” and “o” such as “?x ?y ?z” or “?aa ?ab ?ac” will give the same results as “?s ?p ?o”. Such strings are generally used to fetch all the data stored in the back-end RDF dataset and when the dataset contain millions of triples then such strings can lead to DoS attack. These query strings illustrates how easy it is for an attacker to fetch all the data as these strings does not require any prior knowledge regarding the back-end dataset.

Table VIII: DoS Attack with SPARQL Injection

```

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

```


Figure 5.11: DoS Attack with SPARQL Injection Lesson Interface

XML Injection

Typically web applications especially Ajax based use XML to store data or to exchange messages. XML documents are usually treated as databases that include sensitive information. In web services XML messages are used to send 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. The Semantic Web applications also use RDF/XML (one of the most popular RDF formats on the web) to write graph data. The purpose of implementing this lesson in SemWebGoat is to demonstrate that Semantic Web applications are also vulnerable to XML Injections as they also use XML to transmit sensitive information. This SemWebGoat lesson uses the same scenario that has been used by “XML Injection” of WebGoat. According to the scenario as shown in Figure 5.12, the user gets the list of rewards when the user enters the account.id. The goal is to try to add more rewards to allowed list of rewards by using XML injection. To add more rewards the user would need WebScarab to intercept and modify HTTP response as illustrated in Table IXb.

5.2 Implementation of ModSecurity Rule

A WAF works as a filter that applies a set of rules to an HTTP conversation. Number of attacks can be identified and blocked by customizing the rules according to the applications. We have implemented ModSecurity rules to provide application level protection to Semantic Web applications against

XML Injection

SemWebGoat

Hints Show Params Show Cookies Lesson Plan Show Java Solution

Restart this Lesson

Introduction
Injection Flaws
Numeric SPARQL Injection
String SPARQL Injection
Modify Data with SPARUL Injection
Add Data with SPARUL Injection
Blind Numeric SPARQL Injection
Blind String SPARQL Injection
DoS with SPARQL Injection
[XML Injection](#)

Applications typically use XML to store data or send messages. When used to store data, XML documents are often treated like databases and can potentially contain sensitive information. XML messages are often used in web services and can also be used to transmit sensitive information. The semantics of XML documents and messages can be altered if an attacker has the ability to write raw XML. The XML content can easily be intercepted and altered by using different tools. Developers often do not validate the information that is received. This lesson will teach the attacker to find and modify the HTTP response to add more rewards to your allowed set of rewards.

General Goal(s):

SemWebGoat-Miles Reward Miles show all the rewards available. Your goal is to add more rewards to your allowed set of rewards. Your account ID is '836239'.

Welcome to SemWebGoat-Miles Reward Miles Program.

Rewards available through the program:

- SemWebGoat t-shirt 20 Pts
- SemWebGoat Secure Kettle 50 Pts
- SemWebGoat Mug 30 Pts
- SemWebGoat Core Duo Laptop 2000 Pts
- SemWebGoat Hawaii Cruise 3000 Pts

Enter your Account ID:

The items will be shipped to your address.

Figure 5.12: XML Injection Lesson Interface

Table IXa: XML Injection in WebGoat

```
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>
Modified 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>
```

SPARQL/SPARUL injection attacks. We have configured ModSecurity-2.7 on Apache- 2.4 server and added our rules in modsecurity.conf file. The general rule syntax for ModSecurity has been described as “SecRule VARI-

Table IXb: XML Injection in SemWebGoat

```
Actual HTTP Response: <binding name="rewards">
<literal>WebGoat t-shirt 20 Pts,
WebGoat Secure Kettle 50 Pts,
WebGoat Mug 30 Pts</literal>
</binding>
Modified HTTP Response with XML Injection:
<binding name="rewards">
<literal>WebGoat t-shirt 20 Pts,
WebGoat Secure Kettle 50 Pts,
WebGoat Mug 30 Pts,
WebGoat Core Duo Laptop 2000 Pts,
WebGoat Hawaii Cruise 3000 Pts</literal>
</binding>
```

ABLES OPERATOR ACTIONS”. 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 part specifies what to do if any match occurs.

For writing the ModSecurity rules, all the SPARQL/SPARUL injection attack strings that have been used to exploit the vulnerabilities demonstrated in SemWebGoat have been analyzed. All the attacking strings are using single quote (') to concatenate the malicious query with the input and are ending with closing curly bracket (}) and hash symbol (#) so we implemented ModSecurity rules that could block all the input strings that contain these regular patterns in specific order. The ModSecurity rules mentioned in Table X are using “chain” action to combine the two rules into a single logical rule that is known as *rule chain*. The first rule will match “” in the input string and the second rule will match whether the input string is ending with “}#” or not. In second rule “\s*” is used to match the whitespace zero or more times and “\$” sign is used to match the end of the string. The chained rules are considered similar to AND conditional statements. The actions that are specified in chained rule are only triggered if all the variable checks return true. If any of the variable checks return false then the entire rule chain returns false. After configuring this rule chain if any SPARQL/SPARUL injection attack string will be entered in the input field of SemWebGoat, the ModSecurity will intercept the transaction and in result a HTML error page as shown in Figure 5.13 will appear on the screen.

The description of the ModSecurity chain rule is as follows:

Table X: ModSecurity Rule for Blocking SPARQL/SPARUL Injection Attacks

```
SecRule "REQUEST_BODY" "@rx '" "phase:2,t:urlDecode,
log,auditlog,
redirect:http://localhost:8080/errorpage.html,chain"
SecRule "REQUEST_BODY" "@rx (}\s*#)$" "t:urlDecode"
```

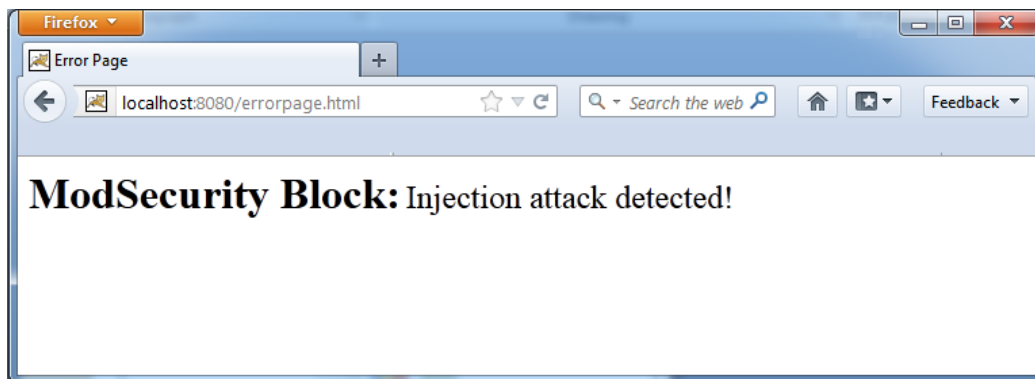


Figure 5.13: HTML Error Page

- **SecRule:** It is used for creating a rule.
- **REQUEST_BODY:** For optimization ModSecurity rules are defined in terms of phases to detect the attacks. ModSecurity can check in the following five phases: Request headers (phase 1), Request body (phase 2), Response headers (phase3), Response body (phase 4) and Logging (phase 5). All the user inputs forms in SemWebGoat are using POST method to retrieve data that is why “REQUEST_BODY” variable has been used in the provided ModSecurity rule. The “REQUEST_BODY” variable is specifying ModSecurity to look at the request message body to match the regular expression.
- **@rx:** Number of operators can be used in rules such as @pm, @rm, @beginsWith, @endsWith, @contains, @within and @streq. Operator starts with @ symbol followed by the operator name. In the given ModSecurity rule @rx is specifying ModSecurity to match the regular pattern that is placed next to it.
- **phase:2:** It is specifying the processing phase action.
- **t:urlDecode:** This transformation function is used for decoding the

URL-Encoded input strings.

- **log:** It is used to log error message on successful rule match.
- **auditlog:** It is used to log in-progress transactions to the auditlog.
- **redirect:** It is used for intercepting the malicious transaction and will redirect it to the specific link.
- **chain:** It is used to chain the rule with the rule that immediately follows it.

This ModSecurity rule has been thoroughly tested against all the valid and malicious user inputs of SemWebGoat based on the MIT Barton dataset and the WebGoat default dataset in the backend and we did not receive any false positives or false negatives. The possibility of false positives occurring is very minor especially for the datasets consisting of words from the English language. Lets assume, case of a login web page that takes user_name and password values such as Oxford's}# or 'abc'def}# but these are rare cases. To completely remove the possibility of false positives these check should be applied at the input validation level where the context is well-known. Other than this, the provided ModSecurity chain rule is a appropriate safeguard measure for protecting Semantic Web applications against number of injection attacks.

Chapter 6

Evaluation

The chapter presents the three evaluation methods that have been used for the testing and evaluation of our work.

For the evaluation of our work we conducted a user study to determine the comprehensiveness and difficulty level of training as well as performed an empirical comparison study against existing penetration testing tools to figure out the overlap in the support for a SPARQL vulnerability analysis. Finally we did performance testing of the web application under realistic traffic workloads and both normal and malicious traffic. Since SemWebGoat is designed to be an e-commerce application deployed in a setting where multiple users would invoke its various functions our goal was to stress test it with and without the ModSecurity controls. For each evaluation method we have used Intel (R) Core(TM) i3 machine with 2.10 GHz CPU, 8GB RAM and Microsoft Windows 7, Service Pack 1 with 64-bit operating system.

6.1 User Study

SemWebGoat was provided to some programmers and post-graduate students and after completing the lessons, users were asked to answer some survey questions. Our survey results show although users were aware of Semantic Web concepts but they were poorly familiar with the vulnerabilities demonstrated in SemWebGoat and the lessons improved their security concepts of Semantic Web applications. 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. Table 6.1 shows a summary of the demographics and Table 6.2 presents the summarized survey results.

Table 6.1: Summary of Demographics

Total Users	Total: 6 Post-graduate Students: 3 (50%) Programmers: 3 (50%)
Age	23-30 years
Working Experience	6 months-4 years
Current General Computer Use	6-10 hours a day
Awareness of Semantic Web and Web Security Tools	Yes: 50% Some Extent: 50%
Work Experience in Domain of Semantic Web	Yes: 50% No: 50%

Table 6.2: Summary of Survey Questions and Results

Survey Questions	Results
The lessons improve my concepts of vulnerabilities in Semantic Web applications.	Yes: 100% Some extent: 0% No: 0%
Each lesson has a well-designed scenario to teach each injection technique.	Yes: 66.7% Some extent: 33.3% No: 0%
Each lesson provides sufficient material, such as hints, java code, solutions for exploiting the vulnerability.	Yes: 100% Some extent: 0% No: 0%
The lesson covers the injection techniques that I would like to know about.	Yes: 100% Some extent: 0% No: 0%
The lessons stimulate my further interest in learning other security technology/concepts.	Yes: 50% Some extent: 50% No: 0%
Were you already aware of the vulnerabilities demonstrated in SemWebGoat.	Yes: 0% Some extent: 50% No: 50%
Application is user-friendly.	Yes: 100% Some extent: 0% No: 0%
Each lesson was related to security of Semantic Web applications.	Yes: 100% Some extent: 0% No: 0%
These lessons provide better understanding of security concepts than the verbal lessons.	Yes: 100% Some extent: 0% No: 0%
Semantic Web Developers should be aware of such vulnerabilities.	Yes: 100% Some extent: 0% No: 0%
The estimated time you spent to complete all lessons.	2 hours
In your view which lesson is most difficult.	Blind Numeric SPARQL Injection Blind String SPARQL Injection
In your view which lesson is the easiest one.	XML Injection

6.2 Experimental Evaluation

In experimental evaluation we have used some of the well-known web application scanners and penetration testing tools to find the vulnerabilities in SemWebGoat. The detail of the scanners and tools is listed in Table 6.3 and the summary of the vulnerabilities detected is presented in Table 6.4. The scanners already had built in support for detecting conventional web attacks such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), SQL Injections (SQLI), ClickJacking, Session Hijacking, HTTP Banner Disclosure, Information Disclosure. Even database vulnerabilities such as SQLI were probed. None of the scanners had support for SPARQL/SPARULI vulnerability. Probing due to its completely different syntax and database structure as compared to SQL. The results of the scanners assured that SPARQL/SPARUL syntax is different from the SQL syntax that is why none of the scanner/tool was able to identify SQLI attacks. The results of the scanners verified that like other web applications, Semantic Web applications are also vulnerable to attacks such as XSS, CSRF and Information Disclosure while on the other hand the evaluation results also addressed the need of a scanner that could automatically detect the possibility of SPARQL/SPARUL injection attacks. Manual Request Editing is available where Editors can be used to supply specific SPARQL values in each HTTP request but it is a time consuming process so we conducted research how to extend scanner to support SPARQL/SPARULI vulnerability probing. We find a “Fuzzer Tool” in WebScarab and Zed Attack Proxy (ZAP) that is used to find vulnerabilities such as SQLI and XSS in any web application by performing automatic substitution of parameter values in the HTTP request that is forwarded to the server. This tool allows a user to load a new “.txt” file that can contain user-specified parameter values (attack strings) which are automatically substituted and tested against the selected HTTP request. We extended the WebScarab and ZAP using its Fuzzer Tool for detecting SPARQL/SPARULI vulnerabilities as a consequence of which detection was made possible (see Table 6.4).

6.3 Performance Testing

Performance testing is an important part of any distributed or Web application testing plan. Early identification of software load limitations helps to configure the system appropriately to avoid unexpected crashes.

Table 6.3: Characteristics of Scanners and Penetration Testing Tools

Name	Version	Type	Scanning Profiles Used
Acunetix WVS	8.0 Free Edition	Standalone	XXS 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 6.4: Vulnerabilities Detected

Name	XXS	CSRF	SQLI	ClickJacking	Session Hijacking	HTTP Banner Disclosure	Info Disclosure	SPARQLI/SPARULI
Acunetix WVS	✗	✗	✗	✗	✗	✗	✗	✗
Netsparker	✗	✗	✗	✗	✓	✓	✓	✗
Websecurify	✗	✗	✗	✗	✗	✓	✓	✗
WebScarab	✓	✓	✗	✓	✓	✓	✓	*
Zed Attack Proxy	✓	✓	✗	✓	✓	✓	✓	*

6.3.1 Evaluation Criteria

A tool named JMeter[39] by Apache is a stress testing tool that can be used to measure the performance of an application under different load type. JMeter can be used to make a graphical analysis of performance or to test a network, server or object under heavy concurrent load. We have used five different test scenarios to evaluate the performance of SemWebGoat and WebGoat under different load environments. For varying the load we have increased the number of threads/users. In each scenario there were six requests per user and all the users were started concurrently. In each test scenario, number of users varied from 1000 to 5000 and the total number of samples/requests that were processed varied from 6000 to 30000. Five different testing scenarios that have been used are:

- SemWebGoat performance with all valid requests.
- WebGoat performance with all valid requests.
- SemWebGoat performance with 50% malicious requests.
- WebGoat performance with 50% malicious requests.
- SemWebGoat performance with 50% malicious requests and with Mod-Security.

For measuring the performance of an application we recorded the following parameters:

- **Throughput:** Total number of requests/sec that the server handled successfully during the test.
- **Error Rate (%):** The percentage of requests that were lost or delayed during the transmission.
- **Response Time (ms):** Time (in milliseconds) taken by the server to process all the requests representing how efficiently the server is handling the load.

6.3.2 Results

Figure 6.1 shows the throughput achieved by the proposed system against number of users. The graph clearly shows that in all test scenarios initially when the number of users were less, the system gave the highest throughput. As the number of users increases, throughput decreases.

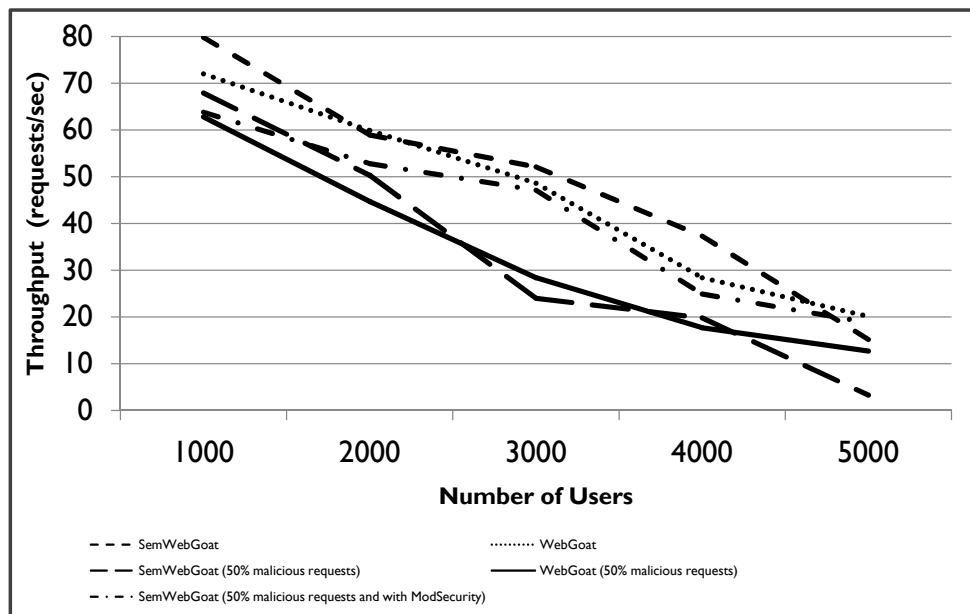


Figure 6.1: Throughput (requests/sec) VS Number of Users

Figure 6.2 shows the error rate against number of users. The graph illustrates that the error rate was low when the users were less but as the number of users increased, the error rate also increased.

Figure 6.3 shows the average response time against the number of users. The graph clearly shows that as the number of users increases, the number of

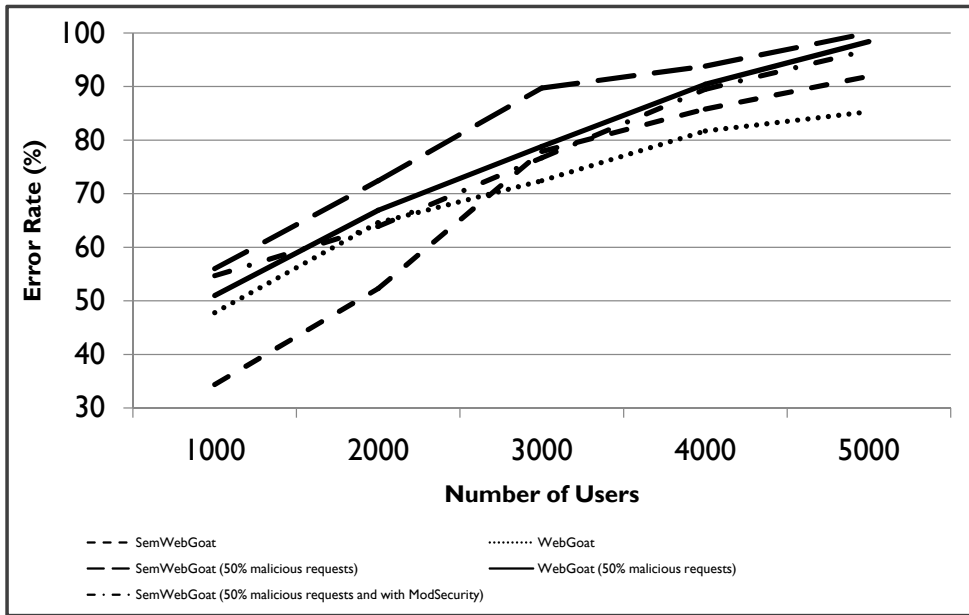


Figure 6.2: Error Rate (%) VS Number of Users

requests increases and therefore the response time for processing all requests increases.

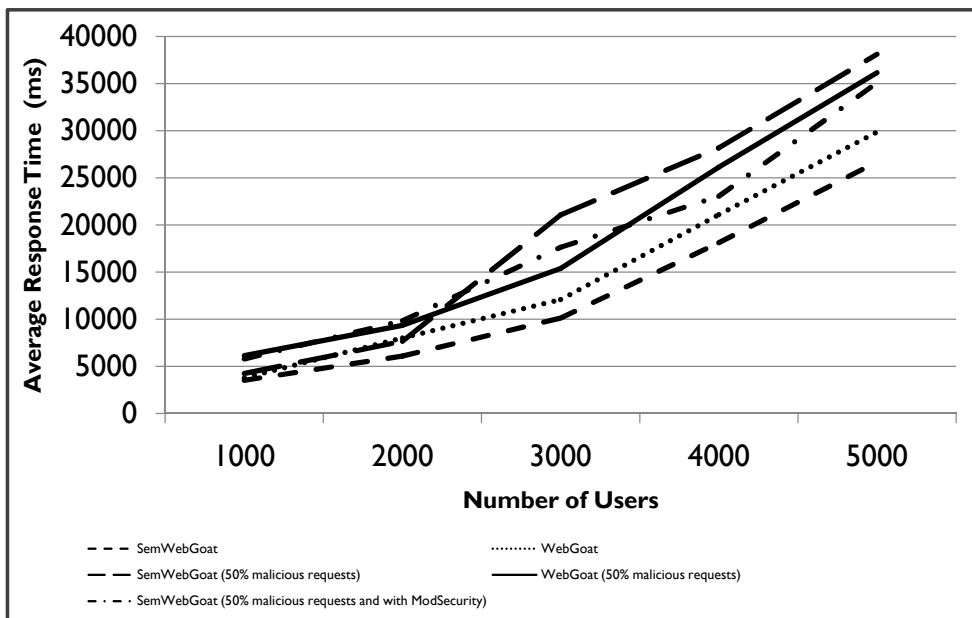


Figure 6.3: Average Response Time (ms) VS Number of Users

While evaluating the performance of application, the stress testing can be stopped when for a measured throughput the measured response time gets too high or when the throughput decreases to 0 requests/sec or when the error rate increases to 100%. The results show that in case of both test scenarios (with 50% malicious requests) the throughput is less whereas the error rate and average response time is greater as compared to other test scenarios because in case of malicious requests server needs to fetch more data for the response. The graphs also show that in case of SemWebGoat (with 50% malicious requests) when the users are increased to 5000, throughput almost decreased to 0% and error rate increased to 100%. In case of the other remaining test scenarios error rate is also near to 100% and throughput is near to 10% when number of users are increased to 5000. The results of SemWebGoat under ModSecurity validates that ModSecurity negligibly affects the performance of an application and in fact improves the error rate and response time in case of 50% malicious traffic so ModSecurity is a suitable solution for protecting applications.

Chapter 7

Conclusion and Future Work

The chapter summarizes overall work done. It also discusses the dimensions in which this work could be further expanded.

7.1 Conclusion

Semantic Web applications are prone to injection attacks that can allow an attacker to access or modify the unauthorized data. It is important for the developers to know how a secure Semantic Web application can be developed, that can assure the integrity, confidentiality and availability of the web application. We presented the basic injection attacks for Semantic Web and explained the implementation of a insecure J2EE Semantic Web application that can be used by developers/penetration testers/students to learn and practice Semantic Web application vulnerabilities in a safe and legal environment. We also provided ModSecurity rules that can be used to detect the SPARQL/SPARUL injection attacks in Semantic Web applications. For the evaluation of our work we conducted a user study as well as carried out experimental evaluation and performance testing. The results of user study concluded that developers should be aware of Semantic Web application vulnerabilities and the SemWebGoat lessons helped the users to understand the Semantic Web application security concepts in a user-friendly environment. The results of experimental evaluation addressed the need of up-gradation in web application scanners so that the SPARQL/SPARUL injections can be detected automatically. The results of performance testing illustrated that SemWebGoat and WebGoat had similar performance under various test scenarios and load types and that the use of ModSecurity firewall improved performance in case of attacks and did not affect the performance of SemWebGoat in a considerable manner overall.

7.2 FutureWork

We are planning to distribute SemWebGoat as an open source software so that every user could make use of it. Moreover in future the second class of vulnerabilities will be identified; for example Stored SPARQL injection attacks and libraries that provide SQL/SPARQL interoperability will be analyzed in Semantic Web applications.

Bibliography

- [1] F. Manola, E. Miller, and B. McBride. RDF Primer. W3c recommendation, World Wide Web Consortium, February 2004. <http://www.w3.org/TR/rdf-primer/>
- [2] B. McBride. Jena: A Semantic Web Toolkit. *IEEE Internet Computing*, vol. 6, no. 6, pp. 55-59, December 2002.
- [3] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. *The Semantic Web*, Sardinia, pp. 54-68, Springer, June 2002.
- [4] O. Erling. Implementing a SPARQL Compliant RDF Triple Store using a SQL-ORDBMS. Technical Report, OpenLink Software Virtuoso, 2001
- [5] N. Harris. 3store: Efficient Bulk RDF Storage. *1st International Workshop on Practical and Scalable Semantic Systems*, Sanibel Island, Florida, pp. 1-15, 2003.
- [6] E. P. Hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3c recommendation, World Wide Web Consortium, January 2008. <http://www.w3.org/TR/rdf-sparql-query/>
- [7] A. Seaborne, G. Manjunath, C. Bizer, J. Breslin, S. Das, I. Davis, S. Harris, K. Idehen, O. Corby, K. Kjernsmo and B. Nowack. SPARQL/Update: A Language for Updating RDF Graphs. W3c recommendation, World Wide Web Consortium, July 2008. <http://www.w3.org/Submission/SPARQL-Update/>
- [8] Database language SQL, Part 2: Foundation (SQL/Foundation). ANSI/ISO/IEC International Standard (IS), September 1999.
- [9] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie and J. Simeon. XML Path Language (XPath) 2.0. W3c recommendation, World Wide Web Consortium, December 2010. <http://www.w3.org/TR/xpath20/>

- [10] J. Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511, June 2006. <http://tools.ietf.org/html/rfc4511>
- [11] OWASP HacmeBank. <https://www.owasp.org/index.php/HacmeBank> (Last accessed: December 2012).
- [12] Category:OWASP WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project (Last accessed: December 2012).
- [13] OWASP Web Application Firewall. https://www.owasp.org/index.php/Web_Application_Firewall (Last accessed: December 2012).
- [14] Using Prepared Statement. <http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html> (Last accessed: December 2012).
- [15] Input Validation Cheat Sheet. https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet (Last accessed: December 2012).
- [16] B. Thuraisingham. Security Standards for the Semantic Web. *J. Comput. Stand. Interfaces*, vol. 27, no. 3, pp.257-268, 2005.
- [17] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. *In ICDE*, pp. 222-233, 2009.
- [18] P. Orduna, A. Almeida, U. Aguilera, X. Laiseca, D. Lopez-de-Ipina, and A. G. Goiri. Identifying Security Issues in the Semantic Web: Injection Attacks in the Semantic Query Languages. *JSWEB*, pp. 4350, Valencia, Spain, September2010.
- [19] X. Yang, Y. Chen, W. Zhang and S. Zhang. Exploring Injection Prevention Technologies for Security-aware Distributed Collaborative Manufacturing on the Semantic Web. *The International Journal of Advanced Manufacturing Technology*, vol.54, pp. 1167-1177, June 2011.
- [20] Injection Attacks. http://www.morelab.deusto.es/code_injection/ (Last accessed: December 2012).
- [21] WebScarab Getting Started. https://www.owasp.org/index.php/WebScarab_Getting_Started (Last accessed: December 2012).
- [22] Firebug Web Development Evolved. <http://getfirebug.com/> (Last accessed: December 2012).

- [23] IEWatch. <http://www.iewatch.com/> (Last accessed: December 2012).
- [24] Wireshark. <http://www.wireshark.org/> (Last accessed: December 2012).
- [25] R. Barnett. WAF Virtual Patching Challenge: Securing WebGoat with ModSecurity. Technical Report, Breach Security, January 2009.
- [26] S. C. Evans (Project Leader), Securing WebGoat Using ModSecurity, Summer of Code 2008. OWASP Beta Level, Version 1.0, Published by OWASP Foundation, November 2008.
- [27] WackoPicko. <https://github.com/adamdoupe/WackoPicko>(Last accessed: December 2012).
- [28] A. Doupe, M. Cova and G. Vigna. 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)*, pp. 111-131, 2010.
- [29] L. Suto. Analyzing the Effectiveness and Coverage of Web Application Security Scanners. Case Study, October 2007.
- [30] L. Suto. Analyzing the Accuracy and Time Costs of Web Application Security Scanners. San Francisco, February 2010.
- [31] H. Peine. Security Test Tools for Web Applications. IESE Report-Nr 48, 2006.
- [32] ARC. <http://www.w3.org/2001/sw/wiki/ARC> (Last accessed: December 2012).
- [33] RDFLib. <http://www.w3.org/2001/sw/wiki/RDFLib> (Last accessed: December 2012).
- [34] Protege. <http://protege.stanford.edu/> (Last accessed: December 2012).
- [35] SQL Injection Prevention Cheat sheet. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet (Last accessed: December 2012).
- [36] SPARQL Injection-Attacking the Triple Store. <https://www.owasp.org/images/0/0f/0nofri-NapolitanoOWASPDaYItaly2012.pdf> (Last accessed: December 2012).

- [37] ARC. <http://www.w3.org/2001/sw/wiki/ARC> (Last accessed: December 2012).
- [38] MIT Barton Catalog MODS. http://archive.org/details/barton_catalog_mods (Last accessed: December 2012).
- [39] Apache JMeter. <http://jmeter.apache.org/> (Last accessed: December 2012).