Essay of <A Search-based Testing Approach for XML Injection Vulnerabilities in Web Applications>, presented by Team 6

(I followed the form for the Coursework#1.)

1. What is the software engineering problem authors are trying to solve?

In most cases, web applications communicate with web services (SOAP & RESTful). Specifically, web applications act as a front-end to the web services, which contain the business logic. A hacker might not have a direct access to those web services, but can still provide malicious inputs to the web application, thus potentially compromising related services.

XML injection (XLMi) attack is a typical example that provide malicious inputs as parameters to the web application to generate a malicious XML output. More specifically, XMLi is an attack technique that aims at manipulating the logic of XML-based applications or services. The aim of this type of attack is to compromise the system itself or other systems that process the malicious XML messages. For instance, executing malicious SQL query generated by XMLi might cause the exposure of confidential information from the back-end databases. For instance, if we have three elements **a**, **b**, and **c** in a sequence where **b** is auto-generated by the application, then one possible injection is inserting "<--" at the end of the input for **a**, inserting "-->" at the front of the input for **c**, and concatenate "malicious parameter<c>~" at the end of the **c.** By commenting out the original with "<-- ... -->", we can replace **b** with our intended malicious parameter. Using this for the SQL query can return all user's information, for instance.

2. What are the technical contributions of the paper?

To detect those vulnerabilities of web applications for XMLi, authors presented their approach to generate test data using search-based testing approach. First they assumed that their test suites (web applications) hides their process to generate XML output from the input so that the attacker can only see the inputs and outputs.

They first pre-generated test objectives (TOs), which are the specific outputs of the system under tests (SUT) – in this case, XML messages to web services – that contain malicious content. If there exists inputs that can lead the SUT to generate such malicious XML outputs, then the SUT is considered to be vulnerable. Then a TO is said to be covered if we can provide inputs which result into the SUT producing the TO. Authors have focused to search for such user input.

Authors defined four types of TOs based on the types of XMLi attacks. The type 1 attack called "deforming" aims to create malformed XML messages to crash the system that process them. The

type 2 attack called "random closing tags" makes resulting XML messages with an extra closing tag to reveal the hidden information about the structure of XML documents or database. The type 3 attack "replicating" and the type 4 attack "replacing" change the content to embed nested attacks. The difference between these two attacks is that the replacing attack requires at least two XML elements where the value of one must be auto-generated by the application.

The method that authors used to generate such inputs is genetic algorithm. This is such a generic algorithm. Each chromosome is composed of a number of string values to be assigned to input parameters of the SUT. Then there are such normal crossover operators and mutation operators. Like two strings can be crossover-ed by exchanging head and tail (l1 = h1+t1, l2 = h2+t2 ➔ l1′ = h1+t2, l2′ = h2+t1), this crossover mixes two parents resulting into two new children. Mutation is also simple; change a character in a chromosome into a new one. Then the fitness function they used is the distance between the output XML message and the TO, where this distance means the "string edit distance", the minimum number of editing operations (inserting, deleteing, or substituting a character) required to transform one into the other. This is because if the distance between the output and the TO is 0, it means that the given input generated the malicious XML content successfully.

They evaluated their method on two sample cases: one is for the controlled sample applications; the other one is for the third-party applications. They have examined 3 or 4 times for each different type of TOS for attack type 1~3, and 1 time for the last attack type 4 if possible. They varied the number of input parameters, the flexibility of the length of the input parameters, and the flag that decides whether they will apply the alphabet restricting process or not. They compared their result with the random method, and concluded that their method is effective; restricting alphabet helps significantly increase the efficiency; increasing the number of input parameters increases the execution time.

3. Why do you like this work, or are interested in this work?

I have heard about the SQL injection, which gives nefarious SQL statements into some entry fields for execution and make database behaves maliciously. For instance, one can make database to dump the contents to the attacker. However, I didn't know how to apply this SQL injection, since most of the databases are hidden from the user so that we cannot inject the query directly to the database. Then I made the interest in this paper (which is actually not the subject of our team) since it has started to introduce the XML injection which can be applied to the SQL injection indirectly. Also, the authors made a simple intuition to define vulnerability of the blackbox system: if someone can generate such an XML output through the given application that is the same XML document (TOs) that is known as malicious one, then the application itself is the malicious one. I thought changing the sight to that direction seems fresh.

4. What were the questions from your team and what were your opinions according to the answer?

- In the case of the controlled sample applications, authors said that there exists an auto-generated part by the system. Then I made a question about that one, how can authors recognize what are the auto-generated values in the general case (third party) and where they exist. I also asked presenters that what possible specific methods that authors could extract that part. Presenters replied that since authors pointed out such specific point, so they might have pointed out that point manually. What I actually wanted to know was, however, what about the general case. For instance, if we insert 1, 2, 3 as the three input arguments and if we find 1, 2, 4, 3 in a row from the result, then we can say that 4 is the auto-generated value (by string matching, for instance), and the name in the <> brackets covering 4 should be the name of that auto-generated value. This is important especially for replacing attack since only after we know the exact location of auto-generated values, we can comment out them. Still, the authors didn't mention how to find auto-generated values from the XML output in their third-party evaluation, so I felt some unclearness from the presenters' answer that they just answered for the controlled case.

- When the presenters pointed out the table that shows the execution time for the genetic algorithm increases when the number of input parameter increases, I have asked for the random algorithm method, "then why the execution time for the random algorithm method tends to decrease?" I thought that since the actual size of the string increases, the execution time should be increased also for the random algorithm case, but authors didn't mention about that. Presenters guessed that it might can stop as early as possible because the randomness for more parameters are too big, but this one just means that each step takes more time, not meaning that the number of steps decreases by that. It would be more helpful if presenters could explain such phenomena more specifically.