String Analysis for the Detection of Web Application Flaws

By

Luca Carettoni, Security Consultant, Secure Network S.r.l.
l.carettoni@securenetwork.it

About this talk

- This research was partially supported by:

ingenio

Ministero del Lavoro e della Previdenza Sociale
Direzione Generale per le Politiche per l'Orientamento e la Formazione

fse
Fondo Sociale Europeo

RegioneLombardia

- *Secure Network* is a start-up company based in Milan, Italy
- Consulting, education and research about IT security
- Right now, I'm working as security researcher in collaboration with the Politecnico of Milan University

securenetwork
il percorso più sicuro.

Politecnico Milano

Input validation flaws 1/2

- Any data handled by a web application should be considered unsafe
- HTTP requests are the primary input feed
- By tampering with the input, an attacker can perform a variety of attacks, for example:
  - injection of SQL code, OS commands, and so on
  - injection of client side scripts to compromise other users' session data and credentials or attack the client machine
  - buffer overflows
  - directory traversal to disclose server-side sensitive info
- Complete input filtering is often too complex to handle

Input validation flaws 2/2

- SQL injection example:

```
$query = sprintf(“SELECT * FROM %s WHERE owner=‘%s’ AND nickname=‘%s’”, $this-
   >table, $this->owner,$alias);
$res = $this->dbh->query($query);
```

What if **$alias** was **‘ UNION ALL SELECT * FROM address WHERE ‘1’=‘1 ?**

- Directory traversal example:

```
<?php $template = 'blue.php';
   if ( is_set( $_COOKIE['TEMPLATE'] ) )
        $template = $_COOKIE['TEMPLATE'];
        include ( "/home/users/phpguru/templates/" . $template ); ?>
```

What if the attacker tampered the HTTP request the following way?

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../etc/passwd
```

How to deal with that?

- The solution is the combination of secure design and development, testing, training and review
- Directly filtering before they reach the application
- Interacting with the application or analyzing its source code using differents approaches: (**IEEE Security&Privacy July/August 2006**)

  -Source Code Analyzer

  -Configuration Scanner

  -Web Application Scanner

  -Binary Analysis Tool

  -Runtime Analysis Tool

  -HTTP Proxy

  -Database Scanner

- Source analysis: pattern matching or **data flow analysis**

Hotspot

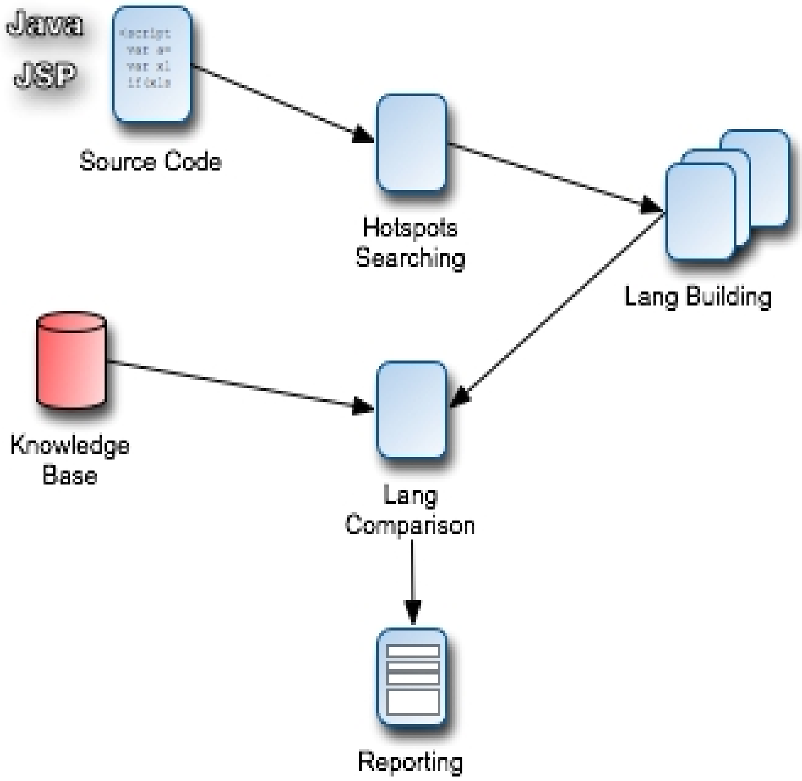
- We use the term **hotspot** to identify the function calls that in a vulnerable application would be exploited as the result of unvalidated input
- Every **hotspot** is associated to a specific signature, composed by ***type of vulnerability***, ***fully qualified method name***, ***number*** and ***type of parameters***
- We are interested in tracing the possible values that String and StringBuffer parameters of hotspots could contain during the application execution
- For example...
  - ***Path traversal:*** methods accessing the filesystem.
    - *java.io.File(java.lang.String)*
    - *java.io.FileReader(java.lang.String), ...*

The main idea

- Input processing in web applications is mainly performed through the exchange of text strings between the client and the server.
  That's why we focus on methods working on strings.
- In a single execution a variable will take, in a specific execution step, a well defined value
- Considering every possible execution we obtain the set of values that the variable could take
- ***Language****: a finite-state automaton* representing the set of those possible values
- The core of our analysis method relies on evaluating the language associated to every hotspots' string parameter.

Analysis method

- **Phase 1**: parsing the application source code looking for hotspots
- **Phase 2**: Building the language associated to every candidate parameter
- **Phase 3**: Comparing those languages with our knowledge base of safe languages



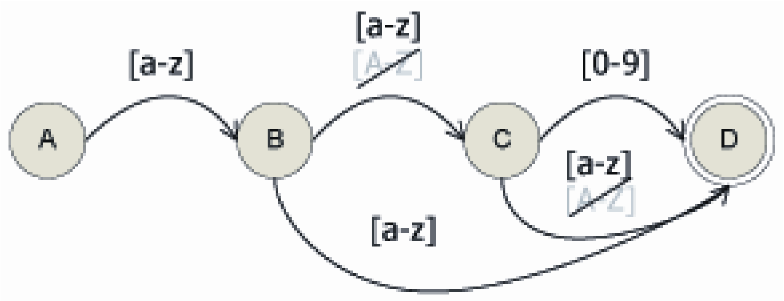
String/Automaton operations

- Each string operation is translated into a specific automaton action:

$$A(L) \xrightarrow{T(f)} A(L')$$

- A simple example, the *toLowerCase()* Java method:

$$L_L = \{\mathrm{x}_1\mathrm{x}_2\ldots\mathrm{x}_n \mid \mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n \in \mathrm{L}_i \wedge \mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n \notin \mathrm{L}_U\}$$



Language comparison

- Using the input vectors (eg. par1) it is possible to modify hotspot parameters (eg. qry)
- The hotspot parameter could then contain a value which isn't valid SQL
- In our knowledge base we defined the safe language for the hotspot as the common SQL language
- If the intersection between language built by analyzing the application data flow and the complement of our safe language is not null then there is a potential flaw

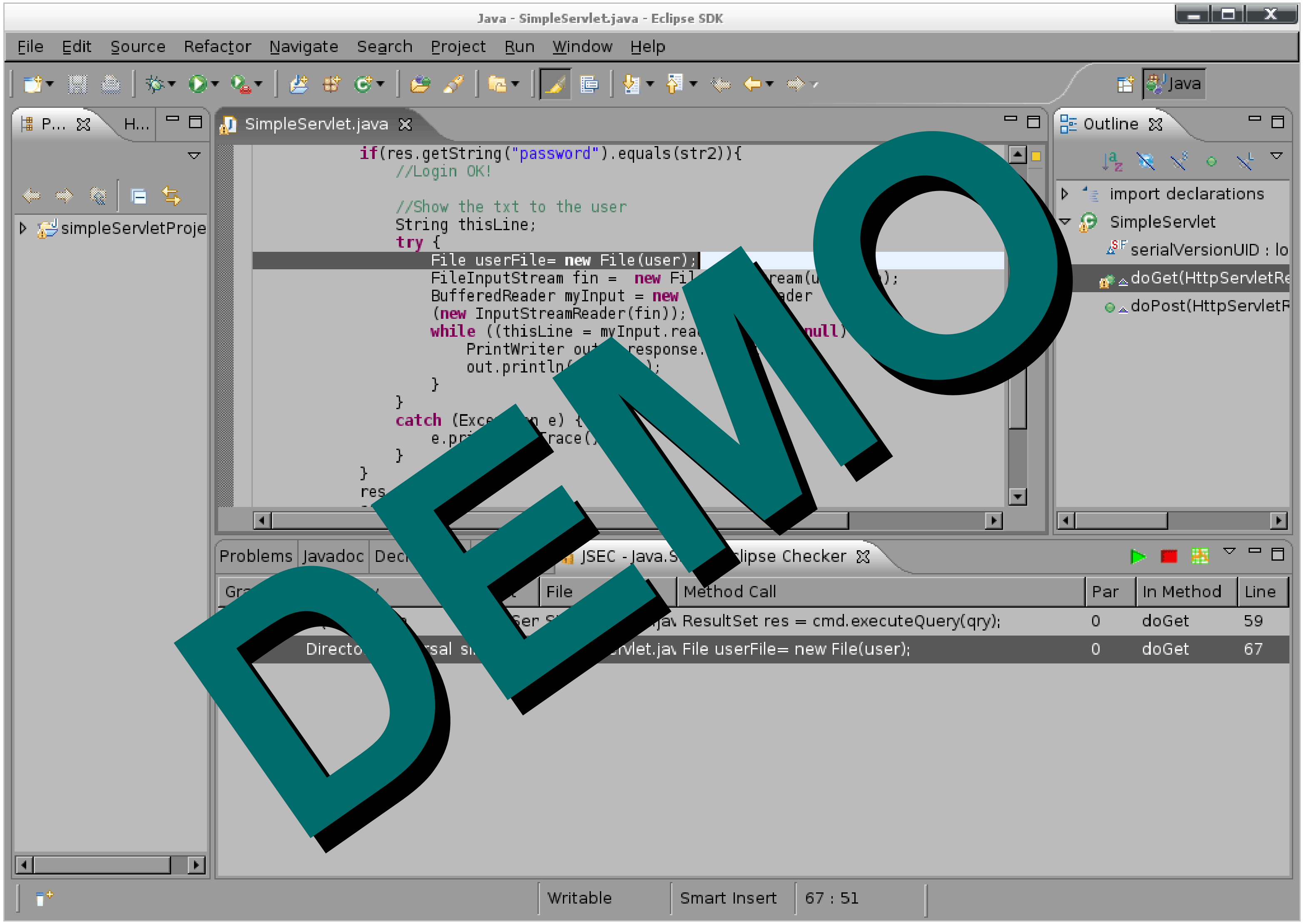
```
import java.servlet.*;
…
public class Servlet extends HttpServlet{

public void doGet(…){
 String str1 =
    request.getParameter(“par1”);
 String qry = “SELECT pass FROM table WHERE
    myRow='“;
 qry = qry.concat(str1);
 qry = qry.concat(“'”);
 …
 Connection cn = … ;
 Statement cmd = cn.createStatement();
 ResultSet res = cmd.executeQuery(qry);
 …
}}
```

$$(L_b \cap \neg L_d) = \varnothing$$

JSEC – **J**ava.**S**tring **E**clipse **C**hecker

- Tightly integrated into the Eclipse IDE
- Code / Compile / Check / Fix
- No user intervention needed in the analysis phase
- Different level of severity in scanning and reporting
- Vulnerabilities defined as plugins that describe the automaton associated
- The analysis is performed using both bytecode (data-flow) and source code (reporting)

JSEC – Java.String Eclipse Checker



```
if(res.getString("password").equals(str2)){
    //Login OK!

    //Show the txt to the user
    String thisLine;
    try {
        File userFile= new File(user);
        FileInputStream fin =  new Fil          ream(u        );
        BufferedReader myInput = new            ader
        (new InputStreamReader(fin));
        while ((thisLine = myInput.rea          null)
            PrintWriter ou    response.
            out.println(         );
        }
    }
    catch (Exce      n e) {
        e.pr        race(
    }
}
res
```

Gr			File	Method Call	Par	In Method	Line
		Ser S	ja\	ResultSet res = cmd.executeQuery(qry);	0	doGet	59
	Directo rsal si		rvlet.ja\	File userFile= new File(user);	0	doGet	67

Summing up

- Source code static analysis cannot completely solve the web app security problem but it's definitely an important step in the right direction
- Our approach is more complex than others but gives more accurate results
- Tightly integrating the security analysis with the IDE can be the key to train the developers about the secure coding practices
- Now: I'm building a detector knowledge base, able to effectively identify the most common vulnerabilities
- Future: Implement the backward slice feature

Questions ?

Feedbacks are welcome

- ***Luca Carettoni*** *- l.carettoni@securenetwork.it*

More info on: http://www.securenetwork.it