String Analysis for the Detection of Web Application Flaws

Luca Carettoni – l.carettoni@securenetwork.it
Claudio Merloni – c.merloni@securenetwork.it

CONFidence 2007 - May 12-13, Kraków, Poland

Web Applications

- Web Applications are everyday more pervasive
- Easy to implement, yet very powerful way to give access to services and content
- Can be made of a handful of simple scripts or a very complex architecture
- Today, web application development often doesn't take into consideration the specific risks coming from the exposure to the web itself

Web Application Security

- Giving access to web application means asking the world to send HTTP request
- Attackers more and more actively look for web application flaws as they are:
  - surprisingly common
  - often the key to subvert the victim's data and networks
  - it is quite easy for an attacker to hide his identity using well known anonymizing techniques

Input Validation - 1

- Every data handled by a web application should be considered unsafe
- HTTP request are the primary input feed
- Attackers can alter any part of an HTTP request: pieces of info coming from a client (also if subject to client side validation) should never be considered safe:
  - GET and POST parameters
  - request headers
  - cookies, and so on.

Input Validation - 2

- Tampering 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 local machine
  - buffer overflows
  - directory traversal to disclose server-side sensitive info
- Complete input filtering is often too complex to handle

Input Validation - 3

- 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
```

Input Validation - 4

- Path Based Access Control

```
public class PathBasedAccessControl extends LessonAdapter {
[...]
String dir = s.getContext().getRealPath( "/images" ); // A
[...]
String file = s.getParser().getRawParameter( FILE, "" ); // B
[...]
File f = new File( (dir + "\\" + file).replaceAll("\\\\","/")); // C
}
```

A: we are in /images/ (Absolute Path on my Linux box: /var/lib/tomcat-5.5/webapps/WebGoat/)
B: from the HTML form, we take the FILE input parameter
C: Creating a File object...

Input Validation - 5

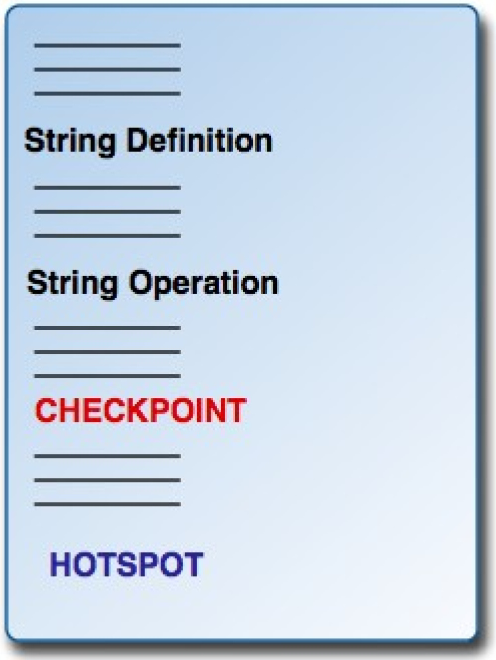
- We can request a file inside the allowed *images* folder:
  - right.gif
- But we can also try to break out of the web root with a correctly crafted path:
  - %2e%2e%2f%2e%2e%2f2e%2e%2f2e%2e%2f2e%2e%2f2e%2e%2fetc/passwd

How to deal with that?

- The solution is the combination of secure desing and development, testing, training and review
- Directly filtering before they reach the application
- Interacting with the application or analyzing its source code:
  - Source Code Analyzer
  - Web Application Scanner
  - Database Scanner
  - Binary Analysis Tool
  - Runtime Analysis Tool
  - Configuration Scanner
  - HTTP Proxy
- Source analysis: pattern matching or **data flow analysis**

Checking the input

- 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.
- Validating the input: **checkpoint**
- **Blacklist**: defining what **bad** input is. Then escaping, substituting, and so on
- **Whitelist**: defining what **good** input is and filtering anything that doesn't match



Hotspot - 1

- 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 hotspots' String and StringBuffer parameters could take during the application execution

Hotspot - 2

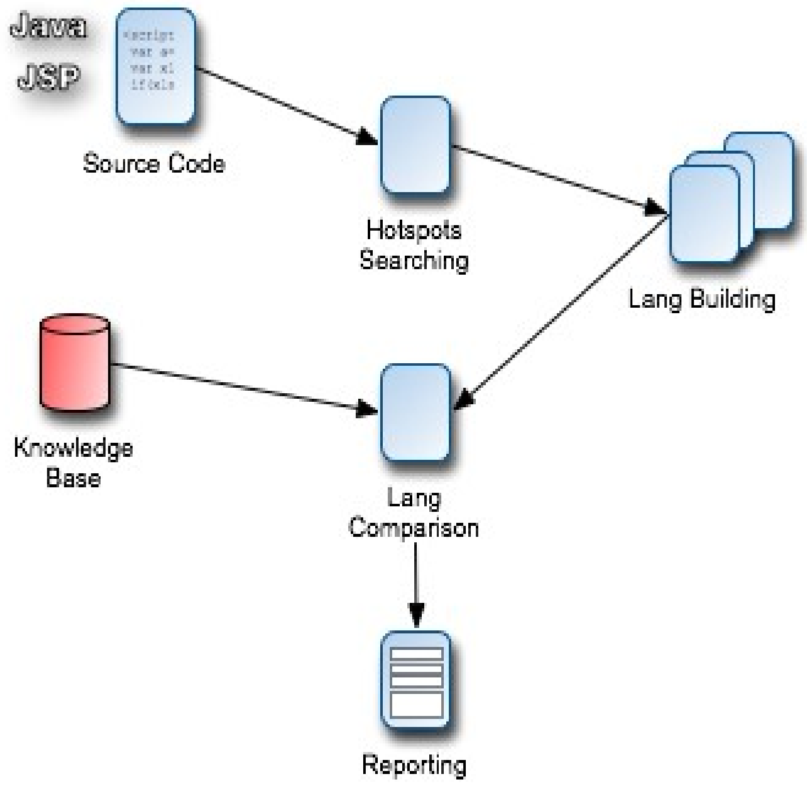
- ***Path traversal:*** methods accessing the filesystem.
  - *java.io.File(java.lang.String)*
  - *java.io.FileReader(java.lang.String), ...*
- ***SQL injection:*** database connectivity.
  - *java.sql.Statement.executeQuery(java.lang.String)*
  - *java.sql.Connection.prepareStatement(java.lang.String), ...*
- ***Command injection:*** command execution, class loading and so on.
  - java.lang.Runtime.exec(java.lang.String, …)
  - java.lang.System.load(java.lang.String), ...

Automaton definition

- 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



Language comparison

- Unvalidated input: using the input vectors (eg. par1) it is possible to modify hotspot parameters (eg. qry)
- The hotspot parameter could then take a value which isn't valid SQL
- In our knowledge base we defined the safe language for the hotspot as the common SQL language
- The complement of this language define the values that qry shouldn't be allowed to take
- 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);
 …
}}
```

Building a tool - 1

- 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

Building a tool - 2

- The analysis is performed using both bytecode (data-flow) and source code (reporting)
- Project resources scanning based both on Eclipse Framework and on raw filesystem analysis:
  - The Eclipse Framework define source locations, classe locations and provide methods to quickly navigate the project structure
  - Filesystem resources can be easily analyzed using both source and class Java reflection

Testing results

- Testing has been conducted on the OWASP WebGoat project (v3.7, 55 Java classes, 16160 lines)
- Our tool:
  - Analysis time: 483 sec.
  - Vulnerabilities found: 16 SQL Injection, 16 Path Traversal
- LAPSE:
  - Analysis time: 32 sec.
  - Vulnerabilities found: 2 Command Injection, 1 Cross-Site Scripting, 13 SQL Injection

Let's see it!

DEMO

Summing up

- It is nowadays critical to enforce security policies on the whole web application lifecycle
- 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

Future work

- Build a detector knowledge base, able to effectively identify at least the most common vulnerabilities
- Automatically parse project resources contained inside j2ee archives.
- Automatically compile Jsp resources to servlets
- Implement the backward slice feature
- Rework the data flow analysis components to make the tool able to process more programming languages

Questions?

Luca "ikki" Carettoni *- l.carettoni@securenetwork.it*
Claudio "paper" Merloni *- c.merloni@securenetwork.it*

SecureNetwork S.r.l.: www.securenetwork.it