All input is evil !

Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

General context

- **“All input is evil!” or “Trust no input!” are common security “mantras”.**
- **Malicious input and the lack of appropriate input validation** is the essencial ingredient of exploits targetting several kinds of software vulnerabilities we will cover:
  - injection vulnerabilities (OS command, SQLi, …)
  - Web-app specific vulnerabilites (XSS, CSRF, …)
  - buffer overflows
  - …

What is an “input” ?

- Input:
  - every item of data that comes from an external source and affects program behavior
- Possible data sources
  - Command line arguments
  - Configuration data (files, environment vars, etc)
  - Network servers
  - Database
  - File system
  - Shared memory
  - Hardware devices
  - …

The “tutorial” from class 1 - client side

```
<html> <he
<script>
function showUser(str) {
if (str=="") {
  document.getElementById("txtHint").innerHTML="";
  return;
}
if (window.XMLHttpRequest) {// code for IE7+, Firefox, Chrome, Opera, Safari
   xmlhttp=new XMLHttpRequest();
} else {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
xmlhttp.onreadystatechange=function()  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200) {
     document.getElementById("txtHint").innerHTML=xmlhttp.responseText;
  }
}
xmlhttp.open("GET","getuser.php?q="+str,true);
xmlhttp.send();
}
</script>
</head>
<body>
<form>
<select name="users" onchange="showUser(this.value)">
<option value="">Select a person:</option>
<option value="1">Peter Griffin</option>
<option value="2">Lois Griffin</opti
<option value="3">Glenn Quagmire</op
<option value="4">Joseph Swanson</op
</select>
</form>
<br>
<div id="txtHint"><b>Person info will be listed here.</b></div>

</body>
</html>
```

entire HTML can be considered the “whole” input

AJAX data source / interface with the server

server output => input to the client side

A “simple” example
“PHP - AJAX and MySQL”
http://www.w3schools.com/php/php_ajax_database.asp

The “tutorial” from class 1 - server side

```php
<?php
$q=$_GET["q"];
$con = mysql_conn                                    'abc123');
if (!$con)
  {
  die('Could not connect: ' . mysql_error());
  }
mysql_select_db("ajax_demo", $con);

$sql="SELECT * FROM user WHERE id = '".$q."'";
$result = mysql_query($sql);

echo "<table border='1'>
<tr>
<th>Firstname</th>
<th>Lastname</th>
<th>Age</th>
<th>Hometown</th>
<th>Job</th>
</tr>";
while($row = mysql_fetch_array($result))
  {
  echo "<tr>";
  echo "<td>" . $row['FirstName'] . "</td>";
  echo "<td>" . $row['LastName'] . "</td>";
  ...
  }
echo "</table>";
mysql_close($con);
?>
```

GET request parameter

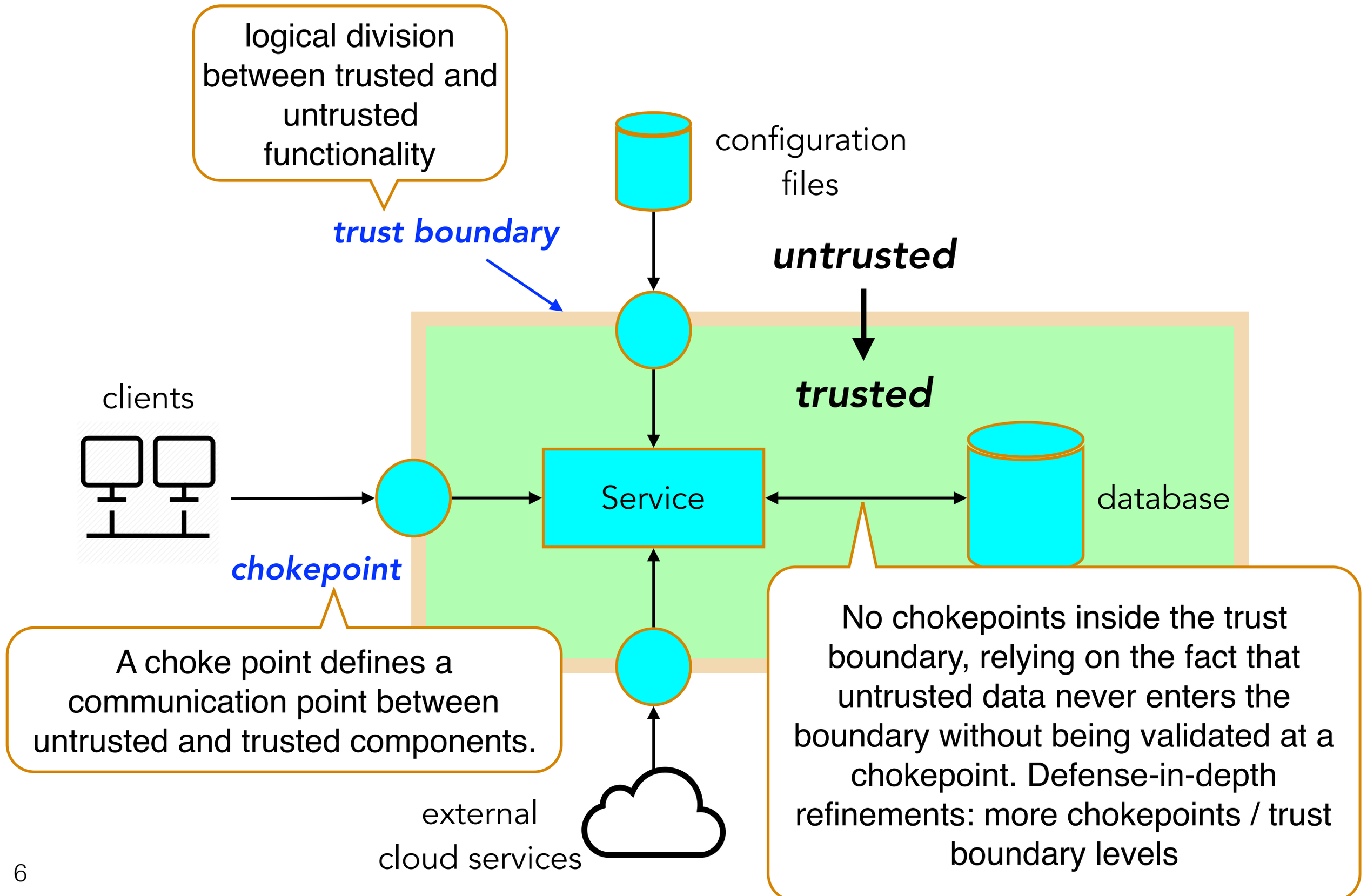
SQL injection possible at this point

database data input (passed on to client)

A “simple” example
“PHP - AJAX and MySQL”
http://www.w3schools.com/php/php_ajax_database.asp

5

Trust boundary



Trust boundary violation

- Trust boundary violation (CWE-501)
  - “A trust boundary can be thought of as line drawn through a program. On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy.”
  - “**A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted**. By combining trusted and untrusted data in the same data structure, it becomes easier for programmers to mistakenly trust unvalidated data.”
- Establishing trust boundaries may be hard due to the diversity of input sources (and misplaced trust in some of them) and he fact that **trust is transitive**: if A trusts B, then A implicitly trust components that are part of B or interface with B Consider for instance program dependencies in the form of external libraries.

Trust boundary violation (2)

Example 1

The following code accepts an HTTP request and stores the username parameter in the HTTP session object before checking to ensure that the user has been authenticated.

Example Language: **Java** *(Bad Code)*

```
usrname = request.getParameter("usrname");
if (session.getAttribute(ATTR_USR) == null) {
  session.setAttribute(ATTR_USR, usrname);
}
```

Example Language: **C#** *(Bad Code)*

```
usrname = request.Item("usrname");
if (session.Item(ATTR_USR) == null) {
  session.Add(ATTR_USR, usrname);
}
```

Without well-established and maintained trust boundaries, programmers will inevitably lose track of which pieces of data have been validated and which have not. This confusion will eventually allow some data to be used without first being validated.

- Example from Trust boundary violation (CWE-501)

Dealing with input

- **Integrity — is input data trustworthy?**
  - Ensure that data has not been tampered and/or comes from a trusted source, e.g. employing encrypted channels, checksums, digital signatures **=> integrity checks usually deployed in association to communication mechanisms.**
- **Validation — is input data valid ?**
  - Ensure that the data is strongly typed, correct syntax, within length boundaries, contains only permitted characters, or if numeric is correctly signed and within range boundaries **=> syntactic checks usually deployed at chokepoint input handling logic.**
  - Mechanisms: sanitization and filtering with associated techniques e.g. data encoding/“escaping”, whitelist and blacklist checks.
- **Application logic — does the input specify a legal operation?**
  - At the application logic layer, does data make sense taking into account the specific context of operation at stake **=> “in-depth” semantic checks throughout the application logic.**
  - Data may be be considered as arriving without tampering from a trusted source and valid, but still be illegal in terms of the operation to execute (e.g., lack of appropriate previleges, unauthorized session, …).

Input validation — generic aspects

- **As we will see in the study of several types of vulnerabilites … (along with domain-specific strategies, mechanisms, and implementation aids)**
- **Strategies**
  - **Rejection** of bad data that does not conform to a certain criteria.
  - **Sanitization**: process data such that potentially malicious fragments are neutralized, e.g. espace special meta-characters, remove known problematic sequences, etc
- **General mechanisms**
  - **While-list**: computational logic identifies valid inputs such that they are accepted / deemed
  - **Black-list** based: computation logic identifies invalid input such that they are rejected, or merely removed/“escaped”
- **Implementation aids**
  - Regular expressions
  - Grammar-based checks e.g. the use of schemas for compliance of XML or JSON data
  - Readily-avaliable sanitization functions/libraries (context-dependent), e.g., OWASP Java Encoder, PHP data filtering, DOMPurify

Whitelisting vs. blacklisting

- Whitelists are generally preferable
  - they concretely define what good inputs are
  - but in some scenarios they can be too restrictive / unfeasible
- Blacklists only identifies a set of bad inputs.
  - The set may be incomplete or hard to enumerate … providing a false sense of security. Chances are that some bad inputs are not filtered out.
  - There is a vulnerably class for incomplete black-lists — CWE-184.
  - Blacklists may however be simpler to implement or more adequate in some cases, e.g., blacklists of domains associated with e-mail spamming.
- Further references (even beyond input validation)
  - Whitelisting vs blacklisting, OWASP “Input Validation Cheat Sheet”
  - Whitelisting vs. Blacklisting , Schneier on Security (short blog article by Bruce Schneier)

Whitelist — simple example

```
private static final String[] ALLOWED_FILE_EXTENSIONS = {
    ".gif", ".jpeg", ".png"
};

static boolean isValidFileExtension(String fileName) {
   for (String ext : ALLOWED_FILE_EXTENSIONS) {
      if (fileName.endsWith(ext)) {
        return true;
      }
   }
   return false;
}
```

- Whitelist defines the set of valid inputs. All others will be rejected.

Blacklist — simple example

```
private static final String[] DISALLOWED_FILE_EXTENSIONS = {
     ".exe", ".com", ".bat"
};

static boolean isValidFileExtension(String fileName) {
    for (String ext : DISALLOWED_FILE_EXTENSIONS) {
      if (fileName.endsWith(ext)) {
        return false;
      }
    }
    return true;
}
```

- Blacklist defines the set of invalid inputs. All others will be accepted.
- Whitelists vs blacklists: what do you think it's best?

Regular expressions — simple example

```
// Regular expression for dates in a YYYY-MM-DD format
  private static final Pattern DATE_PATTERN
      = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})");

  static boolean isValidDate(String s) {
    Matcher m = DATE_PATTERN.matcher(s);
    if (! m.matches() )
      return false;

    int year = Integer.parseInt(m.group(1));
    int month = Integer.parseInt(m.group(2));
    int day = Integer.parseInt(m.group(3));

    return month >= 1 &&
           month <= 12 &&
           day >= 1 &&
           day <= daysInMonth(month, year);
  }
```

- Dates are validated with a YYYY-MM-DD format using a regular expression. Values are subsequently checked to enforce that the date is valid.