Software Quality

Lesson 3:

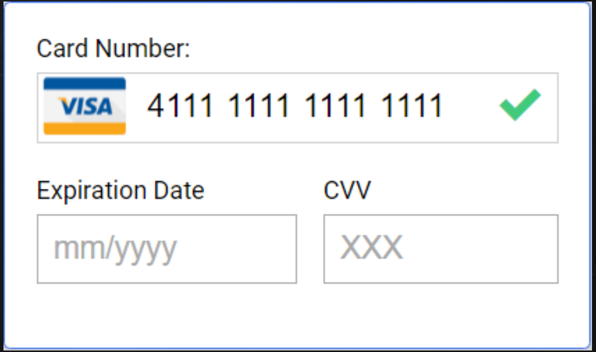
User Input

Introduction

Most dynamic web applications accept some kind of input from the client.

This input: may decide what to do next,
it may be stored somewhere,
included in a new web page,
used in a legacy system,
E-mailed to someone,
and almost everything else, depending on the application.

Without this input, We cannot shop, transfer money, give votes, send web-based greeting cards, use search engines, or any other service that relies on data being passed from the browser to the web server.

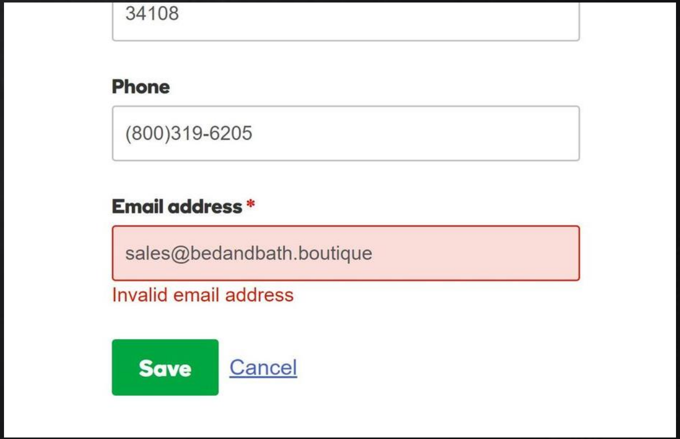

Card Number:
4111 1111 1111 1111
Expiration Date
mm/yyyy
CVV
XXX

Accepting input from the client is probably the greatest threat to the security of a web application.

Accepting wrong input may make the programs make wrong decisions, and the results may vary from harmless, via annoying to devastating.

To make sure our application does not make the wrong decisions, we need to analyze every piece of input.

The analysis is known as input validation.



34108

Phone

(800)319-6205

Email address *

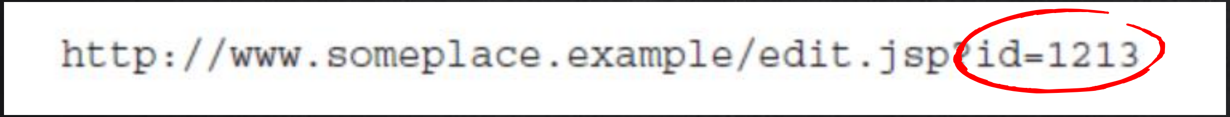
sales@bedandbath.boutique

Invalid email address

Input

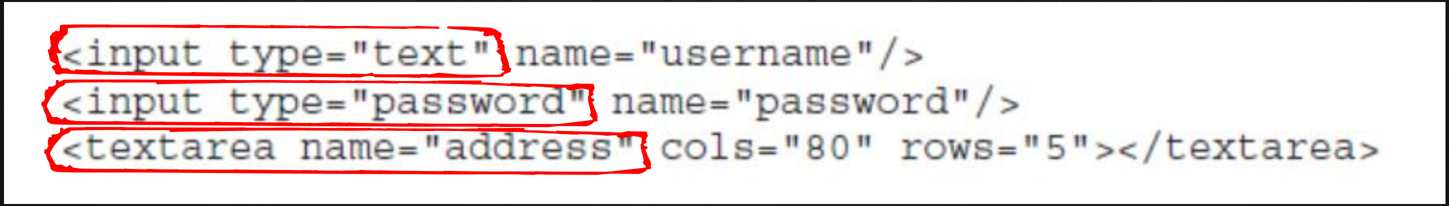
What is Input?

- It is quite clear that URL parameters must be considered as input:



```
http://www.someplace.example/edit.jsp?id=1213
```

- It is also quite obvious that whatever the user enters in text fields and text areas are input to the web application, whether it enters the application through GET or POST.

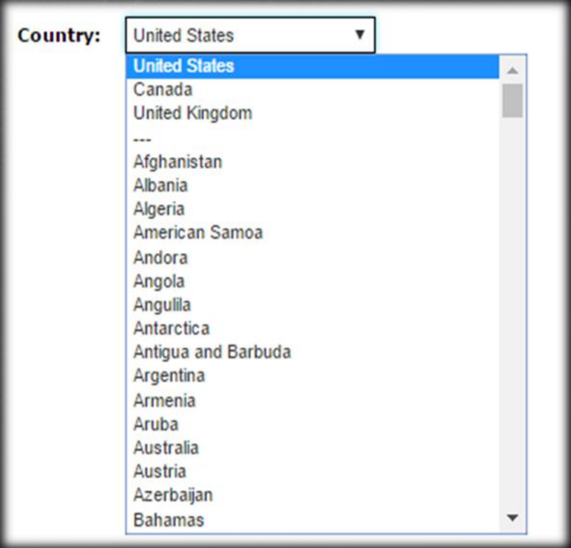


```
<input type="text" name="username"/>
<input type="password" name="password"/>
<textarea name="address" cols="80" rows="5"></textarea>
```

- These are known as `user-generated input`.

- Another kind of input that quite a few developers do not consider "real" input:
  - The user interface lets the user select which of the predefined values to send.
  - The list of possible input values is dictated by the web application rather than by the user.

```
<select name="country">
      :
   <option value="dk">Denmark</option>
      :
   <option value="se">Sweden</option>
      :
</select>
```

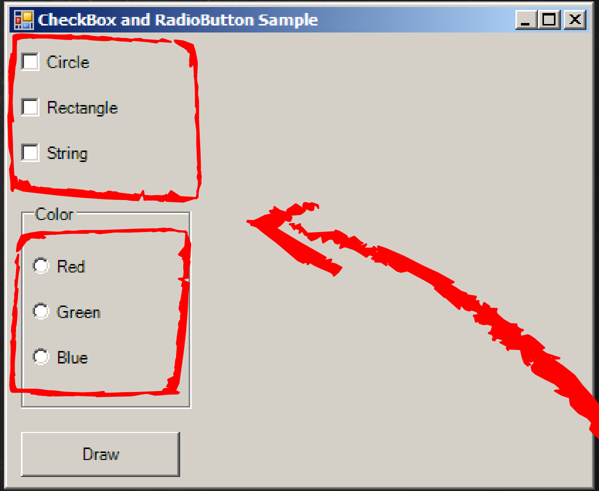

Country: United States ▼

United States
Canada
United Kingdom

Afghanistan
Albania
Algeria
American Samoa
Andora
Angola
Angulila
Antarctica
Antigua and Barbuda
Argentina
Armenia
Aruba
Australia
Austria
Azerbaijan
Bahamas

- Check boxes and radio buttons:

```
<input type="radio" name="gender"
       value="female"/>&nbsp;Female
<input type="radio" name="gender"
       value="male"/>&nbsp;Male
```



- Hidden fields:

```
<input type="hidden" name="userid" value="194423"/>
```

These can be called server-generated input, even if they come from the client, as the values are dictated by our web application.

The user interface does not give the user an opportunity to change the values.

The risk

In most cases, server-generated input will come back to us with a well-defined value, that is the value or one of the possible values that our application included in the HTML.

However, An attacker may have modified the values before sending the request:

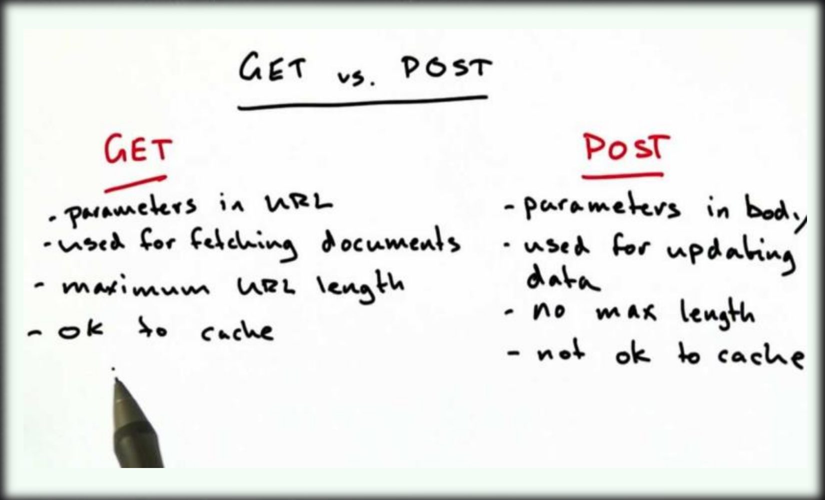
- If a GET request is used, parameter manipulation is just a matter of modifying the URL in the location bar of the browser.
- If a POST request is used, the attacker may have to modify the form details of our HTML before sending the request.

How?

Modifying the HTML is quite simple:

1. Use the browser to save the HTML to a file.
2. Open the file in a text editor.
3. Make the intended changes.
4. If the action attribute of the form is relative, modify it to contain a full URL.
5. Save the file.
6. Open the local file in the browser, and submit the form.

- Some web applications pay no attention to the difference between POST and GET, and accept either of the two.
  - For those applications, the attacker need not go to the trouble of modifying the HTML.
  - Just picks parameters from the form, appends them to the URL given in the action attribute, and puts the resulting URL in the location bar of his browser.



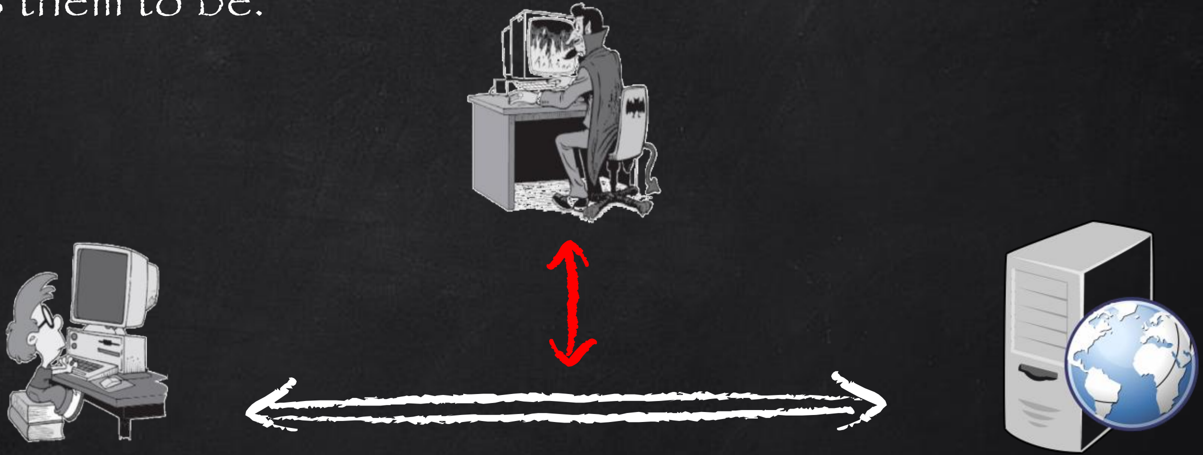
GET vs. POST

GET
- parameters in URL
- used for fetching documents
- maximum URL length
- ok to cache

POST
- parameters in body
- used for updating data
- no max length
- not ok to cache

HOW ABOUT SERVER GENERATED FIELDS?

- Nothing stops an attacker from making country, gender and userid any value he wants them to be.



- So we need to view the server-generated hidden fields, check boxes, radio buttons and select list values as input, just as we see user-generated text fields as input.

Example: on-line Payment

Even HTTP headers, including cookies, must be handled just as carefully as textual input, as the following example will try to show.

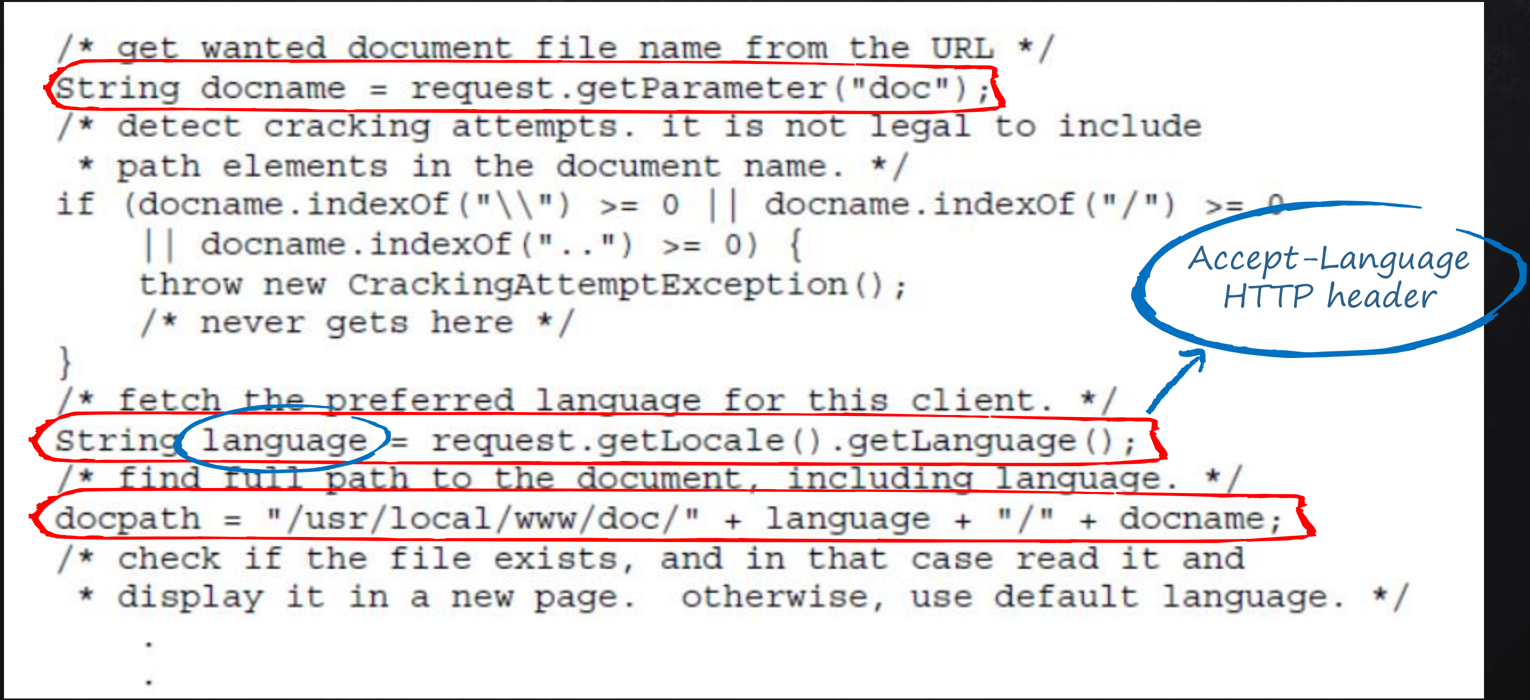
Example: International Documents for online payment

```
/usr/local/www/doc/it/payment.txt
```

If someone wanted to view the payment documentation, they would follow a link to a URL looking like this:

```
http://www.bank.example/help?doc=payment.txt
```

The bank software would determine what language the user preferred based on Locale settings, and read payment.txt from the correct directory. How?



```
/* get wanted document file name from the URL */
String docname = request.getParameter("doc");
/* detect cracking attempts. it is not legal to include
 * path elements in the document name. */
if (docname.indexOf("\\") >= 0 || docname.indexOf("/") >= 0
    || docname.indexOf("..") >= 0) {
    throw new CrackingAttemptException();
    /* never gets here */
}
/* fetch the preferred language for this client. */
String language = request.getLocale().getLanguage();
/* find full path to the document, including language. */
docpath = "/usr/local/www/doc/" + language + "/" + docname;
/* check if the file exists, and in that case read it and
 * display it in a new page.  otherwise, use default language. */
    .
    .
```

How can be attacked?

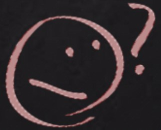
The `language` string is taken from the `Accept-Language` HTTP header, which comes from the client's request.

```
String language = request.getLocale().getLanguage();
docpath = "/usr/local/www/doc/" + language + "/" + docname;
```

```
/usr/local/www/doc/it/payment.txt
```

What would happen if an attacker sent the following lines of HTTP?

```
GET /help?doc=passwd HTTP/1.0
Host: www.bank.example
Accept-Language: ../../../../etc
```



```
/usr/local/www/doc/../../../../etc/passwd
```

What was the programmer's mistake?

Lets Ask Again: What is input?

Anything entering the application from the outside, typically through some request object or request stream, must be considered input.

Input thus includes:

- All URL parameters.
- POST-ed data from textual input, check boxes, radio buttons, select lists, hidden fields, submit buttons and so on.
- Cookies and other HTTP headers used by the application, even those used behind the scenes by the programming platform.

A web application may take input from sources other than the web client. Input may come from files and database tables generated by other parts of the total system.

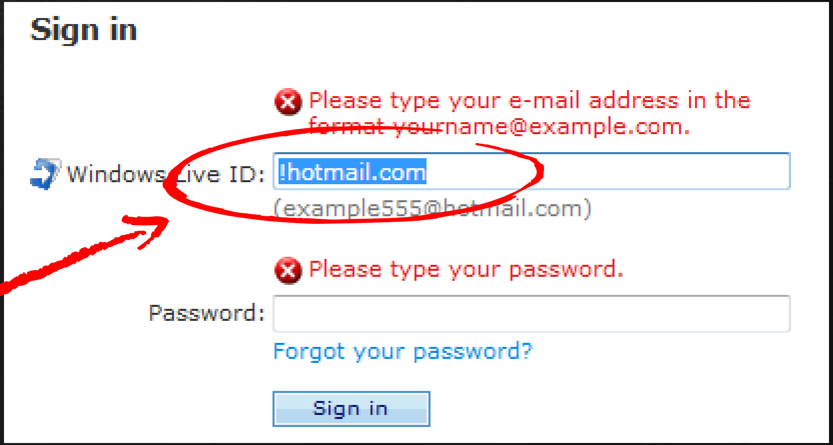
Validating Input

What is the input validation?

Input validation is the process of determining whether an input parameter is valid, according to rules set out by our application.

The validity rules govern domain types rather than programming language data types.

- We may, for instance, say that one particular parameter has a string data type, but the value should be taken as an E-mail address.
- When validating, we check that the format of the parameter matches the required format of an E-mail address.
- The domain type is "E-mail address".



Sign in

Please type your e-mail address in the format yourname@example.com.

Windows Live ID: !hotmail.com

(example555@hotmail.com)

Please type your password.

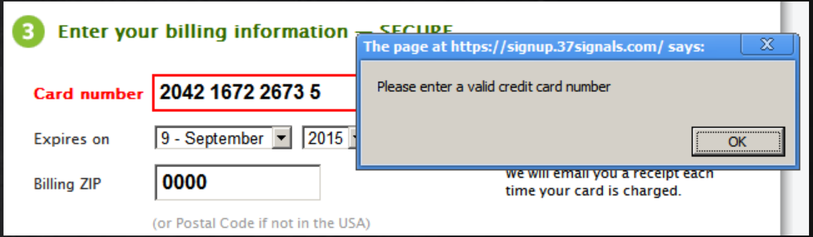
Password:

Forgot your password?

Sign in

Other typical domain types include "account", "country code", "customer ID", "date", "file name", "payment amount", "phone number", "real name", "URL", "user name", "VISA", and so on.

The main goal of input validation **is not** to avoid nasty metacharacter problems such as SQL Injection and Cross-site Scripting.


3 Enter your billing information — SECURE

The page at https://signup.37signals.com/ says:

Please enter a valid credit card number

Card number	2042 1672 2673 5
Expires on	9 - September 2015
Billing ZIP	0000

(or Postal Code if not in the USA)

We will email you a receipt each time your card is charged.

Because for example:

- We cannot, for all possible applications, say that a real name cannot contain <u>single quotes</u> (O'Connor).
- And we cannot forbid <u>less (<) and greater than (>) signs</u> in discussion site notes.

The main goal of input validation is to make sure our application works with data that have the expected format.

Suggestions for Good Input Validation

- Make Sure you Identify and Validate All Input.

  Good input validation depends on a clear understanding of all parameters originating on the client, including hidden fields, option values, cookies and (other) stuff coming from HTTP headers.

- Create Validation Functions.

  Examples:

  - `isValidEMailAddress` and `isValidCustomerID`, returning Boolean values.
  - For server-generated input, parallel functions such as `assertValidEMailAddress` and `assertValidCustomerID` can abort execution if input is invalid.

SUGGESTIONS FOR GOOD INPUT VALIDATION

- Check the Range.

  For certain domain types, particularly the numeric ones, there may be range limitations as well as format limitations.

  Example:

  - The price of an item in a web shop: it must be numeric, but it should not be negative.

- Check the Length.

  You do not wish to allow an infinite number of characters for any input type.

  In a database table, you typically specify an upper length limit for textual fields.

  Always check the input for a reasonable length (database errors and buffer overflows).

Suggestions for Good Input Validation

- Check for the Presence of Null-bytes.

  Null-bytes should never be present in non-binary input.

  As they tend to cause problems for many subsystems, we may just check for them explicitly when validating. (Read Section 2.3 of the textbook).

- Perform Input Validation Before Doing Anything Else.

  Start every request handler by validating all input parameters.

  If validation is delayed until a parameter is used, it is more easily forgotten, and it will not always be clear whether validation has been already done or not.

Suggestions for Good Input Validation

- Perform Authorization Tests Along with Input Validation

  In some cases, input from the client will reference resources that may only be accessed by certain users (e.g. a discussion forum).

  Wise to perform the access control along with input validation, before starting to work on the input.

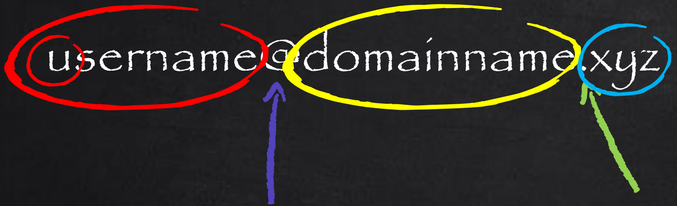
- Try to Automate Input Validation

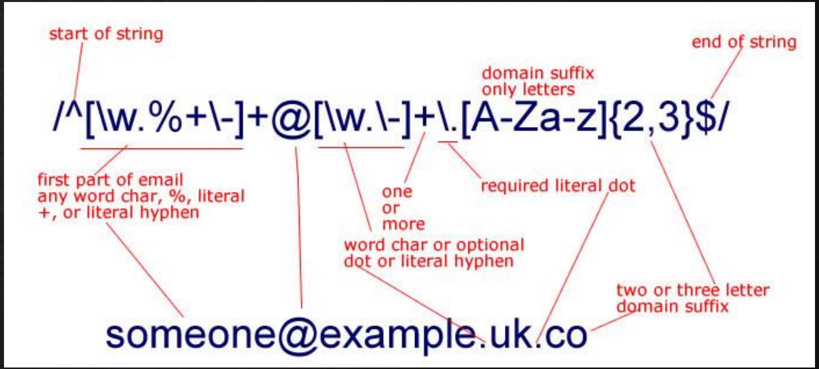
  For projects with many developers, it may be a good idea to create a framework that forbids direct access to the Request object (or similar construct containing request parameters). The framework could handle input validation before the parameters are passed to the main part of the application.

Regular Expressions

Input validation is about deciding whether data are valid or not.

We raise a question that results in true or false, and the answer is based on whether the input matches our expectations.

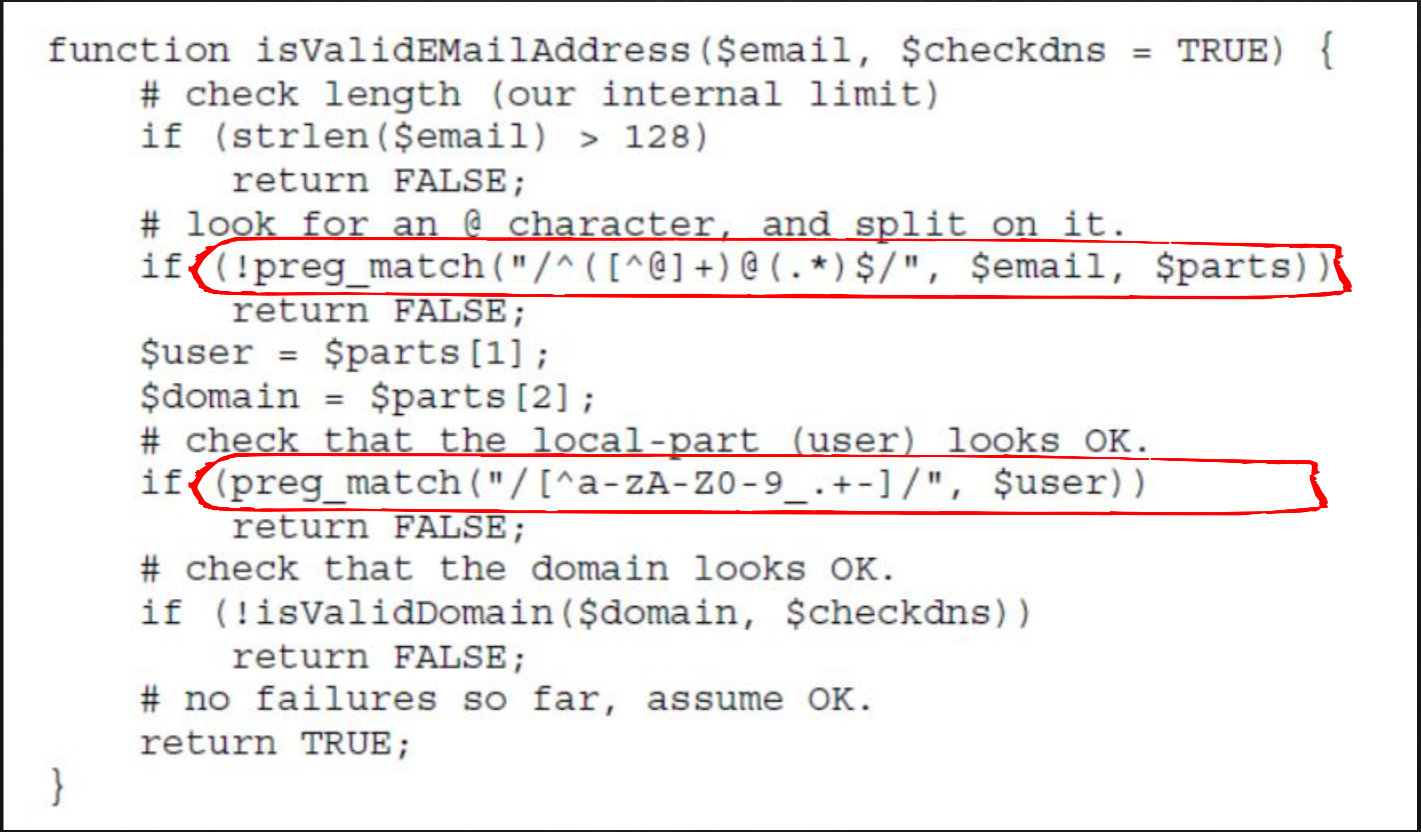



start of string
end of string
domain suffix only letters
/^[\w.%+\-]+@[\w.\-]+\.[A-Za-z]{2,3}$/
first part of email any word char, %, literal +, or literal hyphen
one or more
required literal dot
word char or optional dot or literal hyphen
two or three letter domain suffix
someone@example.uk.co

When it comes to matching text, nothing beats regular expressions (RE).

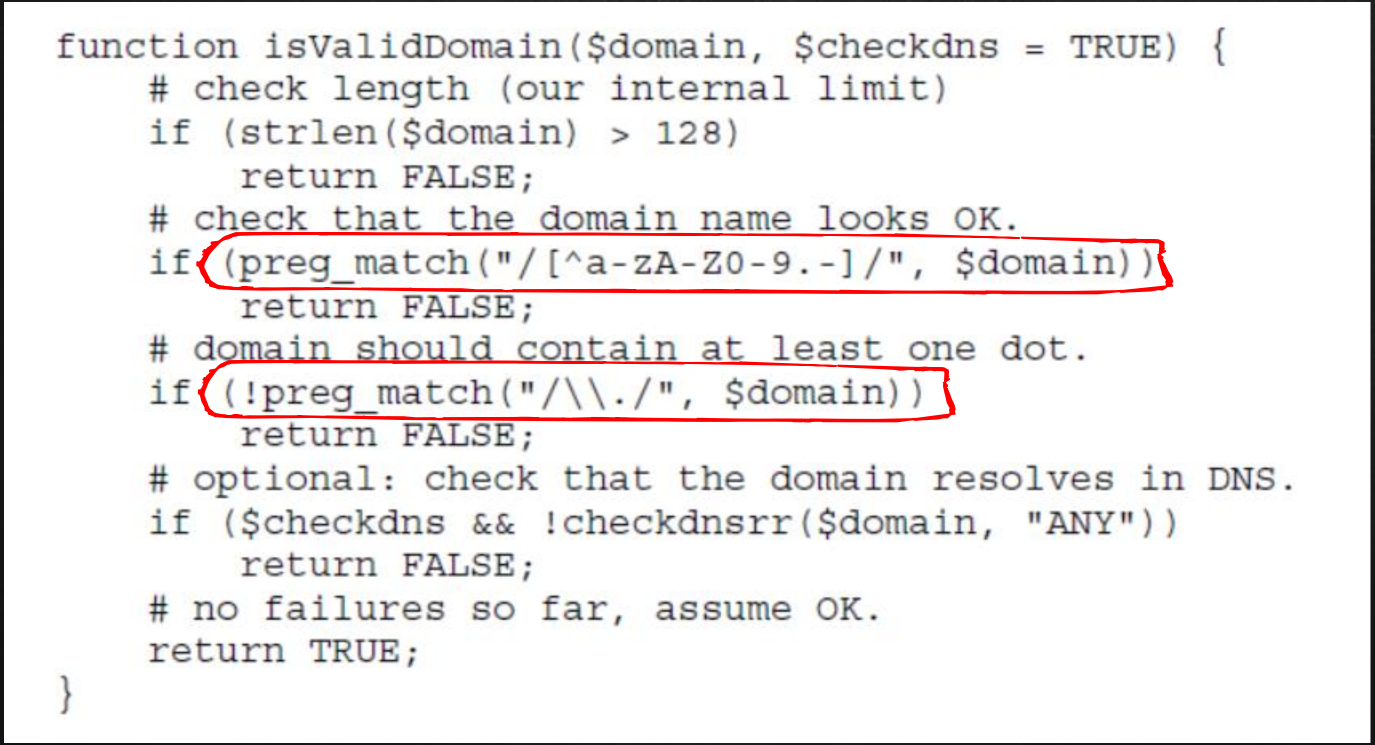
RE is a pattern matching language supported by most programming platforms, either natively, or through third-party addons.

VALIDATION OF AN E-MAIL ADDRESS IN PHP USING RE



```
function isValidEMailAddress($email, $checkdns = TRUE) {
    # check length (our internal limit)
    if (strlen($email) > 128)
        return FALSE;
    # look for an @ character, and split on it.
    if (!preg_match("/^([^@]+)@(.*)$/", $email, $parts))
        return FALSE;
    $user = $parts[1];
    $domain = $parts[2];
    # check that the local-part (user) looks OK.
    if (preg_match("/[^a-zA-Z0-9_.+-]/", $user))
        return FALSE;
    # check that the domain looks OK.
    if (!isValidDomain($domain, $checkdns))
        return FALSE;
    # no failures so far, assume OK.
    return TRUE;
}
```

Validation of a Domain Name in PHP using RE



```
function isValidDomain($domain, $checkdns = TRUE) {
    # check length (our internal limit)
    if (strlen($domain) > 128)
        return FALSE;
    # check that the domain name looks OK.
    if (preg_match("/[^a-zA-Z0-9.-]/", $domain))
        return FALSE;
    # domain should contain at least one dot.
    if (!preg_match("/\\./", $domain))
        return FALSE;
    # optional: check that the domain resolves in DNS.
    if ($checkdns && !checkdnsrr($domain, "ANY"))
        return FALSE;
    # no failures so far, assume OK.
    return TRUE;
}
```

Whitelisting vs. Blacklisting

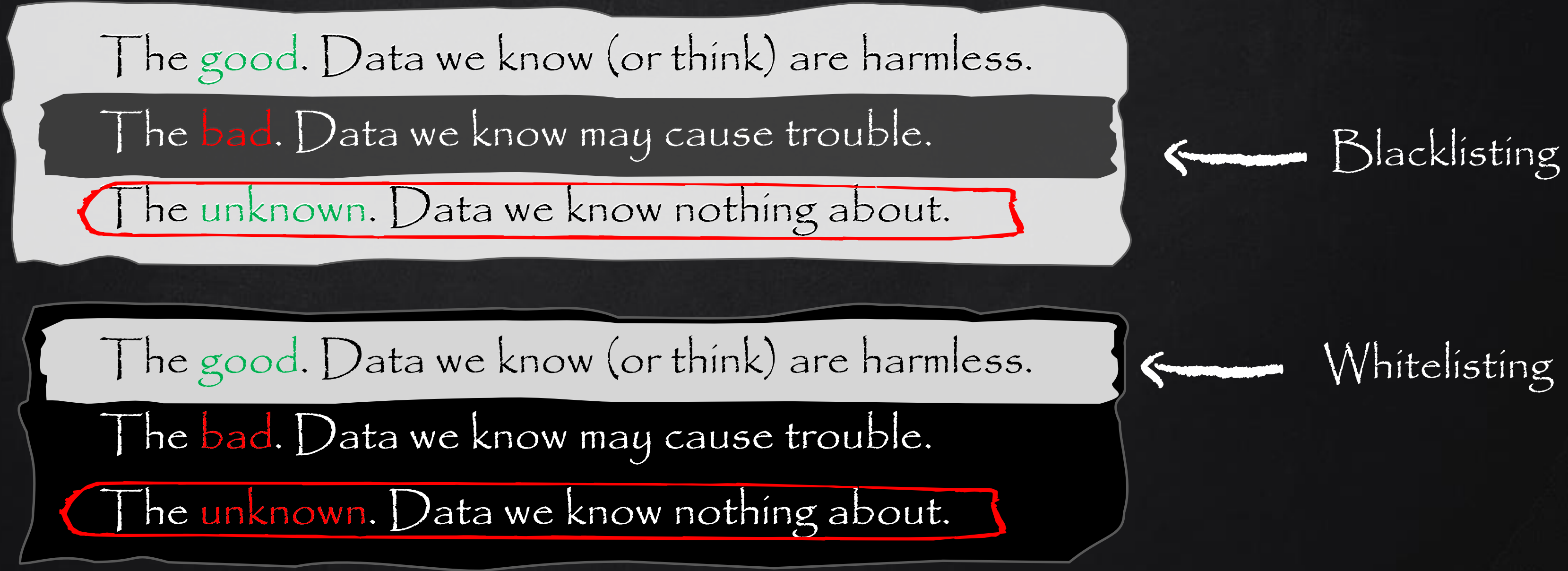
When filtering data, we look at characters or combinations of characters to remove something, rewrite something, or detect something.

The filtering can be done in one of two ways:

- Identify bad data and filter it.
  - The first approach is the most intuitive. We know what data are bad, and look for them. The process known as blacklisting, since we start with a list of things we do not like; a blacklist.
- Identify good data and filter the rest.
  - It start with a list of things we consider harmless. Whenever we see something not on this list, we assume it may be harmful, and filter it. This process is known as whitelisting, as we start with a list of presumed good stuff.

Whitelisting is the preferred approach in a security context. It implements what firewall people would probably call deny by default.

Why?


The good. Data we know (or think) are harmless.
The bad. Data we know may cause trouble.
The unknown. Data we know nothing about.

← Blacklisting

The good. Data we know (or think) are harmless.
The bad. Data we know may cause trouble.
The unknown. Data we know nothing about.

← Whitelisting

Handling Invalid Input

Recap

1. User-generated input is what comes from input fields of type text and password, or from textareas.
   - User-generated input may be invalid due to typing errors.
2. Server-generated input is all the rest, such as hidden fields, URL parameters that are part of an anchor tag, values from selection boxes, cookies, HTTP headers, and so on.
   - Server-generated input, which is not directly modifiable by the user, will never be incorrect during normal usage.
   - If it is incorrect, it means that someone is tampering with values that are normally out of their reach, and not supposed to be changed.

We should handle suspicious user- and server-generated input differently.

How?

- For faulty user-generated input, our application should politely tell the user that something is not right, and encourage him to change his input field.
- For bad server-generated input, we do not need to be that polite.
- In that case, we know that someone has deliberately tried to alter data that are not easily modifiable. The application should abort the operation and log the incident.
- A clean page with "Bad input. Incident logged." is enough.
- It may even stop him from having further attempts.

⚠ CAUTION:

Whatever you do, be very careful if you try to massage or modify the invalid input to make it valid. Why?

An Example to Answer the Question

A European bank provided some static help information to its customers by including the content of text files in nicely formatted web pages.

```
http://www.bank.example/info.asp?file=info1.txt
```

Directory traversal:

```
http://www.bank.example/info.asp?file=../default.asp
```

- If the above URL had been accepted, the attacker would have gained access to the source code of a server-side script.

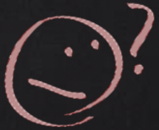
The programmers included code that should prevent directory traversal by getting rid of suspicious parts of the given file name.

Instead of just stopping upon invalid server-generated input, they tried to massage the file name to get rid of path traversal components.

```
filename = Request.QueryString("file")
Replace(filename, "/", "\")
Replace(filename, "..\", "")
```

Looks quite clever, right?

How an attacker can bypass it?



```
http://www.bank.example/info.asp?file=....//default.asp
```

```
....\\default.asp
```

```
..\default.asp
```

The application itself just helped the attacker gain access to a file he shouldn't have access to.

⚠ So, Do not massage invalid input to make it valid!

Summary

Summary

- Input from the client may enter our web applications in many shapes: URL parameters, POSTed form data from text fields, check boxes, selection lists and hidden fields, and from cookies and other HTTP headers.
- We need to identify all input used by our application, both the input we pick up directly from the request, and that we get from more or less well-understood programming platform constructs.
- Some of the input parameters come from user interface elements that let the user dictate the values. We call these parameters user-generated input.

SUMMARY

- Others are not directly modifiable by the user, such as hidden fields, check box values, cookies and so on. We call these server-generated input, as they originate on the server and should be passed back unchanged from the client.
- An attacker may modify both user- and server-generated input, so we must validate both types.
- We should pay particular attention to malformed server-generated input, as it indicates that the user has bypassed the normal user interface and done modifications behind the scene.

SUMMARY

- We should never massage invalid input to make it valid, as an attacker knowing our massaging algorithm may be able to make it work for him.
- Input validation makes sure data has expected values, suitable for our program logic.
- Input validation is not there to prevent metacharacter problems occurring when we pass data to subsystems, although sometimes our validation rules may prevent those problems as a side effect.
- In such cases, input validation gives us defense in depth, at least as long as we follow the rules of always handling metacharacters whenever we pass data along.

Your tasks for this week

Reading:

- "Innocent Code: A Security Wake-Up Call for Web Programmers (Chapter 3).

Building a Security Lab

- Download Lab Week 3 Instruction file from course webpage on GitHub.

https://hogeschool.github.io/Software-Quality/

Read the Instructions and perform the tasks.