Software Security

OWASP Open Source Review Project

Date: Jan 8, 2015

Group: 7

Students: Berry Busser (RU-4492196)
berrybusser@hotmail.com

Mark Vink (RU-4378008)
info@markvink.nl

Niels Harder (RU-4517326)
niels.h.nl@gmail.com

Table of contents

Table of contents 2
1. Organisation 3
2. Verdict 4
V2: Authentication Verification Requirements 4
V3. Session management 9
V4. Access control 11
V5: Malicious input handling verification requirements 13
V6: Output encoding / escaping 15
V7: Cryptography at rest verification requirements 16
V8: Error handling and logging verification requirements 17
V9: Data protection verification requirements 20
V10: Communications security verification requirements 22
3. Refection 24

1. Organisation

Briefly describe the way you organized the review.

The review has been carried out by three team members with varying degrees of knowledge and experience when it comes to PHP. The reviewers have examined the source of the application until each of them had a basic (or even good) understanding of the way it functions.

After getting a basic idea of the workings of the application the work was split by category of security requirements. Team members with more PHP knowledge have checked the more technical requirements.

For every requirement the following steps have been followed:

1. Test the related functionality in a running copy of the application.
2. Use the "Fortify static code analyzer" tool to check for systematically detectable vulnerabilities.
3. Manually verify the findings of the tool to make sure there are no false positives.
4. Identify related source files and manually check these for suspected vulnerabilities.
5. Document the findings and have this reviewed by another team member.

If the Fortify tool has already determined a verification to fail consistently the manual checks as described in step 4 have been deemed unnecessary. The last step has been implemented to make sure that multiple members have reviewed a verification and the chance for false positives and false negatives is decreased.

2. Verdict

Give your judgment for each of the verification requirements, with a short motivation.

V2: Authentication Verification Requirements

Authentication is the act of establishing, or confirming, something (or someone) as authentic, that is, that claims made by or about the thing are true. Ensure that a verified application satisfies the following high level requirements

2.1 Verify all pages and resources by default require authentication except those specifically intended to be public (Principle of complete mediation).

When following the application flow, starting from “testcms-v2/index.php”. It includes bootstrap.php which starts:s

```
/** Handle routing */
TestCMS::run();
```

Within this run() in testcms.php, the code enters an admin area. This area defines three public places/actions. All others places/actions within the admin area include this authentication function.

```
// public admin actions
$public = array('users/login', 'users/amnesia', 'users/reset');

// redirect to login
if(Users::authed() === false and in_array(trim($controller, '_controller').
'/' . $action, $public) === false) {
    return Response::redirect(Config::get('application.admin_folder') .
           '/users/login');
```

Uses::auth() creates a default authentication to pages within the admin area’s except a few public pages. There is no other landing page within the root of “testcms-v2” that bypasses the include of “system/bootstrap.php” Thereby we judge this requirement with a **PASS.**

2.2 Verify that all password fields do not echo the user’s password when it is entered.

A full text search for `$post['pass']` / password / pass within the project reveals **no** echo user’s passwords from password fields. Except on “\testcms-v2\system\admin\theme\users\reset.php” where it seems to echo the password, thereby we judge this requirement as **FAIL**:

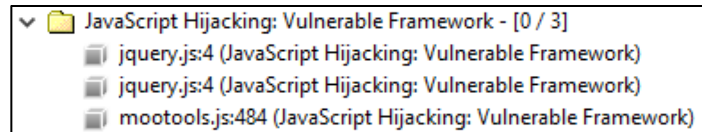
```
<input name="password" id="password" type="password" value="<?php echo
Input::post('password'); ?>">
```

Although this page does not seem to be reachable from the GUI, it does seem to be included from the public functions `reset($hash)` and `recover_password()`:

```
$hash = hash('md5', $user->id . $user->email . $user->password);
$link = Url::build(
      array('path' => Url::make('admin/users/reset/' . $hash))
);
```

2.4 Verify all authentication controls are enforced on the server side.
The authentication control source code is written in PHP. As PHP is server side code, this authentication is thereby enforced on the server. Furthermore there doesn't seem to be any JavaScript code regarding authentication. Although there might be client side issues with the third party JavaScript files jquery.js and mootools.js. The static analysis tool Fortify warns that these files may be vulnerable to JavaScript hijacking:

"Applications that use JavaScript notation to transport sensitive data can be vulnerable to JavaScript hijacking, which allows an unauthorized attacker to read confidential data from a vulnerable application."



Thus judging this requirement with a **FAIL.**

2.6 Verify all authentication controls fail securely to ensure attackers cannot log in.
This depends on the definition of "securely" which could result in judging this requirement with: don't know – security requirement unclear. When trying to log in with wrong credentials, it returns the message "Incorrect details". Which sounds like a good thing to do. Returning detailed messages like "wrong username" or "wrong password" are bad and would give hints to the attacker. Which is not the case with this CMS.

When the database is offline (or with wrong database credentials) when trying to log in, an unhandled exception is returned. This case it returned *"SQLSTATE[HY000] [2002] A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond. in "**C:\xampp\htdocs\testcms-v2\system\classes\db.php** on line **25**."*
Furthermore it returns the stack trace with the source code that causes the error, in this case:

```
#0   C:\xampp\htdocs\testcms-v2\system\classes\db.php(25):   PDO->__construct('mysql:dbname=te...', 'root', 'toor')
```

This an unhandled exception leaks out the database type, local file/webserver location, username, password, small part of the database name and the type of approach within PHP of creating queries (PHP Data Objects in this case). Thereby judging this requirement with a big **FAIL**.

2.7 Verify password entry fields allow, or encourage, the use of passphrases, and do not prevent long passphrases/highly complex passwords being entered.
There are no HTML and JavaScript validations or requirements on the password fields. Nor is there in the PHP function add() within users.php and in the function insert() in db.php which are responsible for creating a new user. At the database level the table "users" contains a "password" field with the type "text". This means there is no defined maximum character length restriction on the database, apart from the maximum character length that the text field itself has, which is 65.535 characters. Because there are no restrictions, this allows the use of passphrases and it does not prevent long passphrases / highly complex passwords from being entered. On the other hand, a single character can be used as a passwords, there are no minimum requirements. But this is not in the scope of this requirement (see requirement 2.27), thereby judging this requirement with a **PASS**.

2.8 Verify all account identity authentication functions (such as update profile, forgot password, disabled / lost token, help desk or IVR) that might regain access to the account are at least as resistant to attack as the primary authentication mechanism.

The edit() function within the user controller, which is triggered when sending a post against "/testcms-v2/index.php/admin/users/edit/#" (where # is the id of the user within MySQL), does not check the authorization and authentication of the action. In other words, the edit() function does not check who you are and if you are allowed to execute this function with the given parameters. All it verifies is if the userid exists and if the form action is a post. It does not even check the old password when updating your password. Neither does the update() function in the user model class have enough checks to prevent a malicious update.

Furthermore as described in 2.1, you can modify anyone's profile as long as you can log in with any account. Thus regain access to your old account. Hereby judging this requirement with a **FAIL**.

2.9 Verify that the changing password functionality includes the old password, the new password, and a password confirmation.

As shown on the image on the right, when updating a user, your own or someone else's, only a new passwords is asked. No old password and a passwords confirmation is required. Anyone can change anyone's password. This does not comply with this requirement. Thereby judging this requirement with **FAIL**.

User details
Create the details for your new user to log in to Test CMS.
Username: admin
The desired username. Can be changed later.
Password:
Leave blank for no change.
Email: admin@email.com
The user's email address. Needed if the user forgets their password.

2.16 Verify that credentials are transported using a suitable encrypted link and that all pages/functions that require a user to enter credentials are done so using an encrypted link.

There is no SSL encryption, but this is because of the assignment which requires us to install the testcms-v2 on our local machines. Furthermore you have to pay for a SSL license. Anyway, when logging in the credentials of the users are send as plain text. They are send as post parameters as: "`user=username&pass=password`". Hereby judging this requirement as a **FAIL.**

2.17 Verify that the forgotten password function and other recovery paths do not reveal the current password and that the new password is not sent in clear text to the user.

The forgotten password function asks only for an email-address. This email-address is then used to send a confirmation about the password change with a link to follow to create a new password. We know that the password is stored in the database as a hash and without a salt with the PHP crypt() function: `$password = crypt($post['password']);`. As the password is hashed and not send in the clear to the user, we judge this requirement with **PASS.**

2.18 Verify that information enumeration is not possible via login, password reset, or forgot account functionality.

Within the "forgot your password" page, you can enter an email address. If such email exists, it will return the message: "We have send you an email to confirm your password change". If the email address doesn't exists, it will return the message: "Account not found". This makes information enumeration possible and thereby this requirement is judged as **FAIL**.

2.19 Verify there are no default passwords in use for the application framework or any components used by the application (such as "admin/password").

This CMS has a default administrator named "admin", but no default password can be found. A password is generated upon installation of the CMS and the database by the following lines of code:

```
$password = random(8);
$sql = str_replace('[[now]]', time(), file_get_contents('test.sql'));
$sql = str_replace('[[password]]', crypt($password), $sql);
```

The function random is a self-defined function that takes an integer as string length for random characters between "0-9a-zA-Z". This password is then fed to the PHP function crypt which returns a hashed string using the standard Unix DES-based algorithm and takes the `$sql` variable as the salt.

Because there is no default password and the admin password upon installation is generated, this requirement is judged as **PASS.**

2.20 Verify that request throttling is in place to prevent automated attacks against common authentication attacks such as brute force attacks or denial of service attacks.

There are no occurrence of the words "http_throttle" and "throttle" within the whole project. Ideally you would not do this in PHP but in Apache, with for example "mod_security" or "mod_throttle". There are other solutions in both hard- and software. But this exceeds the scope of our installation of the testcms-v2 on our local machines for this assignment. Thereby we give this requirement a **NA – check is beyond scope of code.**

2.22 Verify that forgotten password and other recovery paths use a soft token, mobile push, or an offline recovery mechanism.

The recovery paths such as the forgotten password create a MD5 hash of the userid, email and password. This hash is then used to create a link to reset the password. This link is send to the user where they can create a new password without knowing the original password.

```
$hash = hash('md5', $user->id . $user->email . $user->password);
$link = Url::build(array('path' => Url::make('admin/users/reset/'.$hash)));
```

The requirement defines that the forgotten password mechanism must either use a soft token, mobile push or an offline recovery mechanism, but it is not defining requirements on these three mechanisms. It's good that the password is being hashed, although the chosen hashing algorithm could have been better. If sending an email with a link can be considered as a soft token. Thereby judging this requirement as **PASS.**

2.24 Verify that if knowledge based questions (also known as "secret questions") are required, the questions should be strong enough to protect the application.

There is no "secret questions" present in this application. There is no GUI, source code noir database fields that could indicate the use of the "secret questions". Lacking this functionality will result in a **FAIL** judgment.

2.27 Verify that measures are in place to block the use of commonly chosen passwords and weak passphrases.

As mentioned in chapter 2.7, there are no restrictions on passwords. A single character can be used as a passwords, and there are no minimum requirements on the character types. For example if a password should contain an upper letter and a symbol. This means that there are no measures in place to block the use of commonly chosen passwords and weak passphrases. Thereby judging this requirement with a **FAIL**.

2.30 Verify that if an application allows users to authenticate, they use a proven secure authentication mechanism.

This again depends on the definition of a proven secure authentication mechanism. What we know is that within the authentication step the password is check with crypt:

```
if(crypt($post['pass'], $user->password) != $user->password)
```

The PHP documentation on the crypt() function encourages to use the alternative functions instead, functions which are more secure then crypt(). This leads us to believe that the authentication implementation is not a proven secure mechanism. Thereby judging this requirement as a **FAIL**.

2.32 Verify that administrative interfaces are not accessible to untrusted parties

As described in chapter 2.1. There is a default authentication mechanism on the administrative interfaces of the CMS. This prevents untrusted parties, for as long as they don't have an account on the website with the current setup, to access the administrative interfaces. There do not seem to be other ways to access these administrative interfaces then via the default authentication mechanism. Thereby judging this requirement with **PASS**.

V3. Session management

One of the core components of any web-based application is the mechanism by which it controls and maintains the state for a user interacting with it. This is referred to this as Session Management and is defined as the set of all controls governing state-full interaction between a user and the web-based application.

Note, the CMS is using the PHP session functions, thereby the outcome of these requirements depend on the PHP version running on the server which is hosting the CMS. For the current chapter, the environment has PHP version 5.6.14 installed which was released on 01 October 2015.

3.1 Verify that there is no custom session manager, or that the custom session manager is resistant against all common session management attacks.

The CMS does not use a custom session manager. It has a session.php which contains a Session class. But all this class does is calling the PHP session functions, it does not manage it. Thereby judging this requirement with **PASS**.

3.2 Verify that sessions are invalidated when the user logs out.

When the users chooses to log out, the Users::logout() function is triggered. Then short after Session::forget() is called to unset the session key. As the session key is only stored in the PHP session variable which is unset, that session is then invalid. Thereby judging this requirement with **PASS**.

3.3 Verify that session's timeout after a specified period of inactivity.

There seem to be configurations in place which define that the session should expire after 3600 seconds. Which is after one hour. But this configuration never seems to be used. No occurs can be found of $_SESSION['expire'], $_SESSION['last_activity'], $_SESSION['created'] and $_SESSION['start'] which could suggest that the CMS ends the session. There are no cookies regarding session timeouts within the CMS and nothing can be found in relevant classes and full project searches (all searches where done case insensitive). This leads us to believe that the session timeout is not in effect, thereby judging this requirement with **FAIL**.

3.5 Verify that all pages that require authentication have easy and visible access to logout functionality.

Wherever the user may go within the CMS or at the public pages, if the user is logged in, they will always see a message on the top right of the screen that they are logged in, as who they are logged in and with a direct link to log out. It's clearly visible and it's a one click logout functionality. This requirement is judged with **PASS**.

3.6 Verify that the session id is never disclosed in URLs, error messages, or logs. This includes verifying that the application does not support URL rewriting of session cookies.

The session id is not visible in the URL's nor has it been seen in error messages, unhandled exceptions or logs. The application does not support URL rewriting of session cookies. The session id never seems to be disclosed. Thereby judging this requirement with **PASS**.

3.7 Verify that all successful authentication and re-authentication generates a new session and session id.
As the generation of a session within the CMS is handled by the PHP function session_start(), PHP documentation says: "*It creates a session or resumes the current one bases on a session identifier passed via a GET or POST request, or passed via a cookie*". Our session is stored in a cookie which never seems to change. When including the PHP function session_id() which gets the current session id. It is clearly visable that no new session id is generated after one is created. Even after logging out and then logging in as a different user. The session id remains the same. Thereby judging this requirement with **FAIL**.

3.11 Verify that session ids are sufficiently long, random and unique across the correct active session base.
One given session id is "od96aptccasd74lussq95g2f72" which is both 26 characters and bytes long. As the PHP function strlen() returns the number of bytes which is used on the PHP session id. The "Insufficient Session-ID Length" vulnerability within the OWASP vulnerabilities list states that "*Session identifiers should be at least 128 bit long to prevent brute-force session attacks*" 128 equals to 16 bytes. This means that our session id of 26 bytes is sufficiently long. PHP uses a hashing algorithm to create the PHP session id. This creates randomness and enough uniqueness for this requirement. Thereby judging this requirement with **PASS**.

3.12 Verify that session ids stored in cookies have their path set to an appropriately restrictive value for the application, and authentication session tokens additionally set the "HttpOnly" and "secure" attributes
Backed by the Fortify tool: "*The program creates a cookie in cookie.php at line 19, but fails to set the HttpOnly flag to true*". It is missing the 7th parameter to set HttpOnly, thereby judging this requirement with **FAIL**.

3.16 Verify that the application limits the number of active concurrent sessions.
The application does not seem to limit the number of active concurrent sessions. No source code can be found that should limit these sessions. It was possible to be logged into the site on 3 different machines with the same username and password. All 3 machines could browse through the pages without one of the machines getting logged out. Thereby judging this requirement with **FAIL**.

3.17 Verify that an active session list is displayed in the account profile or similar of each user. The user should be able to terminate any active session.
There does not seem to be a GUI which contains a list of active sessions of the user. Nor is there any source code found which could contain such functionality. Users cannot terminate any active session, nor can the admin. Thereby judging this requirement with **FAIL**.

3.18 Verify the user is prompted with the option to terminate all other active sessions after a successful change password process.
The same issues as requirement 3.17, there does not seem to be a GUI or source code which could contain such functionality. Users can't terminate all other active sessions after a successful change password process. Thereby judging this requirement with **FAIL**.

V4. Access control

The approach to this verification started with the default landing page. From there we've looked for pages/resources/directories that should require authentication and checked if they did.

4.1 Verify that the principle of least privilege exists – users should only be able to access functions, data files, URLs, controllers, services, and other resources, for which they possess specific authorization. This implies protection against spoofing and elevation of privilege.
Upon inspection of the Users:authed() method. We've found that it authenticates the user based on having a session key or not. This is actually pretty bad because as we created a new user called "testuser" with a role "User" instead of "Administrator". We were able to access the admin area with the "testuser" user and update the administrator's profile. The different user roles have no effect. There is no principle of least privilege and thereby this requirement is judged with **FAIL**.

4.4 Verify that access to sensitive records is protected, such that only authorized objects or data is accessible to each user (for example, protect against users tampering with a parameter to see or alter another user's account).
Since every registered user has administrator rights, or in other words, every registered user as full access. Then every registered user can read/change/delete everyone's profile. So there is no access control to sensitive records. The only thing that protects the access to sensitive records is by not having an account.
To make matters worse, it is possible to tamper with parameters to alter another user's profile. When updating a profile you can intercept your HTTP request and change the last integer in the POST URL. The POST URL looks like: "POST /testcms-v2/index.php/admin/users/edit/2 HTTP/1.1". The value that is underlined can be changed to 1, then you have overwritten the whole admin's profile. This requirement is judged with **FAIL**.

4.5 Verify that directory browsing is disabled unless deliberately desired. Additionally, applications should not allow discovery or disclosure of file or directory metadata, such as Thumbs.db, .DS_Store, .git or .svn folders.
This is typically a webserver feature concern (Apache, IIS, etc.) that may be on by default and should be turned off. The CMS may have an ".htaccess" file instructing the webserver to turn on or of 'Indexes'. Although an "htaccess.txt" exists, there does not exists an ".htaccess" which is used by Apache. Unfortunately this does not prevent direct disclosures to files as http://localhost/testcms-v2/.gitignore and http://localhost/testcms-v2/htaccess.txt are readable.

We've also created a directory with a single test file and browsed to that directory via the browser. The result is shown in the picture to the right. This proves that directory browsing is enabled and thus not prevented within the CMS or by default within the webserver. Thereby judging this requirement with **FAIL**.

4.8 Verify that access controls fail securely.
Within the base landing page "/testcms-v2/index.php" we have found that they defined:

```
//  Block direct access to any PHP files
define('IN_CMS', true);
```

Next they use the following line of code at the top of each .php document that requires authentication:

```
<?php defined('IN_CMS') or die('No direct access allowed.');
```

These .php documents are:

- "/testcms-v2/system/admin/theme/functions.php"
- "/testcms-v2 /theme/functions.php"
- "/testcms-v2 /config.php"
- "/testcms-v2 /config.default.php"
- All files in "/testcms-v2/system/"
    - except the public pages within "/testcms-v2/system/admin/theme/"

If someone tries to access these files directly, they would be shown the message *"No direct access allowed."* and the code execution is halted. This mechanism fails secure enough to judge this requirement with a **PASS**.

4.9 Verify that the same access control rules implied by the presentation layer are enforced on the server side.
The access controls rules described above in chapter 4.8 are defined in PHP. As PHP is server side code, these rules are enforced on the server. Thereby judging this requirement with **PASS**.

4.13 Verify that the application or framework uses strong random anti-CSRF tokens or has another transaction protection mechanism.
From the "Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet" on the OWASP website, they state that: "*Cross-Site Scripting is not necessary for CSRF to work. However, any cross-site scripting vulnerability can be used to defeat token, Double-Submit cookie, referrer and origin based CSRF defenses.*" As we will further discus XSS vulnerabilities in chapter 5.16, we have stated that "*The application is very vulnerable to XSS attacks*". Thereby making the CMS vulnerable to CSRF. As we were unable to find tokens or other anti-CSRF defenses used within the CMS. We judge this requirement with **FAIL**.

4.16 Verify that the application correctly enforces context-sensitive authorization so as to not allow unauthorized manipulation by means of parameter tampering.
As described in chapter 4.4, it is full possible to make use of parameter tampering. There is no form of authorization to not allow unauthorized manipulation. Thereby judging this requirement with **FAIL**.

V5: Malicious input handling verification requirements

The most common web application security weakness is the failure to properly validate input coming from the client or from the environment before using it. This weakness leads to almost all of the major vulnerabilities in web applications, such as cross site scripting, SQL injection, interpreter injection, locale/Unicode attacks, file system attacks, and buffer overflows.

5.1 Verify that the runtime environment is not susceptible to buffer overflows, or that security controls prevent buffer overflows.

There are a few places with functions that should be checked, these are:

- Using the search bar – functions/search.php & posts.php.
- Logging in as an admin or using the "forgotten password" function – classes/user.php.
- Posting a new comment – classes/comments.php.

When searching the function in posts.php is indirectly called. This function does not have any buffers to be overflowed. The same can be said for the other files mentioned above.

The webserver where this application is hosted will also most likely limit the URL-length, preventing overflows from the search parameter. It should be noted than an arbitrarily large input (1 million+ characters) in a field will produce an internal server error due to exceeding the maximum packet size of MySQL. However, according to the Fortify code analyzer and manual verification there seems to be no indication of a buffer overflow vulnerability. Thereby we judge this verification with a **PASS**.

5.3 Verify that server side input validation failures result in request rejection and are logged.

The application seems to offer logging functionality in the file /classes/log.php. This functionality seems to remain unused as it's supposed to write to system/logs/error.log, which does not seem to exist even after creating errors or providing invalid input. Therefore we judge this verification with a **FAIL**.

5.5 Verify that input validation routines are enforced on the server side

The only client-side input validation to be found is checking for valid e-mail addresses in the relevant parts of the application. This includes leaving a comment, recovering passwords and changing an account's registered e-mail address. In all of these places the e-mail address is verified on the server with the "FILTER_VALIDATE_EMAIL" filter. Therefore we judge this with a **PASS**.

5.10 Verify that all SQL queries, HQL, OSQL, NOSQL and stored procedures, calling of stored procedures are protected by the use of prepared statements or query parameterization, and thus not susceptible to SQL injection.

This can easily be checked by a static analysis tool. That is why we have decided to examine the evidence collected by Fortify. Multiple SQL injection vulnerabilities are found, 5 excluding the vulnerability in the installer of the application. Upon further examination of 'classes/comments.php" we can see the following statement:

```
$sql = "insert into comments (" . implode(', ', $keys) . ") values (" . implode(',
', $values) . ")";
```

The '$values' variable is an array with user-input, clearly this statement is not prepared in a safe way and thus the verification is judged with a **FAIL**.

5.11 Verify that the application is not susceptible to LDAP Injection, or that security controls prevent LDAP Injection.
The Fortify tool is unable to find any LDAP injection vulnerabilities. Furthermore, the application seems to use the MySQL database instead of LDAP. As LDAP does not seem to be used there also seems to be no related security risk. We judge this verification with a **PASS**.

5.12 Verify that the application is not susceptible to OS Command Injection, or that security controls prevent OS Command Injection.
Analyzing the source of the application does not show any places where an OS command is directly called. The Fortify tool also does not seem to have found any command injection vulnerabilities. Therefore we judge this verification with a **PASS**.

5.13 Verify that the application is not susceptible to Remote File Inclusion (RFI) or Local File Inclusion (LFI) when content is used that is a path to a file.
This verification was tested with the application running on the Apache webserver, the application files were in a folder called "htdocs". It seems to be impossible to traverse to the parent folder, which is good.
It should be noted that by default the "install" folder of the CMS is still present in the folder, one can easily open files in the install folder. This folder contains an SQL file, which reveals the structure of the database, and perhaps a skilled attacker could trigger the installer to reinstall the CMS.
However, assuming that the install folder is deleted as intended, this verification is judged with a **PASS**.

5.14 Verify that the application is not susceptible to common XML attacks, such as XPath query tampering, XML External Entity attacks, and XML injection attacks.
From manual verification it seems that the application does not use XML objects for things like database storage. The Fortify tool does not report any XML related errors either and thus this verification is judged with a **PASS**.

5.15 Ensure that all string variables placed into HTML or other web client code is either properly contextually encoded manually, or utilize templates that automatically encode contextually to ensure the application is not susceptible to reflected, stored and DOM Cross-Site Scripting (XSS) attacks.
The application is very vulnerable to XSS attacks. The Fortify tool reports 50 out of 59 critical errors to be XSS related. The vulnerabilities were tested, users can insert any HTML or JavaScript they want into a comment and it will be executed. One of the many possible attacks this could lead to is stealing an administrator's cookies (unnoticed) and gaining complete control of the CMS. The injected JavaScript in the comment even gets executed when the admin opens the CMS to edit the post. This verification is clearly judged as a **FAIL**.

5.22 Make sure untrusted HTML from WYSIWYG editors or similar are properly sanitized with an HTML sanitizer and handle it appropriately according to the input validation task and encoding task.
As stated before, the input fields for a comment can be used to insert any HTML elements into the page. This verification is therefore judged with a **FAIL**.

V6: Output encoding / escaping

This section was incorporated into V5 in Application Security Verification Standard 2.0. ASVS requirement 5.16 addresses contextual output encoding to help prevent Cross Site Scripting.

V7: Cryptography at rest verification requirements

7.2 Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable oracle padding.

The areas of the application making use of cryptography (excluding the installer) are user-related functionality. Namely adding a user, logging in, resetting a password and updating a user.

The application uses DES as a method of encryption, which requires a mode of operation. It is not clear which mode of operation is used, if the Cipher Block Chaining (CBC) mode is used then there is the possibility of an oracle padding attack. It is unclear which mode of operation is used, therefore this verification is judged with **DON'T KNOW**.

7.7 Verify that cryptographic algorithms used by the application have been validated against FIPS 140-2 or an equivalent standard.

The functionality regarding users is reported by Fortify to use weak encryption, namely DES. This is extremely outdated and will be cracked in a short amount of time on modern machines. Furthermore, the following piece of code from the user-creation process shows that password hashes are not salted:

```
$post['password'] = crypt($post['password']);
```

According to the FIPS140-2 standard the DES function is no longer acceptable. It is also worth noting that a "unique application key used for signing passwords" is created when installing the app, but it seems like this key is never used. Therefore this verification is judged with a **FAIL**.

V8: Error handling and logging verification requirements

The primary objective of error handling and logging is to provide a useful reaction by the user, administrators, and incident response teams. The objective is not to create massive amounts of logs, but high quality logs, with more signal than discarded noise.

8.1 Verify that the application does not output error messages or stack traces containing sensitive data that could assist an attacker, including session id, software/framework versions and personal information.
In the base file system/bootstrap.php there is a custom handler registered for the case any errors or exceptions may occur. These handlers point to the class Error in classes/error.php. Method 'native' is called when a usual error occurs and method 'exception' when an exception is thrown. Based on the configuration the error is logged and/or displayed to the visitor. It depends on the configuration set by the website owner if any error is displayed.
The version of the CMS is displayed on the admin login page which is publicly available. Since this has nothing to do with error messages, and the displaying of errors is configurable, this verification is judged as **PASS**.

8.2 Verify that error handling logic in security controls denies access by default.
The only page with security controls is the page where administrators login onto the administrator panel. This page is secured by a form asking the visitor to provide a username-password pair. Posts on this form are handled by the class Users_controller. This specific method displays the login page by default and redirect the user only in case valid credentials were presented. Therefore we judge this verification as **PASS**.

8.3 Verify security logging controls provide the ability to log success and particularly failure events that are identified as security-relevant.
In relevance of the given CMS system, successful and unsuccessful login attempts are considered to be security-relevant. The outcome of the login process is only displayed to the visitor in case of an error (missing or invalid credentials). Since there is no proper logging in place we judge this verification as **FAIL**.

8.4 Verify that each log event includes necessary information that would allow for a detailed investigation of the timeline when an event happens.

```
[info] --> Requested URI: admin/users/login
[info] --> Translated URI: admin/users/login
[info] --> Controller action: users_controller/login
[error] --> Theme file <strong>themes/default/users/login.php</strong> not found. in /Users/mark/Development/testcms-v2/system/classes/template.php on line 56
```

In the table above an example is shown of logged events in case of an error. Events are shown in order of occurrence and this will allow an administrator to dig in the sequence/timeline. Although we believe it's also important to log information about the specific time and visitor. The information may be enough in some cases but will not help in a detailed investigation. Therefore we have to judge this verification as **FAIL**.

8.5 Verify that all events that include untrusted data will not execute as code in the intended log viewing software.

```
[info] --> Requested URI: <script>alert(1)</script>
[info] --> Translated URI: <script>alert(1)</script>
[info] --> Controller action: Routes/<script>alert(1)<
[warning] --> Action does not exist
```

Since the CMS system does log additional information messages like which URI is requested, it is easy to inject executable code (PHP/JavaScript) in the logging file. There is no software provided to inspect loggins; users should rely on own native tools for opening and viewing the log files. Therefore executable code in logging does not form a real threat. We judge this verification as **PASS**.

8.6 Verify that security logs are protected from unauthorized access and modification.

```
[error] --> SQLSTATE[HY000] [1045] Access denied for user 'root'@'localhost' (using password: YES) in /Users/mark/Development/testcms-v2/system/classes/db.php on line 25
```

Log files are not protected from unauthorized access. Every visitor with knowledge about the CMS system can obtain the logs using his web browser by surfing to the right URL. If directoryindex is allowed by the serving server, the user can easily visits the location '/system/logs/' to get the full list of available logs files. If that is not the case then individual log files may accessed by visiting the right location based on the date of the file. For example 'system/logs/2015-12-28.log' contains all errors occurred on December 29, 2015. This behavior can potentially leak confidential information like database usernames as shown in the table above. Therefore we judge this verification as **FAIL**.

8.7 Verify that the application does not log sensitive data as defined under local privacy laws or regulations, organizational sensitive data as defined by a risk assessment, or sensitive authentication data that could assist an attacker, including user's session identifiers, passwords, hashes, or API tokens.
The CMS system does log very minimal information; only level (info/warning/error) and a message. This message does not include any personal or sensitive information unless this message was passed by an exception. The only helping message we could find contained the database name and username. Therefore we judge this verification as **PASS**.

8.8 Verify that all non-printable symbols and field separators are properly encoded in log entries, to prevent log injection.

```
public static function write($severity, $message) {
    ...
     $line = '[' . $severity . '] --> ' . $message . PHP_EOL;

     if($fp = @fopen(PATH . 'system/logs/' . date("Y-m-d") . '.log', 'a+')){
          fwrite($fp, $line);
          fclose($fp);
     }
}
```

Messages which are logged by the CMS are not encoded in any form and therefore is vulnerable to log injection. We have to judge this verification as **FAIL**.

8.9 Verify that log fields from trusted and untrusted sources are distinguishable in log entries.

The CMS system only logs messages with the level info/warning and error. Info and warning messages are only produced by the CMS itself and therefore is considered as a trusted source. Error messages are only logged then passed via an error handler. In that last case we are certainly sure the errors have occurred. Therefore we judge this verification as **PASS**.

8.10 Verify that an audit log or similar allows for non-repudiation of key transactions.

A key transaction can be considered as any action performed by administrative users using the admin panel. Such actions may include:

- Adding and editing blog posts.
- Adding and editing pages.
- Updating and removing comments.
- Adding and editing other administrators.

None of these actions are specifically logged; only visited locations are but this does not tell us if an action has occurred on that location. Furthermore log messages or locations are not bound to corresponding visitors, making it unable to trace back the source. Therefore this verification is judged as **FAIL**.

8.11 Verify that security logs have some form of integrity checking or controls to prevent unauthorized modification.

Log files does not have any property proving the integrity of the files. It is possible to edit the files on the server without the administrator noticing. Therefore this verification is judged as **FAIL**.

8.12 Verify that the logs are stored on a different partition than the application is running with proper log rotation.

Log files are rotated based on the date the error occurred. Server administrators are responsible to archive dated log files. These files are written within the accessible webroot, in the same partition as the application. Therefore this verification is judged as **FAIL**.

V9: Data protection verification requirements

There are three key elements to sound data protection: Confidentiality, Integrity and Availability (CIA). This standard assumes that data protection is enforced on a trusted system, such as a server, which has been hardened and has sufficient protections. The application has to assume that all user devices are compromised in some way. Where an application transmits or stores sensitive information on insecure devices, such as shared computers, phones and tablets, the application is responsible for ensuring data stored on these devices is encrypted and cannot be easily illicitly obtained, altered or disclosed.

9.1 Verify that all forms containing sensitive information have disabled client side caching, including autocomplete features.

Since this is a content management system for managing publicly available websites, there is not much sensitive information posted on forms. There a few forms which may contain sensitive information; the first to sign into the administration interface, the forms for resetting your password, and the form for creating a new admin user. None of these forms have autocomplete disabled, which mean admin users could leak information in their own browser. Therefore this verification is judged as **FAIL**.

9.2 Verify that the list of sensitive data processed by the application is identified, and that there is an explicit policy for how access to this data must be controlled, encrypted and enforced under relevant data protection directives.

There isn't really sensitive information processed except credentials of admin users. There is no policy available on how these credentials are accessed, controlled or stored (encrypted). We don't believe further information is processed that falls under relevance of data protection directives and therefore we have to judge this verification with **N/A**.

9.3 Verify that all sensitive data is sent to the server in the HTTP message body or headers (i.e., URL parameters are never used to send sensitive data).

The hash for resetting a password is embedded in the URL where someone can reset his password. Namely; *'/admin/users/reset/<hash>'*. In our opinion this is even worse than using parameters since full URI's are usually visible in access logs and analytics. Therefore this verification is judged as **FAIL**.

9.4 Verify that the application sets appropriate anti-caching headers as per the risk of the application

```
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Connection: Keep-Alive
Content-Length: 2362
Content-Type: text/html; charset=UTF-8
Date: Wed, 30 Dec 2015 10:40:03 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Keep-Alive: timeout=5, max=100
Pragma: no-cache
Server: Apache
X-Powered-By: PHP/5.6.10
```

The response headers looks like anti-caching headers are in place, although it is unclear if the CMS or web server produces these. Therefore we have to judge this verification with **N/A**.

9.5 Verify that on the server, all cached or temporary copies of sensitive data stored are protected from unauthorized access or purged/invalidated after the authorized user accesses the sensitive data.
There is no caching in place in the CMS. The only type of temporary copies is data stored in the (server) session. The session may contain information about notifications and the current logged-in user. Notifications are removed after they are displayed, thereby this verification is judged as **PASS**.

9.6 Verify that there is a method to remove each type of sensitive data from the application at the end of the required retention policy.
There is no sensitive information stored by the CMS, which should be removed after some kind of timespan; therefore we have to judge this verification with **N/A**.

9.7 Verify the application minimizes the number of parameters in a request, such as hidden fields, Ajax variables, cookies and header values.
There is no unnecessary usage of parameters, hidden fields, cookies or headers. Therefore this verification is judged with **PASS**.

9.8 Verify the application has the ability to detect and alert on abnormal numbers of requests for data harvesting for an example screen scraping.
The application is not capable of detecting abnormal numbers of requests. This could be implemented in the application or on the web server using some kind of request throttling. Thereby we give this requirement a **N/A** - *check is beyond scope of code*.

9.9 Verify that data stored in client side storage - such as HTML5 local storage, session storage, IndexedDB, regular cookies or Flash cookies - does not contain sensitive or PII).
The only local storage, which is used by the CMS, is the default SESSIONID cookie. This cookie only contains a reference to the session stored on the server and does not contain any sensitive information. Therefore this verification is judged with **PASS**.

9.10 Verify accessing sensitive data is logged, if the data is collected under relevant data protection directives or where logging of accesses is required.
No access to data is not logged by the CMS, although the information is not considered to be sensitive. Thereby we give this requirement a **N/A** - *check is beyond scope of system*.

9.11 Verify that sensitive data is rapidly sanitized from memory as soon as it is no longer needed and handled in accordance to functions and techniques supported by the framework/library/operating system.
The PHP function unset() can be used to destroy certain variables and their content. Although this function is called several times, it's not used to sanitize information after it is not needed anymore. Therefore this verification is judged with **FAIL**. Since variables only hold their content for the current request there it does not expose a large risk.

V10: Communications security verification requirements

We are performing a code analysis of a content management system written in PHP. Usually the incoming connection (and whether or not SSL is used) is handled by a webserver like Apache or Nginx, or by a reverse proxy.

10.1 Verify that a path can be built from a trusted CA to each Transport Layer Security (TLS) server certificate, and that each server certificate is valid.
We are not provided with a web server that handles the SSL part and are not provided with certificates to check. **N/A** - *check is beyond scope of system*.

V10.3 Verify that TLS is used for all connections (including both external and backend connections) that are authenticated or that involve sensitive data or functions, and does not fall back to insecure or unencrypted protocols. Ensure the strongest alternative is the preferred algorithm.
The CMS does require a connection with a MySQL database in order to store and retrieve information. It is most likely that a local database will be used but this is not required or enforced. It is possible to configure the CMS to open a connection with a database over a local network or over the Internet. This connection is not secured by SSL on the application level, it wouldn't even be possible to configure without modifying the code yourself. An alternative would be to make a secure tunnel between the webserver and database server. **N/A** - *check is beyond scope of code*.

V10.4 Verify that backend TLS connection failures are logged.
N/A - *check is beyond scope of system*.

V10.5 Verify that certificate paths are built and verified for all client certificates using configured trust anchors and revocation information.
N/A - *check is beyond scope of system*.

V10.6 Verify that all connections to external systems that involve sensitive information or functions are authenticated.
In order to make a connection with the database you will have to configure the database host, name, username and password. Therefore this verification is judged with **PASS**.

V10.8 Verify that there is a single standard TLS implementation that is used by the application that is configured to operate in an approved mode of operation.
The application itself does not use TLS in any form, therefore this verification is judged with **FAIL**.

V10.10 Verify that TLS certificate public key pinning is implemented with production and backup public keys. For more information, please see the references below.
Pinning of certificates has to be configured on the web server and therefore this verification is judged with **N/A** - *check is beyond scope of system*.

V10.11 Verify that HTTP Strict Transport Security headers are included on all requests and for all subdomains, such as Strict-Transport-Security: max-age=15724800; includeSubdomains
The response headers do not include HSTS-headers in order to protect against protocol downgrade attacks and cookie hijacking. Therefore this verification is judged as **FAIL**.

V10.12 Verify that production website URL has been submitted to preloaded list of Strict Transport Security domains maintained by web browser vendors. Please see the references below.
N/A - *check is beyond scope of system*.

V10.13 Ensure forward secrecy ciphers are in use to mitigate passive attackers recording traffic.
N/A - *check is beyond scope of system*.

V10.14 Verify that proper certification revocation, such as Online Certificate Status Protocol (OSCP) Stapling, is enabled and configured.
N/A - *check is beyond scope of system*.

V10.15 Verify that only strong algorithms, ciphers, and protocols are used, through all the certificate hierarchy, including root and intermediary certificates of your selected certifying authority.
N/A - *check is beyond scope of system*.

V10.16 Verify that the TLS settings are in line with current leading practice, particularly as common configurations, ciphers, and algorithms become insecure.
N/A - *check is beyond scope of system*.

3. Refection

Reflect on the whole process, including; the ASVS, the use of static code analysis tools, the way you organised the process and possibly also the TestCMS code.

How good (useful, clear, ...) is the ASVS? How could it be improved?

Only a part of the ASVS has been used for this project, so any conclusions and opinions are based on the part that was used only.

The general opinion is that the ASVS provides a more than decent process for checking the security of an application. This project has shown that following this process will lead to the checking of both obvious and not-so-obvious verifications. The end result seems to be a reasonably thorough check.

But the ASVS is not without its own issues. Some of the verifications are not applicable in the context of the application that you're reviewing. This can make it hard to pass or fail a verification and can get in the way of your final judgment. Some of the other verifications are applicable but very difficult to check for the entire application or too vaguely defined to make a meaningful verification.

And finally the ASVS document has undergone so many revisions that it's become a mess. The verification numbering does not make much sense anymore and chapter six has been completely removed. It would be better if a new version of the document would be released to address this problem.

How useful were code analysis tools? How could they be improved? How did you experience the rates and amounts of false and true positives? How might that be improved?

This project has a reasonably small codebase to review and yet it was clear how much trouble it can be to check the entire source. Code analysis tools address this problem to a certain extent.

Quite a few verifications have quickly received a 'fail' judgment after looking at the results of the used tools, so it has been conceived that the tools do indeed provide a useful overview of vulnerabilities. However, not all verifications could be verified using just the results of the used tools and the results should not be trusted without a certain degree of human verification.

In a small project like this it was already possible to find some (possible) false positives. While the tools do provide a good list of vulnerabilities its' findings can sometimes be hard to understand, which makes it hard to identify false positives. The Fortify tool tries to alleviate this problem by giving useful descriptions of the found vulnerability and allowing you to follow the flow of the vulnerable code. This has proven to be an effective solution for a lot of the found vulnerabilities.

In general the analysis tools have proven to be useful and will probably be even more useful for larger project. But the analysis tools should not be used without human verification, as this will not provide completely accurate conclusions about security.

What were the bottlenecks in doing the security review in your experience?

The greatest bottleneck has been to identify all the relevant parts of the source for all verifications. It cannot be said with one hundred percent certainty that every relevant part of the source has been found for all verifications, as this might prove to be an impossible task. The way to deal with this has been to look at a running instance of the application, identify the relevant and most important parts that seem to deal with the

verification and look at the source for these parts. Of course this still leaves the possibility of missing vulnerabilities.

A smaller but still present bottleneck has been the description of verifications. While most have a clear definition there are some that are too loosely defined or do not provide a concrete method to check. In some cases the OWASP website/wiki provides useful information regarding the subject of the requirement. But often a certain guideline to check the requirement was either missing or to abstract.

Are some (categories of) verification requirements easier to check than others?
Short answer: Most definitely. There can be various reasons for this, like:

- Analysis tools can check some requirements.
- Some requirements can be checked/exploited in a running instance of the application.
- Some requirements are better defined than the other.

For example: It is very easy to find XSS vulnerabilities with a static tool and test these in a running instance of the application. It is much harder to make sure there are no cryptographic modules for which the error handling enables oracle padding, since this is harder to detect for an analysis tool and not visible on a running instance of the application.

If you would have to do something like this again, what would you do differently?
For another (possibly larger) project it would make a great difference to involve one or more of the responsible software engineers in the process. Their role would be to answer questions about the source so it would be easier to identify relevant parts of the source code.

It would also be meaningful to have experts on certain subjects in the team and have them verify the requirements that they're most suited for. For example: Having an expert on cryptography verify the cryptography requirements.

The review has also showed that having results double-checked by different team members yields good results, so this should also be included in approach.

Are there important aspects that could (or should) be changed to improve security? Or aspects that could be changed to facilitate doing a security reviews?
Developers using PHP should use a code analysis tool during development. While this does not guarantee security it should at least help to prevent very obvious and serious vulnerabilities like XSS.

Was it useful for the review to run the application?
Most definitely. Being presented with just the source code is not a very easy way to review. Running the application can help to show the relevant parts that should be tested. For example: When verifying input validation requirements it is easier to run the application and look at the places that deal with input and then look up the matching source files.

For some requirements it has also been useful to run the application and exploit it. For example: If the source code makes it hard to judge to what extent XSS attacks are possible, it is easy to run the application and try to inject some JavaScript.