Security code review of TestCMS

Group 3 (Rob Wu, Peter Wu, Kai-Chun Ning, Michael Yeh)

January 8, 2016

1 Introduction

Security code review is a vital step in software development and is also necessary after software is deployed. Security code review aims to find security issues before the cost of fixing them grows out of control and can provide feedback for future development. Common issues e.g. buffer overflow, SQL injection and more sophisticated issues such as design flaws can be found via code review. There are two types of code review, static and dynamic. Static review is done without actually executing the software by inspecting the source code while dynamic review is performed by running the software. Usually the term static code analysis is used to refer to static code review performed by an automated tool and code review is used to refer to analysis done by human. However, in the rest of the report the term "static review" encompasses both meanings.

Both static and dynamic review are essential in software security testing. Although static review is able to pick up the majority of the security issues in the software. Dynamic review has the advantage of finding security issues caused by the interaction between the software and other parts of the system. Dynamic review usually involves running the software with various inputs and settings to test its behavior. It is thus necessary to understand the software first so valid and intentionally invalid inputs can be provided to the software. As a result, the software still needs to be studied statically first.

In this assignment we focus on static review and start by using a static code analyzer called HP Fortify to review a small PHP software project. After that, the PHP software is manually inspected to evaluate the effectiveness of the static analyzer.

The rest of the report is structured as follows: the scope and the organization of the assignment are explained in section 2. Section 3 lists various security flaws that were found by static review, either with the automated analyzer or by manual inspection. Section 4 draws a conclusion based on the previous analyses and discusses the advantages and disadvantages of the static analyzer.

2 Methodology

2.1 Scope

Before assigning tasks to each of the four group members, we did an initial analysis to determine the project's scope. The project consists of 6k lines of PHP code, 1k lines of JavaScript code, 1.5k lines of CSS code and 13 miscellaneous files (appendix: Initial analysis). It is therefore a relatively small project.

This project is reviewed using the OWASP Application Security Verification Standard (ASVS), version 3.0. An opportunistic security issue verification (level 1) is performed according to verification requirements V2 up to V9.

2.2 Organisation

Analyses are done on a per-category basis, first tackling with V2 requirements from the ASVS document and then continuing with the next verification category requirements.

The work is spread over the team, also based on categories. This allows the reviewers to focus on specific weaknesses. Hints from the static analyzers may be used as a starting point for further inspection, thus starting from low hanging fruit and gradually moving to a more labor-intensive manual analysis for the remaining items.

To make analysis easier, a review of the file relations was conducted which is described (appendix: Architecture).

3 Verdict

3.2 V2. Authentication Verification Requirements

3.2.1 2.1 (Level 1)

FAILED

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

The upgrade page can be accessed without login. Anyone can trigger the web app to download and update its functionality. This can be exploited,

for example, by an attacker who managed to compromise the update server and trick victims into downloading an infected version of the application.

3.2.2 2.2 (Level 1)

FAILED

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

reset.php:14 contains:

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

This code fragment shows that the password is echoed back with HTML when the user tries to reset her password.

3.2.4 2.4 (Level 1)

FAILED

Verify all authentication controls are enforced on the server side.

Access control to update page is not enforced. Anyone can access the page without even logging in. In addition, even normal users (not admin) can add other users to the web app. The web app clearly lacks robust access control mechanism.

3.2.6 2.6 (Level 1)

FAILED

Verify all authentication controls fail securely to ensure attackers cannot log in.

The user-provided input (user and pass) may trigger an exception. Since the control directly passes to the exception handler and bypasses the code that handles failed authentication, attackers may be able to access the unauthenticated section, as the server variable ($errors) had already been set. (See /system/classes/users.php:56)

3.2.7 2.7 (Level 1)

FAILED

Verify password entry fields allow, or encourage, the use of passphrases, and do not prevent long passphrases/highly complex passwords being entered.

The web app does not impose any restriction on the password nor does it encourage user to use complex password. Since not every user will use a long/complex password we consider this requirement failed.

3.2.8 2.8 (Level 1)

FAILED

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.

When a user first register an account, the web application doesn't echo the input password back (system/admin/theme/users/add.php:59). When the user tries to reset her password, her password is echoed (system/admin/theme/users/reset.php:14). Sending the password back is a terrible idea and thus we consider regaining access to the account is apparently less resistant than the primary authentication mechanism.

3.2.9 2.9 (Level 1)

NOT APPLICABLE

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

There is no dedicated password change functionality. Users who wish to change a password can only do so by using the password reset feature.

3.2.16 2.16 (Level 1)

FAILED

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.

The login credentials can be transfered with only http and as POST arguments. Any eavesdroppers can find out the user name and password.

3.2.17 2.17 (Level 1)

FAILED

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 new password is sent in clear text to the user (system/admin/theme/users/reset.php)

The password reset link contains a md5 hash of public information and the password (system/classes/users.php). MD5 hashes can be computed quite fast, so this method allows easy recovery of weak passwords to anyone who spots the reset link.

3.2.18 2.18 (Level 1)

FAILED

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

The system does not rate-limit login or password reset attempts. To every attempt, the server responds whether the given email is known (system/classes/users.php). This allows attackers to enumerate all known email addresses.

3.2.19 2.19 (Level 1)

FAILED

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

By default, the web application uses a default empty password to access the database. (config.default.php:8) If the user does not change it during bootstrapping, then the adversary may access the database directly.

3.2.20 2.20 (Level 1)

FAILED

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.

The login page has no request control, we wrote a simple brute force script(see brute_force.rb) and it worked. The password complexity is only $62^8 = 2^{48}$, totally breakable. Not only brute force is possible, the app does not prevent DDoS either.

3.2.22 2.22 (Level 1)

FAILED

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

Only uses email, and it is insecure as discussed in 2.17.

3.2.24 2.24 (Level 1)

NOT APPLICABLE

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

No secret question design in this web app.

3.2.27 2.27 (Level 1)

FAILED

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

No restriction on password is enforced. (system/admin/theme/users/add.php)

3.2.30 2.30 (Level 1)

FAILED

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

The web app by default creates a weak 8-char-long password for the admin. Although the admin needs to login with this password, all subsequent connections can use the same session id to authenticate. Given the fact that the web app is vulnerable to XSS attack and session hijacking, such authentication mechanism cannot be considered secure. Moreover, user can use http POST to send authentication data to the site and the secret password is totally exposed. Any eavesdropper can easily find out the admin password and gain access to the system.

3.2.32 2.32 (Level 1)

FAILED

Verify that administrative interfaces are not accessible to untrusted parties.

As explained in 2.30.

3.3 V3. Session Management Verification Requirements

3.3.1 3.1 (Level 1)

FAILED

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

The web app uses a customized session manager and is vulnerable to session hijacking attacks. (as shown with csrf.rb)

3.3.2 3.2 (Level 1)

PASSED

Verify that sessions are invalidated when the user logs out.

The web application uses a simple hash to keep track users who has logged in. When a user logs out. It's user name will be removed from the hash

map thus subsequent requests will need to authenticate again.

3.3.3 3.3 (Level 1)

FAILED

Verify that sessions timeout after a specified period of inactivity.

Sessions should timeout in 3600 seconds, which is the default setting in config.php. But due to a possible implementation bug, the cookie we received however has its expire date set in 1981. As a result the timeout has no effect. Even after 1 hour the session is still active.

3.3.5 3.5 (Level 1)

PASSED

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

The logout button is always there at the top right corner.

3.3.6 3.6 (Level 1)

FAILED

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.

Session id is not disclosed in URLs, error messsages or logs. However, it is still included in the HTTP header. Since the web application allows user to connect to with only HTTP, adversaries can obtain session id by eavesdropping. As a result, we consider this security requirement unmet.

3.3.7 3.7 (Level 1)

FAILED

Verify that all successful authentication and re-authentication generates a new session and session id.

The session ID only depends on user ID, email and password, thus a constant value for the same user (See system/classes/users.php, line 29).

3.3.12 3.12 (Level 1)

FAILED

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

PHP's `$_SESSION` is used, which stores cookies without the HttpOnly flag by default. There is no configuration that changes the default.

3.3.16 3.16 (Level 1)

FAILED

Verify that the application limits the number of active concurrent sessions.

The web application does not limit the number of active sessions.

3.3.17 3.17 (Level 1)

FAILED

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.

No such functionality is implemented.

3.3.18 3.18 (Level 1)

FAILED

Verify the user is prompted with the option to terminate all other active sessions after a successful change password process.

No such functionality is implemented.

3.4 V4. Access Control Verification Requirements

3.4.1 4.1 (Level 1)

FAILED

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.

The update web page is unprotected, anyone can access them without logging in.

3.4.4 4.4 (Level 1)

FAILED

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).

Any user can alter other users information, including email address, login password, and so on. The user classification, e.g. admin, normal, and activeness has no effect.

3.4.5 4.5 (Level 1)

FAILED

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.

Anyone can access the database setup script by visiting: install/test.sql The script leaks database layout, a sensitive information.

3.4.8 4.8 (Level 1)

FAILED

Verify that access controls fail securely.

The web application allows any files or dirs that exist to be displayed directly.

(See htaccess.txt). The only access control imposed is by the running apache server.

3.4.9 4.9 (Level 1)

FAILED

Verify that the same access control rules implied by the presentation layer are enforced on the server side.

As in 4.5, the presentation layer does not show any link to test.sql but since there is no access control imposed by the server side. The attacker can access the sensitive file.

3.4.13 4.13 (Level 1)

FAILED

Verify that the application or framework uses strong random anti- CSRF tokens or has another transaction protection mechanism.

No anti CSRF token is implemented. We wrote a script to simulate CSRF attack and were able to add an extra administrator user with the attack. (See csrf.rb)

3.4.16 4.16 (Level 1)

FAILED

Verify that the application correctly enforces context-sensitive authorisation so as to not allow unauthorised manipulation by means of parameter tampering.

No context-sensitive authorization is implemented.

3.5 V5. Malicious input handling verification requirements

3.5.1 5.1 (Level 1)

FAILED

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

The database is susceptible to buffer overflow attack, which is demonstrated with a simple script(See fuzzer.rb)

3.5.3 5.3 (Level 1)

FAILED

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

By default, login failures are not logged. (See config.default.php).

3.5.5 5.5 (Level 1)

PASSED

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

True.

3.5.10 5.10 (Level 1)

FAILED

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

The software is susceptible to SQL injection, see installer.php, line 127 and system/classes/comments.php, line 69.

3.5.11 5.11 (Level 1)

NOT APPLICABLE

Verify that the application is not susceptible to LDAP Injection, or that security controls prevent LDAP Injection.

The web application does not use LDAP protocol.

3.5.12 5.12 (Level 1)

NOT APPLICABLE

Verify that the application is not susceptible to OS Command Injection, or that security controls prevent OS Command Injection.

The web application does not use any user input as OS command.

3.5.13 5.13 (Level 1)

FAILED

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.

The installer is normally used to write strings such as a database host, username and password to a configuration file. This file is then included in the main application via the `require` directive which would then execute the PHP code and generate an array.

The problem with the installer is that it does not sanitize the provided database credentials, allowing a code injection attack. This trivially allows LFI, RFI and more attacks.

3.5.14 5.14 (Level 1)

NOT APPLICABLE

Verify that the application is not susceptible to common XML attacks, such as XPath query tampering, XML External Entity attacks, and XML injection attacks.

The web application does not use XML for storing data, processing request, or including remote files. It does not use XPath either.

3.5.15 5.15 (Level 1)

FAILED

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 web application fails to sanitize user input and is susceptible to various XSS attacks. At line 73 of system/admin/theme/assets/js/comments.js, input is read as text, but then written as HTML at line 90. This is a DOM-based XSS attack.

The comments module also has a stored XSS vulnerability. (see system/admin/theme/posts/edit.php, line 144 and system/classes/comments.php, line 35).

3.5.22 5.22 (Level 1)

FAILED

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 mentioned in section 3.4.9, HTML inputs are not sanitized.

3.7 V7. Cryptography at rest verification requirements

3.7.2 7.2 (Level 1)

FAILED

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

User login does not fail securely (see system/classes/user.php, line 53). The module assumes the user to be valid in the first place and starts to verify the submitted credentials afterwards. Only when an error occurs will the login be rejected. Such a design violates the secure-failing principle. The adversary only needs to come up with a method to keep the $errors structure empty and does not need to obtain valid login credentials. The login procedure should have assumed the user is invalid and the access can only be granted when the credentials are verified to be valid.

3.7.7 7.7 (Level 1)

FAILED

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

The software uses a weak and outdated encryption DES (See installer.php, line 122). The cryptographic algorithm does not pass contemporary standards.

3.8 V8. Error handling and logging verification requirements

3.8.1 8.1 (Level 1)

PASSED

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

The web app can report very detailed information including stack trace, and PHP error message back to user. (See system/admin/theme/error_php.php and system/classes/error.php). By default this feature is not enabled (config.default.php and upgrade/migrations.php).

Note that the above assessment is only valid as long as the `error.log` property is not changed. When it is set to true (as one might want to do during development), then the requirement would fail as the full error message is shown.

3.9 V9. Data protection verification requirements

3.9.1 9.1 (Level 1)

FAILED

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

Failed, install/index.php contains a form with a password field. This password field does not mask the password and the form does not contain any measures to prevent the values from getting saved by a browser's autocomplete feature.

3.9.3 9.3 (Level 1)

FAILED

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).

Even though the web app only uses POST to send form data, users can use HTTP to send them. An eavesdropper can learn everything that is being sent. A careless user who forgets to use HTTPS will still leak all sensitive information.

3.9.4 9.4 (Level 1)

PASSED

Verify that the application sets appropriate anti-caching headers as per the risk of the application, such as the following:
Expires: Tue, 03 Jul 2001 06:00:00 GMT
Last-Modified: {now} GMT
Cache-Control: no-store, no-cache, must-revalidate, max-age=0
Cache-Control: post-check=0, pre-check=0
Pragma: no-cache

In response to a user log in attempt, the server's response includes anti-caching headers:

- Expires=Thu, 19 Nov 1981 08:52:00 GMT (See 3.3)
- Cache-Control=no-store, no-cache, must-revalidate, post-check=0, pre-check=0
- Pragma=no-cache

3.9.9 9.9 (Level 1)

PASSED

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 application does not use client-side storage.

3.10 V10. Communication

NOT APPLICABLE

TLS configuration is an issue for the server administrator, not the application developer. The application did not provide specific instructions on configuring the listening port, so we will defer this analysis.

4 Reflection

In this assignment we first used HP Fortify static source code analyzer to conduct a preliminary vulnerability examination. In particular we used its Source Code Analysis (SCA) Engine to generate result files and further used the Audit Workbench GUI interface to examine the result. The SCA Engine claims to be able to find different types of vulnerabilities, including:

- Data Flow: find data possibly tainted by user input
- Control Flow: find exploitable control flow
- Structural: find exploitable structural design
- Semantic: find dangerous function and API calls
- Configuration: check configuration files
- Buffer: find buffer overflows

Users can fine-tune the engine by changing parameters and rules. In this assignment we sticked to the default settings. The SCA engine translates source code files into an intermediate model, on which the engine applies various rules to find vulnerabilities. The matches are written into the result files and then parsed and displayed by the GUI interface.

We found several design vulnerabilities, which could have been avoided if spotted during the development phrase. Specifically we found issues like a default empty password (Configuration issue), privacy violation (Structural issue) and weak encryption (Semantic issue). An advantage of static source code analysis is its applicability during the development phrase to provide useful feedback to the development team. However, since we conducted the static review after the software implementation was done, there is little we can do with its design.

In theory the static analyzer should be able to find vulnerabilities with low false negative rate, but in reality the state of art still falls short to such expectation. We later found plenty of vulnerabilities overlooked by the tool through manual inspection. However, the static analyzer still serves as a good starting point for deeper examination or at least to get an idea of the maximal quality one can expect.

The static analyzer also fails to correctly detect configuration issues. The default configuration file sets session timeout to 1 hour but during dynamic examination, it never expires (See 3.3.2). Such vulnerabilities are inherently difficult to find with static analysis tools on a dynamic language such as PHP.

We do not consider Fortify able to cope with configuration flaws, which is boasted as one of its key features.

Another shortcoming of Fortify is that often we still need to manually verify the reported vulnerabilities due to high false positive rate. Fortify does provide some arguments to support its findings but those are hard-coded examples used to illustrate the vulnerabilities rather than solid proofs for the security issue. The user may be able to learn the concept of a security issue from the examples but will for advanced developers the additional reporting is just noise and does not help in prioritizing analysis.

Fortify also reported some other vulnerabilities such as Cross Site Scripting, SQL Injection, and Cookie Stealing. We later applied a dynamic analyzer from OWASP called ZAP to verify Fortify's findings. Through actual exploitation ZAP also reported all those issues. We conclude that the static analyzer is surprisingly capable in finding vulnerabilities embedded in scripting language e.g. PHP and SQL.

Despite using both static and dynamic analyzer, we were still unable to verify most of the security requirements listed by OWASP. In the end we turned to manual inspection. Even though the software project is rather small, it still requires quite some time and effort to study its structure, and to verify the security issues. Some of the security requirements apply to the whole software project, so we would need to inspect each and every of the source file to verify it. Such full scale inspection is thus very time-consuming. Alternatively, for some of the security issues, we adopted the sampling method, with which we chose a few source files from the whole project and made our verdict based on them. Another difficulty is that the software is not documented at all. On the one hand it is a strength because we have to perform a reverse-engineering exercise, starting from considering all possible states, gradually shrinking to a smaller set of possibilities. On the other hand, it is also possible that issues are overlooked due to the information overload. We also noticed that the result of static review can be easily replicated while it is very difficult for dynamic review, which depends on the settings and system with which the software executes.

In summary, we used static analyzer Fortify to conduct source code analysis on an existing software project. We found some advantages:

1. Good in findings vulnerabilities of scripting languages
2. Serves as a good starting point for deeper examination

Disadvantages are also found:

1. still unable to tackle vulnerabilities that are intrigue in nature

2. plenty of false positives and false negatives, and can only claim a vulnerability exists but not able to prove it.

3. Generated reports are mechanical in nature and do not really give a judgement.

We however failed to experience the key advantage of static analyzers (at least in the form of Fortify and applied to a PHP application). Since the software development was already finished, we could not learn how valuable the feedback from static analyzers could be during the development phrase. We also noticed the limitation of static and dynamic analyzers, which both failed to examine the software throughly and generated plenty of false positives, and false negatives. Last but not the least, since static analyzer requires source code to work with, we argue that static analyzers are most useful for software developers but not other security specialists such as pentesters and hackers, for whom the source codes are rarely accessible.

A Initial analysis

We extracted `testcms-v2.zip` and categorized the files, as well as the number of lines for code, as follows:

```
$ find -type f | awk -F. '{print $NF}' | sort | uniq -c
      6 css
      6 gif
      1 gitignore
      9 js
     82 php
      2 png
      1 sql
      3 txt
$ find -type f -name '*.php' | xargs cat | wc -l
5975
$ find -type f -name '*.js' | xargs cat | wc -l
1037
$ find -type f -name '*.css' | xargs cat | wc -l
1564
$ find -type f | grep -vE '.(php|js|css)$' | wc -l
13
```

B Architecture

This section tries to document the results of a manual analysis of the application entry point and how it relates to other files.

B.1 Initialization

Related files:

- index.php
- system/bootstrap.php
- system/classes/*.php
- system/admin/controllers/*.php

All requests are first redirected to `index.php` which then includes `system/bootstrap.php`.

The bootstrap file includes some classes from `system/classes/`.

After registering the autoloader, PHP is configured to log all errors and not to show them to the user. This autoloader will search in:

- `system/admin/controllers/FOO.php` for classed named `FOO_controller`.
- `system/classes/FOO.php` otherwise.

Then `config.php` is loaded. If this fails, the literal contents of `system/admin/theme/error_config.php` is shown before the script terminates.

After that, the session is started (`Session::start` via `system/classes/session.php`),
the request is handled (`TestCMS::run` via `system/classes/testcms.php`),
the session is ended (`Session::end` via `system/classes/session.php`) and finally
the request is sent to the user (`Response::send` via `system/classes/response.php`).

B.2 Installation

Related files:

- install/installer.php
- install/diagnose.php
- install/index.php
- config.default.php

If the configuration file is missing, it points you to `./install` (`install/index.php`). No files are further included.

The `install/index.php` file does however mention manual modification of `config.default.php`.

B.3 Admin functions

Related files:

- system/classes/template.php
- system/functions/*.php

`system/classes/templates.php` includes a fixed list of files from `system/functions/` when a path does not contain `system/admin/theme`.

B.4 Themes

Related files:

- system/classes/testcms.php
- system/classes/template.php
- system/classes/themes.php
- themes/default/404.php
- themes/default/posts.php
- themes/default/functions.php
- themes/default/article.php
- themes/default/page.php
- themes/default/search.php

- themes/default/includes/comment_form.php
- themes/default/includes/footer.php
- themes/default/includes/header.php
- system/admin/theme/**.php

The base name of items inside the `themes/` folder are referenced in `install/index.php`. The theme path is set in `system/classes/testcms.php` using `system/classes/template.php`. This is `theme/(themename)` for some theme name or `system/admin/theme/` (when the first URL component equals to `admin`).

The Themes class (`system/classes/themes.php`) is used to parse about.txt files inside the themes folder.

B.5 Upgrade

Related files:

- upgrade/migrations.php
- upgrade/run.php
- upgrade/classes/migrations.php
- upgrade/classes/schema.php
- upgrade/index.php
- upgrade/complete.php

The upgrade folder is not referenced in any other file, but it does use files from `system/classes/` (autoloader, etc.).