SektionEins

Security Report

of Audit of ''Spryker Framework'' on behalf of Spryker Systems

For: Spryker Systems GmbH
- from here on ''Spryker Systems'' -
Julie-Wolfthorn-Straße 1
10115 Berlin

By: SektionEins GmbH
- from here on ''SektionEins'' -
Breite Straße 159
D-50667 Cologne
Germany

Author: Christian Horchert

Contents

Management Summary

Between February 9 and March 30 2016 SektionEins performed a source code audit of the Spryker Framework. The framework is implemented in PHP and consists of *Yves*, the shop front-end application, *Zed*, the back-end application for business logic and the *Rest API*.

During the audit only a few minor and medium rated problems could be identified in the framework itself. Issues found in the demoshop are only added as comments. The shop was tested at an unfinished state and all findings should be seen as recommendations for the use of the Spryker Frameworks. Some of the findings are possible problems in the future, not actual vulnerabilities.

The lack of common vulnerabilities such as SQL injection and the safe use of functions shows that the framework seems robust and secure. The vast majority of source code in this application is well structured and easy to read. Spryker uses a functionality to make sure that no problematic PHP functions are used which can easily extended to cover other probable issues. It is expected that all the issues laid out in this report can be resolved with very little effort.

The use of *assert()* and *unserialize()* should in general be avoided in productive code. This has been fixed by either using casts instead of assertions and JSON functions instead of PHP serialization. Some functions have been identified that use easily predicable strings rather than random data. These functions were used to build for example random strings for the password reset or the subscriber key.

In the session handling for Zed a problem could be identified that leads to a session fixation. The reason is that the token generated is static based on a supplied value by a user. In addition cookie flags should be set at least for all relevant session cookies.

Although not caused by the framework itself Cross-Site Scripting (XSS) has been found in an early version of the demoshop. As already stated this is not rated as a problem. However, it should be noted that proper handling of views for all cases (i.e. output for mail, HTML, JSON) with output is crucial for users of the framework to avoid any problems with malicious input.

For each security vulnerability a risk analysis has been performed that is documented in the risk tables throughout this report. A summary of the risk analysis is given in Appendix 1 including a recommended order of priority for fixes. The problems that are not caused by the framework itself and could be found during the live testing are marked with the LIVE prefix

General

This document describes the source code audit of the "Spryker Framework" that was performed by SektionEins between February 9 and March 30 2016 on behalf of Spryker Systems. All testing was done remotely from the SektionEins office. SektionEins was provided with the application's source code and documents to set up a local demoshop for dynamic testing.

The main goal of the audit was an evaluation of the framework in order to protect it from attackers with and without knowledge of the source code. Security relevant issues were identified in the code and - if possible - verified in the demo system.

The framework consists of roughly 250000 lines of PHP code, without comments or empty lines and external libraries.

1. Persons

The audit has been conducted by Christian Horchert and Stefan Horst. The report has been written by Christian Horchert.

2. Procedure

The audit consisted of the following parts:

- Source Code Audit
- Manual Verification
- Risk Assessment
- Report Generation

2.1. Source Code Audit and Manual Verification

The source code audit is the most important part of the auditing process. This audit mainly consisted of manually reading source code. The procedure was supported by the use of pattern scanning tools that search for specific patterns like the use of functions known to be potentially dangerous. Findings of these tools were marked and then evaluated manually piece by piece.

After this first step the structure of the source code was analyzed to get a better understanding of the different parts of the code. Data and parameter entry and exit points were identified and marked for later analysis. After the enumeration of all data entry points, they were inspected in a top-down approach to understand how incoming data is handled. This was followed by a bottom-up approach that investigates the data exit points to find out how the output is handled. In addition,

the code was searched for missing access control checks and security problems caused by logical flaws.

Aside from data flow guided source code analysis the code was also read line by line to find suspicious code blocks not found during the top-down search. Any such block found was evaluated in depth.

The audit was supplemented by penetration testing tools to test a demo shop built on top of the framework. For the dynamic tests we used the Burp[1] suite of tools.

[1] http://portswigger.net/

After completing the discovery phase of the audit a risk evaluation for all vulnerabilities found in the application is done. The risk evaluation is detailed and follows a DREAD schema. DREAD stands for Damage, Reproducibility, Exploitability, Affected Users and Discoverability. This schema is used to have an indicator which consists of the probability of an attack in relation to the potential harm.

A description of DREAD for risk evaluation can be found on the OWASP website:

`https://www.owasp.org/index.php/Threat_Risk_Modeling#DREAD`

The formula used in this report:
$Risik = (((D_1 + A)/2) + ((R + E + D_2)/3))/2$

where Damage Potential (D_1) and Affected Users (A) are related to Reproducibility (R), Exploitability (E) and Discoverability (D_2).

For every vulnerability an indicator is calculated based on the chance of an attack in relation to its potential damage. The result is the basis for the evaluation where all vulnerabilities up to 2 are comments and up to 4 are low, up to 6 are medium, up to 6 are high and up from 8 are rated critical.

The layout looks like the following table. Every number correlates to a value from 1 to 10 which rates one of the following topics.

1. Damage Potential
   Question: If a threat exploit occurs, how much damage will be caused?
   Rating:
   0 = Nothing. No damage
   5 = Individual user data is compromised or affected.
   10 = Complete system or data destruction

2. Reproducibility
   Question: How easy is it to reproduce the threat exploit?
   Rating:
   0 = Very hard or impossible, even for administrators of the application.
   5 = One or two steps required, may need to be an authorized user.
   10 = Just a web browser's address bar is sufficient, without authentication.

3. Exploitability
   Question: What is needed to exploit this threat?
   Rating:
   0 = Advanced programming and networking knowledge, with custom or advanced attack tools.
   5 = Malware exists on the Internet, or an exploit is easily performed, using available attack tools.
   10 = Just a web browser

4. Affected Users
   Question: How many users will be affected?
   Rating:
   0 = None
   5 = Some users, but not all
   10 = All users

5. Discoverability
   Question: How easy is it to discover this threat?
   Rating:
   0 = Very hard to impossible;
   requires source code or administrative access.
   5 = Can figure it out by guessing or by monitoring network traces.
   9 = Details of faults like this are already in the public domain
   and can be easily discovered using a search engine
   10 = The information is visible in the web browser address bar or in a form.

Beispiel-Risikobewertung mit Hilfe von DREAD						
D	R	E	A	D	RISK	
1	3	3	2	4	2,42	COMMENT LOW MEDIUM HIGH CRITICAL

Functional Analysis

During the source code audit a functional analysis of the source code was performed. It consists of a search for specific function calls and language specific patterns that can introduce security problems if used in a wrong way. These function calls are investigated piece by piece in order to test if any unfiltered input can enter the application. During the functional analysis the code was tested for the following classes of vulnerabilities:

- PHP Code Inclusions Vulnerabilities
- PHP Code Evaluations Vulnerabilities
- Use of cryptographic functions
- File system interfacing
- Functions with Callbacks
- Session Security
- Miscellaneous

1. PHP Code Inclusions Vulnerabilities

PHP Code Inclusion Vulnerabilities are injection vulnerabilities that allow to either influence a part or the full file name used in an include or require statement. Successfully exploited this class of vulnerabilities allows the execution of arbitrary PHP code from remote. The following statements were tested during the audit:

- `include()`
- `include_once()`
- `require()`
- `require_once()`

No vulnerabilities of this class were uncovered during this audit.

2. PHP Code Evaluations Vulnerabilities

Code Evaluation Vulnerabilities are injection vulnerabilities into the parameters of PHP functions or language constructs that allow dynamic PHP code evaluation. Successfully exploited this class of vulnerabilities allows the execution of arbitrary PHP code from remote. The following functions and statements were tested during the audit:

- `assert()`
- `eval()`
- `preg_replace()` with `/e` modifier
- `create_function()`
- `unserialize()`
- `exec()`
- `popen()`
- `proc_open()`
- `passthru()`
- `shell_exec()`
- `system()`
- `mail()`
- `pcntl_exec()`
- Backtick-Operator
- $$ meta variables

During this audit vulnerabilities related to `assert()` and `unserialize()` were uncovered and fixed during this audit.

2.1. assert()

If an expression is passed to *assert()* it is being evaluated at runtime. A problem can arise when `assert()` is used productive code and user input is being evaluated. Spryker used to use `assert()` in numerous locations, but all have either been replaced by casting variables or simply been removed.

EXEC01: assert()						
D	R	E	A	D	RISK	
7	3	2	6	4	4.75	**MEDIUM**

2.2. unserialize()

The function unserialize() was used for building the navigation and filters.

The PHP function `unserialize()` is known to be vulnerable to a number of logical and low-level attacks. User input must never be processed by `unserialize()` in order to prevent such attacks.

The functions calls in question were changed to use JSON instead of PHP serialization.

EXEC02: unserialize()						
D	R	E	A	D	RISK	
7	3	3	3	4	4.17	**MEDIUM**

3. Use of cryptographic functions

Cryptographic functions are used to encrypt or hash data or to produce strong randomness in order to protect data or communication or guard access to certain functions. The use of cryptographic functions has been carefully observed and some minor issues have been found.

3.1. Hash functions

The Spryker Framework uses cryptographic hash functions for security-related functions. Some of the functions are related to external APIs from third parties and can't be changed, but for data that is handled internally stronger algorithms than MD5 and SHA1 should be used, and the framework is already prepared to use stronger hash algorithms in the future.

For the token generation a hashed value based on the user name and password has been used which lead to a static token that would also cause trouble. The way it was implemented lead to different users with the same hash value.

CRYPTO1: Hash functions						
D	R	E	A	D	RISK	
4	4	2	2	5	3.33	**LOW**

3.2. uniqid()

During the audit problems related to `uniqid()` could be identified. It was used for example for the password reset function, customer and subscriber key generator and for the voucher and file writer. It was replaced by a cryptographically secure randomisation function.

`uniqid()` does not create random nor unpredictable strings. This function must not be used for any security-related purpose. A cryptographically secure random function /generator and a cryptographically secure hash function is needed to create unpredictably secure IDs.

CRYPTO2: uniqid()						
D	R	E	A	D	RISK	
4	4	2	2	4	3.17	**LOW**

3.3. rand()

`rand()` was used in some occasions where a cryptographic random function is more appropriate. This included for example the voucher. Spryker Framework now uses a function to ensure stronger random numbers.

CRYPT03: rand()						
D	R	E	A	D	RISK	
4	4	1	2	4	3.00	**LOW**

4. File system interfacing

The Spryker Framework needs read and write access to numerous directories on the server. Therefore it has been analyzed how the access is happening and what preconditions are necessary. One general problem could be identified: The permissions for directories and files written by the framework should be set according to the user and should not be world writeable or readable.

SYS01: File system interfacing						
D	R	E	A	D	RISK	
2	2	1	3	1	1.92	**COMMENT**

5. Functions with Callbacks

The Spryker Framework makes use of functions using callbacks and take other functions as input. One array sort function could be identified where the input needs to be casted as integer in order to avoid possible problems.

CALL01: Functions with Callbacks						
D	R	E	A	D	RISK	
2	2	1	3	1	1.92	**COMMENT**

6. Session Security

During the audit it was checked how the framework handles sessions. Specific checks included session manipulation, hijacking, session fixation, ID regeneration and general session handling.

6.1. Missing httpOnly flag

It was discovered that *zed_session* has set the httpOnly flag. This flag was first introduced by Internet Explorer and is meant to stop JavaScript from accessing a cookie. Nowadays this flag is supported by all modern browsers. Without this flag

a single XSS vulnerability is enough to read the session cookie and send it back to the attacker. The attacker can then try to use the session ID to take over a running session.

It is recommended to make sure that the PHP configuration setting ‘`session.cookie_httponly`‘ is activated.

SESS01: Missing httpOnly flag						
D	R	E	A	D	RISK	
2	4	1	2	7	3.00	**LOW**

6.2. Missing secure flag

The secure flag protects cookies from being transmitted by the browser without SSL encryption. In a production environment this flag must be set and SSL must be enforced.

SESS02: Missing secure flag						
D	R	E	A	D	RISK	
2	4	1	2	7	3.00	**LOW**

6.3. Session Fixation

It was possible to set the session ID for Zed cookie to a user defined value, e.g. `zed_session=XXX` which then yields a page containing a token with a static value which is the same for every same `zed_session` value. Should an attacker manage to set the session ID and token of another user before logging in, e.g. via XSS or other vulnerabilities, the attacker would be able to use the injected session ID to take over the session.

It is highly recommended to regenerate the session id with every change of authorization level, e.g. login or logout and have a token that is random, not predicable.

For the registration in Yves the cookie `vid` can be also fixated. This is a minor problem, but should also be addressed.

Also, it is good practice to allow only session IDs generated by the application. Starting with PHP 5.5.2 this can easily be set in php.ini with `session.use_strict_mode=1`.

SESS03: Session Fixation						
D	R	E	A	D	RISK	
5	6	2	2	7	4.25	**MEDIUM**

7. Miscellaneous

Findings found in this section are things that were marked for further research during the source code audit for checking and that don't fit in any of the other sections.

7.1. Cross-Site Request Forgery (CSRF)

The code was examined for working measures against CSRF. Cross-Site-Request-Forgery is an attack where the attacker is able to execute an action of a web application. This is done by exploiting the trust relationship (session) between a browser and the application. The attacker injects an HTTP request into an already running session of a user. The easiest example would be to load a an image from the attacker's website pointing to an action to be executed, e.g. `html <img src="http://target/action?id=1234"/>`

A common way to transfer specially prepared URLs is a URL shortening service. The attacker does not need to have direct access to the web application, meaning that applications behind firewalls or in private networks or VPNs can be attacked this way.

The application has protection against CSRF for some forms by using a random token. Some functionality in the dynamic test was not properly protected against CSRF and it looks like it is caused by tokens that use static values. Applications build on top of the Spryker Framework need to be checked if all requests that change data sets are protected by CSRF tokens. It is highly recommended to add a suitable CSRF protection for all writing actions, e.g. database changes.

CSRF01: Cross-Site Request Forgery (CSRF)						
D	R	E	A	D	RISK	
2	3	2	3	5	2.92	**LOW**

The application appears to trust the user-supplied host header. By supplying a malicious host header with a password reset request, it may be possible to generate a poisoned password reset link. Consider testing the host header for classic server-side injection vulnerabilities. Depending on the configuration of the server and any intervening caching devices, it may also be possible to use this for cache poisoning attacks.

It is recommended either to make sure that the framework itself makes sure that no arbitrary host headers are accepted or add documentation how to avoid this problem in applications built on top of the Spryker Framework.

HOST02: Host header poisoning						
D	R	E	A	D	RISK	
2	3	2	3	6	3.08	**LOW**

Demoshop

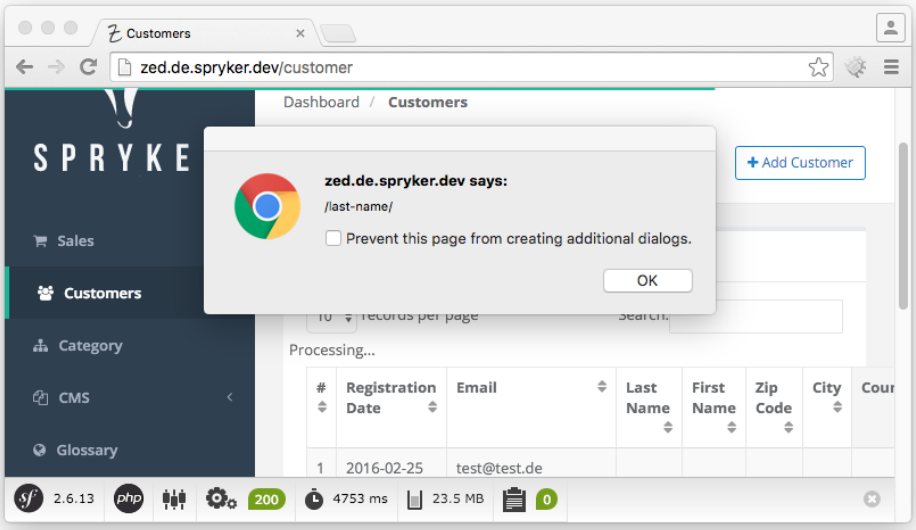
This section describes vulnerabilities found in the demoshop. Most of the things reported in this part don't have to do with the framework itself. But these observations should be helpful to understand how to further secure shops built on top of the Spryker Framework.

1. Cross-Site Scripting (XSS)

During the test of the demoshop Cross-Site Scripting (XSS) in Zed has been found. The reason for the XSS is not framework-related and they are caused by missing encoding to render XSS payloads useless.

The Spryker Framework uses Twig as the template engine. The framework doesn't do much input filtering, so properly dealing with output is essential. It basically relies on proper handling of views for all cases (i.e. output for mail, HTML, JSON).

The most dangerous example could be done by a user in the frontend by crafting a malicious payload in the name and the payload got rendered in the admin interface in Zed.



LIVE01: Cross-Site Scripting (XSS)						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

2. Security-related Headers

This section contains recommendations for the use of certain security headers. The idea of these headers is to force a certain behavior of browsers. These headers were all missing in the demoshop, but these headers should be considered to be used by online shop systems.

2.1. Strict Transport Security

The HTTP Strict Transport Security policy defines a timeframe where a browser must connect to the web server via HTTPS. Without a Strict Transport Security policy the web application may be vulnerable against several attacks:

If the web application mixes usage of HTTP and HTTPS, an attacker can manipulate pages in the unsecured area of the application or change redirection targets in a manner that the switch to the secured page is not performed or done in a manner, that the attacker remains between client and server.

If there is no HTTP server, an attacker in the same network could simulate a HTTP server and motivate the user to click on a prepared URL by a social engineering attack. The protection is effective only for the given amount of time. Multiple occurrencea of this header could cause undefined behaviour in browsers and should be avoided.

A Strict-Transport-Security HTTP header should be sent with each HTTPS response. The syntax is as follows:

```
Strict-Transport-Security:  max-age=<seconds>[; includeSubDomains]
```

LIVE03: Strict Transport Security						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

2.2. Clickjacking (Frameable Response)

An attacker may be able to load the application within an iframe and control the actions of a user e.g. by showing a CSS overlay. This technique is used to circumvent CSRF protection.

In order to prevent Clickjacking, the HTTP header `X-Frame-Options` should be set to `DENY` or `SAMEORIGIN`. In addition the application should include JavaScript code to detect frames.

LIVE04: Clickjacking (Frameable Response)						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

2.3. Disable Content Sniffing

The `X-Content-Type-Option` HTTP header with the value `nosniff` should be set in the response. The lack of this header causes that certain browsers, try to determine the content type and encoding of the response even when these properties are defined correctly. This can make the web application vulnerable against Cross-Site Scripting (XSS) attacks. E.g. the Internet Explorer and Safari treat responses with the content type text/plain as HTML, if they contain HTML tags.

The following HTTP header should be set at least in all responses which contain user input:

```
X-Content-Type-Options: nosniff
```

LIVE05: Disable Content Sniffing						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

2.4. XSS protection header

The `X-XSS-Protection` HTTP header changes the behavior of the XSS filter in browsers. These filters check if the URL contains possible harmful XSS payloads and if they are reflected in the response page. If such a condition is recognized, the injected code is changed in a way, that it is not executed anymore to prevent a successful XSS attack. The downside of these filters is, that the browser has no possibility to distinguish between code fragments which were reflected by a vulnerable web application in an XSS attack and these which are already present on the page. In the past, these filters were used by attackers to deactivate JavaScript code on the attacked web page. Sometimes the XSS filters itself are vulnerable in a way, that web applications which were protected properly against XSS attacks became vulnerable under certain conditions.

It is considered best practice to instruct the browser XSS filter to never render the web page if an XSS attack is detected.

The following header should be set:

```
X-XSS-Protection: 1; mode=block
```

LIVE06: XSS protection header						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

3. Miscellaneous

3.1. Password field with autocomplete

The password fields do not have the `autocomplete='off'` attribute set. Modern web browsers may try to store the password and insert it automatically where appropriate.

If this is not the expected behavior, this attribute should be set accordingly.

LIVE07: Password field with autocomplete						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

3.2. Local name leakage

An applications that was build on top on the framework has been found through Shodan[1] that leaked the name of the development environment through the cookie.

LIVE08: Local name leakage						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

3.3. Bruteforcing user names

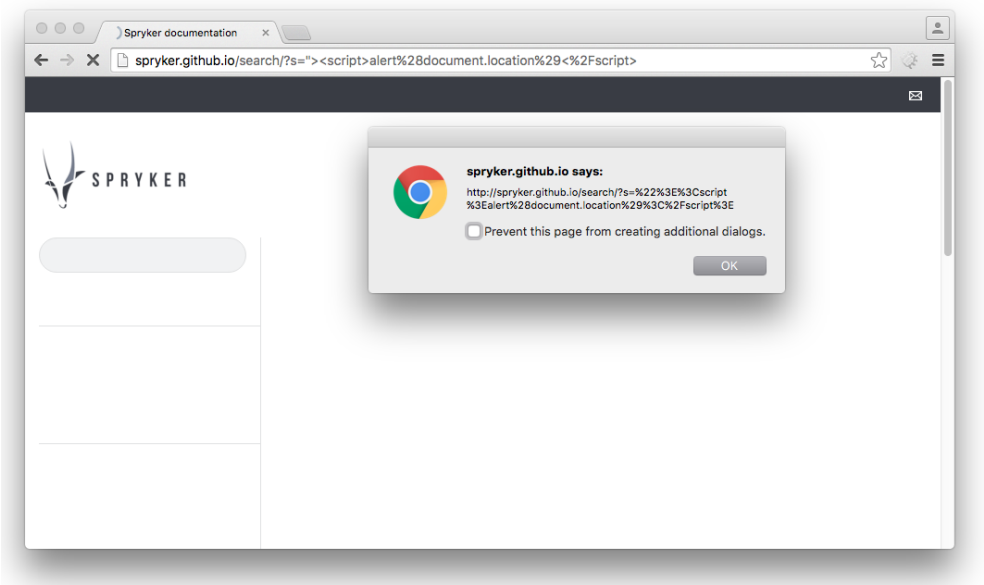
The response of the application indicates if a user is present on the system or not, based on the email address. If a login or password reset is done the response should rather contain a very general note that either user or password are not correct and not a note that a user does not exist or the password for this specific user is wrong.

LIVE09: Bruteforcing user names						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

3.4. XSS in documentation

The documentation found under http://spryker.github.io contains an XSS in the search. Although the impact is minimal and not related at all to the framework it should be fixed.

[1] http://shodan.io/



LIVE10: XSS in documentation						
D	R	E	A	D	RISK	
0	0	0	0	0	0.00	**COMMENT**

Other Tests

Apart from issues and recommendations previously mentioned, a number of other tests have been performed that did not lead to discovering additional security issues:

- Logical Errors, e.g. Step Confusion
- Other Injection Vulnerabilities, e.g. HTTP-Headers, Email, Unicode
- HTTP Response Splitting
- URL Redirection
- Format Strings
- Denial-of-Service, e.g. Crash, Resource Exhaustion, Cookie Poisoning
- Incorrect Filters, e.g. incorrect regex, wrong filter type
- Timing-Attacks
- PHP Configuration Errors
- Password Security: Input, Processing, Transfer, Storage
- Inter-Application Exploits, e.g. Session-ID-Sharing
- SQL injections

Conclusion

During the audit only minor and medium rated problems could be identified. Some of the findings contain fixes for possible problems in the future.

The use of assert() and unserialize() should in general be avoided in productive code. These functions evaluate code which can lead to code execution. Attacking this part of the code is rather complex, but it should still be addressed. It has already been fixed by either using casts instead of assertions and JSON functions instead of PHP serialization.

Some functions have been identified that use easily predicable strings rather than random data. These functions were used to build random strings for password reset or the subscriber key, based on milliseconds. Also hash functions are used in a way that can be attacked or cause problems, i.e. by having strings concatenated that produce the same output). These problems have been fixed by using cryptographic equivalents.

In the session handling for Zed a problem could be identified that leads to a session fixation. The reason is that the token generated is static based on a supplied value by a user. It is crucial that the application only accepts session IDs that are generated by the framework or being otherwise protected against manipulation.

The token generation for the CSRF protection should be revisited to assure that the tokens are unpredictable and renew with new sessions. Only known and valid host headers should be accepted in requests or otherwise ignored or set to a default value.

Although not caused by the framework itself some Cross-Site Scripting (XSS) in Zed have been found. However, it should be noted that proper handling of views for all cases (i.e. output for mail, HTML, JSON) with output is crucial for users of the framework to avoid any problems with XSS.

As a conclusion it can be said that the development of the “Spryker Framework” has had a particular focus flexibility but also on security. The lack of common vulnerabilities such as SQL injection and overall safe use of functions supports that claim. Spryker uses a functionality to make sure that no problematic PHP functions are used.

The vast majority of source code in this application is well structured and easy to read. It is expected that all the issues laid out in this report can be resolved with very little effort.

Appendix 1: Recommended Prioritization

The risk evaluation tables in the previous chapters are an indicator for the recommended order to fix the vulnerabilities found. All vulnerabilities not covered by a risk evaluation table are below a certain risk level and fixing them can therefore be scheduled at the end of the fixing process.

The following table lists the page numbers of vulnerabilities that were discovered during the audit or penetration test and documents the order in which they should be fixed according to our recommendation.

In general the recommendation is to fix vulnerabilities in the order of their risk level. Vulnerabilities on the same risk level should be fixed according to their potential damage impact.

No critical vulnerability could be identified

No high risk vulnerability could be identified

Page 13	SESS03: Session Fixation
Page 10	EXEC02: unserialize()
Page 10	EXEC01: assert()

Page 13	SESS02: Missing secure flag
Page 13	SESS01: Missing httpOnly flag
Page 15	HOST02: Host header poisoning
Page 14	CSRF01: Cross-Site Request Forgery (CSRF)
Page 11	CRYPTO2: uniqid()
Page 11	CRYPTO1: Hash functions
Page 12	CRYPT03: rand()

All remaining vulnerabilities without a separate risk evaluation table should be fixed in an arbitrary order after the vulnerabilities above have been fixed.