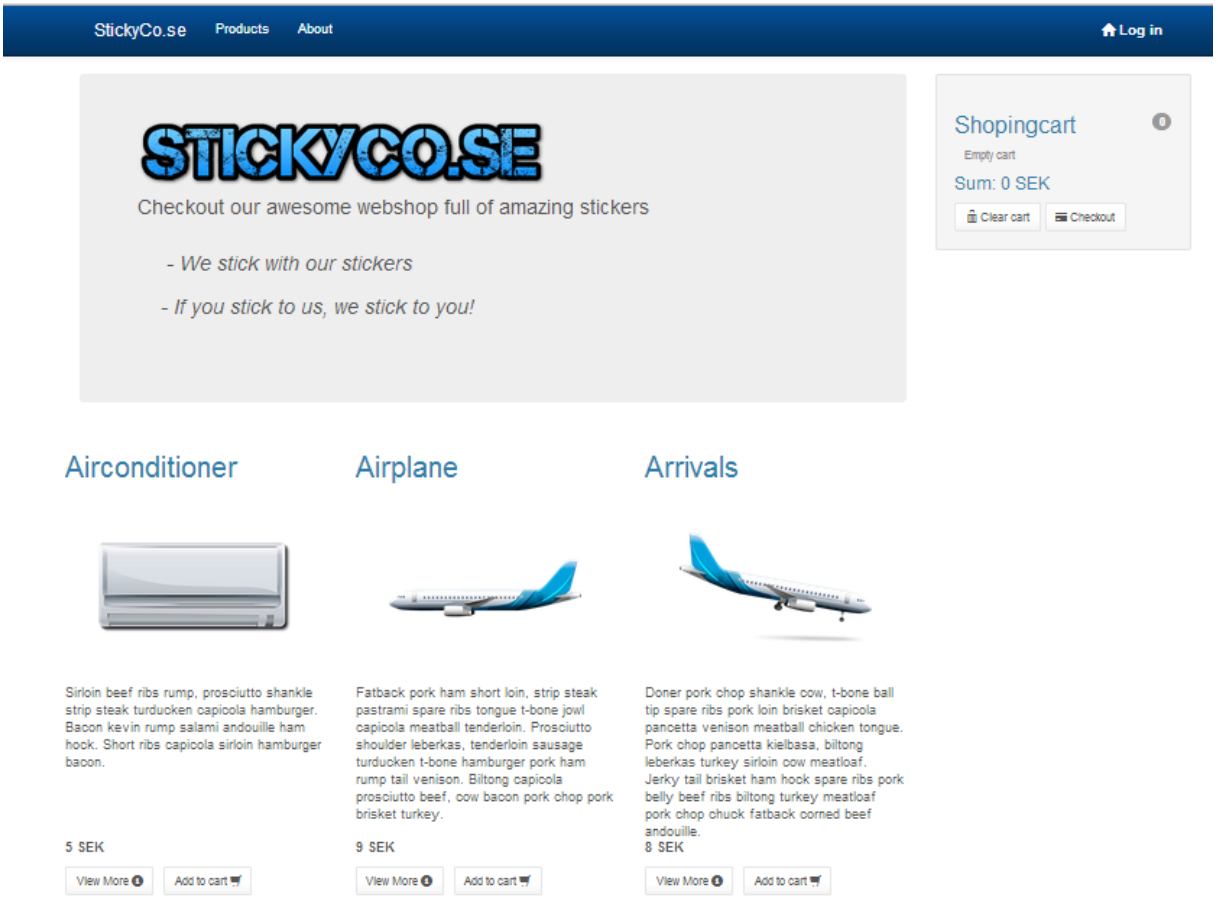
ETIF05 Web Security - A web shop in PHP

Filip Lindqvist & Jakob Berglund
ada10fl2, ada10jbe

October 2013

Demo: http://www.stickyco.se



1 Passwords

When storing passwords in a database or any store for that matter, it is important to not store these in a plaintext since these can be stolen in a attack. The common solution to this is using one-way encryption also known as cryptographic hashing. By hasing the password before storing them one can check the correctness of a password without having the store it in plaintext, since $h(correctpassword) = h(input)$ if $input$ is actually the correct password. The $h(x)$ function are created such that they are inreversible and satify the following conditions:

- Given a hash x it should be very hard to find a m such that $x = h(m)$. (Pre-image resistance)
- Given an specific input i_1 it should be very hard to find another input i_2 such that $i_1 \neq i_2$ such that $h(i_1) = h(i_2)$. (Second pre-image resistance)
- Nothing given it should be very hard to find two different values x_1 and x_2 such that $h(x_1) = h(x_2)$. (Collision resistance)

Examples of these types of hash functions is MD5 and the SHA-family. However MD5 is practically cryptographically broken and unsuitable for usage when storing passwords securely. Also SHA1 is not really considered secure anymore, since there is a few attacks possible to achive collisions in resonable amount of time. SHA1 has sucessors known as the SHA2 and SHA3 that are considered secure. However there is a simple attack on this implemention that is called Time-Memory Trade-Off or Rainbow Table attack. These uses the fact that one can simply compute all different hashes of possible passwords on beforehand and store them locally. Then one can simply reverse lookup the stored hash value to the matching plaintext. The solution to this problem is using a random string also known as a salt, that is concatenated to the password before hasing. This invalidates all pre-calculated values in the mentioned attacks even though the salt is known to the attacker. The best practice of this approach is to use new uniqe salt for each password, making it completely impossible to use pre-calculated tables.

Also one can argue that MD5 and the SHA-family is still bad for hashing of passwords since they are designed to be fast for hashing both small and large datasets. This is and advantage in data processing but a disadvantage in security since they still are vulnerable to brute-force attacks. Today everyone can buy a pair of graphic cards and setup a own cluster to crack passwords within a small budget. With a single consumer medium-end GPU one can compute approximatly 500-600M passwords per second using MD5, respectivley 150-250M passwords per second using SHA1.[1] In conclusion you can break a lowercase alphanumrical password of length 6 in under 10 minutes. Moreover they have a fixed number of computations that in union with Moore's Law will compromise the algorithm as time passes. This leads us to the conslusion that

[1] My results, using http://www.golubev.com/hashgpu.htm

'fast' hashing algoritms should not be used to store passwords. In salvation enters a new group of adaptive hashing algoritms using key derivation functions (KDF) that introduces a work factor to keep up with More's law. A wideley accepted algoritm that uses this is `bcrypt` that as of PHP 5.5 is encorperated in the new password API. This API is both clever, secure and easy to use. It introduces `password_hash()` that creates a single string for database storage containing both the algorithm identifier, the work factor, the salt and the hashed password. As of today `password_hash` uses `bcrypt` with the workfactor of 10, but it also gives developers options to specify the exact algoritm and the work factor. We have manually specified the `bcrypt` algorithm and a work factor of that will change as the time passes. We created the formula $w(year) = 12 * 1.2^{(year-13)}$ for the work factor that may be valid for a few years from now. A another approach may be to let the PHP use its own default values that can be changed by the developers as new versions are released. This will make sure that it will use an algorithm that is secure and updated according to the latest knowledge. On the other hand this requires the PHP to be constantly keept up to date, which hopefully is the case. In addition the API also let the passwords to be lazily rehashed as the workfactor is changed, thanks to `password_needs_rehash()` and the metadata keept in the string.

In addition to the usage of random salts we have also added what we call a code salt. This is 256 bit random salt that is stored in the source code of the webpage and is concatinated with each password. This ensures that a attacker will need both the data from the database and the source code in order to offline brute-force the true password.

2 Cross-Site Scripting

A common attack on the web today is to inject javascript on a page and get this to execute in the users browsers. This can be used to execute Denial of Sevice attacks and information stealing. This is manily caused by bad sanitizing in user input and encoding of output. When developing code it is critical to assuem that all input values in the GET, POST and REQUEST are harmful. They can be changed to illegal values by an attacker building a custom browser sending request at will, thus one can not trust these values. However one must also be careful when writing values into a page from e.g. a database. In this case we must assume that the data is evil since the database may contain dirty data from an another part of the website.

3 Apache and PHP Settings

One of the most important things in building a secure website is making sure that one has a robust configuration of the server. The Apache configuration in controlled by the file httpd.conf that states the different settings used by Apache when serving pages.

3.1 httpd.conf

- `ServerTokens Prod`
  - Ensure that no relevant server version is leaked by Apache.
- `ServerSignature Off`
  - Suppresses the version info in server generated pages.
- `ErrorDocument XXX error`
  - We provide a custom error page for HTTP errors.
- `RewriteEngine On`
  `RewriteCond %{REQUEST_FILENAME} !-d`
  `RewriteCond %{REQUEST_FILENAME}php -f`
  `RewriteRule ^(.*)$ $1.php`
  Rewrites urls without .php to read the actual .php-file.
- `RewriteCond %{SERVER_PORT} !443`
  `RewriteRule .* https://%{HTTP_HOST}%{REQUEST_URI}`
  Force all traffic go use SSL port 443.
- `SSLEngine on`
  - Enable SSL Engine.
- `SSLCertificateFile "conf/ssl.crt/ssl.crt"`
  - Server certificate for transfer to the client
- `SSLCertificateKeyFile "conf/ssl.key/ssl.key"`
  - Private server certificate key
- `SSLProtocol -ALL +SSLv3 +TLSv1`
  - Force SSL version 3 or TLS version 1.
- `SSLCipherSuite -ALL:TLS_RSA_WITH_AES_256_CBC_SHA:HIGH:!aNULL:!MD5`
  - Enable only the stongest ciphers, e.g. disable MD5 MACs.
- `SSLHonorCipherOrder on`
  - Let the server decide what cipher to use. Thus selecting TLS_RSA_WITH_AES_256_CBC_SHA if it is available.

3.2 php.ini

- `max_execution_time=10`
  - Maximum execution time of each script, in seconds.
- `max_input_time=10`
  - Maximum amount of time each script may spend parsing request data.
- `max_input_nesting_level=32`
  - Maximum amount of nested objects.

- `memory_limit=32M`
  - Maximum amount of memory a script may consume (32MB).
- `post_max_size=1M`
  - Maximum size of POST data that PHP will accept, avoid DoS.
- `file_uploads=Off`
  - Deny HTTP file uploads. We do not use this in your webshop.
- `expose_php=Off`
  - Remove the `X-Powered-By` HTTP header
- `safe_mode=Off`
  - Safe mode is deprecated in PHP 5.5+ due to the false sense of security that was easily breached.
- `disable_functions=`
  `fsocket_open,exec,passthru,···,popen`
  - Disable all dangerous functions that have security issues a total of 88 functions.
- `disable_classes=`
  `splfileobject`
  - Disable dangerous classes that have security issues
- `enable_dl=Off`
  - Disable loading of dynamic extensions.
- `extension_dir = ···`
  `extension=php_pdo_mysql.dll`
  - To get better control we remove all unused extensions
- `error_reporting=E_ALL & ~E_DEPRECATED & ~E_STRICT`
  - Log all errors
- `display_errors=Off`
  - Do not display errors, this can leak information
- `display_startup_errors=Off`
  - Do not display errors, this can leak information
- `track_errors=On`
  - We still want a stacktrace
- `log_errors=On`
  - Log all errors that are encountered
- `log_errors_max_len=32768`
  - Keep a long log, this is a resource in security analysis and threat detection

- `register_globals=Off`
  - Disable the embarassing option to register all varaiables in a single global one.
- `register_long_arrays=Off`
  - Disable the deprecated long $HTTP_*_VARS predefined variables
- `register_argc_argv=Off`
  - Disable the register of $argv & $argc variables
- `auto_globals_jit=On`
  - Lazy load ENV, REQUEST and SERVER
- `magic_quotes_gpc=Off`
  `magic_quotes_runtime=Off`
  `magic_quotes_sybase=Off`
  - Disable Magic quotes are a preprocessing feature of PHP, it is deprecated.
- `default_mimetype=''text/html''`
  - Default type of data that the PHP produces.
- `default_charset = ''UTF-8''`
  - Use UTF-8 as character encoding.
- `allow_url_fopen=Off`
  - Disallow PHP to treat URLs (like http:// or ftp://) as files.
- `allow_url_include=Off`
  - Likewise for the include/require commands.
- `default_socket_timeout=1`
  - Default timeout for socket based streams (seconds).
- `session.save_handler = files`
  - Store Sessions in files, to increase the security this can be handled by the database, where one can set more fine-grained permissons.
- `session.save_path = ···`
  - This path should be only be readable by Apache user only.
- `session.use_cookies=1`
  `session.use_only_cookies=1`
  - Use and use only session cookies to store SID.
- `session.cookie_secure=On`
  - Force HTTPS for cookie transfer.
- `session.cookie_httponly=On`
  - Don't make session cookies reachable from javascript.

- `session.hash_function=7`
  - Hash algorithm used to generate the session IDs, 7=sha512 in hash_algos().
- `session.name="SIDC3195C68B6D5F13E40"`
  - Set a different name for the PHP session cookies, makes it harder for an attacker to use pre-built attacks
- `session.entropy_length=512`
  - How many random bytes to read from the source
- `session.use_strict_mod=1`
  - Force regeneration of uninitialized session IDs. Protects from session fixation by session adoption
- `session.use_trans_sid=Off`
  - Make sure PHP dosen't write PHPSESSID in GET-query
- `open_basedir=wwwpath,settingsstorage`
  - Allow "www-path"' where all PHP code is located and also allow the "settings-storage" files which we store in a separate path.

4 Apache security

It is important to run the Apache server under a non-admin user to prevent attackers from running code as a high privileged user. Furthermore we want the restrict the paths that PHP can access on the filesystem. By using the `open_basedir` we only allow acces to the PHP source files and a second directory where we store the database credentials and the PHP session data. These folders should only be readable by the restricted non-admin user. Since we use files to store the sessions we have less control of the access to this files. A better approach would be to store the sessions the database where we can fine-grain the access control, also giving us only one point to defend which is good from a security and safety perspective. Also this is performance wise better.

5 PHP Sessions

To prevent session hijacking we firstly use HTTPS which makes it harder for an attacker to eavesdrop traffic and sidejack sessions. In addition we force all sessions cookies to transfer over HTTPS. Secondly we set the HttpOnly flag on the session cookie to ensure that it is not possible to reach it from javascript. This prevents XSS attacks where a javascript is injected into a site to fetch the session id. However not all browsers implement this flag so it is not a complete protection against these attacks. Moreover we disable the ability to set the session id through GET parameters, which prevents attackers from sending Alice a link that changes her session id. Furthermore we read 512 bytes of random data to generate and SHA512 the session id. Moreover we force uninitialized

session ids to be regenerated before they reach the PHP code. Finally we watch sessions to make sure that the user has the same "UserAgent" and "IP-address" during the entire session, else we close the session thereby forcing a user logout. Also we keep track of the users activity and let the login sessions timeout after 30 minutes if no activity are detected. Altough this approach has impact on the user experience, however this is not relevant in the topic of this context.

6 SQL injections

To avoid these attacks we use PHP Data Objects that is in a sense prepared statments much like `MySQLi` but build to be used with many common databases not just MySQL. It precompiles and escapes all queries before they are executed. Not only is faster to let statements compile and be cached, but the escaping prevents injections of bad strings. PDO prevents what is called first order injections but not injections of second order injections. First order injections means that an attacker gets a result in a single query. Second order means that injections first are put into the database and explotied in a later query by succeding to request the injections string. If one uses prepared statments everywhere one is also secure against the second order. If one uses prepared statments in one query but not in others a second order attack is possible. In a sense first order injection is forgetting that input data may be dirty and second order injection is forgetting that database may be dirty as well. In our experience it all boils down to enforcing input restictions and proper data validation before interacting with the database. Thus it is good to enforce a abstraction pattern that creates a singel entry point to the database or the tables, which has the responsibilty to validate and clean dirty input before it reaches the database. We have solved this by putting all database code in a single PHP class that is instantiated and called to exeute queries. In addition we have also enforce a single class for validating user data, with strict RegEx patterns for each input types. This is then called from both the PHP page frontend to provide user feedback and and the PHP database abstraction class to ensure proper input, in the case of a less careful developer.

7 Resources

The webshop is build on three javascript frameworks used to simplify and beautify the page.

- Bootstrap 3, a sleek, intuitive, and powerful mobile first front-end framework for faster and easier web development.
  http://getbootstrap.com/
- JsRender, a library to render JSON objects into HTML.
  http://www.jsviews.com/

- JQuery, a fast, small, and feature-rich JavaScript library. http://jquery.com/

We have used free images from these sources in our webshop. The texts was generated using http://baconipsum.com/api.

- http://dryicons.com/free-icons/preview/travel-and-tourism-part-1/
- http://dryicons.com/free-icons/preview/travel-and-tourism-part-2/
- http://dryicons.com/free-icons/preview/christmas-icons-set/

We have used the PHP Security Cheat Sheet from Open Web Application Security Project (OWASP) and Symantec to strengthen your configuration and code.

- https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet
- http://www.symantec.com/connect/articles/securingphpstepstep
- http://www.symantec.com/connect/articles/securingapachestepstep

Furthermore we have read a few blog posts expliaining the vurnabilities and security issues with PHP and password hashing.

- http://me.veekun.com/blog/2012/04/09/phpafractalofbaddesign/
- http://codahale.com/howtosafelystoreapassword/
- http://web.securityinnovation.com/appsec-weekly/blog/bid/79067/Disable-DangerousFunctionsinPHP