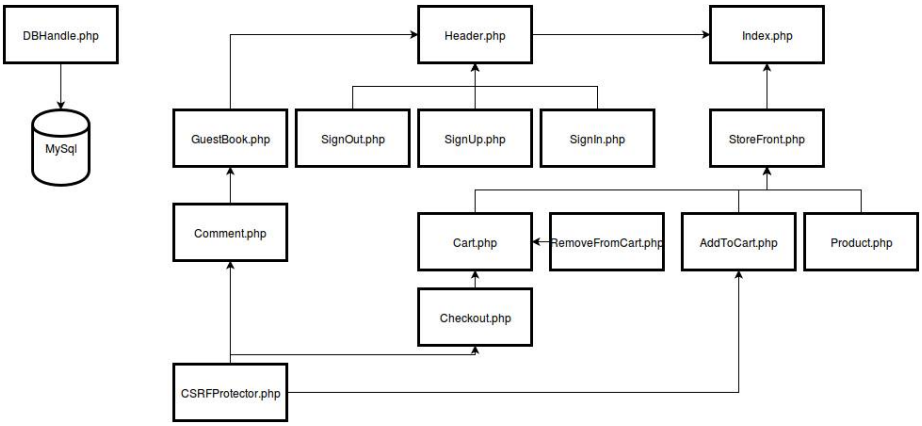
Project in Web Security, EITF05

Joakim Brorsson	b.joakim@gmail.com
Martin Gunarrson	dat11mgu@student.lu.se
Magnus Sunesson	htx12msu@student.lu.se
Amar Vrbanjac	dic13avr@student.lu.se

1 Architectural overview

1.1 Illustration



1.2 Overview

We have chosen to use Apache, MySQL, HTML and PHP in our implementation of our web shop.

- `Index.php` shows `Header.php` and `StoreFront.php`
- `Header.php` contains the forms `SignUp.php` and `SignIn.php`
  - `SignUp.php` shows the form for signing up in the header
  - `SignIn.php` shows the form for signing in in the header
- `StoreFront.php` shows the available products from `Products.php`, it also gives the ability to add a product to the cart
  - `Product.php` is a class describing a product
    - `RemoveFromCart.php`
    - `Checkout.php`
  - `AddToCart.php` adds a product to the cart and relays the user to `Cart.php`
    - `Cart.php` shows products in the cart and gives the ability to remove an product from the cart, go to the `index.php` or to checkout
      - `RemoveFromCart.php` removes an item from the cart
      - `Checkout.php` shows the receipt
- `Guestbook.php` contains a function for retrieving comments put into the database, it also shows the comments on the webpage
  - `Comment.php` is a class describing a comment

- `DBHandle.php` contains all methods used for accessing, retrieving, inserting and deleting data in the database
- `CSRFProtector.php` authenticates `HTTP GET` when the user submits a comment, adds a product to the cart or checkouts from the cart

2 Security considerations

2.1 Eavesdropping

2.1.1 Physical

Eavesdropping is when an attacker is looking over the victims shoulder to eavesdrop the password. In our implementation the risk of a successful attack of this kind is reduced by covering the input password with asterisks. This however does not protect against eavesdropping on the physical keyboard, which can be filmed and thus giving the password.

2.1.2 Digital

Session sniffing is when the traffic between user and server is being eavesdropped by an attacker. Since `HTTP` request are sent unencrypted, cookies can be read and sessions stolen. [1] By using `SSL` in our implementation, and thereby encrypting the traffic between server and user, it is not world-readable.

2.2 Man-In-The-Middle Attack (MITM)

This attack is when the attacker is acting as a, as the name implies, man-in-the-middle between two endpoints A and B. A and B think they are communicating directly to each other but the attacker is actually relaying the data, meaning it has access to the information sent. The website we have built use SSL, which when used correctly protects against man-in-the-middle. This is due to the fact that SSL make use of certificates and certificate authorities, CA. The data sent is encrypted, and the attacker will not be able to decrypt it without a valid certificate, signed by a certificate authority.

2.3 SQL-Injections

In an `SQL-injection`, the attacker makes his attack by injecting queries to a website, i.e. in a comment field and receive data from the database. There is a PDO module implemented in PHP 5 and above, that makes it possible to use prepared statements. Which is protection against `SQL-injection`, as it properly makes use of parameterized queries. This is also what we have implemented in our web page.

[1] http://www.eit.lth.se/fileadmin/eit/courses/eitf05/lecture_notes/LN_2015_webbappsec.pdf

2.4 Cross-Site Request Forgery (CSRF)

In CSRF the attacker has the user perform actions on a website already authenticated to. [2] Our protection is as follows; first check if the session is active, if not active abort. Every time the client makes a GET-request, i.e. downloads the page, a hash is created that the server sends to the client. The hash is created by combining session ID, a timestamp and a secret. Then, when the client makes an input to the server, this hash is sent along with the data to the server. The server keeps track of valid hashes, and as such it will only accept input from the user mapped to the hash of any message. An attacker will not be able to recreate it, as it does not know the secret.

2.5 Cross-site scripting (XSS)

In XSS, the attacker injects (malicious) code/scripts to a server for the purpose of, for example, stealing cookies. It is done via a browser. [3] Our protection against this is to filter all input, making some special characters "unusable" which limits which kind of input is possible, making it harder to inject code.

2.6 Predictable cookies

If the cookie use a predictable parameter, such as `username`, it is predictable and attackers can create one themselves. A protection against this is to simply not use predictable parameters. We use randomly generated session-ID:s instead, which is unpredictable. As such, the attacker can't make its own cookies as easily.

2.7 Path traversion

Path traversion: Access files by making requests to server. For example, a user asking for a file in path `/.../etc/passwd` can work if the server does not validate the input, giving the user access to `etc/passwd`. [4] Apache. We never use user input as a file path, in fact we never use system calls to access files. All handling of files is done through GET. We have some folders in the document root. These folders have however had their access restricted. Apache will prevent any user of the site to access the protected folders and files.

2.8 Dictionary, brute force and rainbow-table attacks

The brute force attack tries all combinations of passwords within a set of possible characters. Whereas the dictionary attack uses a list with predefined pass-

[2] http://www.eit.lth.se/fileadmin/eit/courses/eitf05/protected/L3B_2_per_page.pdf
[3] http://www.eit.lth.se/fileadmin/eit/courses/eitf05/lecture_notes/LN_2015_webbappsec.pdf
[4] http://www.eit.lth.se/fileadmin/eit/courses/eitf05/lecture_notes/LN_2015_webbappsec.pdf

words, i.e. the most common passwords, and traverses this list to try and find a match.

2.8.1 Online attacks

In our implementation we have chosen to lock the users account for 30 minutes after three failed login attempts. Thus making a online brute force attack very time consuming and would also be of high cost for the attacker.

An alternative solution to this could be to prompt a `CAPTCHA` to the user instead of the 30 minute lock.

2.8.2 Offline attacks

Our implementation uses the `BCrypt` function where we choose amount of iterations to hash the password, it also uses unique salt for each stored password.

Assumed that our user database can be stolen, neither a dictionary, brute force or rainbow-table attack would be applicable. In an offline attack the attacker would need to guess and precompute hashes of every possible password and any possible salt, `hash(password+salt)`. With a unique and large salt for every password it is infeasible to precompute.

By having an high iteration amount for the `BCrypt` function the cost becomes high for the attacker, who would need to use great resources to precompute the necessary amount of hashes. This also applies for the brute force attack, where the attacker would need to `hash(password+salt)` in real-time.

2.9 Denial-of-Service, DoS

Our website does not provide any protection against Denial-of-Service attacks. This is an attack in which the attacker makes it hard for users to access the service under attack, for example by overloading the server. Protection against DoS-attacks usually consists in use of firewalls and similar rather than implementations in the code. As such, we have not made any protection against it in this website.