Web Security

André Restivo

Index

Introduction

Attacks and Vulnerabilities

- A **vulnerability** is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application.
- **Attacks** are the techniques that attackers use to exploit the vulnerabilities in applications.

Reference: Open Web Application Security Project

OWASP Top 10 (2013)

- Injection
- Broken Authentication and Session Management
- Cross-Site Scripting (XSS)
- Insecure Direct Object References
- Security Misconfiguration
- Sensitive Data Exposure
- Missing Function Level Access Control
- Cross-Site Request Forgery (CSRF)
- Using Components with Known Vulnerabilities
- Unvalidated Redirects and Forwards

OWASP Top 10 - 2013

OWASP Top 10 (2017)

- Injection
- Broken Authentication
- Sensitive Data Exposure
- XML External Entities
- Broken Access Control
- Security Misconfiguration
- Cross-Site Scripting (XSS)
- Insecure Deserialization
- Using Components with Known Vulnerabilities
- Insufficient Logging & Monitoring

OWASP Top 10 - 2017

OWASP Top 10 (2021)

- Broken Access Control
  including **CSRF**
- Cryptographic Failures
  including **no HTTPS**
- Injection
  including **SQL Injection** and **XSS**
- Insecure Design
- Security Misconfiguration
- Vulnerable and Outdated Components
- Identification and Authentication Failures
  including bad **password management**
- Software and Data Integrity Failures
- Security Logging and Monitoring Failures
- Server-Side Request Forgery (SSRF)

OWASP Top 10 - 2021

Security Impact

- Financial losses
- Intellectual property theft
- Brand reputation compromise
- Fraud
- Legal exposure
- Extortion

Path Traversal Attack

Path Traversal Attack

Using the .. and / symbols to gain access to files and directories that are not intended to be accessed.

```
http://www.foo.com/../../database.db
```

Normally web servers are well protected against these types of attacks but the application can also be targeted:

```
http://www.foo.com/page.php?page=../../database.d
```

```
http://www.foo.com/viewimage.php?path=viewimage.p
```

Preventing

```
http://www.foo.com/index.php?page=news
```

Replace:

```
include('header.php');
include($_GET['page']);
include('footer.php');
```

With:

```
include('header.php');
if ($page == 'news') include('news.php');
if ($page == 'login') include('login.php');
include('footer.php');
```

SQL Injection

SQL Injection

Insertion of a SQL query via the **input data** from the client to the application.

SQL injection attacks allow attackers to:

- spoof identity
- tamper with existing data
- allow the complete disclosure of all data on the system
- become administrators of the database server

Disclosure of data

```
// $username has the name of the logged in user
$dbh->query("SELECT * FROM items
             WHERE owner = '" . $username . "'");
```

Create an account with username: `johndoe' OR 1 = 1--`

```
SELECT * FROM items WHERE owner = 'johndoe' OR 1
```

Spoof identity

```
// verifies if username and password are correct
$dbh->query("SELECT * FROM users WHERE " .
      "username = '" . $username . "' " .
      "AND password ='" . $password . "'");
```

Use these credentials to login:

`username: "johndoe" and password: "' OR 1 = 1; --"`

```
SELECT * users
WHERE username = 'johndoe'
AND password = '' OR 1 = 1; --'
```

Gain privileges

```
// searches for specific item
$dbh->query("SELECT * FROM items WHERE title = '"
```

Navigate to URL:

```
http://foo.com/search.php?title='; INSERT INTO us
('johndoe', 'password', true); --
```

Third parameter has admin status of user:

```
SELECT * FROM items WHERE title = ''; INSERT INTO
('johndoe', 'password', true); --'
```

Preventing

- Use of **Prepared Statements** (Parameterized Queries).
- Use of Stored Procedures.
- Escaping all User Supplied Input.

```
$stmt = $dbh->prepare('SELECT * FROM items WHERE
$stmt->execute(array($title));
$items = $stmt->fetchAll();
```

Cross-site Scripting (XSS)

Cross-site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of **injection**, in which **malicious scripts** are injected into otherwise benign and trusted websites.

Types

- **Persistent** XSS generally occurs when user input is **stored on the target server**, such as in a database, in a message forum, visitor log, or comment field, ...
- **Reflected** XSS occurs when user input is **immediately returned by a web application** in an error message, search result, or any other response, **without permanently storing** the user-provided data.
- **DOM Based** XSS occurs when the data flow **never leaves the browser**. For example, the malicious script could be in the URL and the page inserts it into the DOM adding malicious code.

Cross-site Scripting (XSS) : Persistent

```
<?php
 $stmt = $dbh->prepare("INSERT INTO comment
                        VALUES (DEFAULT, ?, ?, ?)
 $stmt->execute(array($_POST['postid'], $_POST['t
                      $_SESSION['username']));
?>
```

```
<?php
 $stmt = $dbh->prepare("SELECT * FROM comment WHE
 $stmt->execute(array($_POST['postid']));
 $comments = $stmt->fetchAll();
?>

<?php foreach($comments as $comment) {?>
  <div class="comment"><?=$comment['text']?></div
<?php } ?>
```

Comments can contain malicious code that is stored and shown to all users:

```
comment.php?postid=10&text=<script>alert("hacked"
```

Cross-site Scripting (XSS) : Reflected

```php
<?php
echo "You searched for: " . $_GET["query"];
// List search results
?>
```

```
http://foo.com/search.php?query=<script>alert("ha
```

Preventing

Never put untrusted data:

- directly in a **script**
- inside an **HTML comment**
- in an **attribute name**
- in a **tag name**
- directly in **CSS**
- inside **non-safe attribute** values

Preventing

Validate: If the input contains unexpected characters reject it:

```
if ( !preg_match ("/^[a-zA-Z\s]+$/", $_GET['name'
  // ERROR: Name can only contain letters and spa
}
```

Filter: Or just filter the unexpected characters:

```
$name = preg_replace ("/[^a-zA-Z\s]/", '', $_GET[
```

Preventing

Encode: When showing untrusted data encode it first using htmlspecialchars() or htmlentities():

```
<?=htmlentities($post['text'])?>      // encodes a
<?=htmlspecialchars($post['text'])?> // encodes o
```

So that this:

```
<script>alert("hacked")</script>
```

Becomes this:

```
&#x3C;script&#x3E;alert(&#x22;hacked&#x22;)&#x3C;
```

Preventing

Encode: When using untrusted data to create URLs encode it first using urlencode():

```
<a href="search.php?q=<?=urlencode($_GET['q'])?>"
```

So that this:

```
search.php?q=<script>alert("hacked")</script>
```

Becomes this:

```
search.php?q%3D%3Cscript%3Ealert(%22hacked%22)%3C
```

Preventing

- To write untrusted data in other locations (attributes, tag names, comments, ...), use a **context-aware encoder** like PHP-ESAPI.
- If you want to allow some HTML, strip_tags() **might not be enough**.
- Use a more advanced **HTML filter library** like HTML Purifier.

Preventing in Javascript

HTML Escape Before Inserting Untrusted Data into HTML Element Content

```
const entityMap = {
  "&": "&amp;",
  "<": "&lt;",
  ">": "&gt;",
  '"': '&quot;',
  "'": '&#39;',
  "/": '&#x2F;'
};

function escapeHtml(string) {
  return String(string).replace(/[&<>"'\/]/g, fun
    return entityMap[s];
  });
}
```

Not enough in all locations; use **context-aware encoders**. For example OWASP ESAPI for Javascript.

Cookies

- Preventing all XSS flaws is hard.
- To mitigate the impact of an XSS flaw on your site, set the HTTPOnly flag on your session cookie using session-set-cookie-params before starting your session:

```
session_set_cookie_params(0, '/', 'www.fe.up.pt',
```

If the HttpOnly flag is included in the HTTP response header, the cookie cannot be accessed through a client-side script.

XSS Mantra

> "filter input, encode output"

Read more:

- OWASP XSS Prevention Cheat Sheet
- OWASP DOM Based XSS Prevention Cheat Sheet
- OWASP XSS Filter Evasion Cheat Sheet
- A comprehensive tutorial on cross-site scripting

Cross-site Request Forgery (CSRF)

Cross-site Request Forgery (CSRF)

The application allows a user to submit a **state-changing request** that does **not include anything secret**.

```
http://foo.com/transfer.php?amount=1500&destinati
```

The attacker constructs a request that will **transfer money** from the victim's account to the attacker's account, and then **embeds** this attack in an image request stored on various sites under the attacker's control:

```
<img
  src="http://foo.com/transfer.php?amount=1500&de
  width="0" height="0" />
```

If the victim visits any of the attacker's sites while already authenticated to *foo.com*, these forged requests will automatically include the user's session info, authorizing the attacker's request.

Preventing (NOT)

- Using a Secret Cookie
- Only Accepting POST Requests
- Multi-Step transactions
- URL Rewriting

These methods **DO NOT WORK**

Preventing

- **Generate** a random token per session
- **Store** this token as a session variable
- **Send** this token as part of every (sensitive) request
- **Verify** the token is correct on every page

Preventing

```
function generate_random_token() {
  return bin2hex(openssl_random_pseudo_bytes(32))
}
```

```
session_start();
if (!isset($_SESSION['csrf'])) {
  $_SESSION['csrf'] = generate_random_token();
}
```

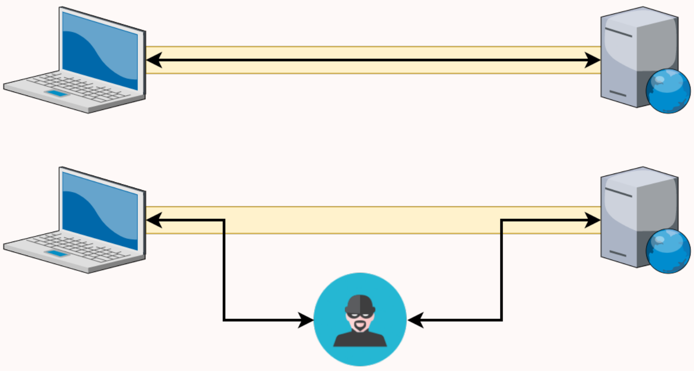
```
<form action="transfer.php">
  <input type="hidden" name="csrf" value="<?=$_SE
</form>
```

```
session_start();
\\...
if ($_SESSION['csrf'] !== $_POST['csrf']) {
  // ERROR: Request does not appear to be legitim
}
```

Man-in-the-middle Attack

Man-in-the-middle Attack

- **Intercept** a communication between two systems.
- Using different techniques, the attacker **splits** the original TCP connection into 2 new connections, one between the client and the attacker and the other between the attacker and the server
- Once the TCP connection is intercepted, the attacker acts as a **proxy**, being able to read, insert and modify the data in the intercepted communication.

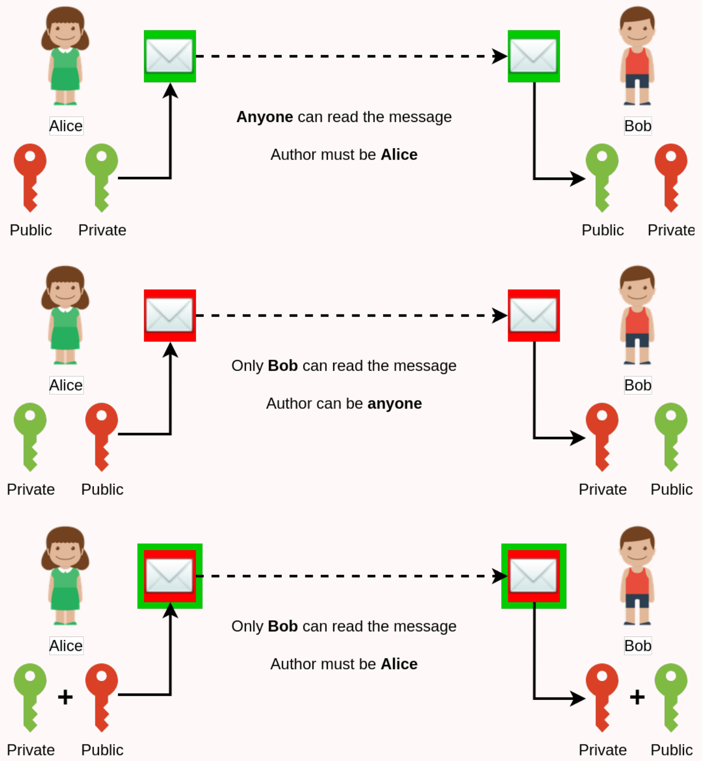


Public-key Cryptography

Also known as **asymmetric** cryptography, is a class of cryptographic algorithms that requires two separate keys, one of which is private and one of which is public.

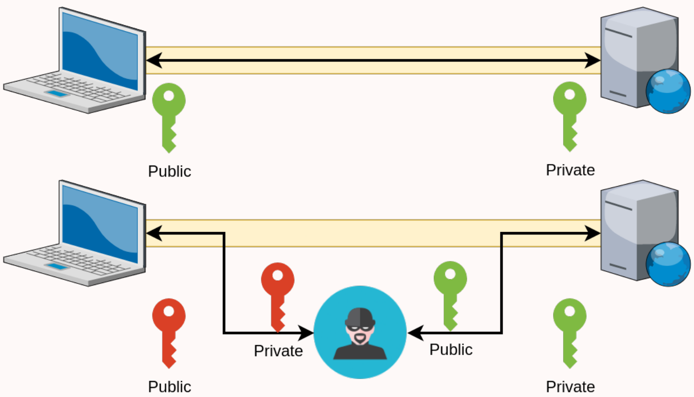
- If the sender **signs** a message with his private key, any receiver can **verify** that the message was sent by him.
- If a sender **encrypts** a message with a public key, **only** the receiver having the private key can read that message.
- Let's see how this works without going too deep into the math behind it.

Public-key Cryptography



Man-in-the-middle (again)

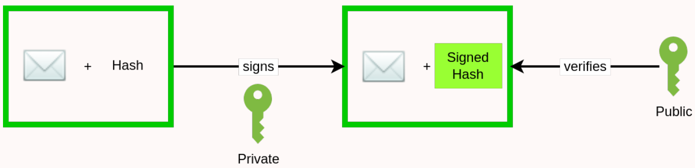
- Using encryption is **not enough** because every encryption method requires an additional exchange or transmission of information over a secure channel (e.g. the public key).



- The solution is to use public keys that have been signed by a **certificate authority** (CA).

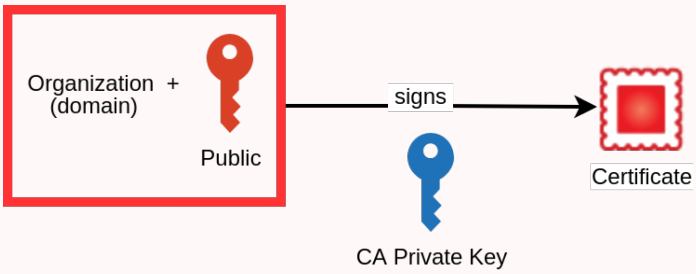
Digital Signature

- Digital signatures are a scheme that allows the demonstration of a message's **authenticity**.
- For efficiency reasons, **normally only a hash** of the original message is signed.



Certificates

- Certificates are small data files that digitally **bind** a **cryptographic key** to an **organization**.
- By signing a certificate, a **Certificate Authority (CA)** states that it **verified** the organization's information.

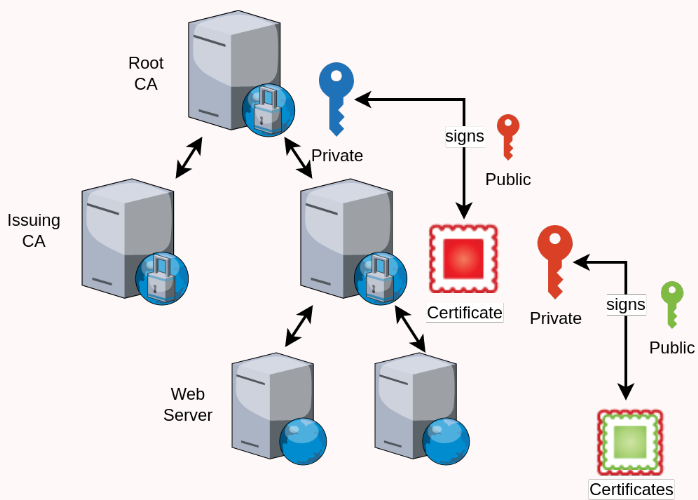


Certificate Authority

- Web browsers **trust** websites based on CAs that come **pre-installed** (Verisign/Comodo/Microsoft/...).
- The user trusts the CA to **vouch** only for **legitimate websites**.
- The website **provides** a **valid** certificate, which means it was signed by a trusted authority.
- The certificate **correctly identifies** the website.
- The user trusts that the protocol's encryption layer (TLS/SSL) is sufficiently **secure** against eavesdroppers.

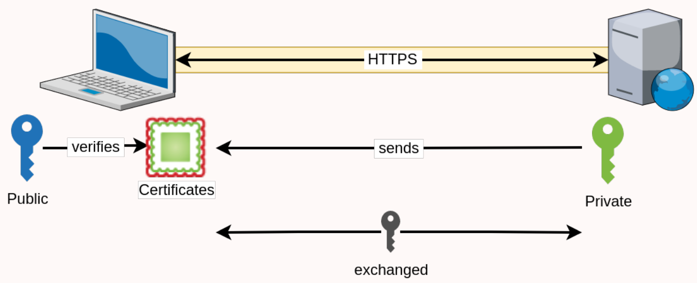
Chain of Thrust

- A **certificate chain** is an **ordered list** of certificates, where each one certifies the next until a root certificate is reached.
- This allows browsers to only **pre-install a few** root certificates.



HTTPS

- **H**yper**t**ext **T**ransfer **P**rotocol **S**ecure (HTTPS) is just HTTP on top of the **SSL/TLS** protocol.
- The browser uses the **pre-installed CAs certificates** to verify the authenticity of the **server's public key**.



For **efficiency** reasons, public-key cryptography is used to exchange a **symmetric key** that is used for the rest of the session (SSL handshake).

Credential Storage

Password Transmission

Passwords have to be sent from the browser to the server. But they should **never**:

- Be sent over **HTTP** (only HTTPS) to prevent man-in-the-middle attacks or eavesdropping.
- Be sent using **GET** parameters as they will be displayed in the URL.
- Be encrypted in the browser. Being able to capture the encrypted password would be the same as capturing the plain text password.

Hashing

In the case of a database breach, having passwords stored in **clear text**, allows the attacker to have **instant** access to **all** user passwords.

So passwords should be stored as hashes:

- Hash algorithms are **one-way** functions. They turn any amount of data into a **fixed-length** *fingerprint* that **cannot** be reversed.
- Small changes in the original text produce **completely different hashes**.

Hashing Workflow

- The user **creates** an account by entering a username and password.
- Their password is **hashed** and **stored** in the database.
- When the user attempts to login, the **hash** of the password they entered is **checked** against the **hash** of their real password.
- If the hashes **match**, the user is granted **access**. If not, the user is told they entered invalid login credentials.

```
$stmt = $db->prepare(INSERT INTO users VALUES (?,
$stmt->execute(array($username, md5($password));
```

```
$stmt = $db->prepare('SELECT * FROM users
                      WHERE username = ? AND pass

$stmt->execute(array($username, md5($password));

if ($stmt->fetch() !== false) {
  $_SESSION['username'] = $username;
}
```

Cracking Hashes

- **Brute Force Attacks** - Try every possible combination of characters up to a given length.
- **Dictionary Attack** - Try every password and variants from a file. These files come from dictionaries and real password databases.
- **Lookup Tables** - Pre-computed tables containing passwords hashes in a password dictionary.
- **Rainbow Tables** - Rainbow tables are a time-memory trade-off technique. Slower but can store more hashes. Examples.

Using Salt

- Lookup tables and rainbow tables only work because each password is hashed the **exact same way**.
- We can prevent this by **appending** a string to each password making pre-existing rainbow tables useless.
- This is called adding **salt** to a password. "Everything is better with salt."

Salt Reuse

Using the same salt for every user is **ineffective**:

- Two users with the **same password** will still have the same hash.
- The attacker can generate a **rainbow table** for that specific salt.
- Finding the salt is relatively easy (especially if the salt is short).

Double Hashing

Double hashing passwords, sometimes with different hashing algorithms, can make hashes **less secure**.

Hashing Algorithm

- There are several hashing algorithms available. Some of them are currently considered **weaker** (MD5, SHA1).
- More secure hashing functions should be used like SHA256, SHA512 or bcrypt (blowfish).

Slow Hash Functions

- High-end graphics cards (GPUs) and custom hardware can compute **billions of hashes per second** making brute force attacks still very effective.
- The goal is to make the hash function **slow enough** to impede attacks, but still **fast enough** to not cause a noticeable delay for the user.
- Key stretching is implemented using a special type of **CPU-intensive** hash function (e.g. **bcrypt**).
- These algorithms take a **security factor** or iteration count as an argument. This value determines how slow the hash function will be.

Secret Key

- By adding a **secret fixed key** to all passwords, we prevent an attacker that only gained access to the database, to even try to crack the passwords.
- This key has to be **kept secret** from an attacker even in the event of a breach.
- The key must be stored in an **external system**, such as a physically separate server dedicated to password validation.
- One can even use special **dedicated hardware** to store this secret key.

Passwords Done Right

Salt

- Salt should be generated using a Cryptographically Secure Pseudo-Random Number Generator (**CSPRNG**).
- The salt needs to be **unique** per user.
- The salt needs to be **long**.

Generating

- Prepend the **salt** to the **password** and **hash** it with a standard cryptographic hash function such as **bcrypt**.
- Save both the salt and the hash in the **user's database record**.

Validating

- Retrieve the user's **salt** and **hash** from the database.
- Prepend the **salt** to the given **password** and **hash** it using the same hash function.
- Compare the **hash** of the given password with the **hash** from the database.

Read more: Hashing Security

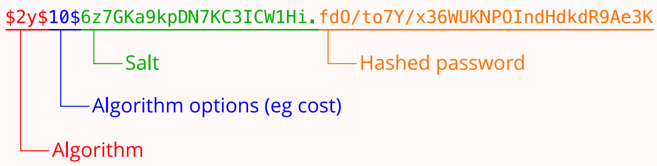
Passwords in PHP

The recommended method to hash and validate passwords in PHP is by using the password-hash and password-verify functions.

```
string password_hash (string $pwd , integer $algo
```

```
boolean password_verify ( string $pwd , string $h
```

- These functions generate their own salt.
- The **hash** function returns the used algorithm, cost and salt as part of the hash. Therefore, all information that's needed to verify the hash is included in it.
- This allows the **verify** function to verify the hash without needing separate storage for the salt or algorithm.



PHP Example

```
<?php
  $options = ['cost' => 12];
  $stmt = $db->prepare(INSERT INTO users VALUES (
  $stmt->execute(array(
    $username,
    password_hash($password, PASSWORD_DEFAULT, $o
  );
```

```
<?php
  $stmt = $db->prepare('SELECT * FROM users WHERE
  $stmt->execute(array($username));
  $user = $stmt->fetch();

  if ($user && password_verify($password, $user['
    $_SESSION['username'] = $username;
  }
```

The current default algorithm is **bcrypt**.

More on Passwords

- Make sure your usernames/userids are case **insensitive** (even emails).
- Implement proper **password strength** controls.
- Do **not** apply short or no length, character set, or encoding restrictions on the entry or storage of credentials.
- Design password storage **assuming** eventual compromise.

OWASP Authentication Cheat Sheet

OWASP Password Storage Cheat Sheet