Sanitizzazione dell'Input, Gestione degli Errori e Sicurezza di Base in PHP

Prof. Fedeli Massimo - IIS Fermi Sacconi Cpia

13 gennaio 2026

Documento tecnico per sviluppatori web

Corso di Sicurezza Informatica – Livello Introduttivo

Redatto il 13 gennaio 2026

Indice

1 Introduzione

La sicurezza delle applicazioni web è una componente fondamentale nello sviluppo software moderno. Con l'aumento delle minacce informatiche, è essenziale che ogni sviluppatore comprenda i rischi legati all'input utente non controllato, alla gestione impropria degli errori e alle vulnerabilità comuni come le *injection* e gli attacchi *Cross-Site Scripting* (XSS).

PHP, pur essendo uno dei linguaggi più diffusi per lo sviluppo web, è spesso soggetto a cattive pratiche che possono compromettere la sicurezza di interi sistemi. Questo documento ha l'obiettivo di introdurre concetti chiave relativi alla sanitizzazione dell'input, alla corretta gestione degli errori e alle principali vulnerabilità di sicurezza nel contesto PHP, fornendo esempi pratici e linee guida per scrivere codice sicuro.

2 Sanitizzazione dell'Input

2.1 Cos'è la sanitizzazione?

La **sanitizzazione dell'input** consiste nel processo di pulizia, validazione e trasformazione dei dati provenienti dall'esterno (es. form HTML, parametri URL, cookie, header HTTP) prima che vengano utilizzati dall'applicazione. L'obiettivo è prevenire l'inserimento di dati malevoli che potrebbero compromettere la logica dell'applicazione o la sicurezza del sistema.

In PHP, ogni dato proveniente da:

- `$_GET`
- `$_POST`
- `$_COOKIE`
- `$_REQUEST`
- `$_SERVER`

deve essere considerato **non affidabile** finché non è stato adeguatamente validato e/o sanificato.

2.2 Validazione vs Sanitizzazione

È importante distinguere tra:

- **Validazione**: verifica che l'input rispetti determinati criteri (es. formato email, lunghezza, tipo numerico).
- **Sanitizzazione**: modifica l'input per renderlo sicuro (es. rimozione di caratteri speciali, encoding HTML).

Entrambi i passaggi sono complementari e spesso necessari.

2.3 Funzioni utili in PHP

PHP offre diverse funzioni native per la sanitizzazione:

- `filter_var()`: valida e sanifica variabili con filtri predefiniti.
- `htmlspecialchars()`: converte caratteri speciali in entità HTML.
- `trim()`, `strip_tags()`: rimuovono spazi bianchi o tag HTML.

2.3.1 Esempio con `filter_var`

```
1     $email = $_POST['email'] ?? '';
2
3     if (filter_var($email, FILTER_VALIDATE_EMAIL)) {
4       echo "Email valida: " . htmlspecialchars($email, ENT_QUOTES,
    'UTF-8');
5     } else {
6       echo "Email non valida.";
7     }
8
```

Listing 1: Esempio di validazione email

2.3.2 Esempio con `htmlspecialchars`

```
1     $user_input = $_GET['comment'] ?? '';
2     $safe_output = htmlspecialchars($user_input, ENT_QUOTES |
    ENT_SUBSTITUTE, 'UTF-8');
3     echo "<p>Commento: $safe_output</p>";
4
```

Listing 2: Sanitizzazione per output HTML

2.4 Filtri di input con `filter_input`

Una pratica migliore è usare `filter_input()` invece di accedere direttamente a `$_GET` o `$_POST`:

```
1     $id = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT);
2     if ($id === false || $id < 1) {
3       http_response_code(400);
4       die("ID non valido.");
5     }
6
```

Listing 3: Uso di filter$_{i}$*nput*

Nota: A partire da PHP 8.1, `FILTER_SANITIZE_STRING` è deprecato. Si consiglia di usare `htmlspecialchars()` o librerie esterne come HTML Purifier per contenuti complessi.

3 Gestione degli Errori

3.1 Perché gestire gli errori in modo sicuro?

Mostrare messaggi di errore dettagliati agli utenti finali può rivelare informazioni sensibili sul sistema (percorsi di file, strutture di database, versioni di software), facilitando attacchi mirati.

3.2 Configurazione iniziale

In ambiente di produzione, è fondamentale disattivare la visualizzazione degli errori:

```
display_errors = Off
log_errors = On
error_log = /var/log/php_errors.log

```

Listing 4: Configurazione sicura in php.ini

Oppure tramite codice (sconsigliato in produzione, ma utile per test):

```
ini_set('display_errors', 0);
ini_set('log_errors', 1);

```

3.3 Gestione personalizzata degli errori

È buona pratica implementare un gestore di errori personalizzato:

```
function customErrorHandler($errno, $errstr, $errfile, $errline
) {
  error_log("[$errno] $errstr in $errfile on line $errline");
  if (!headers_sent()) {
    http_response_code(500);
    echo "<h1>Errore interno del server</h1>";
    exit;
  }
}

set_error_handler("customErrorHandler");

```

Listing 5: Gestore di errori personalizzato

3.4 Eccezioni

PHP supporta le eccezioni (`try/catch`). È consigliabile usarle per gestire errori prevedibili:

```
try {
  $pdo = new PDO($dsn, $user, $pass);
} catch (PDOException $e) {
  error_log("Connessione DB fallita: " . $e->getMessage());
  http_response_code(500);
  die("Servizio temporaneamente non disponibile.");
```

```
    }

```

Listing 6: Gestione eccezioni

4 Concetti di Sicurezza di Base

4.1 Cross-Site Scripting (XSS)

4.1.1 Cos'è XSS?

L'attacco **Cross-Site Scripting** (XSS) si verifica quando un'applicazione include input utente non sanificato in una pagina HTML. Un attaccante può iniettare script JavaScript malevoli che vengono eseguiti nel browser delle vittime.

Esistono tre tipi principali:

1. **XSS riflesso**: lo script è incluso nella richiesta (es. URL) e immediatamente restituito.
2. **XSS persistente**: lo script viene salvato nel database e mostrato a tutti gli utenti.
3. **XSS basato su DOM**: la vulnerabilità è lato client (JavaScript).

4.1.2 Esempio di XSS riflesso

```
    // CODICE NON SICURO
    $search = $_GET['q'];
    echo "<p>Risultati per: $search</p>";

```

Listing 7: XSS riflesso - codice vulnerabile

Se un utente visita: `pagina.php?q=<script>alert('XSS')</script>`, lo script verrà eseguito.

4.1.3 Prevenzione

Usare sempre `htmlspecialchars()` per qualsiasi output dinamico in HTML:

```
    $search = $_GET['q'] ?? '';
    $safe_search = htmlspecialchars($search, ENT_QUOTES |
ENT_SUBSTITUTE, 'UTF-8');
    echo "<p>Risultati per: $safe_search</p>";

```

Listing 8: Prevenzione XSS

Altre misure:

- Usare Content Security Policy (CSP) negli header HTTP.
- Validare l'input (es. solo caratteri alfanumerici per una ricerca).

4.2 SQL Injection

4.2.1 Cos'è SQL Injection?

L'**SQL Injection** avviene quando un attaccante inserisce codice SQL malevolo in un input che viene poi concatenato direttamente in una query. Ciò permette di leggere, modificare o cancellare dati arbitrari.

4.2.2 Esempio vulnerabile

```
$id = $_GET['id'];
$query = "SELECT * FROM users WHERE id = $id";
$result = mysqli_query($conn, $query);

```

Listing 9: SQL Injection - codice pericoloso

Un attaccante potrebbe usare: `pagina.php?id=1 OR 1=1 -`, ottenendo tutti gli utenti.

4.2.3 Prevenzione con Prepared Statements

La soluzione più efficace è usare **prepared statements** con PDO o MySQLi:

```
$id = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT);
if ($id === false || $id < 1) {
  http_response_code(400);
  die("ID non valido.");
}

$stmt = $pdo->prepare("SELECT * FROM users WHERE id = ?");
$stmt->execute([$id]);
$user = $stmt->fetch();

```

Listing 10: Prevenzione con PDO

Mai concatenare input utente direttamente nelle query!

4.3 Altre Injection Comuni

Oltre a SQL Injection, esistono altre forme di injection:

- **Command Injection**: esecuzione di comandi di sistema tramite `exec()`, `shell_exec()`, ecc.
- **LDAP Injection**: manipolazione di query LDAP.
- **XPath Injection**: attacchi su query XML.

La regola generale è: **mai fidarsi dell'input utente** e usare API sicure (es. prepared statements, librerie validate).

5 Best Practice Generali

1. **Non fidarsi mai dell'input utente**. Trattalo sempre come potenzialmente malevolo.
2. **Sanitizzare in base al contesto**: HTML, SQL, comandi di shell richiedono tecniche diverse.
3. **Usare librerie consolidate**: non reinventare la ruota (es. HTML Purifier per HTML complesso).
4. **Aggiornare PHP e dipendenze**: molte vulnerabilità sono patchate nelle nuove versioni.
5. **Disattivare errori in produzione**.
6. **Usare HTTPS** per proteggere i dati in transito.
7. **Applicare il principio del minimo privilegio**: l'utente DB non deve avere permessi di amministratore.

6 Strumenti e Risorse Utili

- **OWASP Top 10**: `https://owasp.org/www-project-top-ten/`
- **HTML Purifier**: `http://htmlpurifier.org/`
- **PHPStan**: `https://phpstan.org/`
- **ModSecurity**: `https://modsecurity.org/`

7 Conclusioni

La sicurezza web non è un optional. Anche piccole disattenzioni nella gestione dell'input o nella configurazione degli errori possono portare a gravi compromissioni. In PHP, grazie a funzioni native e buone pratiche consolidate, è possibile scrivere applicazioni robuste e sicure.

Ricorda: la sicurezza è un processo continuo, non un prodotto finito. Testa regolarmente il tuo codice, mantienilo aggiornato e forma te stesso e il tuo team sulle minacce emergenti.

Appendice A: Esempio Completo di Form Sicuro

```
<!-- form.html -->
<form method="post" action="process.php">
<label>Email: <input type="email" name="email" required></label>
<label>Commento: <textarea name="comment" required></textarea></label>
<button type="submit">Invia</button>
</form>
```

```php
<?php
// process.php
$email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
$comment = htmlspecialchars(trim($_POST['comment'] ?? ''), ENT_QUOTES, 'UTF-8');

if (!$email) {
  http_response_code(400);
  die("Email non valida.");
}

// Salva nel DB con prepared statement
$stmt = $pdo->prepare("INSERT INTO comments (email, text) VALUES (?, ?)");
$stmt->execute([$email, $comment]);

echo "Grazie! Commento inviato.";
?>
```

Listing 11: Form HTML + elaborazione sicura

Appendice B: Configurazione Consigliata per php.ini (Produzione)

```
; Disabilita visualizzazione errori
display_errors = Off
display_startup_errors = Off

; Abilita logging
log_errors = On
error_log = /var/log/php_errors.log

; Disabilita funzioni pericolose
disable_functions = exec,passthru,shell_exec,system,proc_open,popen

; Limita upload e memoria
upload_max_filesize = 2M
post_max_size = 3M
memory_limit = 128M

; Imposta fuso orario
date.timezone = Europe/Rome
```

Riferimenti Bibliografici

1. OWASP Foundation. *OWASP Top Ten Project.* `https://owasp.org/www-project-top-ten/`
2. PHP.net. *The PHP Manual – Security.* `https://www.php.net/manual/en/security.php`
3. Mozilla Developer Network. *Cross-site scripting (XSS).* `https://developer.mozilla.org/en-US/docs/Glossary/Cross-site_scripting`
4. Evans, C. (2020). *Web Security for Developers.* No Starch Press.