X-Frame-Options: All about Clickjacking?

“How else do X-Frame-Options protect my website”

A poem by Frederik Braun (Mozilla) and Mario Heiderich (Cure53)

The X-Frame-Options header is known to be a good measurement against those so called Clickjacking attacks. You know, this kind of attack where some other website loads important parts of your website inside an Iframe or even frameset, makes everything invisible and overlays attractive looking links and buttons with your invisible website. But - is it really all about Clickjacking? Let us find out about that!

Introduction

Clickjacking essentially means that people can be tricked into clicking on things triggering important actions. Usually they are authenticated at the website containing the elements they are tricked to click on. The technique is not new - not at all. No later than 2002, Jesse Ruderman complained about the risks of transparency for Iframes with similar things in mind. He correctly noted that the way browsers handle Iframes might compromise the security promises a website gives - without having any reliable way to defeat against it[1]:

> "*This is bad for security, because it means I can show a page like Yahoo in an iframe and add items to that page.*" Ruderman, 2002

It was only six years later in 2008, when Jeremiah Grossman and Robert Hansen "re-discovered" the issue and revealed an attack pattern which involves fooling users into clicking on seemingly harmless buttons[2]. The trick behind this attack is to place the targeted website (e.g., the Adobe Flash settings panel[3], a Facebook "like" button or a shopping cart) inside an Iframe and overlay it with a completely different webpage. This overlay might then entice users to click on certain buttons or links. This coined the term Clickjacking. Meanwhile, the colorful world of information security has accumulated a rather large array of related attack techniques - double-click jacking, like-jacking, cursor-jacking and what not. But we don't want to cover these today.

Based on the large number of possible exploitation scenarios, a prevention technique was developed and deployed - initially in MSIE 8 - and meanwhile supported by all modern browsers. We are talking about the X-Frame-Options HTTP header[4]: This particular header allows a web server to announce, that a specific document is either only to be framed from a certain URL (ALLOW-FROM uri), from sites of the same origin (SAMEORIGIN) or not to be framed at all (DENY) and thereby more or less defeat Clickjacking by removing its very foundation[5]. So - to wrap it up:

- `X-Frame-Options: DENY` « won't allow the website to be framed by anyone
- `X-Frame-Options: SAMEORIGIN` « No one can frame except for sites from same origin
- `X-Frame-Options: ALLOW-FROM %uri%` « Only this one URI can frame. No one else

While this security header has been mostly invented as a reaction to Clickjacking, it is indeed useful to prevent other, far more dangerous yet less well-known things. And this is what our article is about.

---

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=154957
[2] http://jeremiahgrossman.blogspot.de/2008/10/clickjacking-web-pages-can-see-and-hear.html
[3] http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html
[4] http://tools.ietf.org/html/rfc7034
[5] http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx

The Docmode-Problem

It's a pretty well known fact that Microsoft Internet Explorer (IE) ships several different *document modes* - meaning different ways to render a HTML document[6]. These modes (we call them Docmodes - not to be confused with Doctypes) are meant to provide a fallback in case your precious website renders incorrectly after a new IE version was released. Developers experienced this effect for example when IE5.5 was superseded by IE6 and even more when IE6 was overtaken by IE7 - and so on. So, a decision was made to provide those developers with an easy way out: Simply set IE to a rendering mode that resembles an older version, yet has all security fixes shipped with the more modern versions.

This even made perfect sense! If for instance a web developer spent a lot of time to build around one of those common CSS bugs - Peek-a-boo[7], Guillotine[8] and what not, then a new version of IE where this bug was fixed could actually have devastating effects - and the once costly and time consuming fix working around the old bug would actually destroy the layout rendered by IE instead of fixing it. So - if you indeed need the old bugs to work because you coded your way around them, then the older document modes might be something that saves you a lot of time. How to set them? Easy as pie - with a META tag:

```
<meta http-equiv="X-UA-Compatible" content="IE=7" />
```

Or of course via HTTP Header:

```
X-UA-Compatible: IE=7
```

But, along with old layout bugs one can also reactivate ancient APIs and features hidden deep inside Internet Explorer's DLLs. One of them are CSS Expressions - an easy and convenient way for attacks to turn an injection into CSS context into an exploitable JavaScript execution[9]. This is how it looks:

```
<DIV style="color: red; foo: expression(open(alert(1)))">XSS via
CSS</DIV>
```

Note that this *only* works, in case the document is loaded in IE5 Quirks or IE7 document mode. Any other mode does not support CSS expressions anymore. And the news are even better: IE11 doesn't even support CSS Expressions in the Internet Zone - a page must be marked to be running in the "Trusted Zone" to make them work. But to get back to our actual topic - now what does all this have to be with *X-Frame-Options*? Well. In IE, a document mode will be given from parent frame to child frame - classic inheritance. If a website is loaded in "Standards Mode" but another website frames it, then the document mode will be inherited from the parent website. If that document mode happens to be IE5 or IE7, then CSS expressions will work again! Even though your website is usually running in "Standards Mode". Good to know, right?

---

[6] http://msdn.microsoft.com/library/cc288325%28v=vs.85%29.aspx
[7] http://www.positioniseverything.net/explorer/peekaboo.html
[8] http://www.positioniseverything.net/explorer/guillotine.html
[9] http://msdn.microsoft.com/en-us/library/ms537634%28v=vs.85%29.aspx

Now - if you prohibit framing via *X-Frame-Options*, then no one can ever force your website to be running in an older document mode and re-activate CSS expressions[10]. Well, unless you have a huge header- or HTML-injection problem anyway. So we can see here - XFO is yet for another time safeguard against nasty script injections. Oh - and also if you use the HTML5 doctype (that short one... `<!doctype html>`) in your website, the support for CSS expressions is turned off even if you get docmode-inherited![11] Magic. So don't stick with the ancient HTML4 doctypes. Exchange them first thing tomorrow and use the HTML5 doctype - perfect also in case your website's business model forbids you to forbid getting framed.

Oh - and Microsoft just recently announced, that Visual Basic Script will be disabled in standards mode in IE11. Which is great. VBS allows an attacker to have an XSS be successful even if no parentheses or other necessary characters are echoed by the injectable part if the website. So disabling support for it of course decreases the size of the attack surface. Not dramatically but by a significant quantity. Now, with the mentioned docmode inheritance, we can easily reactivate VBS too. So again, Iframes are able to bring back features that might harm you - and there's nothing you can do against that aside from setting your XFO headers and be fine. by the way, CSS Expressions are more or less dead in IE11 - they only work in the "Trusted Site" zone. Rest in peace, CSS expressions. We'll miss you!

Drag'N'Drop XSS

You like browser-games? Now what if you navigate to a website that invites you to play a little game. All you have to do is grab a "Basketball" with your mouse cursor (or your sweaty greasy index finger if you are into touch-devices, bah!) and drag it over into a basket. Everybody loves that! Now, if you do that and you are a wee bit lucky, you'll win a free iPad. Sounds perfectly legit, right? Only that the basketball is not a ball - and the basket is not a basket.

What the attacker does - yes, the website was prepared by one of those gloomy minded protagonists - is the following: She places another one of these invisible Iframes pointing to a website your are logged into and waits, that you drag, let's say a maliciously prepared SVG into it! You might think it's the basketball but it's payload and once the payload is being dropped on the invisible Iframe, it executes in the framed website's context and you have yet another XSS. Or worse. The Iframe could as well point to an Intranet resource or a router interface. Luckily, browser vendors realized the danger of these cross-domain drag&drop attacks and either block cross-origin drag&drop completely or internally sanitize everything that leaves one domain and gets dropped into another. Lucky you.

But there are edge-cases. Some browsers prohibit cross-domain drag&drop in the same view - the same window. But as soon as you have several browser windows opened, it will work again.

[10] http://blogs.msdn.com/b/dross/archive/2011/06/30/enforcing-standards-mode-with-x-frame-options.aspx
[11] http://www.w3.org/TR/html5/syntax.html#the-doctype

Copy&Paste XSS

Then there is copy&paste - which is still tolerated by browsers even if it happens across domains. The W3C proposed a sanitation checklist for what they labeled the Clipboard API[12]. It urges browser vendors to cleanse Rich-Text in the clipboard to make sure, a copy & paste operation doesn't result in a full blown XSS without the user having any chance to find out what happened.

But if you read those specs closely, you might quickly realize that they are in fact proposing a blacklist. Script-tags need to go, JavaScript URIs, Applets and what not. Many bypasses were reported and fixed in the past and it's likely that there will be more in the future[13] [14]. Some of them can be seen below:

```
<body contenteditable>copy <form id=test></form><button form=test
formaction=javascript:alert(1)>then click me!</button>me into a x-origin
window

<body contenteditable>copy <iframe style="height:0;width:0;opacity:0"
srcdoc="<img src=x onerror=alert(domain)>"></iframe>me into a x-origin
window
```

There's more like these. In several browsers - even if fully patched. Bottom line is: if you don't prevent framing, then your website might be prone to become victim of a copy&paste XSS. Note - JavaScript-based framebusters do not help here - they can be deactivated easily by a skilled attacker. It has to be the good old XFO again. If you wish to play with cross-domain copy&paste, an ugly yet working tool is available for you - created by yours truly[15].

Invisible Site-Wide XSS

Remember Krzysztof Kotowicz's masterpiece of an attack based on the HTML5 history API[16]? Well, what he did was the following. He claimed, that he can expand the impact of a single reflected XSS to the entire domain - and make every single page on the affected domain be vulnerable too. What he did was basically an old trick combined with a new technique to make it absolutely stealthy.

He took a URL pointing to a document with a reflected XSS. Then he created an Iframe via XSS - framing the actual website - without Injection. Important here: The Iframe must fill the entire window so the user cannot distinguish between actual page and Iframed page. The XSS'ed website and the framed website are on the same domain - so they can talk with each other, the SOP allows that. If the user clicks on a link, the Iframe navigates - but not the top frame. The top frame still carries the XSS payload and thereby observes and controls anything the user does in the Iframe. That's pretty nasty already and allows indeed to expand the attack surface from one reflected XSS to cover the entire domain. But! The user can

---

[12] http://www.w3.org/TR/clipboard-apis/#pasting-html-and-multi-part-data
[13] http://code.google.com/p/chromium/issues/detail?id=136497
[14] https://code.google.com/p/chromium/issues/detail?id=171392
[15] http://html5sec.org/copypaste/xdom
[16] http://blog.kotowicz.net/2010/11/xss-track-got-ninja-stealth-skills.html

easily notice that something is going on. By observing, that even when a click on a link is being performed, the URL shown in the address bar doesn't change.

And that's where the HTML5 History API comes into play[17]. With this API we can change the content of the address bar without actually navigating! This way the XSS in the outer frame can read which link the user clicks in the inner frame and then take that info and show it in the address bar. So there's no way for the user to see that each and every click is being tracked and each and every page of the inner frame is compromised by the original, harmless reflected XSS. Iframes - you gotta love 'em. Further note that this might even bypass XFO - when set to more tolerant SAMEORIGIN string. DENY would have mitigated the attack. Not the XSS - but the dramatic expansion of the attack surface.

Cross-Site Scripting & Length Restrictions

Cross-Site Scripting (XSS) seems to be a completely different topic compared to Clickjacking and most people will agree that this should be prevented by properly escaping output. But! We're going to tell you now that one can also use the *X-Frame-Options* as an additional layer of defense against XSS of many kinds. Imagine this: An attacker can inject JavaScript/HTML into a vulnerable web page but is hindered to carry out a dangerous attack because his input is limited in size. Bloody length restrictions. Let's assume the XSS payload lands within a tag attribute but may not be longer than, say, 25 characters. For example: http://vulnpage.com/?i=foo

```
<img src='foo' />
```

How to exploit with a 25 character limits? Here's the trick: By loading this vulnerable web page within an iframe, we can control the window's name, as explained earlier. The injection can therefore simply perform a call to `eval(name)`. So, the URL *http://vulnpage.com?i=x' onerror=eval(name)%20* should give us the following result in HTML code:

```
<img src='x' onerror=eval(name) ' />
```

See how the `src` attribute has been closed and a new one was opened? Let's put this all in an iframe:

```
<iframe name="alert('XSS or something very harmful but long')"
src="http://vulnpage.com?i=' onerror=eval(name)%20"></iframe>
```

Note though, that the same could be accomplished with a "detached view", also known as a popup. With a popup we can of course freely set a name for the window - by just setting the second parameter or `window.open()`. And even easier, we can just take a classic link, apply a "target" attribute and done. So, while XFO provides a good bit of protection against XSS payload delivery via name, it's not a silver bullet. People who really want to be safe from these kinds of attacks randomize their window's name to become an unguessable value - or at least reset it by deploying `<script>window.name='';</script>`. We'll cover this again later on.

[17] https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Manipulating_the_browser_history

JavaScript à la Carte

When you allow websites to frame you, you basically give them full permission to decide, what part of JavaScript of your very own script can be executed and what cannot. That sounds crazy right? So, let's say you have three script blocks on your website. The website that frames you doesn't mind two of them - but really hates the third one. maybe a framebuster, maybe some other script relevant for security purposes. So the website that frames you just turns that one script block of - and leave the other two intact. Now how does that work?

Well, it's easy. All the framing website is doing, is using the browser's XSS filter to selectively kill JavaScript on your page. This has been working in IE some years ago but doesn't anymore - but it still works perfectly fine in Chrome. Let's have a look at an annotated code example.

Here is the evil website, framing your website on example.com and sending something that looks like an attempt to XSS you! Only that you don't have any XSS bugs. The injection is fake - and resembles a part of the JavaScript that you actually use on your site:

```
<iframe src="//example.com/index.php?
code=<script>alert(1)</script>"></iframe>
```

Now we have your website. The content of the code parameter above is part of your website anyway - no injection here, just a match between URI and site content:

```
<!doctype html>
<h1>HELLO</h1>
<script>alert(1)</script>
<script>alert(2)</script>
```

The effect is compelling. The `alert(1)` will be blocked by Chrome's XSS Auditor - but the `alert(2)` will be allowed and still execute. In IE10, none of the alerts execute. Firefox? Well, there is no XSS filter but the NoScript XSS filter cannot be tricked by this as the injection itself is being converted into a harmless version - and the existing script on the website is not harmed by that.

So, allowing other websites to frame you effectively allows those websites to decide, which part of your JavaScript should execute and which shouldn't. Yes - similar things are possible with the Iframe Sandbox - but this here is far more fine-grained and might actually cause great harm once you start using JavaScript for security or crypto purposes.

Frame Busting

Of course there are ways to prevent being framed without X-Frame-Options. These techniques, mostly using JavaScript check if they are framed and bail out if otherwise. A commonly found snippet looks like this

```
if (window.location != top.location) {
        top.location = window.location }
```

These tricks have some common flaws, as Rydstedt et al. have shown in "Busting Framebusting"[18]. As explained in our previous section, these code snippets can be easily removed by abusing an XSS Filter. Other methods suggested in the paper would involve disabling JavaScript in the iframe completely, using the sandbox[19] attribute or overwriting the location attribute via DOM Clobbering[20]. As a conclusion the authors of the mentioned paper created a better frame busting JavaScript code, which had some problems of its own[21]. The conclusion is, of course, to use X-Frame-Options. Outsourcing this problem from the DOM and passing it over to the browser itself helps - and there's nothing against using both, just in case your website still attracts a lot of IE6 and Firefox 2.0 users. They are out there - and it's hard to fully ignore them.

Side Channels & Cross-Origin Leaks

Side Channel attacks using Iframes have been published all over the place. And there is no cure against them aside from making sure your website cannot be framed. Look at the research by Paul Stone[22], Eduardo[23] and one of the fellow authors of this article[24]. Even CSS shaders weren't safe from Iframe-based timing attacks as proven by Kotcher et al[25]. We can easily read pixels across domains using SVG filters and timings, we can guess words written by a framed website by decreasing the Iframe size and deriving information from scrollbar appearance timing, shown by Eduardo and we can use malicious fonts, scrollbars and CSS animations to read words and numerical sequences, even CSRF token from an Iframed website.

Heck, we don't even have to delve into academic complexity to see how Iframes can be side channels and reveal information, that otherwise would not be available to the attacker. Simple event handling, load time measuring, scrollbars, relative positioning - there's just so many ways. Remember how Iframes in combination with CSS Media Queries were used to determine the existence of a scrollbar and thereby allow reading characters and strings across domains[26]? Now there's also sandboxed Iframes allowing the parent site to control, which kind of active content the framed site can execute - which is not bad at all but

---

[18] http://seclab.stanford.edu/websec/framebusting/
[19] https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#attr-sandbox
[20] http://www.thespanner.co.uk/2013/05/16/dom-clobbering/
[21] https://www.codemagi.com/blog/post/194
[22] https://media.blackhat.com/us-13/US-13-Stone-Pixel-Perfect-Timing-Attacks-with-HTML5-WP.pdf
[23] http://sirdarckcat.blogspot.de/2013/09/matryoshka-wrapping-overflow-leak-on.html
[24] http://www.slideshare.net/x00mario/stealing-the-pie
[25] http://www.robertkotcher.com/pdf/TimingAttacks.pdf
[26] http://html5sec.org/scrollbar/test/

also extends the capabilities the attacker has. And last but not least there's of course SOP bypasses that make use of Iframes. And if you followed the security advisories of the past ten years, you might notice that there were some of them. Some many. So - Iframes do not only leverage Clickjacking, XSS, "Old Feature Re-Activation" (now called OFR - no, just kidding) and other attack vectors. They also allow the attacker to unfold a huge array of timing and other side channel attacks to get hands on private data of your fellow users. So we can see, the number of reasons why you definitely don't want to get framed increases dramatically.

Bye bye, CSP

Currently the Gecko engine is plagued by a problem involving Iframes and CSP. Once a website using CSP is being placed inside a sandboxed Iframe, the browser engine decides to ignore CSP and re-enable injections:

```
//evil.com
<iframe sandbox="allow-scripts" src="//victim.com/csp.html">

//victim.com
<?php
   header('X-Content-Security-Policy: default-src \'self\'');
   header('Content-Security-Policy: default-src \'self\'');
   header('X-Webkit-CSP: default-src \'self\'');
   header('Set-Cookie: abc=123');
?><!doctype html>
<body onclick="alert(1)">
Click me
</body>
```

If we set the sandbox settings to be very liberal, we can of course execute arbitrary injected JavaScript despite restrictive CSP rules. They simply don't apply anymore. The problem was reported by Deian Stefan and a ticket was filed several weeks ago[27]. At the time of writing, no fix was available - the problem still existed in Firefox Nightly versions.

[27] https://bugzilla.mozilla.org/show_bug.cgi?id=886164

Conclusion

So, we can see - setting your X-Frame-Options headers and setting them right doesn't exclusively protect against Clickjacking attacks but rather eliminates an entire range of nasty an attacker can do to your website. It doesn't really matter how good your XSS filter is - once framed it only takes a copy&paste or similar operations to inject arbitrary markup and get it to execute. CSP might be disabled by sandboxed Iframes, a single reflected XSS enables address bar spoofing and stealthy site-wide XSS and more.

And there is even more things to can do in case you need that certain extra bit of security. For example to check whether there is an *opener* property[28] in your DOM and someone caused your website to be opened in a new window, detached view, popup or alike. Not everybody wants that and loading a website in a popup often enables similar attacks to the ones we already saw. only that the attacker cannot necessarily control the visibility of your website as it's possible with Iframes. Well, attackers can gain partial control over what the victim would see from your website loaded in a popup. Maybe have a look at *window.open()*[29] and the feature parameter to see what popups can do and which of these work across domains - there might be some surprises in there for you. Also don't forget: Randomize your name! Some of the described attacks not only work in Iframes but also new windows or detached views. To fully mitigate them you need to make sure that you randomize *window.name* so no one can illegitimately et a hold on your window and navigate it to places where the sun won't shine.

And last but not least: This discussion is not over yet - this article shows a whole bunch of reasons why you want to make sure your website cannot be framed. But there's certainly more - and we'd be happy to hear from you. What other reasons do you know why XFO might be a perfect safeguard? What other attacks did you come up with, what kind of evil Iframe games have you observed in the past and what risks do you see in the future? Let us know in the comments[30] and keep the discussion active for good. Let's give security managers and developers more firepower in arguing why XFO makes sense and needs to be deployed - and help them to make their websites a bit more secure.

[28] http://help.dottoro.com/ljauuavo.php
[29] http://msdn.microsoft.com/en-us/library/ie/ms536651%28v=vs.85%29.aspx
[30] http://tinyurl.com/xfo-clickjacking