Clickjacking Defense Cheat Sheet

Brought to you by OWASP

Clickjacking Defense Cheat Sheet

Brought to you by OWASP Cheat Sheets

This cheat sheet is focused on providing developer guidance on Clickjack/UI Redress attack prevention.

The most popular way to defend against Clickjacking is to include some sort of "frame-breaking" functionality which prevents other web pages from framing the site you wish to defend. This cheat sheet will discuss two methods of implementing frame-breaking: first is X-Frame-Options headers (used if the browser supports the functionality); and second is javascript frame-breaking code.

Defending with X-Frame-Options Response Headers

The X-Frame-Options HTTP response header can be used to indicate whether or not a browser should be allowed to render a page in a <frame> or <iframe>. Sites can use this to avoid Clickjacking attacks, by ensuring that their content is not embedded into other sites.

X-Frame-Options Header Types

There are three possible values for the X-Frame-Options headers:

- **DENY**, which prevents any domain from framing the content.
- **SAMEORIGIN**, which only allows the current site to frame the content.
- **ALLOW-FROM uri**, which permits the specified 'uri' to frame this page. (e.g., ALLOW-FROM http://www.example.com) The ALLOW-FROM option is a relatively recent addition (circa 2012) and may not be supported by all browsers yet. BE CAREFUL ABOUT DEPENDING ON ALLOW-FROM. If you apply it and the browser does not support it, then you will have NO clickjacking defense in place.

Browser Support

The following browsers support X-Frame-Options headers.

Browser	DENY/SAMEORIGIN Support Introduced	ALLOW-FROM Support Introduced
Chrome	4.1.249.1042	Not supported/Bug reported
Firefox (Gecko)	3.6.9 (1.9.2.9)	18.0
Internet	8.0	9.0

Explorer		
Opera	10.50	
Safari	4.0	Not supported/Bug reported

References:

- Mozilla Developer Network
- IETF Draft
- X-Frame-Options Compatibility Test - Check this for the LATEST browser support info for the X-Frame-Options header

Implementation

To implement this protection, you need to add the X-Frame-Options HTTP Response header to any page that you want to protect from being clickjacked via framebusting. One way to do this is to add the HTTP Response Header manually to every page. A possibly simpler way is to implement a filter that automatically adds the header to every page.

OWASP has an article and some code that provides all the details for implementing this in the Java EE environment.

The SDL blog has posted an article covering how to implement this in a .NET environment.

Common Defense Mistakes

Meta-tags that attempt to apply the X-Frame-Options directive DO NOT WORK. For example, <meta http-equiv="X-Frame-Options" content="deny">) will not work. You must apply the X-FRAME-OPTIONS directive as HTTP Response Header as described above.

Limitations

- **Per-page policy specification**

The policy needs to be specified for every page, which can complicate deployment. Providing the ability to enforce it for the entire site, at login time for instance, could simplify adoption.

- **Problems with multi-domain sites**

The current implementation does not allow the webmaster to provide a whitelist of domains that are allowed to frame the page. While whitelisting can be dangerous, in some cases a webmaster might have no choice but to use more than one hostname.

- **Proxies**

Web proxies are notorious for adding and stripping headers. If a web proxy strips the X-Frame-Options header then the site loses its framing protection.

Best-for-now Legacy Browser Frame Breaking Script

One way to defend against clickjacking is to include a "frame-breaker" script in each page that should not be framed. The following methodology will prevent a webpage from being framed even in legacy browsers, that do not support the X-Frame-Options-Header.

In the document HEAD element, add the following:

First apply an ID to the style element itself:

```
<style id="antiClickjack">body{display:none !important;}</style>
```

And then delete that style by its ID immediately after in the script:

```
<script type="text/javascript">
   if (self === top) {
       var antiClickjack = document.getElementById("antiClickjack");
       antiClickjack.parentNode.removeChild(antiClickjack);
   } else {
       top.location = self.location;
   }
</script>
```

This way, everything can be in the document HEAD and you only need one method/taglib in your API.

Reference: https://www.codemagi.com/blog/post/194

window.confirm() Protection

The use of x-frame-options or a frame-breaking script is a more fail-safe method of clickjacking protection. However, in scenarios where content must be frameable, then a window.confirm() can be used to help mitigate Clickjacking by informing the user of the action they are about to perform.

Invoking window.confirm() will display a popup that cannot be framed. If the window.confirm() originates from within an iframe with a different domain than the parent, then the dialog box will display what domain the window.confirm() originated from. In this scenario the browser is displaying the origin of the dialog box to help mitigate Clickjacking attacks. It should be noted that Internet Explorer is the only known browser that does not display the domain that the window.confirm() dialog box originated from, to address this issue with Internet Explorer insure that the message within the dialog box contains contextual information about the type of action being performed. For example:

```
<script type="text/javascript">
   var action_confirm = window.confirm("Are you sure you want to delete your
youtube account?")
   if (action_confirm) {
       //... perform action
   } else {
       //...  The user does not want to perform the requested action.
   }
</script>
```

Non-Working Scripts

Consider the following snippet which is **NOT recommended** for defending against clickjacking:

```
<script>if (top!=self) top.location.href=self.location.href</script>
```

This simple frame breaking script attempts to prevent the page from being incorporated into a frame or iframe by forcing the parent window to load the current frame's URL. Unfortunately, multiple ways of defeating this type of script have been made public. We outline some here.

Double Framing

Some frame busting techniques navigate to the correct page by assigning a value to parent.location. This works well if the victim page is framed by a single page. However, if the attacker encloses the victim in one frame inside another (a double frame), then accessing parent.location becomes a security violation in all popular browsers, due to the **descendant frame navigation policy**. This security violation disables the counter-action navigation.

Victim frame busting code:

```
if(top.location!=self.locaton) {
  parent.location = self.location;
}
```

Attacker top frame:

```
<iframe src="attacker2.html">
```

Attacker sub-frame:

```
<iframe src="http://www.victim.com">
```

The onBeforeUnload Event

A user can manually cancel any navigation request submitted by a framed page. To exploit this, the framing page registers an onBeforeUnload handler which is called whenever the framing page is about to be unloaded due to navigation. The handler function returns a string that becomes part of a prompt displayed to the user. Say the attacker wants to frame PayPal. He registers an unload handler function that returns the string "Do you want to exit PayPal?". When this string is displayed to the user is likely to cancel the navigation, defeating PayPal's frame busting attempt.

The attacker mounts this attack by registering an unload event on the top page using the following code:

```
<script>
window.onbeforeunload = function()
{
  return "Asking the user nicely";
}
</script>
<iframe src="http://www.paypal.com">
```

PayPal's frame busting code will generate a BeforeUnload event activating our function and prompting the user to cancel the navigation event.

No-Content Flushing

While the previous attack requires user interaction, the same attack can be done without prompting the user. Most browsers (IE7, IE8, Google Chrome, and Firefox) enable an attacker to automatically cancel the incoming navigation request in an onBeforeUnload event handler by repeatedly submitting a navigation request to a site responding with \204 - No Content." Navigating to a No Content site is effectively a NOP, but flushes the request pipeline, thus canceling the original navigation request. Here is sample code to do this:

```
var preventbust = 0
window.onbeforeunload = function() { killbust++ }
setInterval( function() {
  if(killbust > 0){
  killbust = 2;
  window.top.location = 'http://nocontent204.com'
  }
}, 1);
<iframe src="http://www.victim.com">
```

Exploiting XSS filters

IE8 and Google Chrome introduced reflective XSS filters that help protect web pages from certain types of XSS attacks. Nava and Lindsay (at Blackhat) observed that these filters can be used to circumvent frame busting code. The IE8 XSS filter compares given request parameters to a set of regular expressions in order to look for obvious attempts at cross-site scripting. Using "induced false positives", the filter can be used to disable selected scripts. By matching the beginning of any script tag in the request parameters, the XSS filter will disable all inline scripts within the page, including frame busting scripts. External scripts can also be targeted by matching an external include, effectively disabling all external scripts. Since subsets of the JavaScript loaded is still functional (inline or external) and cookies are still available, this attack is effective for clickjacking.

Victim frame busting code:

```
<script>
if(top != self) {
  top.location = self.location;
}
</script>
```

Attacker:

```
<iframe src="http://www.victim.com/?v=<script>if''>
```

The XSS filter will match that parameter "<script>if" to the beginning of the frame busting script on the victim and will consequently disable all inline scripts in the victim's page, including the frame busting script. The XSSAuditor filter available for Google Chrome enables the same exploit.

Clobbering top.location

Several modern browsers treat the location variable as a special immutable attribute across all contexts. However, this is not the case in IE7 and Safari 4.0.4 where the location variable can be redefined.

IE7 Once the framing page redefines location, any frame busting code in a subframe that tries to read top.location will commit a security violation by trying to read a local variable in another domain. Similarly, any attempt to navigate by assigning top.location will fail.

Victim frame busting code:

```
if(top.location != self.location) {
  top.location = self.location;
}
```

Attacker:

```
<script> var location = "clobbered";
</script>
<iframe src="http://www.victim.com">
</iframe>
```

Safari 4.0.4

We observed that although location is kept immutable in most circumstances, when a custom location setter is defined via defineSetter (through window) the object location becomes undefined. The framing page simply does:

```
<script>
  window.defineSetter("location" , function(){});
</script>
```

Now any attempt to read or navigate the top frame's location will fail.

Restricted zones

Most frame busting relies on JavaScript in the framed page to detect framing and bust itself out. If JavaScript is disabled in the context of the subframe, the frame busting code will not run. There are unfortunately several ways of restricting JavaScript in a subframe:

- **In IE 8:**

  ```
  <iframe src="http://www.victim.com" security="restricted"></iframe>
  ```

- **In Chrome:**

  ```
  <iframe src="http://www.victim.com" sandbox></iframe>
  ```

- **In Firefox and IE:** Activate designMode in parent page.