# On the fragility and limitations of current Browser-provided Clickjacking protection schemes

Sebastian Lekies
*SAP Research*
*sebastian.lekies@sap.com*

Mario Heiderich
*University Bochum*
*mario.heiderich@rub.de*

Dennis Appelt
*SAP Research*
*dennis.appelt@sap.com*

Thorsten Holz
*University Bochum*
*thorsten.holz@rub.de*

Martin Johns
*SAP Research*
*martin.johns@sap.com*

## Abstract

An important and timely attack technique on the Web is *Clickjacking* (also called *UI redressing*), in which an attacker tricks the unsuspicious victim into clicking on a specific element without his explicit knowledge about where he is actually clicking. In order to protect their websites from being exploitable, many web masters deployed different countermeasures to this kind of attack.

In this paper, we explore the limitations and shortcomings of current anti-clickjacking approaches and present several bypasses of state-of-the-art tools, including an attack we call *Nested Clickjacking* that enables us to perform Clickjacking against the social network Google+. Furthermore, we present the results of a large scale empirical study on the usage of current anti-clickjacking mechanisms on about 2 million web pages. The results of our analysis show that about 15 % of the analyzed web sites protect themselves against Clickjacking.

## 1 Introduction

In the last several years, we have observed a move from server-side attacks to client-side attacks. Especially the web browser has become an attractive target for attackers and many different attack vectors ranging from Cross-Site-Scripting (XSS) over Cross-Site Request Forgery (CSRF) to pharming attacks and many others have been developed (see for example [2, 11–14, 21]). Since being introduced by Hansen and Grossman in 2008 [8], the Web attacks which are summarized under the term *clickjacking* (also called *UI redressing* [35]) have received considerable attention both from attackers as well as defenders. The basic idea behind such attacks is that an adversary can lure an unsuspicious user into interacting with an authenticated user interface to conduct state changing actions or to extract sensitive information without the user's consent. This is typically achieved by changing the user interface in such a way that the user is tricked into clicking on a specific element without his explicit knowledge about where he is actually clicking.

Clickjacking is actually an important problem in practice. For instance, only recently Facebook opened a lawsuit against a company that is suspected to conduct clickjacking attacks against Facebook users. Solely in this case the adversaries were able to earn about $1,200,000 per month according to estimations by Facebook [6]. Clickjacking attacks are often reported in practice and typically social media sites are targeted by these attacks. In order to counter the growing menace imposed by Clickjacking, several protective measures such as for example JavaScript-based frame-busting [30] or the `X-Frame-Options` header [22] were proposed.

Unfortunately, these mechanisms are only able to protect against attacks in some cases, but not in others as we will show in this paper. We have performed an empirical analysis of the currently deployed anti-clickjacking mechanisms in respect to their flexibility, ability to protect against novel attack variants, and their current limitations. We were able to identify several weaknesses of the current state-of-the-art tools that enable us to perform Clickjacking attacks despite deployed countermeasures. For example, we introduce an attack vector called *nested Clickjacking* that allows Clickjacking via cross-domain framing despite of a deployed `X-Frame-Options` response header field.We successfully performed such an attack against the social network Google+ which demonstrates the practical relevance of these attacks. To assess the attack surface and to examine the usage of Clickjacking protection mechanisms in the wild, we also performed a large-scale study in which we analyzed more than 2 million web pages and studied how widely deployed anti-clickjacking mechanisms are. We found that from the investigated Top 20,000 Alexa web sites almost three thousand domains utilize at least one approach for defending against such attacks. The majority of sites use some kind of frame busting code, which unfortunately does not offer a robust mitigation.

In summary, we make the following key contributions in this paper:

- We perform an in-depth analysis of the current methods to counter Clickjacking attacks. To this end, we consolidate existing shortcomings of state-of-the-art tools and introduce novel attacks, including *Nested Clickjacking*, which we successfully demonstrate against Google+. Furthermore, we present bypasses of the *ClearClick* technique introduced by NoScript [20] and an implementation problem of Safari's HTML5 offline application cache that leads to a circumvention of HTTP header-based protection schemes such as the X-Frame-Options header.
- To study the deployment of current countermeasures, we conducted an empirical study of more than 2 million web pages with regards to Clickjacking protection mechanisms. We found that many sites actually use current anti-clickjacking mechanisms which unfortunately have their specific limitations.

The rest of the paper is structured as follows. While we outline the basic attack scenario in Section 2, Section 3 cover the latest Clickjacking protection mechanisms. In Section 4 we then show that the presented mechanisms are either not applicable in many environments or can be circumvented due to technical flaws. Finally, in Section 5 we investigate the usage of Clickjacking protection among the Alexa Top 20,000 websites in order to gain insights into the threats imposed by Clickjacking and ineffective protection capabilities.

## 2 Technical Background: Clickjacking

*Clickjacking* (also called *UI redressing*) [4, 8, 29, 35] is an attack that lures an unsuspicious user into clicking on an element that is different to what the user perceives to click on. The attacker tricks the user to perform such a click, often in conjunction with an authenticated user interface to conduct state changing actions or to extract sensitive information without the user's consent. This kind of attack was first reported in June 2002 by Ruderman, who noted in the Mozilla bug tracking system that transparent iframes can lead to security problems [29]. In 2008, Hansen and Grossman rediscovered this problem and coined the term *Clickjacking* [8]. They presented several attack vectors and this work was the starting point for more research in this area. Callegati and Ramilli provide an overview of the attack concept and discuss implementation details [4].

In general, a typical clickjacking attack can be broken down into three different steps:

1. Forcing the browser to render a third party UI to which the user is authenticated (i.e., where the user is able to access sensitive information or to conduct state changing actions) within a container that is controlled by the attacker. Examples for such a container are popup windows and the iframe element (see Section 4 for more details).

2. Disguising the third party UI in a fashion that the user is unable to recognize it. For instance, this can be achieved by making it transparent, by totally or partly covering it with other elements, by reducing its size, or by displaying it for only a very short amount of time (e.g., through rapid page navigation).

3. Luring the user into interacting with the disguised UI *without* the user being aware of the presence of the third party UI.

Contrary to the popular belief, Clickjacking attacks are not limited to the use of invisible iframes, but can be conducted in a variety of different ways. The dangerous nature of Clickjacking lies in the fact that it can be conducted against web applications that are free of any technical flaws since the main attack is performed against a web browser, in which the victim clicks on an element without realizing that he in fact interacts with another website. Furthermore, it is in general harder to detect on the server-side whether a request was conducted by the user intentionally or whether the user was tricked into clicking a certain UI element. Nevertheless, server-side approaches have the appealing advantage that they enable an administrator to deploy a protection mechanism for the whole site, without caring of individual web browsers, and thus they are widely used in practice.

An exemplary Clickjacking attack could therefore be executed as follows: an attacker creates a website called `funnykittengame.org` which contains a game that lures an unsuspicious user into interacting with the website. While the user plays the game, the attacker's website dynamically creates a transparent iframe element pointing to the user's banking application at `secure-banking.com`. During the game, the attacker places the iframe directly underneath the user's cursor. As the iframe is transparent, the user is not able to spot the iframe and therefore believes that he is still interacting with the game UI. However, when he clicks the mouse the next time, the corresponding click is not sent to the game UI, but to his banking application and can cause state changes on that site. More technical details and examples are available in the literature [4, 8].

## 3 Current Defenses Against Clickjacking

After having introduced the necessary background information, we now consolidate the state-of-the-art of combating Clickjacking attacks and review related work in this area. In addition to server-side approaches, which are the main focus of this paper, we also discuss client-side protection to provide a comprehensive overview of the defensive landscape. This serves as a basis for the attacks we introduce in the next section.

### 3.1 Client-side Approaches

**NoScript ClearClick:** One client-side approach to mitigate the risk imposed by Clickjacking is the Firefox extension NoScript, created and maintained by Maone [20]. This tool provides an additional array of protective features for Firefox users, including a configurable, domain-sensitive and selective script execution blocker, mechanisms to enforce secure transport protocols, a tool to prohibit access for websites to possible Intranet origins, and an add-in called *ClearClick*. This tool aims for Clickjacking protection by extending the browser's functionality and this is achieved in the following way: *ClearClick* monitors clicks and other user interactions occurring on a website framed by a page from a different origin. Once a user attempts to click a link on a framed website that appears to be victim of obfuscation attempts from its parent and cross-origin document, the interaction will be blocked. Consequently, a warning dialog will be shown, requiring the user to perform at least two confirming interactions to finally allow the possibly malicious process.

Our research shows that *ClearClick* can be circumvented and we presented in the next section an attack that involves drag&drop interaction. Another attack vector was recently presented by Kotowicz et al., that also demonstrates the limitations of *ClearClick* [15]. In contrast, our approach enables a robust handling and prevention of Clickjacking attacks.

**Alternative Browser Designs:** There are alternative browser designs like Gazelle [33], OP [7], or the secure web browser [10] that propose novel designs of web browsers that take security considerations into account. While such clean-slate approaches offer nice properties and can also protect against many kinds of attacks, we aim at techniques that are deployed in practice today.

### 3.2 Server-side Approaches

Due to the fragile nature of client-side approaches, a complimentary line of work attempts to solve the problem at the server-side as we discuss in the following.

**Frame Busting:** So called *frame busting* was one of the first countermeasures that was deployed against Clickjacking. The basic idea behind frame busting is to avoid the unauthorized framing of a web page. This is achieved using a small snippet of JavaScript code, which first checks if the page that contains the script is currently framed. If this is the case, the script navigates the top document of the framing hierarchy away from the framing page towards the URL of the script's including page, effectively "busting out" of the frame [30]. Listing 1 depicts an example of framebusting code. Note that many other ways exist to bust out of a frame, but the basic idea is mostly the same.

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

Listing 1: Example for JavaScript framebusting code.

Unfortunately, the vanilla frame busting technique, as shown above, exposes flaws that can be potentially utilized to circumvent the protection mechanism [30]. Such circumvention can be achieved through two distinct techniques. First, the adversary can attempt to avoid the navigation of the top window, e.g., via *204 flushing* [5], double framing [30], or by simply asking the user not to leave the website via the `onbeforeunload` event. Second, the attacker can try to disable the frame busting code itself, for example, by misusing modern XSS-Filters [25] or by using sandboxed iframes [34].

After an investigation of the frame busters used by the Alexa Top 500 websites, Rydstedt et al. proposed an improved frame busting code that avoids the identified weaknesses [30]. The main idea of their approach is to disable the rendering process unless the framing test was successfully executed. This can be implemented by dynamically setting cascading style sheets (CSS) properties (see Listing 2 for details). At the time of this writing, this method represents (to the best of our knowledge) the most secure frame busting solution [26]. However, Clickjacking attacks can also be carried out in other ways as we show in the next section and thus frame busting does not offer a full protection.

```
<style>
  body { display:none;}
</style>
<script>
  if (self == top) {
    document.getElementsByTagName("body")
        [0].style.display = "block";
  } else {
    top.location = self.location;
  }
</script>
```

Listing 2: Improved framebusting code from [30]

**X-Frame-Options:** Microsoft introduced the so called `X-Frame-Options` header in order to counter the

growing threat imposed by Clickjacking [22]. Similar to frame busting, the `X-Frame-Options` header also aims at preventing framing. However, it does not rely on JavaScript, but instead it is implemented as a native capability of the web browser. By attaching an `X-Frame-Options` HTTP response header to an outgoing request, a web server can influence the framing behavior of the corresponding document. The header's value can take one of two different tokens: `DENY` and `SAMEORIGIN`. While `DENY` prevents the browser from rendering the document within a frame completely, `SAMEORIGIN` allows the browser to display a resource within a frame whenever the top frame was served by the same origin. Although, the `X-Frame-Options` header is nowadays supported by every major browser, it is still not standardized, leaving room for browser incompatibilities. For instance, Internet Explorer supports a third header value called `ALLOW-FROM`, which takes exactly one additional parameter, specifying a domain which is allowed to frame the delivered resource [17].

**Content Security Policy (CSP):** The *Content Security Policy* is a mechanism that was developed to mitigate the risks imposed by content injection vulnerabilities such as cross-site scripting [32]. The mechanism itself consists of a declarative policy that is deployed on the server side and enforced by the client. Thereby, the policy contains a set of directives that restrict the functionality of a web application to the minimum that is required to run the application within the browser. Earlier revisions of CSP contained a directive called *frame-ancestors*, which aims at combatting Clickjacking. Originally, this directive allowed developers and site administrators to supply a list of comma separated domains including wildcard identifiers.

However, the current revision of the Content Security Policy specification does not cover this directives addressing protection from frame-based Clickjacking attacks anymore. Based on several problems with CSP's implementation, the directive was flagged deprecated and cannot be found in current versions of the specification draft. The Internet Draft by Ross on HTTP Header Frame Options and the W3C Editor's Draft by van Kesteren on the `From-Origin` Header are currently discussed as possible successors for either the `X-Frame-Options` header and the CSP `frame-ancestors` directive [28, 31]. However, at the time of writing, it still remains unclear what approach will be used eventually.

# 4 Open Issues With Clickjacking

While the countermeasures presented in Section 3 are able to prevent standard attack scenarios, they typically fail in more complex situations. In this section, we provide a comprehensive review of their current limitations: For one, we describe application scenarios in which the current mitigation strategies are not applicable. Furthermore, we cover and introduce sophisticated attacks that are able to circumvent the current protective approaches.

## 4.1 Protection Despite Framing

The core assumption of the current defense mechanisms is that Clickjacking can be avoided by preventing cross-domain framing of web content. As a result, each of the described server-side mechanisms provides two different configuration options: either framing of a webpage can be forbidden completely, or framing can be limited to same origin pages only.

However, this is too limiting in practice for various use cases in which framing is an essential aspect. Especially in commercial environments, such as corporate portal solutions or online advertisement, cross-domain framing is often required and essential. Hence, the current anti-framing-based solutions are not applicable here. Nevertheless, those pages can still be a valuable target for Clickjacking attacks due to their often sensitive nature.

## 4.2 Double Clickjacking

Besides utilizing cross-domain iframes, there are further, lesser known ways to conduct Clickjacking attacks. One such method is *Double Clickjacking* [9]. Instead focusing on frames, double Clickjacking relies on opening the cross-domain content in a new window. More precisely, the attacked page is opened within a "pop-under" window, i.e., a window that is hidden under the main window immediately after it was created via JavaScript. In general, this process is so fast that the user is not able to spot the window and its content before it is hidden. After opening such a window, the attacker's page lures the user into double clicking an element on his page (e.g., by letting the user play a game). The first click hits the element on the attacker's page. This action immediately brings the pop-under window to the front (and the clickable element directly under the user's cursor). The second click then hits the attacked page and triggers the targeted state changing action. Finally, after a very short amount of time, the attacker's page then hides and closes the pop-under window again, preventing the user from recognizing the true nature of the attack.

## 4.3 Clickjacking via History Navigation

A second Clickjacking attack method that does not rely on framing was presented by Zalewski [36]. It utilizes JavaScript's ability to navigate forward and backward within the browser window's history via the

JavaScript `history` object. When a victim visits the attacker's page, the attacker opens another window (via `window.open()`) containing the page that is being attacked. As the attacker's main page receives the window handle from `window.open()`, it is able to instruct the pop up to navigate to arbitrary URLs. Immediately after opening it, the main page triggers the other window to navigate away from the attacked page to another attacker controlled site. Due to the fact that the attacked site was opened first, the currently opened site is able to navigate back to it via `history.back()`. Now the attacker again lures the victim into clicking on different elements of his page. In the right moment (i.e., just before the user clicks) the page calls `history.back()`. Therefore, the browser window navigates back and the click hits the attacked page that is immediately loaded from cache. Right after the click the main window again triggers a navigation, so that the user is not able to recognize that his click was hijacked. As for the attack presented before, the current protection mechanisms are not effective at all against this attack.

## 4.4 Nested Clickjacking

During our research, we explored how current web browsers handle sites that carry an `X-Frame-Options` header. We identified a vulnerability within the `X-Frame-Options` mechanism that allows Clickjacking via cross-domain framing despite of a deployed `X-Frame-Options` response header field.

**Attack Description:** The root cause of this vulnerability is the way in which browsers verify the frame origin when the `X-Frame-Options` header is set to `SAMEORIGIN`. In that case, browsers only compare the origin of the framed page to the origin of the top window. Thereby, the top window is not necessarily the window that embeds the frame, but only the topmost window within the framing hierarchy. In between the framed page and the top window, there could be multiple other frames of different origins. If an attacker controls a framed page he is, thus, able to conduct Clickjacking attacks against the embedding page despite of a deployed `X-Frame-Options:SAMEORIGIN` header. Due to the nature of this attack, we call it *Nested Clickjacking*. To illustrate the attack we now cover a real-world attack scenario.

**Real-World Example – Exploiting Google+ via Google's Image Search**[1]**:** To verify the validity of our finding, we examined web applications that both utilize frames themselves and use the `X-Frame-Options` header. This way, we discovered a Clickjacking vulnerability on `google.com`. On the main page Google offers a feature that allows an authenticated user to share content on Google+. If unprotected, this feature could for example be abused to trick a user into sharing photos from his cellphone via Clickjacking. Hence, to prevent abuse, Google deployed `X-Frame-Options:SAMEORIGIN` headers on this site.

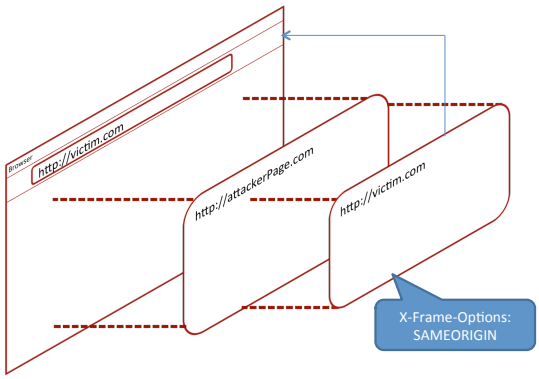


Figure 1: Illustration of Nested Clickjacking

Although direct framing of Google's homepage is now forbidden to third-party websites, it is still possible to clickjack the "share" feature via Goggle's Image Search, which utilizes iframes to display image search results in the context of their original page. The image search is either available through `images.google.com` or via `google.com/imgres`. While the domain `images.google.com` is different from `google.com`, the URL `google.com/imgres` can be used to smuggle an attacker controlled frame onto the `google.com` domain: the attacker uploads a random image to a web page that is indexed by Google's search engine. When searching for the image, Google previews the attacker controlled site within the preview frame. The attacker is now able to include a frame pointing back to `google.com` on his site (see Figure 1 for an illustration). As the top window's domain is also `google.com`, the browser will not stop the rendering process despite of the deployed `X-Frame-Options` header. Hence, the attacker is now able to conduct Clickjacking attacks by luring a user onto the search result page.

**Assessment:** At first sight, it seems to be unlikely that an attacker is able smuggle a frame onto a third-party website, but there are multiple ways to realize such a scenario. On the one hand, an adversary could trigger the creation of a frame element himself or he could take control of an already existing one. The former can be achieved by utilizing a method such as shown in the previous example or by misusing an HTML injection vulnerability at the target site where XSS is not exploitable due to an XSS filter or any other circumstances. The later can also be achieved in multiple ways. An attacker could for example buy large-scale banner advertisements

[1]This issue has been reported to Google

that are loaded within an iframe on a vulnerable websites. Furthermore, the attacker could misuse a persistent XSS vulnerability at a page that is framed by the target website. This is important in practice as it demonstrates how a vulnerability at one site can suddenly open a Clickjacking vulnerability at another site that would otherwise be protected by the `X-Frame-Options` header.

In order to avoid Nested Clickjacking vulnerabilities a Web site could utilize *Deny* instead of *Same-Origin* for its security sensitive Web pages. However, this is only possible if those sensitive Web pages are not subject to framing within the Web application.

## 4.5 Circumventing Header-based protection mechanisms in Safari

Additionally to the nested Clickjacking vulnerability, we discovered a way to remove custom HTTP response headers within Safari for HTML5 offline applications[2]. HTML5 offers a new mechanism that allows a Web application to cache certain resources for offline usage. In order to do so, the application needs to specify a manifest file that tells the browser to store certain HTML documents within the so called application cache[3]. Whenever an HTTP request is conducted towards a cached document, it is not loaded via HTTP but directly loaded from the App Cache (even if an internet connection is available). Hence this feature can be used to reduce bandwidth consumption and to make Web pages offline available.

During our research we discovered that Safari's App Cache only stores the HTML body of the cached document; HTTP response headers are simply dropped. So, whenever a document is loaded from App Cache custom HTTP response headers such as the X-Frame-Options header are not present although the original document carried these headers. Hence any header-based (security) mechanism is useless when used in conjunction with the HTML5 Offline Application feature within Safari.

Given Apple's (and therewith Safari's) market leadership with mobile devices and the fact that offline features are often deployed in mobile versions of Web applications [16], this behavior represents a critical and potentially widespread vulnerability. For example, both GMail and Hotmail utilize X-Frame-Options in conjunction with HTML offline features in their mobile versions. Hence, both applications are vulnerable to Clickjacking when accessed via the mobile Safari browser.

[2]This issue has been reported to Apple. However, at the time of this writing a fix was not available yet

[3]Note: the App Cache is different from the standard browser cache

## 4.6 Circumventing ClearClick

Finally, we studied the NoScript ClearClick behavior to evaluate its robustness in protecting users from classic Clickjacking attacks. During our research, we noticed the following classification features that ClearClick allies to tell potentially malicious interactions apart from benign framing and element overlapping:

- ClearClick analyzes whether an element (collection) containing an Iframe, object, or embed element overlaps with any other element that is receiving a mouse or keyboard event. This is based on the principles of classic Clickjacking attacks discussed in Section 3. Several checks are being applied on behalf of the overlapping element, to help ClearClick tell apart potentially malicious intent from benign interaction.

- ClearClick analyzes the opacity value of the element and checks for a specific threshold. Once the opacity has reached a value below 0.3, the click on elements framed by the overlapping element is being considered a possible Clickjacking attack and a warning dialog is displayed. In general, ClearClick attempts to judge upon the overlapping element visibility determined by opacity, size and other factors to decide whether to display the warning and block the click or not.

In essence, ClearClick attempts to find out, if the element receiving the click is by any chance visible to the user. If the element is hardly visible or not visible at all, ClearClick assumes an attack scenario and informs the user with the mentioned dialog window. The click is being prevented, the event interrupted.

One exploit to bypass ClearClick has recently been developed by Kotowicz et al. that exploits the visibility check by hiding the clicked elements in unexpected ways [15]. The bypass technique is based on a variation of the formerly published CursorJacking attack by Bordi [3]. By using the CSS directive `cursor:none`, the cursor is being hidden and replaced by a fake JavaScript based representation that appears at a different location having the unsuspecting user think it is the actual pointer. The actual cursor nevertheless still exists, it is simply invisible for the victim. By pointing the fake cursor over a link or button and performing a click, the victim will unknowingly click with the invisible cursor pointing to a different element than the visible fake cursor. A proof of concept website has been developed and made publicly available [15].

We also developed a bypass of ClearClick that involves the necessity for a drag&drop interaction by the victim. The attacker can trick the victim into dragging

an element into another element and drop it there. After that, the click needs to be initiated. The workings of this exploit are based on the fact that Firefox allows dragging embedded style elements. The unsuspecting user will drag a CSS *style* element into an Iframe, where it potentially sets all elements of the framed document to a state of opacity. Note that the framed website itself is applied with an opacity of zero to be hidden, but since no visibility impairing measurements have been applied for the Iframe *itself*, ClearClick must assume a legitimate action based on its heuristic and detection rules. Therefore the click will not be stopped and the Clickjacking attack can be performed successfully. Both attacks have been reported to Maone, and are being prevent since NoScript versions 2.2.7 and 2.2.8. [19]. The code snippet shown in Listing 3 demonstrates our bypass based on the CSS drag & drop vulnerability.

```
// evil.com
<script>
  window.onload = function()
    document.execCommand('SelectAll', null,
        true)
</script>
<h2 contenteditable=true
  ondragend=test.style.opacity=1>
  Drag <style>#foo *{opacity:0} #target{
      position:
    absolute;top:0;left:0;height:150;
    width:300;display:block}</style> Me
</h2>
<object id=test
  style=overflow:none;opacity:.1;
  data="http://victim.com/"></object>

//victim.com
<body id=foo>
  <h1 contenteditable=true>drop me here</h1>
  <h1><a id=target href="http://www.test.de
      /">
  CLICK</a></h1>
</body>
```

Listing 3: Bypassing NoScript ClearClick with Drag&Drop CSS

Ultimately, we discovered a third novel attack vector against ClearClick, which is utilizing a quirky behavior of the Gecko rendering engine when handling invalid SVG filter URIs applied via CSS. Once the filter URI given via CSS is pointing to an invalid or nonexisting SVG filter set, the element requesting those filters is being rendered invisible. For ClearClick, opacity and dimensions as well as other criteria are not matching the Clickjacking detection rule-set; therefore clicks on an element made invisible via invalid SVG filters appeared perfectly visible for ClearClick and accordingly no Clickjacking alert was raised. Listing 4 shows an example attack vector to demonstrate the issue. We submitted this bug to the NoScript author who created a fix within few hours; the problem has been marked resolved with NoScript 2.3.1rc3. Note that this vulnerability, unlike the aforementioned ones, neither requires exotic user interaction nor displays any visible traces of the attack before it occurs.

```
<style>
iframe{
    height: 100px;
    width: 100px;
    opacity: .3;
    filter: url(invalid);
}
</style>
<iframe src="http://example.com/victim.html
    "></iframe>
```

Listing 4: Bypassing NoScript ClearClick with invalid SVG filters

# 5 An Empirical Study on Clickjacking Protection in the Wild

To investigate the perceived threat imposed by Clickjacking, we examined the usage of Clickjacking protection mechanisms in the wild. The usage of protection mechanisms will yield insights into how web masters assess the risk imposed by this kind of attack. If the risk is considered high, we expect that more websites use some kind of protection measures. In several related studies, different researchers analyzed the prevalence of malicious web pages and drive-by downloads attacks on the Internet [18, 23, 24, 27]. Closely related to the work presented in this section is a study by Balduzzi et al., who introduced a system to automatically detect Clickjacking attacks and analyzed over one million unique web pages for such attacks [1]. Their system simulates user clicks on all clickable elements of a given website and detects the consequences of these clicks in terms of Clickjacking attacks. We also perform a large-scale study, but study the empirical deployment of three Clickjacking protection mechanisms as discussed next. This provides a thorough overview of current mitigation techniques and extends previous studies in this area.

## 5.1 Methodology

In order to assess the current usage of Clickjacking protection mechanisms, we decided to conduct a large-scale measurement study of the *Alexa Top Sites* since these sites are an interesting target for this kind of attacks. Thereby, we were mainly interested in the following research questions when crawling these sites:

**(RQ1)** How many websites make use of frame busters, X-Frame-Options or CSP?

**(RQ2)** How many websites have cross-domain frames deployed on their websites (and hence are vulnerable to Nested Clickjacking)?

**(RQ3)** How many websites are framed by cross-domain websites and are thus not able to deploy `X-Frame-Options` header?

**Crawling Scope:** As stated by Rydstedt et al., we expect that frame busting code is often not placed on the main page of a website, but on login or password reset pages [30]. Therefore, we decided to focus on the *Alexa Top 20,000* websites and the first level subpages of each domain. The subpages where discovered by following any link on the main page that pointed towards a resource on the same domain or a subdomain of the corresponding website.

**Frame Busting Detection:** In order to detect frame busting code on a website, we conducted a simple but effective test: we simply framed the page under investigation and checked whether the top frame conducted some kind of redirect. If such a redirect was observed, we conclude that frame busting was present on the studied page.

## 5.2 Results

In this section, we present the results of our survey and discuss these results in the context of the identified research questions.

### 5.2.1 General Overview

In total, our crawling infrastructure was able to crawl 2,039,679 unique web pages, from which 139,216 (6,8 %) returned an error code or where unreachable at the time of analysis. Therefore, we were able to successfully investigate 1,900,463 web pages for Clickjacking protection mechanisms. Out of the investigated 20,000 Alexa web sites, a total of 2,975 (14.88 %) domains utilized at least one approach for defending against Clickjacking. Note that some of the sites actually utilize multiple defense mechanisms. While 972 (4.86 %) domains deployed `X-Frame-Options` headers, a total of 2,230 (11.15 %) sites utilized a JavaScript-based approach and only two (0.01 %) made use of CSP for preventing framing. Table 1 provides a summarized overview of these numbers.

| Mechanism | Pages | Websites | % Sites |
|---|---|---|---|
| X-Frame-Options | 10,982 | 972 | 4.86 % |
| Frame-Busting | 87,685 | 2,230 | 11.15 % |
| CSP | 13 | 2 | 0.01 % |

Table 1: General overview of crawling results

In the following, we discuss the results for the individual research questions mentioned above.

### 5.2.2 RQ1: Protection mechanisms

In total, we discovered that 87,685 unique web pages busted out of the frame in our testing scenario. Compared to the total number of 1,900,463 pages, this represents only 4.61 %. However, if we aggregate those numbers for each website (i.e., mainpage + subpages), we can observe that 2,230 sites (which corresponds to 11.15 % of all sites) deploy frame busters on at least one subpage. This means that web masters are not deploying frame busters throughout all of their pages, but only to certain spots. This assumption is strengthened by another observation: while 899 websites deploy frame busting directly on the main page, 1,331 only protect some of their subpages.

As already stated in Section 3.2, future versions of the Content Security Policy (CSP) will not support the anti-framing directive `frame-ancestors` any more. Therefore, it is not surprising that only two websites utilize this feature for header-based framing protection. More interesting are the numbers gained for the `X-Frame-Options` header: while we were able to identify 972 websites utilizing this header, only 265 of these sites deployed it on the main page. The vast majority of 707 sites deployed it only on some of their subpages. As discussed above, a very similar behavior can also be observed for frame busting code. This very telling data point raises an interesting question: Why are web masters not rolling out the protective measurements on a wide scale, but only on some very specific spots? Wouldn't it be easier for them to configure their web servers in a less granular fashion? One answer to this question are potential limitations of the investigated approaches: as most anti-clickjacking protections follow the "disable-framing-to-be-secure" approach, many webmasters could be forced to trade off between functionality (by enabling framing) and security (by disabling framing). Hence, protection is only applied to neuralgic points of a website, where the desired functionality can still be achieved with deployed protective measures.

### 5.2.3 RQ2 & RQ3: Framing Behavior

Besides the protection mechanisms itself, we are also interested in the framing behavior of the investigated sites.

| Value | Pages |
|---|---|
| SAME-ORIGIN | 7,906 (72 %) |
| DENY | 3,076 (28 %) |

Table 2: X-Frame-Options values

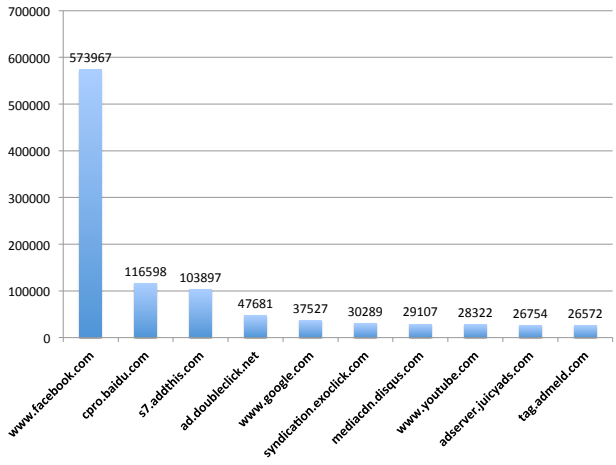


Figure 2: Top Ten of the framed domains

On the one hand, this enables us to learn more about the applicability of frame-based security solutions, and on the other hand we are able to measure the impact of the Nested Clickjacking attack presented in Section 4.4.

In total, we were able to count 4,007,176 million iframe elements on 14,449 (72.25 %) of the 20,000 Alexa sites and their subpages. About 2,812,274 (70.18 %) of those frames are pointing towards cross-domain resources (we exclude subdomain frames from this number) and hence are potentially vulnerable to the Nested Clickjacking attack. Given the fact, that 72% of all the observed X-Frame-Options header (See Table 2 for details) utilize the *same-origin* value, it is very likely that Nested Clickjacking attacks can successfully be conducted in practice.While the raw number of cross-domain iframes is surprisingly high, the number of framed web sites is small. In total, we identified 17,496 unique domains that were being framed. While these sites are not able to deploy frame-based anti-clickjacking solutions, they are also a valuable target for attackers to conduct a nested Clickjacking attack. Especially, if an attacker is able to gain control over a domain that is framed often, he is able to increase the available attack surface by the amount of frames pointing to that domain. As shown in Figure 2, the top ten domains, for example, are represented in about 36.29 % of all the investigated cross-domain iframe elements.

## 6 Conclusion

In this paper we investigated the current state-of-the-art browser-provided Clickjacking protection schemes. Clickjacking is an attack in which an adversary tricks the unsuspicious victim into clicking on a specific element without the victim's explicit consent. Thereby, the current defensive tools can be divided into client-side and server-side approaches. While we investigated NoScript Clearclick as a representative for the client-side, we conducted an in-depth analysis of Frame Busting, the X-Frame-Options header and the Content Security Policy for the server-side.

We consolidated little known bypasses and limitations and also discovered two novel attack scenarios including *Nested Clickjacking* which we successfully demonstrate against Google+, and a CSS-based bypass of the ClearClick technique introduced by NoScript.

Furthermore, we studied the deployment of these countermeasures by conducting an empirical study of more than 2 million web pages. Thereby, we discovered that many sites protect themselves against Clickjacking attacks. However, we also discovered that many websites are still susceptible to attacks due to the identified bypasses and weaknesses of the presented countermeasures. These problems clearly demonstrate that the nature of Clickjacking attacks is far more complex than previously assumed by the research community. Novel countermeasures are needed that take these more complex scenarios and attacks into consideration to successfully counter the threats imposed by Clickjacking.

## References

[1] BALDUZZI, M., EGELE, M., KIRDA, E., BALZAROTTI, D., AND KRUEGEL, C. A solution for the automated detection of clickjacking attacks. In *ACM Symposium on Information, Computer and Communications Security (AsiaCCS)* (2010).

[2] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *ACM Conference on Computer and Communications Security* (2008).

[3] BORDI, E. Cursorjacking. http://eddy.bordi.fr/securite/cursorjacking.html, February 2011.

[4] CALLEGATI, F., AND RAMILLI, M. Frightened by links. *IEEE Security & Privacy 7* (2009), 72–76.

[5] CODERRR. Preventing frame busting and click jacking (ui redressing). http://coderrr.wordpress.com/2009/02/13/preventing-frame-busting-and-click-jacking-ui-redressing, February 2009.

[6] FACEBOOK SECURITY. Facebook, Washington State AG Target Clickjackers. [online], https://www.facebook.com/notes/facebook-security/facebook-washington-state-ag-target-

clickjackers/10150494427000766, last accessed 02/12/12, January 2012.

[7] GRIER, C., TANG, S., AND KING, S. T. Secure Web Browsing with the OP Web Browser. In *IEEE Symposium on Security and Privacy* (2008).

[8] HANSEN, R., AND GROSSMAN, J. Clickjacking. [online], `http://www.sectheory.com/clickjacking.htm`, last accessed 02/13/12, August 2008.

[9] HUANG, L.-S., AND JACKSON, C. Clickjacking attacks unresolved. White paper, CyLab, July 2011. Available online.

[10] IOANNIDIS, S., AND BELLOVIN, S. M. Building a secure web browser. In *USENIX Technical Conference* (2001).

[11] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting browsers from DNS rebinding attacks. In *ACM Conference on Computer and Communications Security* (2007).

[12] JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. Protecting browser state from web privacy attacks. In *World Wide Web Conference Series* (2006).

[13] JOHNS, M., AND WINTER, J. Requestrodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference* (May 2006).

[14] KARLOF, C., SHANKAR, U., TYGAR, J. D., AND WAGNER, D. Dynamic pharming attacks and locked same-origin policies for web browsers. In *ACM Conference on Computer and Communications Security* (2007).

[15] KOTOWICZ, K. Cursorjacking again. `http://blog.kotowicz.net/2012/01/cursorjacking-again.html`, January 2012.

[16] KRUEGER, D. Create offline web applications on mobile devices with html5. `http://www.ibm.com/developerworks/web/library/wa-offlineweb/`, may 2010.

[17] LAW, E. Combating clickjacking with x-frame-options. `http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx`, March 2010.

[18] LU, L., YEGNESWARAN, V., PORRAS, P. A., AND LEE, W. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *ACM Conference on Computer and Communications Security* (2010).

[19] MAONE, G. Noscript changelog. http://noscript.net/changelog, January 2012.

[20] MAONE, G. Noscript clearclick. `http://noscript.net/faq#clearclick`, January 2012.

[21] MARTIN, M., AND LAM, M. S. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *USENIX Security Symposium* (2008).

[22] MICROSOFT. Ie8 security part vii: Clickjacking defenses, 2009.

[23] MIN WANG, Y., BECK, D., AND JIANG, X. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)* (2005).

[24] MOSHCHUK, A., BRAGIN, T., GRIBBLE, S. D., AND LEVY, H. M. A crawler-based study of spyware in the web. In *Network and Distributed System Security Symposium (NDSS)* (2006).

[25] NAVA, E. V., AND LINDSAY, D. Our favorite xss filters and how to attack them. http://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf, July 2009.

[26] OWASP. Clickjacking. https://www.owasp.org/index.php/Clickjacking#Best-for-now_implementation, July 2011.

[27] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The ghost in the browser analysis of web-based malware. In *Usenix Security Symposium* (2007).

[28] ROSS, D. draft-gondrom-frame-options-01 - HTTP header frame options. http://tools.ietf.org/html/draft-gondrom-frame-options-01#section-2.4, September 2011.

[29] RUDERMAN, J. Bug 154957 – iframe content background defaults to transparent, June 2002. `https://bugzilla.mozilla.org/show_bug.cgi?id=154957`.

[30] RYDSTEDT, G., BURSZTEIN, E., BONEH, D., AND JACKSON, C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)* (2010).

[31] V. KESTEREN, A. The From-Origin header. http://dvcs.w3.org/hg/from-origin/raw-file/tip/Overview.html, December 2011.

[32] W3C. Content security policy, November 2011. `http://www.w3.org/TR/CSP/`.

[33] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The Multi-Principal OS Construction of the Gazelle Web Browser. In *USENIX Security Symposium* (2009).

[34] WHATWG. The iframe element, 2010.

[35] ZALEWSKI, M. Arbitrary page mashups (UI redressing). [online], `http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_(UI_redressing)`, last accessed 02/12/12.

[36] ZALEWSKI, M. X-frame-options is worth less than you think. Website, December 2011. Available online: http://lcamtuf.coredump.cx/clickit/.