Tamper-Resistant LikeJacking Protection*

Martin Johns and Sebastian Lekies

SAP Security Research
Germany
http://www.websand.eu

Abstract. The ClickJacking variant *LikeJacking* specifically targets Web widgets that offer seamless integration of third party services, such as social sharing facilities. The standard defense against ClickJacking is preventing framing completely or allowing framing only in trusted contexts. These measures cannot be taken in the case of LikeJacking, due to the widgets' inherent requirement to be available to arbitrary Web applications. In this paper, we report on advances in implementing LikeJacking protection that takes the specific needs of such widgets into account and is compatible with current browsers. Our technique is based on three pillars: A JavaScript-driven visibility check, a secure in-browser communication protocol, and a reliable method to validate the integrity of essential DOM properties and APIs. To study our protection mechanism's performance characteristics and interoperability with productive Web code, we applied it to 635 real-world Web pages. The evaluation's results show that our method performs well even for large, non-trivial DOM structures and is applicable without requiring changes for the majority of the social sharing widgets used by the tested Web applications.

1 Introduction

The days, in which a single application provider provided the code, as well as, the content of a single Web application are long gone. Nowadays, mixing services by multiple parties in the context of a single Web document is the norm and not any longer the exception [23]. A major driving force of this development are seamless sharing widgets, such as *like buttons* provided by social networks like Facebook or Google Plus. These widgets allow one-click interaction with the network without leaving the context of the page which hosts the widget. Potential uses for such widgets are not reduced to social sharing but are increasingly adopted by unrelated services. For instance, the micropayment service Flattr offers similar widgets[1] to initiate payments directed to the widget's hosting page.

While significantly lowering the barrier to interact with the widget provider's services, such widgets also open the door for abuse: In the recent past, a variant of the ClickJacking [1, 7, 12] attack, aptly named *LikeJacking*, appears in the

* This work was in parts supported by the EU Project Web- Sand (FP7-256964).

[1] Flattr tools: http://developers.flattr.net/tools/

S.J. Stolfo, A. Stavrou, and C.V. Wright (Eds.): RAID 2013, LNCS 8145, pp. 265–285, 2013.

wild repeatedly [21, 28] and has received considerable attention [12, 30]. As we will discuss in Section 3.1, preventing LikeJacking attacks is non-trivial and, unlike the `X-Frames-Option`-header [20] in the case of general ClickJacking, no applicable, browser-based security measure exist.

Up to now, there is no reliable countermeasure against LikeJacking available, forcing service operators either to expose their users to the risk or to break the widget's seamless interaction model [30]. For this reason, in this paper, we investigate a protection approach that is specifically targeted at LikeJacking attacks, to mitigate this currently unsolved security problem.

Contributions: In this paper, we make the following contributions:

- We propose a novel LikeJacking protection methodology that relies only on JavaScript capabilities already present in today's Web browsers, and hence, can be adopted immediately. The proposed protection mechanism is based on JavaScript-based checking of visibility conditions (see Sec. 4) and a secure communication protocol between the protection script and the embedded widget (see Sec. 5).
- Furthermore, we present a methodology to reliably check the integrity of an existing DOM tree instance and the corresponding DOM APIs (see Sec. 6). This methodology effectively enables a JavaScript to validate its embedding DOM, even in the context of untrusted Web documents. Furthermore, we document how this technique can be implemented in a cross-browser fashion and document that the process performs well even for large DOM tree structures (see Sec. 7.2).
- Finally, as part of the protection measure's evaluation, we report on a practical study, which examines how popular Web sites handle social sharing widgets in respect to visibility properties (see Sec. 7.2).

2 Technical Background

2.1 Social Sharing Widgets

In the beginning of the Web, the content of a single HTML document was static and originated from exactly one source: the hosting server. This changed soon during the evolution of the Web. Nowadays, Web sites often include a multitude of services from many different third parties [23]. Due to the Same-Origin Policy, however, interaction and integration of such third party services is not straightforward. The technical methods of choice, for this purpose are script includes and `iframe` elements, which are nowadays omnipresent in the Web [16]. Nevertheless, when visiting a Web site, a non-technical user is not able to recognize all the iframe elements as many Web sites use this technology to seamlessly integrate third part content. Thereby, CSS style declarations are used to style the iframe elements in a way that the content of the iframe appeals to be part of the embedding page. Besides advertisement, this technique is increasingly used to provide seamless interaction capabilities between different Web applications. One such integration feature that received special attention lately are

Social Sharing widgets. These widgets can be used to share arbitrary content with your friends on your favorite social networks. Thereby, the social network provides the sharing functionality in the form of a simple Web document that can be embedded via an iframe into the page. As the social network's cookies are attached to all requests initiated by and within the iframe, the iframes UI controls and scripts act in the name of the user towards the social network. One important requirement for such a scenario is that the user is encouraged to use the widget as both, the social network and the embedding page, have an interest in the user's social interaction. Therefore, the functionality should be as easy as possible and ultimately only consist of one single click.

2.2 Click- and LikeJacking

The underlying security problem of Clickjacking was first discovered by Ruderman in 2002 [25]. In the Mozilla bug tracking system he noted that transparent iframes can lead to security problems. However, it took another 6 years until the term Clickjacking was coined by Hansen and Grossman [7].

The term ClickJacking denotes a class of attacks, that aim to trick users into interacting with cross-domain Web UIs without their knowledge. In general, ClickJacking utilizes `iframes` which are hidden to the user, using varying techniques. Instead the user is presented a completely different UI which is positioned by the attacker either over or under the iframe. Hence, when attempting to interact with the attacker's fake UI, the user is actually clicking elements in the hidden iframe. In particular, the following attack implementations have been discussed and demonstrated:

Hiding the iframe via CSS: Several CSS properties, such as `opacity` or `mask` can be used to render the target iframe completely transparent. This allows it to position the attacker's crafted GUI below the iframe. When the user tries to click the fake elements, his click is received by the overlaying iframe.

Obstructing the iframe with overlaying elements: Alternatively to an invisible iframe and underlying fake GUI, also the opposite scenario is possible: The adversary can also place his GUI elements on top of the iframe, thus, completely or partially obstructing it. In such situation, he could either cover everything but the button, that he wants the victim to click, or he could cover it completely and set the overlay's *pointer-events*-property to `none`, which causes the clicks received by the overlay to be seamlessly passed on to the underlying DOM elements, i.e., the target iframe.

Moving the iframe under the mouse pointer: Finally, the attacker could render the iframe outside of the screen's visible regions. Then, when he anticipates a click from the user, e.g., in the context of a game, he can quickly position the iframe under the user's mouse.

2.3 Countermeasure

The currently established countermease against Clickjacking is *frame busting.* The goal of frame busting is to forbid an untrusted site to frame a security sensitive Web page. This can be achieved by including a small snippet of JavaScript into the security sensitive page. The script checks wether the page is framed and if so it redirects the top browser window away from the untrusted site towards the security sensitive site effectively busting out of the frame. As shown by Rydstedt et al. [26] many problems exist with practical implementations that allow an attacker to circumvent the protective measures.

The `X-Frame-Options` response header also follows the idea of forbidding framing to third-party Web sites [20]. The mechanism is not implemented in JavaScript, but browser itself prevents the untrusted framing. Furthermore, if there is an existing trust relationship between the involved sites, a Web document can selectively allow being framed by some-origin pages or specifically whitelisted sites, using the corresponding values for the header [20].

3 LikeJacking Protection via Visibility Proofs

3.1 Problem Statement

As discussed above, all currently available ClickJacking countermeasures require a pre-existing trust relationship between the widget and the including domain. On the most basic and best-supported level, this trust relationship is limited to the widget's 'own' domain, using the `X-Frame-Option` header's `same-origin` directive. In the foreseeable future, as soon as the header's `Allow-from` option receives wider support, the widget can define a whitelist of domains that are permitted to include the widget's hosting frame.

However, in situations, in which a widget is designed to be included in arbitrary domains, as it is the case with social sharing widgets, the whitelisting approach does not work anymore. As it stands today, the widget is at the mercy of the including page: It has to allow being framed generally and has only very limited means to obtain information on the actual framing context via its `referrer` information, which is known to be unreliable [3,14].

In [12] Huang et al. propose a browser provided mechanism to ensure that visibility conditions of specified Web UI elements are ensured. Huang's core technique is currently under standardization by the W3C as a potential extension of the Content Security Policy (CSP) mechanism [19]. If this technique would receive broad browser support in the future, it could be used as a suiting mitigation strategy. Unfortunately, it is unknown if, when, and to which degree the technique will actually be implemented in the Web browsers. Similar techniques, which are discussed now for years, still have no broad browser support. For instance, the highly useful `Allow-from` directive for the `X-Frame-Options`-header, is still not fully supported by all browser, and up to now, there is no definite commitment that Internet Explorer will implement CSP. Hence, it is reasonable to assume, that native browser supported security measures will take a considerable time.

Thus, for the time being, browser-provided means do not offer the needed flexibility and security properties for the outlined Web widget use-cases. However, as motivated in the beginning of this paper, LikeJacking is a real threat today. For this reason, we investigated a solution that can be built with the means that

Web browser offer today. In the remainder of this paper we propose a solution that satisfies the following criteria:

Visibility proof: The Web widget receives validation that its UI was visible to the user during the user's interaction with the widget.

Legacy browser compatibility: The aim of the proposed technology is to provide protection today that is compatible with at least a significant majority of the currently deployed Web browsers. Thus, relying on future browser features is out of scope for this paper.

Tamper resistance: Even under the assumption, that the widget is included in an actively malicious page, the protection and validation mechanism should either hold, or in unrecoverable cases, reliably detect potentially malicious situation, so that the widget can react accordingly.

No disruption: In case of legitimate usage of the widget, the hosting page should remain as unaffected as possible.

Based on these requirements, several implementation characteristics can be deducted immediately: For one, it follows directly from the legacy browser compatibility requirement that the measure will rely on JavaScript to enforce the desired properties. Furthermore, as the visibility of the widget is governed by the hosting document, the solution's script will have be executed, at least partially, in the context the hosting page. Finally, based on these implications, the solution has to anticipate potential JavaScript-driven attacks from the hosting page, to fulfill the tamper resistance goals.

3.2 The Big Picture

In this section, we give a high level overview on our protection approach. The emphasis is on its general functionality, without going into deep technical detail.

The core of our methodology is a JavaScript library that is included in the hosting Web document (see Fig. 1). The script ensures that the widget's predefined visibility conditions are met. This is done through the utilization of DOM APIs, which provide access to the widget's rendering conditions, such as position, size or CSS properties. The specifics of this process are discussed in Section 4.

The widget itself is included in the hosting page using a standard `iframe`-element. However, all user interaction of the widget is disabled until it has been verified that the frame is clearly visible to the user.

If the JavaScript library can verify, that the visibility requirements are indeed met, the script signals the widget, that it is safe to enable user interaction (see Sec. 5). From this point on, clicks received by widget are handled seamlessly. To prevent a malicious site to alter the widget's rendering after the initial visibility check, the validation is repeated in a randomized pattern.

3.3 Security Considerations and Resulting Technical Challenges

Our system relies on running a script in the scope of a Web document that is controlled by an untrusted third party. We do not have control over when

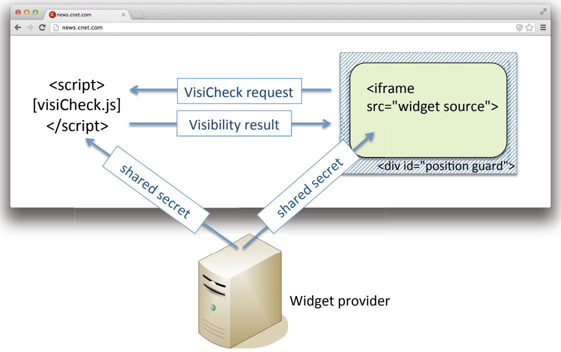


Fig. 1. Overview of the protection system

or how our JavaScript is included in the page. Thus, a potentially malicious party has the opportunity to apply changes to the DOM's global object and the corresponding DOM APIs, for instance via wrapping the APIs or creating new DOM properties, that shadow the native implementations (see Sec 6.2). Hence, under the assumption, that the integrating party (from now on "the attacker") is actively malicious, the resulting technical challenges are as follows:

(C1) No reliance on the elements in the global JavaScript scope: We cannot control when our script is included. Hence, we do not know which changes to the global scope have been conducted by the attacker.

(C1) No assumptions about the integrity of global DOM objects and methods: Due to JavaScript's highly dynamic characteristics, the attacker can overwrite all global properties, functions, and objects within the scope of the Web document, with only few notable exceptions, such as the `location` DOM object. For this reason, our mechanism cannot make any assumptions regarding the state or behavior of these objects. Instead, it has to ensure their integrity before utilization.

(C1) Careful handling of confidential data: All JavaScript in a Web document is executed in a shared global space. This means that all unscoped objects, functions, and values can be accessed by any JavaScript running in the context of the document. In case data values exist that have to be kept secret from the attacker, precautions have to be taken to avoid information leakage.

In Section 6, we discuss how our solution ensures the integrity of the required DOM APIs as well as how sensitive information are kept out of the attacker's reach.

3.4 A Defensive UI Interaction Strategy to Prevent LikeJacking

Based on the reasoning above, we now define our proposed UI interaction strategy for Web widgets:

The widget allows seamless user interaction only when the following conditions are satisfied:

1. *The predefined visibility conditions have been successfully checked.*
2. *The integrity of the required DOM APIs, which are needed to execute the visibility check, has been verified.*
3. *Both condition above have to be fulfilled for at least a pre-defined timespan before the actual user interaction happens (e.g., 500 ms), to avoid quick property changes through the adversary immediately before the user interaction.*

If one of these conditions has not been met, the widget either prevents user interaction or executes a secondary verification step through safe UI, such as confirmation pop-ups, Captchas, or similar measures.

In certain situations, the hosting page has legitimate reasons to temporarily violate the visibility conditions. For instance the widget could be contained in an initially hidden portion of the site, which is only visible after explicit user interaction, e.g., via hovering the mouse over a menu. For such cases, the protection mechanism provides an API to signal the widget, that its visibility condition has changed. This allows the protection script to re-execute the checking algorithm and, in case of a positive result, re-enabling direct user interaction.

4 Verifying of Visibility Conditions

In general there are four different conditions, that could lead to a DOM element not being visible to the user: Either CSS properties have been set, that cause the element to be invisible, obstruction DOM elements are rendered in front of the element, the element's rendering dimensions are reduced to a nearly invisible size, or the element's position is outside the current viewport's boundaries.

In the following sections, we discuss how these conditions can be reliably detected.

4.1 CSS-Based Visibility Prevention

Several CSS properties exist, that influence the visibility of DOM elements. See Table 1 for a comprehensive overview. For each of the properties, unambiguous visibility conditions can be defined, for instance, the condition that an element's `opacity` value has to be above a certain threshold. Checking these properties via JavaScript is possible via the `window.getComputedStyle()` API, which computes an element's final CSS property values that result after applying all matching CSS rules. While some properties are inherited directly (in our case mainly the `visibility` property), most properties have to be checked both for the element itself as well as for its direct DOM ancestor chain. With the exception of `opacity`, all checked CSS values are absolute, i.e., the element's visibility is determined through a set of enumerable options. For instance in the case of the `visibility` property, the possible values are `visible`, `hidden`, or `collapse`. As an exception, the `opacity` property value is a composite property,

Table 1. Relevant DOM and CSS properties (excluding vendor prefixed variants)

CSS Property	Check condition	Appl. elements	Method
visibility	value	element only	`getComputedStyle()`
display	value	DOM chain	`getComputedStyle()`
mask	value	DOM chain	`getComputedStyle()`
opacity	threshold	DOM chain	`getComputedStyle()`
position[a]	value	offset chain	DOM properties
dimension[a]	minimum	DOM chain	DOM properties

[a]: Values influenced by CSS and DOM position, calculated via DOM properties

that has to be calculated via multiplying the individual `opacity` values present in the element's DOM ancestor chain. If a diversion of the predefined condition for one of these CSS properties could be identified, a potential attack is flagged and communicated to the widget.

4.2 Obstructing Overlays

CSS allows the positioning of DOM elements both in a relative and an absolute fashion. This permits Web developers to create overlays in which one DOM element is rendered on top other elements. This allows the adversary to (partially) obstructed the widget with opaque overlays. Furthermore, through setting the overlay's `pointer-events` CSS property to `none`, the overlay will pass all received user interaction to the underlying element, i.e., to the widget. This effectively enables a ClickJacking condition which leaves the widget's own CSS properties untouched.

To detect such situations, all intersecting DOM elements have to be identified. To do so, the checking algorithm iterates over the embedding DOM tree's nodes and calculates the nodes' position and dimensions. For all (partially) overlapping elements, the `pointer-event` CSS property is obtained. If overlapping elements with disabled `pointer-events` could be found, a potential attack is flagged. Likewise, in the case where significant portions of the widget are obstructed by standard elements. At the first glance, this process exposes potential for a performance issue. However, due to the efficient DOM implementations of today's browsers, this process scales very well even for non-trivial DOM trees with more than several thousand nodes (see Sec. 7.2 for details).

4.3 Element Size and Position

Side effects of the DOM rendering process can also influence an element's visibility: For one, the rendered dimensions of an element are of relevance. E.g., through setting both the rendering height and width to zero the element can effectively be hidden. To avoid such conditions, the widget can define minimum value for width and height. To ensure, that the desired minimum dimensions are met, the effective size of an element has to be computed. An elements size depends on two factors: The element's own dimensions, determined through the DOM properties `offsetWidth` and `offsetHeight`, and the dimensions of its DOM ancestors, under the condition, that on of these ancestors has set its `overflow` CSS property to `hidden`. Thus, via walking through the widgets DOM ancestor chain, its effective size can be obtained.

Furthermore, the position of an element can be outside of the currently displayed viewport, hence, effectively hiding it from the user. In general, such a situation is not necessarily an indication that the page actively attempts to conceal the element. As most pages are bigger than the available screen estate, parts of the Web page are rendered legitimately outside of the current viewport. This especially holds true for page height, i.e., page regions below the currently viewed content. Hence, we have to take further measures to tell apart benign from malicious situations.

4.4 Position Guarding

As outlined in Sec. 2.2, one of the ClickJacking variants moves the click target quickly under the victims mouse pointer, just before a click is about to happen. With visibility checks at isolated, discrete points in time, this attack variant is hard to detect reliably. Hence, for position-changing based attack scenarios, we utilize an additional indicator: After the other visibility verification steps have concluded correctly, the script injects an absolutely positioned, transparent DOM overlay of it own, completely covering the widget as well as a small area surrounding it (see Fig. 1).

The overlay has the purpose to register intended interaction with the widget beforehand. This is achieved with a `mouse-over` event handler. Whenever the user targets the widget with his mouse pointer, he automatically enters the protection overlay. This causes the execution of the overlay's eventhandler. The eventhandler now conducts three steps: First, based on the received `mouse event`, it verifies that its own position within the DOM layout has not changed. Then it checks that the widget's visibility and position have not been tampered with. If these two tests terminated positively, the overlay temporarily disable its `pointer-events`, to allow interaction with the widget. Furthermore, the exact time of this event is recorded for the final verification step (see Sec. 5.3).

4.5 Unknown Attack Variants

The presented visibility checking algorithms have been designed based on documented attack methods as well as on a systematical analysis of relevant DOM-mechanisms. However, it is possible, that attack variants exist which are not yet covered by the outlined checks. Especially, the versatility and power of CSS has the potential of further, non-obvious methods to influence the visibility of DOM elements. However, due to the nature of such attack variants, they will in any case leave traces in the involved elements' DOM or CSS properties. Thus, it can be expected that adding checks for these indicators will be straight forward. Furthermore, as the overlay-checking step (see Sec. 4.2) already requires probing properties of all DOM elements, newly discovered characteristics that need to be validated, should at worst add a linear factor to the performance overhead.

5 Trusted Communication between the Protection Script and the Widget

As motivated in Section 3.4, initially the widget disables all direct interaction, until the visibility verification script in the hosting page sends the signal, that all required conditions have been met. In this section, we outline this communication channel's implementation. As the protection script runs in an untrusted context, specific measures have to be taken to ensure message integrity and authenticity. For this purpose, we rely on two language features of JavaScript: The `PostMessage`-API and local variable scoping.

5.1 PostMessage

The *PostMessage API* is a mechanism through which two browser documents are capable of communicating across domain boundaries in a secure manner [27]. A PostMessage can be sent by calling the method `postMessage(message, targetOrigin)` of the document object that is supposed to receive the message. While the `message` attribute takes a string message, the `targetOrigin` represents the origin of the receiving document.

In order to receive such a message, the receiving page has to register an event handler function for the "message" event which is triggered whenever a PostMessage arrives. Particularly interesting for our protection mechanism are the security guarantees offered by this API:

1. *Confidentiality:* The browser guarantees that a PostMessage is only delivered to the intended recipient, if the `targetOrigin` specified during the method call matches the recipient window's origin. If confidentiality is not required, the sender may specify a wildcard (*) as `targetOrigin`.
2. *Authenticity:* When receiving a message via the event handler function, the browser additionally passes some metadata to the receiving page. This data includes the origin of the sender. Hence, the PostMessage API can be used to *verify* the authenticity of the sending page.

Effectively, this implies that whenever a widget receives a PostMessage from it's embedding page, it is able to obtain reliable information about its embedding context.

5.2 Information Hiding via Closure Scoping

In general, the protection scripts runs in the origin of the adversary's page. Hence, according to the JavaScript's Same-origin Policy, his scripts have unmitigated access to the shared global object space. Thus, all potentially secret

Listing 1 Anonymous function creating a closure scoped shared secret

```
// Anonymous function without reference in the global object
(function(){
   // Constructor for the checker object
   var VisiCon = function(s){
      var secret = s;  // not visible outside of the object
      [...]
  }
  // Store the secret upon initialization in the closure
  window.VisiChecker = new VisiCon([[...shared secret...]]);
  ...
})();
```

information, such as shared secrets between the protection script and the widget have to kept out of reach for the adversary's code. As Crockford has documented [5], this can be done with JavaScripts closure scoping. All information stored in closures, such as the `VisiCon` object in Lst. 1, are not accessible from the outside. Furthermore, as the encapsulating anonymous function leaves no reference in the global scope, its source code cannot be accessed via `toString()` and, hence, the secret value is effectively kept out of reach for the adversary.

5.3 Resulting Communication Protocol

The protection script is implemented in the form of an anonymous function as depicted above (see Lst. 1). Encapsulated in this function is a secret value, which was provided by the script's host and is shared with the widget. This value will be used to prove the script's authenticity to the widget (see Fig. 1).

Upon initialization, the protection script retrieves the widget's iFrame element from the DOM and conducts the visibility verification process. After successful completion of visibility (see Sec. 4) and DOM integrity (see Sec. 6) checks, the script sends a `postMessage` to the widget with the signal, that it is safe to enable user interaction. Included in this message is the shared secret, to proof the messages authenticity. This approach is secure, as the `PostMessage`-API guarantees that only scripts running in the widget's origin can read the message and the shared secret is kept in a closure with no connection to the global object.

From this point on, the protection script re-executes the visibility and integrity checking process at randomized times, to detect if the widget's visibility or position have been actively tampered with after the initial positive validation.

Finally, a concluding `PostMessage` handshake is conducted when the widget receives actual user interaction, e.g., through clicking: Before acting on the click, the widget queries the protection script, to ensure that the visibility and integrity properties have not been violated in the meantime. As the widget's position guard (see Sec. 4.4) must have been triggered right before the interaction with the widget occurred, this information is fresh and reliable. In case the guard has not been triggered, this is a clear indication that the widget has been moved since the last periodic check, which in turn is a clear sign of potentially malicious actions. Only in case that the guard has been triggered and the visibility conditions are intact, the protection script answers the widget's enquiry. In turn, the widget only directly acts on the click, if this answer was received.

6 Validating DOM Integrity

6.1 Redefinition of Existing Properties and APIs

JavaScript is a highly dynamic language, which allows the redefinition of already existing elements and methods. This can be done in two fashions: For one an element can be redefined through direct assignment. Alternatively, `Object.defineProperty` can be utilized to change properties of existing objects. The latter method cannot only redefine the behavior of methods, but also of object properties, through the definition of the internal [[Get]], [[Set]], and [[Value]] properties. In addition, setting its internal property [[Configurable]] to `false` prevents deletion and further changes.

6.2 Resulting Potential DOM Integrity Attacks

Redefinition of existing methods and properties is not restricted to objects that have been created through script code. Also the Web browser's native APIs and objects can be changed this way. It is possible to overwrite global APIs, such as `alert()`, with custom functions. It has been shown in the past, how this technique can be used to detect [2] and mitigate [8,17,24] XSS attacks.

However, in our case, the adversary could potentially use this technique to obfuscate LikeJacking attempts. As discussed in Sections 4 and 5 our system relies on several native DOM APIs, such as `window.getComputedStyle()` and properties of DOM elements, such as `parent` or `offsetWidth`. Through redefining these DOM properties to return false information, the attacker can effectively undermine the visibility check's correctness.

Challenge: Validating DOM Integrity. To ensure the correctness of the visibility checking algorithm, we have to conduct two steps: For one, we need to compile a complete list of all native APIs and DOM properties which are used by the process, including the applicable checking scope (see Table 2). Secondly, for each element of this list, a reliable methodology has to be determined, which validates that the method or property has not been redefined by the adversary.

6.3 Built-In Objects and the Semantics of the `delete` Operator

To handle potential DOM tampering attacks, JavaScript's `delete` operator plays a central role. In [17] Magazinius et al. noted, that redefined DOM APIs revert back to their original state if they are *deleted*. The reason for this lies in the method how native DOM elements and APIs are exposed to the JavaScript:

Table 2. List of required DOM APIs and properties

Name	Type	Checking scope
`getComputedStyle`	DOM method	`window`
`getElementById, getElementsByTagName`	DOM method	`document`
`defineProperty`	DOM method	all DOM nodes[1]
`addEventListener`	DOM method	`window` & position guard
`contentDocument, postMessage`	DOM property	widget iframe
`parentNode, offsetParent`	DOM property	all DOM nodes
`offsetLeft, offsetTop`	DOM property	all DOM nodes
`offsetHeight, offsetWidth`	DOM property	all DOM nodes

[1] : Google Chrome only

The actual implementation of these properties are within the built-in host objects, which are immutable. These built-ins serve as the prototype-objects for the native DOM objects, such as `window`, `Object`, or `document`. The DOM-space instances of these objects merely provide references to the native implementations. The `delete` operator removes a property from an object. If this operation succeeds, it removes the property from the object entirely. However, if a property with the same name exists on the object's prototype chain, the object will inherit that property from the prototype, which in the case of host objects is immutable [22]. Thus, redefining native DOM APIs creates a new property in the native object's current DOM-space instance, which effectively shadows the native prototype. Through deletion of this shadowing property, the prototype's implementation reappears (please refer to [32] for further information on this topic). However, deleting properties is potentially destructive. It is known that redefinition or wrapping of native API can be used for legitimate reasons, e.g., to provide the developer with enhanced capabilities. Thus, whenever possible, our mechanism attempts to detect but not to undo changes to the essential APIs and properties (see Sec. 6.4). If such changes could be detected, the mechanism concludes that the DOM integrity can't be validated and instructs the widget to disable seamless interaction (according to the strategy defined in Sec. 3.4).

6.4 Integrity of Native DOM APIs

As explained above, native DOM APIs cannot be deleted and a redefinition merely creates a DOM-space reference with the same name. Thus, a straightforward check for redefined native APIs works like this (see also Lst. 2):

1. Store a reference to the checked API in a local variable. In the tampering case, this variable will point to the DOM-space implementation.
2. `delete` the API and check the outcome. If the operation returned `true` continue to step 4.
3. If the operation returned `false`, the deletion failed. As deleting unchanged references to host-APIs always succeeds, the failing of the operation is a reliable indicator, that the corresponding property of the hosting object was

Listing 2 Tamper checking DOM APIs (simplified sketch)

```
// Keep a copy for reference
var copy = window.getComputedStyle;

// deletion of unchanged host APIs always returns 'true'
if (delete window.getComputedStyle){
  // Check if the function has changed
  if (window.getComputedStyle == copy)
      [... all is ok ...]
  else
      error("tampered!");
} else {   // delete failed
  // Redefined property with [[Configurable]] set to 'false'
  error("tampered!");
}
```

redefined with `defineProperty`, while setting the internal [[Configurable]] property to `false` (see Sec. 6.1). Hence, the API has been redefined. Terminate.

4. Compare the API to the local copy. If both point to the same implementation, the API's integrity is validated. Terminate positively.
5. If they differ, the API has been overwritten. Restore the local copy to the host object, in case the redefinition has legitimate reasons (non-disruptive approach) and terminate the integrity validation with negative result.

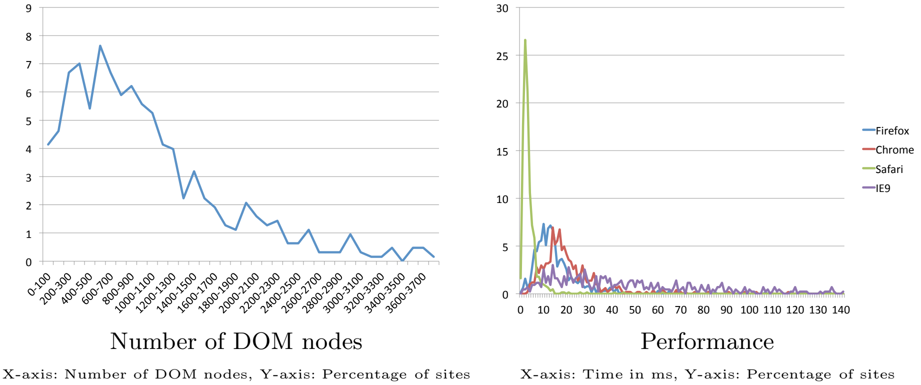
We practically validated this algorithm with Internet Explorer 9, Firefox 19, and Safari 5.

A subtle bug in Google Chrome: The behavior described above is universally implemented in all browsers, with one exception: Current versions of Google Chrome (in our tests version 26) allow destructive deletion of *some* native DOM APIs, mainly the ones attached to `Object`, such as `getOwnPropertyDescriptor`. However, for affected APIs, Chrome APIs can be verified by applying the same test to the API's respective `toString()` method, as the `Function` prototype exposes the correct behavior. This means, Chrome DOM APIs can be checked via applying the method discussed above to the APIs `toString()` method, instead to the APIs themselves.

6.5 Native DOM Property Integrity

While all browsers act (mostly) identical in respect to the redefinition of native DOM APIs, they expose differences when it comes to the properties of DOM elements, such as `parentNode` or `offsetHeight`.

Firefox & Internet Explorer 9 treat DOM properties in the exact same fashion as DOM APIs (see Sec. 6.4). Hence, for these browsers, the same algorithm can be applied.



Number of DOM nodes

X-axis: Number of DOM nodes, Y-axis: Percentage of sites

Performance

X-axis: Time in ms, Y-axis: Percentage of sites

Fig. 2. Results of the performance evaluation

Google Chrome's native DOM properties are immutable. This means, direct overwriting or redefining via `defineProperty` has *no* effect on the property. The property's value remains untouched by attempts to change it. Unfortunately, Chrome allows the irreversible deletion of DOM properties. Furthermore, after such deletion, a new property with the same name can be added to the hosting object again, now under full control of the attacker. However, the new property has the same characteristic as all 'normal' JavaScript properties, namely its internal [[Configurable]] property acts as specified: If it is set to `true`, the property can be redefined, if it is set to `false` a redefining step fails with an error message. Both cases differ noticeably from the legitimate behavior and, thus, can be utilized for a reliable test.

Safari & Internet Explorer 8 are strict about DOM integrity and do not allow direct overwriting or deleting of DOM properties. This also applies to using the `defineProperty` method. Thus, in the case of these two browsers, nothing has to be done, as malicious undermining of the DOM integrity is impossible.

7 Evaluation

7.1 Security Evaluation

In this section we discuss, based on the attack description in Sec 2.2, how our measure is able to defend the widget. Please note: This security evaluation only covers attack variants, which have been previously documented. In respect to yet to-be-discovered attacks, please refer to Sec. 4.5.

Hiding the iframe via CSS: The visibility checking process identifies all potential conditions that would render the widget invisible to the user (see Sec 4.1) and, thus, notifies the widget about the potentially malicious settings.

Obstructing the iframe with Overlaying Elements: Our mechanism finds all DOM elements that overlap with the widget (see Sec 4.2). Therefore, potential obstructing elements can be identified and acted upon.

Moving the iframe under the Mouse Pointer: The position guard overlay (see Sec. 4.4) enforces that the relative position of the widget in the page does not change after the visibility check has concluded. Therefore, this attack method is effectively disarmed.

Furthermore, the correct functioning of the visibility checking process is ensured through the system's DOM integrity checking methodology even in the context of an actively malicious embedding page (see Sec 6).

In this context, it has to be stressed, that the boundaries between Click/-LikeJacking and pure social engineering are fluid. Under suiting circumstances related attacks might be possible without resorting to overlays or other visibility influencing techniques, i.e., through hiding a visible element in plain sight via surrounding it with many similar looking elements. In such situations, the proposed protection method is powerless.

7.2 Functional and Performance Evaluation

To examine our approach's performance and interoperability characteristics, we conducted a practical evaluation. For this purpose, we selected a set of 635 sites out of the Alexa Top 1000, based on the characteristic that the sites included at least one JavaScript library directly from Facebook, as such a script-include is a necessary precondition to integrate Facebook's "like button". Furthermore, we implemented our visibility- and tamper-checking algorithms in a fashion, that it becomes active automatically after the page finished its rendering process. This means for every page, which includes our measure, the script automatically identifies all included social sharing widget (from the Facebook, Goole and Twitter) and validates their respective visibility state. Finally, we created a small program that causes a browser to successively visit the test sites and a userscript, which injects our script in every page this browser loads. For this, we used the following browser extensions: Greasemonkey[2] for Firefox 19, NinjaKit[3] for Safari 5, and IE7Pro[4] for Internet Explorer 9. Google Chrome has native support for userscripts and, hence, did not require a dedicated browser extension. All experiments were conducted on a MacBook Pro (Os X 10.7.2, Core i7, 2,2 GHz, 8GB RAM). The Internet Explorer evaluation was done using a Windows 7 virtual machine, running in VMWare Fusion 5. For all sites, the DOM integrity validation was performed and for all encountered widgets, also the visibility check.

One of the evaluation's goals was to examine to which degree real-world Web code is compatible with our protection approach. For no site out of the test bed, the DOM integrity check failed. Furthermore, as it can be seen in Table 4

[2] Greasemonkey: `https://addons.mozilla.org/de/firefox/addon/greasemonkey/`

[3] NinjaKit: `https://github.com/os0x/NinjaKit`

[4] IE/Pro: `http://www.ie7pro.com/`

Table 3. Browser performance measurements

Browser	**Min**^{5}	**Max**^{5}	**Average**^{5}	**Median**^{5}
Firefox^1	1	135	15.0	13
Google Chrome^2	3	117	21.0	18
Safari^3	1	62	3.0	3
Internet Explorer^4	1	141	52.0	40

^{1x}: Firefox 19.0.2 / OsX 10.7, ^{2}: Chrome 26.0.1410.43 / OsX 10.7,
^{3}: Safari 5.1.2 / OsX 10.7, ^{4}: IE 9.0.8112 / Win7 (VMWare),

^{5}: All times in milliseconds

for the vast majority of the widgets (1537 out of 1648), the visibility could be verified. For the remaining 111 widgets, manual analysis in respect to providing interoperability would be required.

Furthermore, as documented in Table 3 and Figure 2, our protection mechanism only causes negligible performance costs, with a general median overhead of less then 40ms and worst case scenarios well below 200ms, even for large, non-trivial DOM structures with up to 3000 nodes.

8 Related Work

Further Attack Variants: Besides the basic attack, which utilizes invisible iFrames, several different forms of Clickjacking attacks were discovered. For one, Bordi and Kotowicz demonstrated different methods to conduct a so called Cursorjacking attack [4,15]. Thereby, the real mouse cursor is hidden and fake cursor is presented to the user at a different position. When interacting with the Web site the user only recognizes the fake cursor. When clicking the mouse, the click event does not occur at the position of the visible fake cursor but at the position of the hidden cursor. Therefore, the user is tricked into clicking an element that he not intended to click.

Adding protection against such attacks to our countermeasure is straight forward: The CSS styling of the mouse pointer can be added to the forbidden visibility conditions.

Furthermore, Clickjacking attacks are not limited to invisible iFrames. Zalewski and Huang showed that it is also possible to use popup windows instead of frames [11,31]. While Zalewski's approach utilizes the JavaScript history API and a timing attack, Huang came up with the so called Double Clickjacking attack. Thereby, a Web site opens a popup window, behind the actual browser window. Then the Web site lures the user into double clicking on the visible Web site. When the first click hits to page the popup window is brought to the front and therefore, the second click hits the page that was loaded within the popup window. After a few millisecond the Web site closes the popup window and therefore the user does not recognizes the attack.

Our mechanism is secure against Huang's double-click attack: As the position guard overlay (see Sec. 4.4) does not receive the required mouse-over event, it does not change its `pointer-events` and, hence, catches the click before it can

Table 4. Compatibility testing with deployed widgets

Widget provider	Sites[1]	Total[2]	Visible	Hidden	CSS[3]	DOM[4]	Obstructed[5]
Facebook	391	837	779 (93%)	58 (7%)	34	8	16
Google+	167	277	255 (92%)	22 (8%)	4	13	5
Twitter	207	534	503 (94%)	31 (6%)	22	1	8

[1]: Number of sites that include at least one widget of the provider (out of 635) [2]: Total number of found widgets
Reasons for failed visibility check: [3]: CSS properties (see Sec 4.1),[4]: DOM properties (see Sec 4.3),
[5]: Obstructing overlays (see Sec 4.2)

reach the widget. Also, even if the mouse is slightly moved between the clicks, the entering position of the mouse pointer will be in the middle of the overlay and not at the borders, which is a clear indicator for suspicious behavior.

Server-Side Countermeasures: Besides the general ClickJacking-focused approaches discussed in Sec 2.3, some mechanism have been proposed that also take Likejacking into account. When the first Likejacking attacks were conducted, Facebook implemented some countermeasures to detect "malicious likes" [30]. When ever a malicious situation is detected, the user is asked to confirm the action, instead of seamlessly processing the "like request". Unfortunately, precise details on the implementation are not available and the problem still exists in the wild.

Another approach was proposed by Brad Hill [9]. He suggested to utilize user interface randomization as an anti-clickjacking strategy. Thereby, a Web widget renders its buttons in different location each time it is loaded.Therefore, the attacker cannot be sure in which position the button is being placed and is only able to use a trial and error approach to conduct the attack. By analyzing the first click success rate, a Widget provider would be able to detect Likejacking campaigns very soon, as in the legitimate use case the first click success rate is significantly higher than in the trial and error Clickjacking attack. However, randomizing the user interface decreases user experience and might distract user's from using a widget. Furthermore, the method is not applicable to more complex widgets.

Client-Side Countermeasures: The first client-side countermeasures was the NoScript ClearClick Firefox plug-in [18]. ClearClick detects a Clickjacking attack by creating two screenshots and comparing the results. One screenshot is taken from the plugin object or the framed page the user attempts to click on. The second screenshot shows how the page/object is embedded into the page. If the two screenshots differ, the object's visibility is somehow tampered and therefore ClearClick shows a warning to the user. Furthermore, ClickIDS, a related, experimental browser extension, was presented in [1].

In 2012 Brad Hill suggested to introduce a new type of control that requires more user interaction than just a click (e.g. a Swipe, Scrub, or holding the mouse for a certain amount of time, etc) [10]. While the user interacts with the control, the browser forces the corresponding markup to become completely visible. While doing so, the browser could even dim or hide other elements so that these elements do not overlap or hide the security sensitive control. However, until now this idea has not been implemented by any major browser.

Besides these mechanisms a few other client-side mechanisms were proposed to stop Clickjacking attacks in the form of alternative browser designs (e.g Gazelle [29], the OP Web browser [6] or the secure Web browser [13]). For the time being, none of these proposals have been adopted by the major browsers.

9 Conclusion

In this paper, we presented a novel methodology to protect Web widgets against LikeJacking attacks. Our approach does not require browser modifications and is fully interoperable with today's JavaScript capabilities. Using a practical evaluation of 635 site, we demonstrated our technique's compatibility with productive Web code and showed that the approach's performance scales well, while causing negligible overhead.

Outlook: Because of the closeness of LikeJacking to social engineering (see Section 7.1) and the highly flexible nature of CSS, the visibility validation step of our approach has to be regarded as its most fragile component. However, when approaching the topic from a wider angle, it becomes apparent that LikeJacking is only one instance in a lager problem space:

The underlying challenge occurs every time, when a third party service requires reliable information on the Web execution context in which it is included. Hence, the more significant contribution of this paper is the general methodology, that allows third party components to trustworthy collect evidence on the state of the integrator page and securely communicate the result, with visibility validation being only one example for such an evidence collecting process.

References

1. Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., Kruegel, C.: A solution for the automated detection of clickjacking attacks. In: AsiaCCS (2010)
2. Barnett, R.: Detecting Successful XSS Testing with JS Overrides. Blog post, Trustwave SpiderLabs (November 2012), `http://blog.spiderlabs.com/2012/11/detecting-successful-xss-testing-with-js-overrides.html` (last accessed April 7, 2013)
3. Barth, A., Jackson, C., Mitchell, J.C.: Robust Defenses for Cross-Site Request Forgery. In: CCS 2009 (2009)
4. Bordi, E.: Proof of concept - cursorjacking (noscript), `http://static.vulnerability.fr/noscript-cursorjacking.html`
5. Crockford, D.: Private Members in JavaScript (2001), `http://www.crockford.com/javascript/private.html` (Janauary 11, 2006)
6. Grier, C., Tang, S., King, S.T.: Secure Web Browsing with the OP Web Browser. In: IEEE Symposium on Security and Privacy (2008)
7. Hansen, R., Grossman, J.: Clickjacking (August 2008), `http://www.sectheory.com/clickjacking.htm`

8. Heiderich, M., Frosch, T., Holz, T.: IceShield: Detection and mitigation of malicious websites with a frozen DOM. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 281–300. Springer, Heidelberg (2011)
9. Hill, B.: Adaptive user interface randomization as an anti-clickjacking strategy (May 2012)
10. Hill, B.: Anti-clickjacking protected interactive elements (January 2012)
11. Huang, L.-S., Jackson, C.: Clickjacking attacks unresolved. White paper, CyLab (July 2011)
12. Huang, L.-S., Moshchuk, A., Wang, H.J., Schechter, S., Jackson, C.: Clickjacking: attacks and defenses. In: USENIX Security (2012)
13. Ioannidis, S., Bellovin, S.M.: Building a secure web browser. In: USENIX Technical Conference (2001)
14. Johns, M., Winter, J.: RequestRodeo: Client Side Protection against Session Riding. In: OWASP Europe 2006, refereed papers track (May 2006)
15. Kotowicz, K.: Cursorjacking again (January 2012), `http://blog.kotowicz.net/2012/01/cursorjacking-again.html`
16. Lekies, S., Heiderich, M., Appelt, D., Holz, T., Johns, M.: On the fragility and limitations of current browser-provided clickjacking protection schemes. In: WOOT 2012 (2012)
17. Magazinius, J., Phung, P.H., Sands, D.: Safe wrappers and sane policies for self protecting javaScript. In: Aura, T., Järvinen, K., Nyberg, K. (eds.) NordSec 2010. LNCS, vol. 7127, pp. 239–255. Springer, Heidelberg (2012)
18. Maone, G.: Noscript clearclick (January 2012), `http://noscript.net/faq#clearclick`
19. Maone, G., Huang, D.L.-S., Gondrom, T., Hill, B.: User Interface Safety Directives for Content Security Policy. W3C Working Draft 20 (November 2012), `http://www.w3.org/TR/UISafety/`
20. Microsoft. IE8 Security Part VII: ClickJacking Defenses (2009)
21. Mustaca, S.: Old Facebook likejacking scam in use again, Avira Security Blog (February 2013), `http://techblog.avira.com/2013/02/11/old-facebook-likejacking-scam-in-use-again-shocking-at-14-she-did-that-in-the-public-school/en/`
22. Mozilla Developer Network. delete (February 2013), `https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/delete`
23. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In: CCS 2012 (2012)
24. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting javascript. In: ASIACCS 2009 (2009)
25. Ruderman, J.: Bug 154957 - iframe content background defaults to transparent (June 2002), `https://bugzilla.mozilla.org/showbug.cgi?id=154957`
26. Rydstedt, G., Bursztein, E., Boneh, D., 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)
27. Shepherd, E.: window.postmessage (October 2011), `https://developer.mozilla.org/en/DOM/window.postMessage`

28. SophosLabs. Clickjacking (May 2010), `http://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/` (last accessed July 4, 2013)
29. Wang, H.J., Grier, C., Moshchuk, A., King, S.T., Choud-hury, P., Venter, H.: The Multi-Principal OS Construction of the Gazelle Web Browser. In: USENIX Security Symposium (2009)
30. Wisniewski, C.: Facebook adds speed bump to slow down likejackers (March 2011)
31. Zalewski, M.: X-frame-options is worth less than you think. Website (December 2011), `http://lcamtuf.coredump.cx/clickit/`
32. Zaytsev, J.: Understanding delete (January 2010), `http://perfectionkills.com/understanding-delete/`