Out of the Dark: UI Redressing and Trustworthy Events

Marcus Niemietz, Jörg Schwenk

Horst Görtz Institute for IT Security
Chair for Network and Data Security
Ruhr-University Bochum
marcus.niemietz@rub.de, joerg.schwenk@rub.de

Abstract. Web applications use *trustworthy events* consciously triggered by a human user (e.g., a left mouse click) to authorize security-critical changes. Clickjacking and UI redressing (UIR) attacks trick the user into triggering a trustworthy event unconsciously. A formal model of Clickjacking was described by Huang et al. and was later adopted by the W3C UI safety specification. This formalization did not cover the target of these attacks, the *trustworthy events*.
We provide the first extensive investigation on this topic and show that the concept is not completely understood in current browser implementations. We show major differences between widely-used browser families, even to the extent that the concept of trustworthy events itself becomes unrecognizable. We also show that the concept of *trusted events* as defined by the W3C is somehow orthogonal to *trustworthy events*, and may lead to confusion in understanding the security implications of both concepts. Based on these investigations, we were able to circumvent the concept of trusted events, introduce three new UIR attack variants, and minimize their visibility.

1 Introduction

UI Redressing attacks are powerful attacks which can be used to circumvent browser security mechanisms like sandboxing and the Same-Origin Policy (SOP). They are far less intrusive than, for example, Phishing mails because the user thinks he performs a legal action on an innocent-looking web page. In 2008 Grossmann et al. had to cancel their OWASP talk about a new attack technique called Clickjacking [10]: it turned out that they were able to bypass a major protection mechanism of Adobe's Flash – Clickjacking allowed the attacker's website to automatically get access to the camera and microphone of the victim without any explicit permission. According to Adobe, Clickjacking had the "highest level of damage potential that any exploit can have" [26].

In contrast to Clickjacking that is usually associated with left-click mouse events only, the broader term UIR also covers events from the keyboard and even touch gestures [12,30]. In the past years, many attacks and defense mechanisms were published by the industry as well as the academic community (e.g., [28,33,4,3], and [18]).

Formal Definition of UIR. Huang et al. [12] defined Clickjacking to be an attack that violates the integrity of either the *visual context* or the *temporary context* of a trustworthy user action on a sensitive element of the web application. *Visual context integrity* may either be violated by making the sensitive element invisible (e.g., by placing it in fully transparent mode above some other element), or by hiding the fact that the user is actually clicking on such an element (e.g., by modifying the image of the mouse pointer, also referred to as Cursorjacking [15]). *Temporal context integrity* can be violated by replacing a non-sensitive element, just before the user clicks on it, by the sensitive element.

The definitions from Huang et al. [12] can easily be extended to the broader class or UIR attacks. However, the treatment of trustworthy events becomes more complex because in addition to left-click events, also right-click-and-select, keyboard and inter alia touch events must be taken into account.

Events in Web Applications. Events can be triggered by humans (e.g., by clicking on a button or moving the mouse pointer), by network operations, or automatically with the help of scripts. From the network, events like `load` or the status change events in XMLHttpRequest queries can be triggered. Purely script based are, for example, those triggered by the `setTimeout()` or `setIntervall()` method. For human interaction, a distinction must be made between events that the user consciously starts (e.g., `click` or `keydown`), and events that he may not notice (e.g., `mouseover`). Event-handlers are procedures with an `on`-prefix; they are called when the corresponding event occurs. For example, the `onclick` event-handler is called whenever a `click` event occurs.

Events are managed in the event system of the browser and there exist many differences across browsers. For example, the event `wheel` will only be executed on the event system of Internet Explorer (IE) when the method `addEventListener()` is used. The event system of Google Chrome (GC) will recognize this event with the same conditions when the event-handler `onwheel` is used. To foster interoperability, there exists a working draft of an UI event specification designed by the World Wide Web Consortium (W3C) [13]. The specification describes event systems and subsets of different event types.

Trusted vs. Trustworthy Events. Trusted events are defined by the W3C as follows: "Events that are generated by the user agent, either as a result of user interaction, or as a direct result of changes to the DOM, are trusted by the user agent with privileges that are not afforded to events generated by script through the `createEvent()` method, modified using the `initEvent()` method, or dispatched via the `dispatchEvent()` method. The `isTrusted` attribute of trusted events has a value of *true*, while untrusted events have a `isTrusted` attribute value of *false*." ([40], Section 3.4).

This definition is very broad and therefore not suitable for a distinction between events that may be allowed to cause security critical changes, and those that may not. For example, the `mouseover` and `click` events are both "trusted" according to the W3C definition when caused by a human user; however, displaying a pop-up window or sending the contents of an HTML form simply

because the mouse pointer crossed over a certain area of the browser window (`mouseover`) seems far too permissive. Our definition is more specific: *a trustworthy event is an event that is triggered by a conscious user action* (e.g., by left-click, right-click, or keystroke).

Unreliability of `isTrusted`. To mark trusted actions, the DOM Level 3 specification of the W3C mentions a read-only property called `isTrusted`, which returns a boolean value depending on the dispatched state [39]. In Section 4 we show that this property cannot be used to distinguish *trustworthy* events from other events, since pop-ups are blocked even if `isTrusted=true`, and are allowed even if `isTrusted=false`.

Trustworthy Event Scenarios. Trustworthy events are used in different security critical scenarios. ***User consent*** in activating potentially dangerous browser features (e.g., activating the webcam via Adobe's Flash) was the main target in previously described UIR attacks. ***Pop-up windows*** are usually blocked when there was no former click with the mouse pointing device. One reason is that pop-up windows are used by the advertisement industry and thus they might disturb the user or they may even trick him to install malware. The ***clipboard*** should only be accessible by user initiated keyboard or mouse pointing events. If the clipboard would be accessible by JavaScript code only, an attacker's website could steal saved data like passwords stored in a password manager (paste action). ***Drag-and-drop*** is a scenario where a user is able to move data cross-origin. Again, if this feature was accessible by JavaScript code only, the SOP could be circumvented. *Additional scenarios:* in Firefox (FF), the deprecated XML User Interface Language (XUL) handlers and commands can only be triggered by trustworthy events like click and touch [23]. In modern browsers such as GC, forms can be filled out automatically by using the autofill feature that could be activated by trustworthy events like keystrokes and left-clicks [34,42].

Investigation of Trustworthy Events. We study *(1)* all mouse events including (a) different left-clicks (`click`, `dblclick`, `mousedown`, `mouseup`), (b) right-click, (c) mouse movements (`mouseover`), (d) drag (`drag`, `dragstart`) and (e) `wheel`; *(2)* the keyboard events `keydown`, `keyup`, `keypress`, and *(3)* combinations of mouse and keyboard events. We show that many of these user-triggered actions have different interpretations as trustworthy or non-trustworthy events in the different browser families.

We investigate the three lesser-researched application areas of trustworthy events: pop-up windows bypassing pop-up blockers, escaping the browser sandbox via copy-and-paste to and from the clipboard, and bypassing the SOP via drag-and-drop.

Research questions. In this work we investigate the following questions:

Which events are recognized as trustworthy by a modern web browser?
How is trustworthy event handling implemented in modern web browsers?
Could the knowledge of these implementations lead to new UIR variants?

Contribution. The contributions of this paper are as follows:

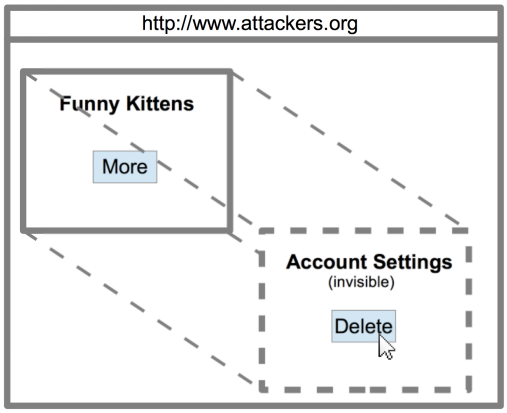

Fig. 1: Illustration for a Classic Clickjacking attack.

- We systematically evaluate trustworthy events in web applications originating in a mouse device, the keyboard, or a combination thereof, and describe differences in modern browsers implementations.
- We thoroughly analyze three security critical trustworthy event scenarios (pop-up windows, drag-and-drop, and clipboard), both same and cross-origin.
- We introduce and discuss three new UIR attack variants by making use of particularities of trustworthy event implementations in modern browsers.

2 UI Redressing

The initial Clickjacking attack of Grossman et al. raised a lot of attention due the hijacking possibilities of the webcam and microphone, but they also discovered a general security problem. As listed by Niemietz et al. [28], UIR is a set of attacks that include Clickjacking as a subset. Next to Classic Clickjacking there are other attacks like Sharejacking and Likejacking (e.g., to attack Facebook [35]), and inter alia Cursorjacking [15,7]. UIR does not only cover clicks, it also covers drag operations (drag-and-drop attacks [38]), keystrokes (Strokejacking [43]) and even maskings (SVG-based attacks [27]).

In a classic Clickjacking attack illustrated in Figure 1, the victim has opened the attacker's website, which consists of two Iframes. The first Iframe ("Funny Kittens") is loading a visible HTML document to lure the victim into clicking on the `More` button. The second Iframe loads the target "Account Setting" website, but this frame is rendered invisibly (e.g., with the help of the property `opacity =0`) on top of the visible frame. Because of invisible Iframe's position above the `Funny Kittens` Iframe, the victim will actually click on `Delete` instead on `More`.

UIR Contexts. According to Huang et al., the definition of UIR is that "an attacker application presents a sensitive UI element of a target application out of context to a user and hence the user gets tricked to act out of context" [12]. This definition describes the root cause of UIR.

Visual Context. This context defines what the user sees. It does not include actions (e.g., clicking) on sensitive elements (e.g., buttons). To ensure target *display integrity*, sensitive elements must be fully visible to the user. In contrast, *pointer integrity* requires that input mechanisms and their resulting actions are fully visible to the user.

Temporal Context. The timing of a user's action is known as the temporal context. To ensure *temporal integrity*, the user's action is actually intended by the user. To compromise temporal integrity, a visible button could be replaced by the attacker right before the victim is clicking on it (e.g., with a Facebook *Like* button).

These context definitions provide an important insight on how UIR attacks work in the important case that the user does simple events such as a single left-click. However, in reality there exists a much broader set of user events (e.g., keystroke, right-click, and a chain of left-clicks). This could lead to new attack variants and therefore different events must be considered (shown in Section 6).

3 Events in Web Applications

Browser events can be divided into different event types according to the W3C working drafts for handling browser events [13,14]. In the following, we map common event types into different event type groups. To the best of our knowledge, we completely cover all commonly used user interactions.

All event types can be either triggered by *user* or *script* actions. To name one example, a user can consciously trigger a `click` event by explicitly clicking on a button with the event-handler `onclick`. In addition, a script can also trigger this event automatically by using the DOM's `click()` method (e.g., `document .getElementById("button").click()`).

Resource Events. These are frame or object events that are triggered by HTTP events. Examples for resource events are `error` (failed to load), `load` (finished loading), and `unload` (unloading of a document or depending resource).

Mouse Events. Consciously created mouse events are usually left and right clicks. In addition, mouse events can also be generated unconsciously when the pointer is moved or when drag-and-drop actions are done. The most deeply nested element is always the target of a mouse event. Except for user interactions on a virtual keyboard, touch events act similar to mouse events and are thus included in the mouse event set. Examples for mouse events are `click` (button has been pressed and released), `mousemove` (moved pointing device), and `drag` (dragged element or text).

Keyboard Events. This event type is for example triggered when a user is pressing (`keydown`) or releasing a key (`keyup`). Virtual keyboards, from input devices like touch screens, trigger keyboard events and are therefore also in this even type group.

Multiple Events. Some events cannot be assigned to only the mouse or keyboard; they can also be triggered by both variants. As an example, a user can

select text in an `input` element by using the mouse cursor (click and mark) and also the keyboard (shift and arrow keys).

Based on these event types, we provide a definition for trustworthy events:

Definition 1. *An event is called* trustworthy *when it was triggered by a* conscious *user action.*

4 DOM Property isTrusted

The W3C specification ([40], Section 3.4) describes a boolean attribute `isTrusted`: "The `isTrusted` attribute of trusted events has a value of *true*, while untrusted events have a `isTrusted` attribute value of *false*." We investigate this attribute in detail and show that it is not related to *trustworthy* events.

Different isTrusted Implementations. According to the W3C, the DOM property `event.isTrusted` only returns `true` when an event was dispatched by the user agent [39]. According to the Mozilla Developer Network (MDN), the property is defined as *true* "when the event was generated by a user action, and *false* when the event was created or modified by a script or dispatched via dispatchEvent" [24]. IE is an exception because all events are *true* except they are created with `createEvent()`. This JavaScript feature can be used to create an event object and simulate an event type such as a mouse event (e.g., an automatically fired click on a button for testing web applications).

isTrusted=false, but Pop-Ups are Allowed. Listing 1.1 contains a button and a hyperlink. If the button is clicked by the user, the `onclick` event-handler calls `document.getElementByID("test").click()`, and this JavaScript function selects the hyperlink (which has `id="test"`), and performs a script-generated click event on it. Consequently, the value of `isTrusted`, which is shown in the `alert()` window, is `false`, as described in the W3C specification. Nevertheless, `window.open()` is executed, and a pop-up window is displayed.

```
1 <button onclick="document.getElementByID("test").click()">
    </button>
2 <a href="#" id="test" onclick="alert('isTrusted: '+
    event.isTrusted); window.open('http://example.org', 'rub
    ','height=200,width=200');">Trusted Click</a>
```

Listing 1.1: Pop-ups are not blocked although `isTrused` is `false`.

isTrusted=true, but Pop-Ups are Blocked. Listing 1.2 provides an example with the `<video>` element introduced with HTML5. It contains an `onloadstart` event-handler, which executes code when the browser starts looking for the video file given in line 2. Thus, JavaScript code will be directly executed without any real user interaction. Due to this reason, the JavaScript code generated pop-up will be blocked. The alert-window with `event.isTrusted` displays `true` on all browsers although the only user interaction was an initial opening of the page (e.g., FF, GC, and Edge).

```
<video onloadstart="window.open('http://example.org', null,
    'height=200,width=400,status=yes,toolbar=no,menubar=no
   ,location=no');alert(event.isTrusted);">
<source src="movie.mp4" type="video/mp4">
</video>
```

Listing 1.2: Pop-ups are blocked although `isTrusted` is `true`.

Inheritance of Trustworthiness. Our evaluation of the behavior of `isTrusted` and the displaying of pop-up windows shows an interesting result; events occurring within a delay of one second after an initial trustworthy event are also treated as trustworthy events, although they may be triggered purely by JavaScript.

More formally: let $P_t = true$ denote the fact that the pop-up window opened at time t was *not* blocked by the pop-up-blocker. Let $iT = t_0$ denote the fact that a trustworthy event was initiated by the user at time t_0. Then we have:

$$P_t := \begin{cases} true, \ if \ (iT = t_0) \wedge (|t - t_0| \leq 1 \ sec) \\ false, else \end{cases}$$

The interesting discovery is that a pop-up window will not be blocked in the event that there was *once* a (real) user's click in the chain of events. This behavior was observed for the tested versions of FF and Safari (SA).

5 Trustworthy Scenarios

The W3C UI Events specification [40] does not recommend actions that are allowed after a trustworthy event. As shown by Huang et al. [12], a missing formal definition could lead to different browser implementations and thus to browser bugs and vulnerabilities.

Next to our trustworthy event definition, we address this issue by providing a description of three different trustworthy scenarios. We believe that the scientific community and browser vendors will get a valuable overview about this currently not examined area and thus derive new attack variants and countermeasures (cf. Section 6).

5.1 Pop-Up Scenario

Need of Trustworthy Events. In the past, JavaScript code was able to automatically open pop-up windows when the user simply opened a website. The advertising industry used this feature to show unwanted ads to the user and thus modern browsers distinguish between wanted and unwanted pop-up windows: a pop-up window should only be shown when a trustworthy event (e.g., `click`) was used to call the required JavaScript pop-up-code (e.g., `window.open`).

Evaluation. Table 1 lists four different types of events with each event type containing different events. Each event type includes different events. The test

cases for these events were executed in four different browsers: IE 11, FF 47, GC 54, Opera (OP) 41, and SA 10. Our test function for pop-ups is given in Listing 1.3. It tries to create up to five pop-up windows in case that the code is indeed called. If this is the case, all five pop-ups are displayed in FF and SA; in contrast, only one pop-up with a warning window in IE, GC, and OP.

```
<script>
function createPopups(){
  for (i=1;i<6;i++) {
    window.open('//evil.org', i, 'width=50,height=50');
  }
}
</script>
```

Listing 1.3: Our test function for pop-ups.

In the first event type group, *resource* events are given. These events are inter alia triggered by loading the browser's window or by simply reloading it. The user does not use an input device like a mouse or a keyboard and thus pop-up windows are not displayed.

Mouse events are the second type of events. Our test cases cover left-clicks, right-clicks, mouse movements, dragging actions, and the usage of the mouse wheel. In the event of a left-click, pop-ups will be shown. A right-click only leads to pop-up windows in IE. Mouse movements and dragging actions do not let the tested browser open pop-up windows. The event `wheel` is triggered when the wheel rolls up or down over an HTML element; it does not lead to the displaying of pop-up windows in FF, GC, and OP. Furthermore, this event is not supported in IE.

With the third defined type called *keyboard events*, only GC and OP act in a pop-up scenario. IE and FF behave differently, pop-ups will be blocked.

The fourth type called *multiple events* consists of events that can be triggered in different ways like keyboard actions and left-clicks. It shows that there are events which act different across browsers; only some browsers allow access to the pop-up scenario and IE only in case of a left-click in combination with the event `select`. In IE 11 and FF 47, a left-click in combination with `focus` or `blur` does not lead to a pop-up execution. As another example, FF grants access when an `input` event in combination with a right-click for copy-and-paste is used. This is not the case when this event is used in combination with a keyboard action. GC and OP act exactly in the opposite way.

5.2 Clipboard Scenario

Need of Trustworthy Events. Clipboard data may contain sensitive information that should not be shared with an arbitrary website. For example, password managers usually save stored passwords into the clipboard such that they could be inserted into login forms (e.g., for banking or shopping). Therefore, JavaScript code that is able to automatically read clipboard data could copy the password

Events	Type	IE 11	FF 47	GC 54	OP 41
load, error, unload	Resource	✗			
click, dblclick, mousedown, mouseup (left-click)	Mouse	✓			
contextmenu (right-click)		✓		✗	
mouseenter, mouseleave, mousemove, mouseout, mouseover (movement)		✗			
drag, dragstart (dragging)		✗			
wheel		✗			
keydown, keyup, keypress	Keyboard	✗		✓	
search (keyboard, left-click)	Multiple	–		(✗,✓)	
select (keyboard, left-click)		(✗,✓)		✓	
input (keyboard, right-click paste)		✗	(✗, ✓)	(✓, ✗)	
focus (keyboard, left-click)		✗		✓	
focusin, focusout (keyboard, left-click)		✗	–	✓	
blur (keyboard, left-click)		✗		✓	(✗, ✓)
scroll (keyboard, wheel)		✗			

Table 1: Events and their triggered pop-up windows. ✓ indicates that the pop-up was shown, ✗ that it was blocked. For the category of multiple events, "keyboard" denotes all events of type "Keyboard", and (✓,✗) means that a keyboard event did result in a pop-up, whereas the mentioned click event did not.

from the clipboard and send it to the attacker. For this reason, browsers should only allow access to clipboard data after a conscious user action, i.e. after a trustworthy event. A moderate security problem arises in the event of copy and cut operations to the clipboard; a website should not overwrite clipboard data without an explicit permission of the user.

Evaluation. As shown in Table 2, the clipboard always allows copy, cut, and paste operations with the help of a keyboard or mouse pointing device (no script execution). In the event of automatically executed scripts, it is usually not possible to access the user's clipboard. IE is an exception as it allows access to copy, cut, and paste operations (see Listing 1.4) by showing the user a confirmation window which only gives access when the user explicitly clicks on `Allow access`.

```
1  //read data of type ``Text'' from clipboard
2  window.clipboardData.getData("Text");
3  //write data of type ``Text'' to the clipboard
4  var input = "This text is written to the clipboard";
5  window.clipboardData.setData("Text",input);
```

Listing 1.4: JavaScript functions to access the clipboard.

By looking at the results from the pop-up scenario (cf. Table 1), JavaScript code can act on a higher privileged authorization level in case that the script was triggered by a trustworthy event. We found that the clipboard copy and cut capabilities are also enabled when a trustworthy event calls JavaScript code. To name an example, a listener on the event `click` can be used to copy data into

the clipboard via `clipboardData.setData`. Except IE, event handlers which are able to open pop-up windows are also able to access the clipboard API with copy and cut capabilities within a delay of one second (e.g., via the `EventTarget .addEventListener()` method) [25]. Thus, our pop-up definition with P_t (cf. Section 4) also applies to these kinds of clipboard API access.

Paste operations can only be accessed with the help of JavaScript code when the user triggers a trustworthy `paste` event via `Ctrl+V` and `Edit->Paste`. This clipboard API [37] paste event behavior is important from the security perspective (discussed in Section 6.3).

<table>
<tr><th>Action via</th><th>IE 11</th><th>FF 47</th><th>GC 54</th><th>OP 41</th></tr>
<tr><td colspan="5">Copy / Cut</td></tr>
<tr><td>Right mouse-click then copy/cut</td><td colspan="4">✓</td></tr>
<tr><td>Keyboard: <code>Ctrl+C</code></td><td colspan="4">✓</td></tr>
<tr><td>Script</td><td rowspan="2">(✓)</td><td colspan="3">✗</td></tr>
<tr><td>Trustworthy Event and then script</td><td colspan="3">cf. Table 1</td></tr>
<tr><td colspan="5">Paste</td></tr>
<tr><td>Right mouse-click then paste</td><td colspan="4">✓</td></tr>
<tr><td>Keyboard: <code>Ctrl+V</code></td><td colspan="4">✓</td></tr>
<tr><td>Script</td><td rowspan="2">(✓)</td><td colspan="3">✗</td></tr>
<tr><td>Trustworthy Event and then script</td><td colspan="3">(✗)</td></tr>
</table>

Table 2: Clipboard handling. ✓ denotes that the text is copied, ✗ that it is not copied. (✓) denotes that the text is copied, but a warning is displayed. The reference to Table 1 means that any trustworthy event that could be used to trigger a pop-up in FF 47, GC 54, or OP 41 can be used, in combination with the JavaScript code given in Listing 1.4, to write text to the clipboard.

5.3 Drag-and-Drop Scenario

Need of Trustworthy Events. Drag-and-drop operations can be done same-origin or cross-origin. Thus, the usual access limitations of the SOP in the HTML context does not apply in this scenario. Modern browsers like GC even allow the user to drag content from the desktop into the browser's website (e.g., for file uploads). Without trustworthy events, arbitrary data from another window and environment could be stolen automatically with the help of JavaScript code.

JavaScript DOM Access. An example for transferring data via drag-and-drop is given in Table 3. In this table, the host document (HD) shown in Listing 1.5 includes the embedded document (ED) displayed in Listing 1.6.

The first part of Table 3 illustrates that the code of Listing 1.5 can be used, in the same-origin case, to copy the word `Test` into the input field of Listing 1.6. This is possible because we select this word by using the ID `HDt` and afterwards

we copy it into the input field with the ID `EDi`. To do this, one must select the embedding element with the ID `EDf`. In the cross-origin case, the browser does not allow the copy-action.

From an attacker's perspective, it is interesting to know whether it is possible to do actions which are restricted by the SOP [31,5]:

1. We trigger the JavaScript function of Listing 1.5 by dragging the content of `<div>` to trigger the JavaScript function `copy()` with the help of the `ondragstart` event-handler. In this case, only same-origin access from the HD to the ED is allowed.
2. Cross-origin drag-and-drop operations are allowed in two browsers: IE 11 and FF 47. Trustworthy events like selecting the text `test` with the mouse, dragging it into the Iframe's `input` field and dropping the selected text into this field allows to do actions that are (cross-origin) restricted with JavaScript code. GC and OP also allowed these actions in former versions (cf. Section 6).

```
<i id="HDt">Test</i><br>
<iframe id="EDf" src="http://example.org/form.html"></i
    frame>
<div draggable="true" ondragstart="copy()">Drag me</div>
<script> function copy() {
  document.getElementById("EDf").
      contentDocument.getElementById("EDi").value =
      document.getElementById("HDt").innerHTML;
} </script>
```

Listing 1.5: The HD executes JavaScript code when a `dragstart` event occurs.

```
<form action="action.php">
<input type="text" id="EDi"><br>
</form>
```

Listing 1.6: HTML code of the ED.

Iframe access	IE 11	FF 47	GC 54	OP 41
JavaScript				
Same-Origin (SO)	✓			
Cross-Origin (CO)	✗			
Mouse Events				
Click calls function (SO)	✓			
Click calls function (CO)	✗			
Drag&Drop (SO, CO)	✓		✗	

Table 3: A HD wants to transfer data to the Iframe's web page (✓ access, ✗ no access).

6 New UIR Attack Variants

Based on the described trustworthy scenarios, we demonstrate that known UI redressing techniques in combination with trustworthy events can be used to derive attacks with a higher attack surface. We construct three new attack variants and evaluate their practicability on modern browsers.

6.1 Optimized Drag-and-Drop Attack

In 2010 Stone published a Clickjacking attack that makes use of the HTML5 drag-and-drop API [38]. In a proof of concept, he showed a website with a frog and a blender. By using social engineering, he lured the victim intro dragging the frog into the blender. What the victim actually does is a cross-origin-drag of attacker defined content into another website. This bypasses protection mechanisms against Cross-Site Request Forgery and could be used in webmail application, document editors, or even to set passwords as shown by Niemietz et al. [29,21].

Drag-and-drop across windows was supported between browsers and therefore an attractive feature which could be abused by attackers. Nowadays, this feature is disabled in modern browsers like GC, SA, and OP; it still works in IE, Edge, and FF. In the following, we derive an attack variant which highlights the importance of different UIR contexts. It points out that trustworthy events play an indispensable role in browser security.

In Stone's initial attack of dragging a frog into a blender, the victim had to clearly visible move the mouse cursor a certain distance (frog to blender) such that the victim might know that it initialized a drag action. The following described attack shrinks the cursor distance to a minimum (e.g., two pixels). Thus, the victim might not notice that any drag actions occurred.

Attack Summary. By looking on the left-hand side of Figure 2, the attacker's website without any user action is displayed (cf. Listing 1.8). This website could be opened by the user due to a click on a link in a phishing mail. What the victim does is that it slightly moves the button causing a *cross-origin* drag-and-drop injection to occur (cf. Listing 1.7 and Listing 1.9). For demonstration purposes, an alert-window generated by JavaScript code appears with the injected content (cf. Listing 1.9).

Attack Structure. With the help of Listing 1.8, there is a web page shown, making use of social engineering techniques. By showing an image with a button that should be moved, attacker defined content will be dragged but not the selected image. By dragging the image, the function *hover* of Listing 1.7 will also be called. This function places an invisible Iframe directly under the mouse cursor such that an drop action attempts to put the attacker defined content into the Iframe's document.

```
1 function hover(e){
2   var x=document.getElementsByTagName("iframe")[0].style;
```

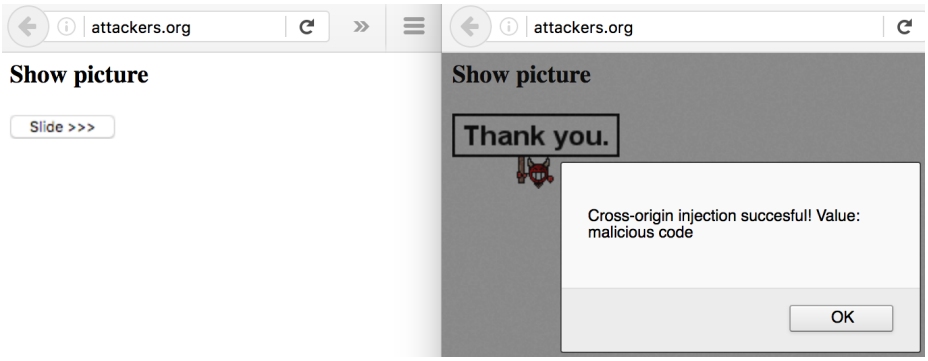

Fig. 2: Attacker defined content can be cross-origin injected.

```
  x.left=(e.clientX-60)+"px";
  x.top=(e.clientY-10)+"px";
  x.display='inline';
  x.opacity='0.0';
}
```

Listing 1.7: JavaScript code of the HD (scenario: drag-and-drop attack).

```
  <h3>Show picture</h3>
  <iframe src="a.html" style="position:fixed; display:none">
      </iframe>
  <div id="d" style="background-image:url('evil.png');
      height:1px; width:127px; opacity:0"></div>
  <img src="s.png" draggable="true" ondragstart="this.src='';
       event.dataTransfer.setData('text/plain','malicious
      code'); hover(event); var d=document.getElementById('d
      ').style; d.height='57px'; d.opacity='1'">
```

Listing 1.8: HTML code of the HD (scenario: drag-and-drop attack).

The Iframe's content is shown in Listing 1.9. It only consists of an input area and JavaScript code which shows an alert-window on the condition that the attacker defined content is dropped. Thus, the alert-window only appears in case that the proof-of-concept functions as expected. In a real world application, there could be a search engine in the background which automatically looks up the dropped user input by pulling XMLHttpRequest leading to a code injection, and thus to Cross-Site Scripting.

```
<script>
var t = setInterval(function() {
  if (document.getElementsByTagName("input")[0].value) {
    alert('Cross-origin injection succesful! Value: '+
        document.getElementsByTagName("input")[0].value);
    clearInterval(t);
  }
}, 500);
```

```
</script>
<input type="text" style="position:absolute; top:0px;
    left:0px">
```

Listing 1.9: HTML and JavaScript code of the ED (scenario: drag-and-drop attack).

6.2 Multiple Pop-Up Attack

As shown in Table 1, a pop-up window can be generated with a trustworthy event like a `click` within a delay which is shorter than one second. For FF and SA, we evaluated that more than one pop-up window will not be blocked once a single pop-up is generated. In contrast, GC, OP, IE, and Edge show one pop-up window and an additional warning window as an information about the blocking of the other pop-up windows.

```
<script>
function makePopups(){
  for (i=1;i<1000;i++) {
    window.open('x.html',i,'width=500,height=500');
  }
}
</script>
<a href="#" onclick="makePopups()">Spam</a>
```

Listing 1.10: HTML and JavaScript code of the ED (scenario: multiple pop-up attack).

An example is given in Listing 1.10. After a click on `Spam` the trustworthy event `click` is triggered and thus the function `makePopups()` is called. The function includes a for-loop which generates 1,000 windows that could be either pop-ups (this example) or new tabs (by removing the third parameter with `width` and `height`). In FF and SA, all of these windows are shown to the user. This behavior leads to a heavy memory consumption and thus heavily slows down the underlying system's speed. It is likely that a victim will close all browser windows simultaneously and for this reason, it may also lose existing browser sessions (e.g., in other tabs). Another use case is click-fraud by creating multiple pop-ups with advertisements; an attempt to close these unwanted windows could lead to an unintended click and thus a successfully clicked advertisement.

The behavior of FF unexpected due to browser settings that are reachable via `about:config`. Firstly, the property `dom.popup_maximum` (maximum number of pop-up windows) has a default value of `20`. We are clearly able to generate more windows with trustworthy events. Secondly, the property `dom.popup_allowed_events` (events that spawn pop-ups) has the value `change click dblclick mouseup notificationclick reset submit touchend`.

As shown in Table 1, we could also use other events like a left-click triggered `select` (not listed within `dom.popup_allowed_events`). Therefore, there is a lack of handling pop-up windows properly. We have reported these problems to Mozilla.

6.3 Hijacking Clipboard Data

In contrast to browsers like FF, GC, and even Edge, IE allows full access to the clipboard after a confirmation on a warning window (cf. Table 2). Clickjacking can be used to attack an IE user and thus to get access to the saved clipboard data that may contain sensitive data like a password.

We introduce two new attack sub-variants to steal clipboard data. Firstly by stealing the second click from a double-click scenario which was described by Huang et al. [12]. Secondly by just using a single click; this highlights the importance to look on different trustworthy events.

The first variant is displayed in Listing 1.11. With the help of social engineering, the attackers lures a user to make a double click on the displayed button. The first click of the double click triggers the `onclick` event-handler, which shows the accessed clipboard data in an alert window (as a proof-of-concept). For the clickjacking attack, the second click of the double-click actually occurs on the `Allow access` button of the confirmation window. To ensure that a user always hits the `Allow access` button, the `Double Click` button will always be positioned in the middle of the screen (with slide adjustments).

The second variant is targeting an impatient user. It consists nearly of the same code and displayed Figure, except for two changes. The `Double Click` button is named `DL in X` where X is a counter with a number in seconds which decreases until zero. An impatient user will wait until the button's counter reaches zero to download a file, and thus the click will be correctly timed. The attacker will therefore show the confirmation dialog 300ms before the button's counter reaches zero, such that the click will be successfully hijacked.

The limitation of both attack variants is that the confirmation window must be visible for at least 300 milliseconds; this is the lower bound we measured. The Human Benchmark Project[1] recorded over 51 million clicks and measured that the average reaction time of a human is 282 milliseconds (where the user was aware of being timed). Therefore, it is very likely that a user is not able to cancel the hijacked click on the confirmation window.

```
1 <style> button { position: fixed; top: 50%; left: 50%;
2   margin-top: 15px; margin-left: -20px; } </style>
3 <button onclick="if (window.clipboardData.getData('Text').
    length > 0) { alert('Hijacked Clipboard data: '+
    window.clipboardData.getData('Text')); }">
4 Double Click</button>
```

Listing 1.11: HTML and JavaScript code of the ED (scenario: clipboard attack).

7 Defenses Discussion

We have evaluated that trustworthy events are implemented differently across browsers. Our formal definition of trustworthy events and the thereby derived

[1] `http://www.humanbenchmark.com/tests/reactiontime/statistics`

descriptions of three different scenarios might help browser vendors to minimize the high number of event handling differences.

An approach to help browser vendors to avoid bugs and features that may lead to security vulnerabilities is to compare their browser result with the result of the majority of other modern browsers. For example, it may be suspicious if just one out of seven tested browsers allows access (or a particular interaction) after a trustworthy event; for clarification reasons, the set of browsers could be extended (e.g., by considering more browsers like Brave and Chromium).

Drag-and-Drop Attack. Drag-and-drop actions are known since the introduction of web browsers, which still allow restricted draggings of for example text elements (selected text), images (image URL), and anchor-elements (anchor URL). Moreover, HTML5 has introduced a drag-and-drop API [41] that is nowadays integrated in all modern web browsers.

We constructed a drag-and-drop attack variant that can be executed in three (IE 11, Edge 20, and FF 47) tested browsers. A simple but effective countermeasure is to prohibit drag-and-drop frame attacks by *disallowing drag operations with data across frames with different origins*. Browser vendors like Google and Opera allowed cross-frame drag-and-drop operations in the past; nowadays, this is not anymore possible due to security reasons (cf. Section 6.1)

Pop-Up Attack. FF is the only tested browser which allows creating hundreds of pop-ups after a trustworthy event like a left-click within the measured delay of one second. All other tested browsers disallow the execution of multiple pop-ups and therefore the user will not be annoyed when, for example, they appear unintentionally. The majority of our tested browser behavior results can therefore be used to derive a countermeasure for FF; this browser should *only show one pop-up window after a trustworthy event.*

Clipboard Data. Our clipboard data attack variant on IE showed that a user should not get an unlimited control over the whole clipboard data by just executing JavaScript code. For this reason, there are different access types (copy, cut, paste) that are implemented in modern browsers due to the W3C clipboard API [37]. However, the behavior of IE underlined that read access should only be allowed with a trustworthy event like a keystroke combination (e.g., `STRG+V`).

The countermeasure of disallowing clipboard read access is very strict and it might be more convenient to get only read access if the user explicitly gives the permission by *showing a clipboard permission window for a time that is significantly higher than the human response time*; this should be longer than the short display time of the IE permission window (cf. Section 6.3).

According to the Human benchmark project, only a negligible amount of the measurements (<0,1%) have a longer human response time than 500 milliseconds. As a consequence, a browser implementation should only activate the `Allow access` button of the permission window after a trustworthy event and a delay of at least half a second. This ensures with a high probability that the second click will not be hijacked by an attacker.

8 Related Work

Definitions & Specifications. Huang et al. [12] discussed UIR attacks and defenses with a definition of UIR. They developed a defense called in InContext to mitigate UIR attacks. The W3C created a UI safety specification [20] that is based on the ideas of InContext. Similar UI contexts are mentioned in the W3C UI security and visibility API [14]. These foundations of describing trusted events do not consider conscious user actions, which we define as trustworthy events. Without these events, UIR attacks could not be executed.

By looking at the concept of zones and scenarios, IE includes predefined zones like *Internet*, *Local Intranet*, and *Trusted Sites* [22]. This concept is partially adopted between browsers by explicitly white-listing trusted sites [11]. Trusted site lists can be used to manage whether certain actions should be automatically executed (e.g., generate cryptographic keys, play Flash files, and show pop-ups).

Attacks & Countermeasures. Grossman et al. [10] introduced Clickjacking as an attack which is nowadays considered as a class of attacks which relies on the broader set of UIR attacks. Although the attack on Flash received high media attention and several bugfixes since 2008 [2], it was successfully attacked years later (e.g., in 2011 [1]). Next to JavaScript-based frame busters [33], the HTTP Header `X-Frame-Options` [16,8], and nowadays even the Content-Security-Policy [36] can be used to defend against many types of UI redressing. In an evaluation about different JavaScript-based UIR protection mechanisms, Rydstedt et al. [33] pointed out that there exist attacks which can be used to attack protection mechanism and thus disable them. Balduzzi et al. [4] designed and implemented an automated system to analyze Clickjacking attacks. Niemietz et al. [29] evaluated the security of home routers and found that none of them are protected against UIR. Rydstedt et al. [32] published a paper about UIR on mobile sites and also on home routers.

Lekies et al. [17] presented bypasses for Clickjacking defense tools like NoScript's ClearClick. Furthermore, they introduced a new attack technique called nested Clickjacking. By showing that UI time delays as defense mechanisms are not sufficient to protect the user, Akhawe et al. [3] created examples which bypass the W3C UI safety specification [20].

Mobile Devices. Lin at al. [19] published Screenmilker, which analyzes the user interface of an Android device. By using the Android debug bride (ADB), they showed that Screenmilker is able to make screenshots during user interactions and they were able to steal secrets like passwords. Bianchi et al. [6] published a study on Android-based graphical user interface confusion attacks [128]. These attacks concentrate on phishing and privacy violations. Niemietz et al. enumerated different UIR attacks [27] and their countermeasures. Furthermore, they provide a Tapjacking attack to compromise Android devices [28]. Based on this work, Fratantonio et al. [9] created malicious apps that completely control the UI feedback loop. They furthermore showed with a user study that none of the created attacks could be detected by a user.

9 Conclusions

In this paper, we provide a definition of *trustworthy events*, which are the target of UI Redressing attacks. We show that this concept is significantly different from the concept of *trusted events* as defined by the W3C. Interpretations of events as being trustworthy differ significantly between browser families, and by a non-documented inheritance mechanism trustworthiness may be transferred, within the time frame of one second, from a trustworthy event to a sequence of events triggered by JavaScript. This, for example, allowed us to circumvent the FF pop-up blocker.

We investigated three scenarios where trustworthy events play a major role in protecting the security of web applications: pop-ups, drag-and-drop, and copy-and-paste. In all three scenarios, differences in the interpretation of trustworthy events could be shown. We refined one new example attack variant in each scenario, based on a more detailed investigation of these scenarios. Finally, we discuss defense mechanisms by analyzing the causes of our trustworthy event attacks. With the definition and description of *trustworthy events*, we hope that this paper will contribute to a better understanding of UIR attacks, and thus improved web application security.

References

1. Aboukhadijeh, F.: Spy on the webcams of your website visitors. `http://feross.org/webcam-spy/` (October 2011)
2. Aharonovsky, G.: Malicious camera spying using clickjacking. `MaliciouscameraspyingusingClickJacking` (October 2008)
3. Akhawe, D., He, W., Li, Z., Moazzezi, R., Song, D.: Clickjacking revisited: A perceptual view of ui security. In: 8th USENIX Workshop on Offensive Technologies (WOOT 14). USENIX Association, San Diego, CA (Aug 2014), `https://www.usenix.org/conference/woot14/workshop-program/presentation/akhawe`
4. Balduzzi, M., Egele, M., Kirda, E., Balzarotti, D., Kruegel, C.: A solution for the automated detection of clickjacking attacks. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security. pp. 135–144. ASIACCS '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1755688.1755706`
5. Barth, A.: The Web Origin Concept. IETF, RFC 6454 (December 2011), `http://tools.ietf.org/html/rfc6454`, `http://tools.ietf.org/html/rfc6454`
6. Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., Vigna, G.: What the app is that? deception and countermeasures in the android user interface. In: IEEE Symposium on Security and Privacy. Department of Computer Science, University of California, Santa Barbara (2015)
7. Bordi, E.: Cursorjacking proof of concept. `http://static.vulnerability.fr/noscript-cursorjacking.html` (August 2010)
8. Braun, F., Heiderich, M.: X-Frame-Options: All about Clickjacking? `https://cure53.de/xfo-clickjacking.pdf` (2013)
9. Fratantonio, Y., Qian, C., Chung, S., Lee, W.: Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In: Proceedings of the IEEE Symposium on Security and Privacy (Oakland). San Jose, CA (May 2017)

10. Hansen, R., Grossman, J.: Clickjacking attack. http://www.sectheory.com/clickjacking.htm (December 2008)
11. Help, G.C.: Allow or block content settings for certain sites. https://support.google.com/chrome/answer/3123708?hl=en (March 2017)
12. Huang, L.S., Moshchuk, A., Wang, H.J., Schecter, S., Jackson, C.: Clickjacking: Attacks and defenses. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). pp. 413–428. USENIX, Bellevue, WA (2012), https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/huang
13. Kacmarcik, G., Leithead, T.: Ui events – w3c working draft. https://www.w3.org/TR/uievents/ (August 2016)
14. Kaminsky, D., Huang, D.L.S., Maone, G.: W3c – user interface security and the visibility api. https://www.w3.org/TR/UISecurity/ (June 2016)
15. Kotowicz, K.: Cursorjacking again. http://blog.kotowicz.net/2012/01/cursorjacking-again.html (January 2012)
16. Lawrence, 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)
17. Lekies, S., Heiderich, M., Appelt, D., Holz, T.: On the fragility and limitations of current browser-provided clickjacking protection schemes. In: Presented as part of the 6th USENIX Workshop on Offensive Technologies. USENIX, Berkeley, CA (2012), https://www.usenix.org/conference/woot12/workshop-program/presentation/Lekies
18. Lekies, S., Heiderich, M., Appelt, D., Holz, T., Johns, M.: On the fragility and limitations of current browser-provided clickjacking protection schemes. In: in Usenix Workshop on Offensive Technologies (wOOt 2012) (2012)
19. Lin, C.C., Li, H., Zhou, X., Wang, X.: Screenmilker: How to milk your android screen for secrets. Network and Distributed System Security (NDSS) Symposium 2014 (2014)
20. Maone, G., Huang, D.L.S., Gondrom, T., Hill, B.: W3c – user interface security directives for content security policy. https://dvcs.w3.org/hg/user-interface-safety/raw-file/tip/user-interface-safety.html (June 2014)
21. Mayer, A., Niemietz, M., Mladenov, V., Schwenk, J.: Guardians of the clouds: When identity providers fail. CCSW 2014: The ACM Cloud Computing Security Workshop (2014)
22. Microsoft: How to use security zones in internet explorer. https://support.microsoft.com/en-us/help/174360/how-to-use-security-zones-in-internet-explorer (June 2012)
23. Needham, K.: The future of developing firefox add-ons. https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/ (August 2015)
24. Network, M.D.: Event.istrusted (February 2017), https://developer.mozilla.org/en-US/docs/Web/API/Event/isTrusted
25. Network, M.D.: Web apis – document.execcommand(). https://developer.mozilla.org/de/docs/Web/API/Document/execCommand (January 2017)
26. Niemietz, M.: Clickjacking und UI-Redressing - Vom Klick-Betrug zum Datenklau: Ein Leitfaden für Sicherheitsexperten und Webentwickler. dpunkt-Verlag (2012)
27. Niemietz, M.: UI Redressing:Attacks and Countermeasures Revisited. In: in CONFidence (May 2011)

28. Niemietz, M., Schwenk, J.: UI Redressing Attacks on Android Devices. https://media.blackhat.com/ad-12/Niemietz/bh-ad-12-androidmarcus_niemietz-WP.pdf (Dezember 2012)
29. Niemietz, M., Schwenk, J.: Owning your home network: Router security revisited. In: Web 2.0 Security & Privacy 2015, San Jose (CA). http://ieee-security.org/TC/SPW2015/W2SP/papers/W2SP_2015_submission_9.pdf (2015)
30. Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H., Cowan, C.: User-driven access control: Rethinking permission granting in modern operating systems. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 224–238 (May 2012)
31. Ruderman, J.: The same origin policy. Online, http://www-archive.mozilla.org/projects/security/components/same-origin.html (2008)
32. Rydstedt, G., Bursztein, E., Boneh, D.: Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization. In: in Usenix Workshop on Offensive Technologies (wOOt 2010) (2010), http://seclab.stanford.edu/websec/framebusting/tapjacking.pdf
33. Rydstedt, G., Bursztein, E., Boneh, D., Jackson, C.: Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In: in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010) (2010), http://seclab.stanford.edu/websec/framebusting/framebust.pdf
34. Sherman, I.: Making form-filling faster, easier and smarter. https://webmasters.googleblog.com/2012/01/making-form-filling-faster-easier-and.html (January 2012)
35. Sophos: Facebook worm - "likejacking". http://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/ (May 2010)
36. Stamm, S., Sterne, B., Markham, G.: Reining in the web with content security policy. In: Proceedings of the 19th International Conference on World Wide Web. pp. 921–930. WWW '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1772690.1772784
37. Steen, H.R.M.: W3c – clipboard api and events. https://www.w3.org/TR/clipboard-apis/ (December 2016)
38. Stone, P.: Next generation clickjacking – new attacks against framed web pages. https://www.contextis.com/documents/5/Context-Clickjacking_white_paper.pdf (April 2010)
39. W3C: W3c dom4: Dom event istrusted. https://www.w3.org/TR/dom/ (November 2015)
40. W3C: Ui events. https://w3c.github.io/uievents/ (January 2016)
41. WHATWG: Html, living standard – drag and drop. http://www.whatwg.org/specs/web-apps/current-work/multipage/dnd.html#dnd (November 2013)
42. WHATWG: Form control infrastructure. https://html.spec.whatwg.org/multipage/form-control-infrastructure.html (July 2017)
43. Zalewski, M.: Strokejacking. http://lcamtuf.blogspot.de/2010/06/curse-of-inverse-strokejacking.html (June 2010)