Clickjacking: Attacks and Defenses

Lin-Shung Huang
Carnegie Mellon University
linshung.huang@sv.cmu.edu

Alex Moshchuk
Microsoft Research
alexmos@microsoft.com

Helen J. Wang
Microsoft Research
helenw@microsoft.com

Stuart Schechter
Microsoft Research
stuart.schechter@microsoft.com

Collin Jackson
Carnegie Mellon University
collin.jackson@sv.cmu.edu

Abstract

Clickjacking attacks are an emerging threat on the web. In this paper, we design new clickjacking attack variants using existing techniques and demonstrate that existing clickjacking defenses are insufficient. Our attacks show that clickjacking can cause severe damages, including compromising a user's private webcam, email or other private data, and web surfing anonymity.

We observe the root cause of clickjacking is that an attacker application presents a sensitive UI element of a target application *out of context* to a user (such as hiding the sensitive UI by making it transparent), and hence the user is tricked to act out of context. To address this root cause, we propose a new defense, *InContext*, in which web sites (or applications) mark UI elements that are sensitive, and browsers (or OSes) enforce *context integrity* of user actions on these sensitive UI elements, ensuring that a user sees everything she should see before her action and that the timing of the action corresponds to her intent.

We have conducted user studies on Amazon Mechanical Turk with 2064 participants to evaluate the effectiveness of our attacks and our defense. We show that our attacks have success rates ranging from 43% to 98%, and our InContext defense can be very effective against the clickjacking attacks in which the use of clickjacking is more effective than social engineering.

1 Introduction

When multiple applications or web sites (or OS principals [44] in general) share a graphical display, they are subject to *clickjacking* [13] (also known as *UI redressing* [28, 49]) attacks: one principal may trick the user into interacting with (e.g., clicking, touching, or voice controlling) UI elements of another principal, triggering actions not intended by the user. For example, in *Likejacking* attacks [46], an attacker web page tricks users into clicking on a Facebook "Like" button by transparently overlaying it on top of an innocuous UI element,

such as a "claim your free iPad" button. Hence, when the user "claims" a free iPad, a story appears in the user's Facebook friends' news feed stating that she "likes" the attacker web site. For ease of exposition, our description will be in the context of web browsers. Nevertheless, the concepts and techniques described are generally applicable to all client operating systems where display is shared by mutually distrusting principals.

Several clickjacking defenses have been proposed and deployed for web browsers, but all have shortcomings. Today's most widely deployed defenses rely on *framebusting* [21, 37], which disallows a sensitive page from being framed (i.e., embedded within another web page). Unfortunately, framebusting is fundamentally incompatible with embeddable third-party widgets, such as Facebook Like buttons. Other existing defenses (discussed in Section 3.2) suffer from poor usability, incompatibility with existing web sites, or failure to defend against significant attack vectors.

To demonstrate the insufficiency of state-of-the-art defenses, we construct three new attack variants using existing clickjacking techniques. We designed the new attack scenarios to be more damaging than the existing clickjacking attacks in the face of current defenses. In one scenario, the often-assumed web-surfing-anonymity can be compromised. In another, a user's private data and emails can be stolen. Lastly, an attacker can spy on a user through her webcam. We have conducted the first clickjacking effectiveness study through Amazon Mechanical Turk and find that the aforementioned attacks have success rates of 98%, 47%, and 43%, respectively.

Learning from the lessons of existing defenses, we set the following design goals for our clickjacking defense:

- Widget compatibility: clickjacking protection should support third-party widgets.
- Usability: users should not be prompted for their actions.
- Backward compatibility: the defense should not break existing web sites (e.g., by disallowing exist-

ing functionality).

- Resiliency: our defense should address the root cause of clickjacking and be resilient to new attack vectors.

The root cause of clickjacking 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. For example, in the aforementioned Likejacking attack scenario, an attacker web page presents a false visual context to the user by hiding the sensitive "Like" button transparently on top of the "claim your free iPad" button.

To address the root cause and achieve the above goals, our defense, called *InContext*, lets web sites mark their sensitive UI elements and then lets browsers enforce the *context integrity* of user actions on the sensitive UI elements. The context of a user's action consists of its visual context and temporal context.

- *Visual context* is what a user should see right before her sensitive action. To ensure visual context integrity, both the sensitive UI element and the pointer feedback (such as cursors, touch feedback, or NUI input feedback) need to be fully visible to a user. We refer to the former as *target display integrity* and to the latter as *pointer integrity*.

- *Temporal context* refers to the timing of a user action. Temporal integrity ensures that a user action at a particular time is intended by the user. For example, an attack page can compromise temporal integrity by launching a *bait-and-switch* attack by first baiting the user with a "claim your free iPad" button and then switching in a sensitive UI element right before the anticipated time of user click.

We implemented a prototype of InContext on Internet Explorer 9 and found that it is practical to use, adding at most 30ms of delay for verifying a click. We evaluated InContext through Amazon Mechanical Turk user studies, and our results show that InContext is very effective against attacks in which the use of clickjacking is vital to attack effectiveness.

2 Threat Model

The primary attacker against which InContext defends is a *clickjacking attacker*. A clickjacking attacker has all the capabilities of a web attacker [17]: (1) they own a domain name and control content served from their web servers, and (2) they can make a victim visit their site, thereby rendering attacker's content in the victim's browser. When a victim user visits the attacker's page, the page hides a sensitive UI element either visually or temporally (see Section 3.1 for various techniques to achieve this) and lure the user into performing unintended UI actions on the sensitive element out of context.

We make no attempt to protect against social engineering attackers who can succeed in their attacks even when the system is perfectly designed and built. For example, a social engineering attacker can fool users into clicking on a Facebook Like button by drawing misleading content, such as images from a charity site, *around* it. Even though the Like button is not manipulated in any way, a victim may misinterpret the button as "liking" charity work rather "liking" the attacker web site, and the victim may have every intention to click on the button. In contrast, a clickjacking attacker exploits a system's inability to maintain context integrity for users' actions and thereby can manipulate the sensitive element visually or temporally to trick users.

3 Background and Related Work

In this section, we discuss known attacks and defenses for clickjacking, and compare them to our contributions. Below, we assume a victim user is visiting a clickjacking attacker's page, which embeds and manipulates the *target element* residing on a different domain, such as Facebook's Like button or PayPal's checkout dialog.

3.1 Existing clickjacking attacks

We classify existing attacks according to three ways of forcing the user into issuing input commands out of context: (1) compromising *target display integrity*, the guarantee that users can fully see and recognize the target element before an input action; (2) compromising *pointer integrity*, the guarantee that users can rely on cursor feedback to select locations for their input events; and (3) compromising *temporal integrity*, the guarantee that users have enough time to comprehend where they are clicking.

3.1.1 Compromising target display integrity

Hiding the target element. Modern browsers support HTML/CSS styling features that allow attackers to visually hide the target element but still route mouse events to it. For example, an attacker can make the target element transparent by wrapping it in a `div` container with a CSS `opacity` value of zero; to entice a victim to click on it, the attacker can draw a decoy *under* the target element by using a lower CSS `z-index` [13]. Alternatively, the attacker may completely cover the target element with an opaque decoy, but make the decoy unclickable by setting the CSS property `pointer-events:none` [4]. A victim's click would then fall through the decoy and land on the (invisible) target element.

Partial overlays. Sometimes, it is possible to visually confuse a victim by obscuring only a part of the target element [12, 41]. For example, attackers could overlay their own information on top of a PayPal checkout `iframe` to cover the recipient and amount fields while leaving the "Pay" button intact; the victim will thus have

incorrect context when clicking on "Pay". This overlaying can be done using CSS `z-index` or using Flash Player objects that are made topmost with Window Mode property [2] set to `wmode=direct`. Furthermore, a target element could be partially overlaid by an attacker's popup window [53].

Cropping. Alternatively, the attacker may crop the target element to only show a piece of the target element, such as the "Pay" button, by wrapping the target element in a new `iframe` that uses carefully chosen negative CSS position offsets and the Pay button's width and height [41]. An extreme variant of cropping is to create multiple 1x1 pixel containers of the target element and using single pixels to draw arbitrary clickable art.

3.1.2 Compromising pointer integrity

Proper visual context requires not only the target element, but also all pointer feedback to be fully visible and authentic. Unfortunately, an attacker may violate pointer integrity by displaying a fake cursor icon away from the pointer, known as *cursorjacking*. This leads victims to misinterpret a click's target, since they will have the wrong perception about the current cursor location. Using the CSS `cursor` property, an attacker can easily hide the default cursor and programmatically draw a fake cursor elsewhere [20], or alternatively set a custom mouse cursor icon to a deceptive image that has a cursor icon shifted several pixels off the original position [7].

Another variant of cursor manipulation involves the blinking cursor which indicates keyboard focus (e.g., when typing text into an input field). Vulnerabilities in major browsers have allowed attackers to manipulate keyboard focus using *strokejacking* attacks [50, 52]. For example, an attacker can embed the target element in a hidden frame, while asking users to type some text into a fake attacker-controlled input field. As the victim is typing, the attacker can momentarily switch keyboard focus to the target element. The blinking cursor confuses victims into thinking that they are typing text into the attacker's input field, whereas they are actually interacting with the target element.

3.1.3 Compromising temporal integrity

Attacks in the previous two sections manipulated visual context to trick the user into sending input to the wrong UI element. An orthogonal way of achieving the same goal is to manipulate UI elements after the user has decided to click, but before the actual click occurs. Humans typically require a few hundred milliseconds to react to visual changes [34, 54], and attackers can take advantage of our slow reaction to launch timing attacks.

For example, an attacker could move the target element (via CSS position properties) on top of a decoy button shortly after the victim hovers the cursor over the decoy, in anticipation of the click. To predict clicks more effectively, the attacker could ask the victim to repetitively click objects in a malicious game [1, 3, 54, 55] or to double-click on a decoy button, moving the target element over the decoy immediately after the first click [16, 33].

3.1.4 Consequences

To date, there have been two kinds of widespread clickjacking attacks in the wild: Tweetbomb [22] and Likejacking [46]. In both attacks, an attacker tricks victims to click on Twitter Tweet or Facebook Like buttons using hiding techniques described in Section 3.1.1, causing a link to the attacker's site to be reposted to the victim's friends and thus propagating the link virally. These attacks increase traffic to the attacker's site and harvest a large number of unwitting friends or followers.

Many proof-of-concept clickjacking techniques have also been published. Although the impact of these attacks in the wild is unclear, they do demonstrate more serious damages and motivate effective defenses. In one case [38], attackers steal user's private data by hijacking a button on the approval pages of the OAuth [10] protocol, which lets users share private resources such as photos or contacts across web sites without handing out credentials. Several attacks target the Flash Player webcam settings dialogs (shown in Figure 1), allowing rogue sites to access the victim's webcam and spy on the user [1, 3, 9]. Other POCs have forged votes in online polls, committed click fraud [11], uploaded private files via the HTML5 File API [19], stolen victims' location information [54], and injected content across domains (in an XSS spirit) by tricking the victim to perform a drag-and-drop action [18, 40].

3.2 Existing anti-clickjacking defenses

Although the same-origin policy [35] is supposed to protect distrusting web sites from interfering with one another, it fails to stop any of the clickjacking attacks we described above. As a result, several anti-clickjacking defenses have been proposed (many of such ideas were suggested by Zalewski [51]), and some have been deployed by browsers.

3.2.1 Protecting visual context

User Confirmation. One straightforward mitigation for preventing out-of-context clicks is to present a confirmation prompt to users when the target element has been clicked. Facebook currently deploys this approach for the Like button, asking for confirmation whenever requests come from blacklisted domains [47]. Unfortunately, this approach degrades user experience, especially on single-click buttons, and it is also vulnerable to double-click timing attacks of Section 3.1.3, which could trick the victim to click through both the target element

and a confirmation popup.

UI Randomization. Another technique to protect the target element is to randomize its UI layout [14]. For example, PayPal could randomize the position of the Pay button on its express checkout dialog to make it harder for the attacker to cover it with a decoy button. This is not robust, since the attacker may ask the victim to keep clicking until successfully guessing the Pay button's location.

Opaque Overlay Policy. The Gazelle web browser [45] forces all cross-origin frames to be rendered opaquely. However, this approach removes all transparency from *all* cross-origin elements, breaking benign sites.

Framebusting. A more effective defense is *framebusting*, or disallowing the target element from being rendered in `iframes`. This can be done either with JavaScript code in the target element which makes sure it is the top-level document [37], or with newly added browser support, using features called `X-Frame-Options` [21] and CSP's `frame-ancestors` [39]. A fundamental limitation of framebusting is its incompatibility with target elements that are intended to be framed by arbitrary third-party sites, such as Facebook Like buttons.[1] In addition, previous research found JavaScript framebusting unreliable [37], and in Section 4.2, we will show attacks that bypass framebusting protection on OAuth dialogs using popup windows. Zalewski has also demonstrated how to bypass framebusting by navigating browsing history with JavaScript [55].

Visibility Detection on Click. Instead of completely disallowing framing, an alternative is to allow rendering transparent frames, but block mouse clicks if the browser detects that the clicked cross-origin frame is not fully visible. Adobe has added such protection to Flash Player's webcam access dialog in response to webcam clickjacking attacks; however, their defense only protects that dialog and is not available for other web content.

The ClearClick module of the Firefox extension NoScript also uses this technique [23], comparing the bitmap of the clicked object on a given web page to the bitmap of that object rendered in isolation (e.g., without transparency inherited from a malicious parent element). Although ClearClick is reasonably effective at detecting visual context compromises, its on-by-default nature must assume that all cross-origin frames need clickjacking protection, which results in false positives on some sites. Due to these false positives, ClearClick prompts users to confirm their actions on suspected clickjacking attacks, posing a usability burden. An extension called

ClickIDS [5] was proposed to reduce the false positives of ClearClick by alerting users only when the clicked element overlaps with other clickable elements. Unfortunately, ClickIDS cannot detect attacks based on partial overlays or cropping, and it still yields false positives.

Finally, a fundamental limitation of techniques that verify browser-rendered bitmaps is that cursor icons are not captured; thus, pointer integrity is not guaranteed. To address this caveat, ClearClick checks the computed `cursor` style of the clicked element (or its ancestors) to detect cursor hiding. Unfortunately, cursor spoofing attacks can still be effective against some users even if the default cursor is visible over the target element, as discussed in Section 7.2.

3.2.2 Protecting temporal context

Although we're not aware of any timing attacks used in the wild, browser vendors have started to tackle these issues, particularly to protect browser security dialogs (e.g., for file downloads and extension installations) [34]. One common way to give users enough time to comprehend any UI changes is to impose a delay after displaying a dialog, so that users cannot interact with the dialog until the delay expires. This approach has been deployed in Flash Player's webcam access dialog, suggested in Zalewski's proposal [51], and also proposed in the Gazelle web browser [45]. In response to our vulnerability report, ClearClick has added a UI delay for cross-origin window interactions [24].

Unresponsive buttons during the UI delay have reportedly annoyed many users. The length of the UI delay is clearly a tradeoff between the user experience penalty and protection from timing attacks. Regardless, UI delay is not a complete answer to protecting temporal integrity, and we construct an attack that successfully defeats a UI delay defense in Section 4.3.

3.2.3 Access Control Gadgets

Access control gadgets (ACG) [30] were recently introduced as a new model for modern OSes to grant applications permissions to access user-owned resources such as camera or GPS. An ACG is a privileged UI which can be embedded by applications that need access to the resource represented by the ACG; authentic user actions on an ACG grant its embedding application permission to access the corresponding resource. The notion of ACGs is further generalized to application-specific ACGs, allowing applications to require authentic user actions for application-specific functionality. Application-specific ACGs precisely capture today's web widgets that demand a clickjacking defense.

ACGs require clickjacking resilience. While Roesner et al's design [30] considered maintaining both visual and temporal integrity, they did not consider pointer in-

---

[1] `X-Frame-Options` and `frame-ancestors` both allow specifying a whitelist of sites that may embed the target element, but doing so is often impractical: Facebook would have to whitelist much of the web for the Like button!

Figure 1: **Cursor spoofing attack page.** *The target Flash Player webcam settings dialog is at the bottom right of the page, with a "skip this ad" bait link remotely above it. Note there are two cursors displayed on the page: a fake cursor is drawn over the "skip this ad" link while the actual pointer hovers over the webcam access "Allow" button.*

tegrity and did not evaluate various design parameters. In this work, we comprehensively address these issues, and we also establish the taxonomy of context integrity explicitly.

3.2.4 Discussion

We can conclude that all existing clickjacking defenses fall short in some way, with robustness and site compatibility being the main issues. Moreover, a glaring omission in all existing defenses is the pointer integrity attacks described in Section 3.1.2. In Section 5, we will introduce a browser defense that (1) does not require user prompts, unlike ClearClick and Facebook's Likejacking defense, (2) provides pointer integrity, (3) supports target elements that require arbitrary third-party embedding, unlike framebusting, (4) lets sites opt in by indicating target elements, avoiding false positives that exist in ClearClick, and (5) is more robust against timing attacks than the existing UI delay techniques.

3.3 Our contributions

The major contributions of this paper are in evaluating the effectiveness of existing attack techniques as well as designing and evaluating a new defense. Our evaluation uses several new attack variants (described in Section 4) which build on existing techniques described in Section 3.1, including cursor manipulation, fast-paced object clicking, and double-click timing. Whereas most existing proof-of-concepts have focused on compromising target display integrity, we focus our analysis on pointer integrity and temporal integrity, as well as on combining several known techniques in novel ways to increase effectiveness and bypass all known defenses.

4 New Attack Variants

To demonstrate the insufficiency of state-of-the-art defenses described above, we construct and evaluate three

attack variants using known clickjacking techniques. We have designed the new attack scenarios to be potentially more damaging than existing clickjacking attacks in the face of current defenses. We describe each in turn.

4.1 Cursor spoofing attack to steal webcam access

We first crafted a cursor spoofing attack (Section 3.1.2) to steal access to a private resource of a user: the webcam. In this attack, the user is presented with an attack page shown in Figure 1. A fake cursor is programmatically rendered to provide false feedback of pointer location to the user, in which the fake cursor gradually shifts away from the hidden real cursor while the pointer is moving. A loud video ad plays automatically, leading the user to click on a "skip this ad" link. If the user moves the fake cursor to click on the skip link, the real click actually lands on the Adobe Flash Player webcam settings dialog that grants the site permission to access the user's webcam. The cursor hiding is achieved by setting the CSS `cursor` property to `none`, or a custom cursor icon that is completely transparent, depending on browser support.

4.2 Double-click attack to steal user private data

Today's browsers do not protect temporal integrity for web sites. We show in our second attack that even if a security-critical web page (such as an OAuth dialog page) successfully employs framebusting (refusing to be embedded by other sites), our attack can still successfully clickjack such a page by compromising temporal integrity for popup windows.

We devised a bait-and-switch double-click attack (Section 3.1.3) against the OAuth dialog for Google accounts, which is protected with `X-Frame-Options`. The attack is shown in Figure 2. First, the attack page baits the user to perform a double-click on a decoy button. After the first click, the attacker switches in the Google OAuth pop-up window under the cursor right before the
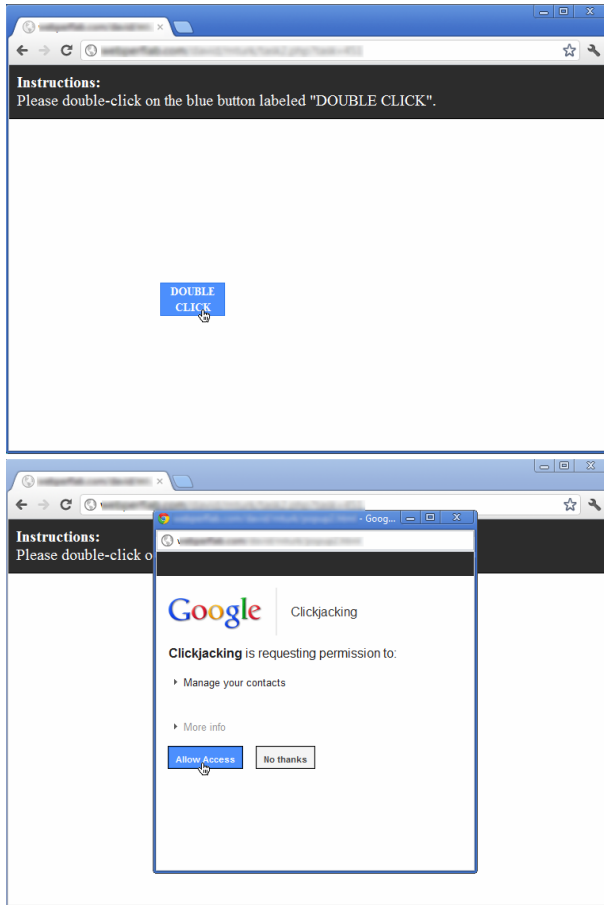
Figure 2: **Double-click attack page.** *The target OAuth dialog popup window appears underneath the pointer immediately after the first click on the decoy button.*

second click (the second half of the double-click). This attack can steal a user's emails and other private data from the user's Google account.

The double-click attack technique was previously discussed in the context of extension installation dialogs by Ruderman [33].

4.3 Whack-a-mole attack to compromise web surfing anonymity

In our third attack, we combine the approaches from the previous two attacks, cursor spoofing and bait-and-switch, to launch a more sophisticated whack-a-mole attack that combines clickjacking with social plugins (e.g., Facebook Like button) to compromise web surfing anonymity.

In this attack, we ask the user to play a whack-a-mole game and encourage her to score high and earn rewards by clicking on buttons shown at random screen locations as fast as possible. Throughout the game, we use a fake cursor to control where the user's attention should be. At a later point in the game, we switch in a Facebook Like button at the real cursor's location, tricking the user to
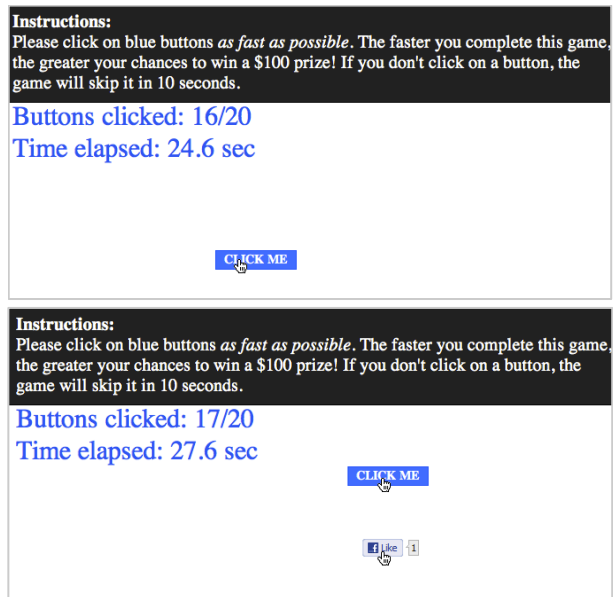

Figure 3: **Whack-a-mole attack page.** *This is a cursor spoofing variant of the whack-a-mole attack. On the 18th trial, the attacker displays the target Like button underneath the actual pointer.*

click on it.

In 2010, Wondracek et al. [48] showed that it is feasible for a malicious web site to uniquely identify 42% of social network users that use groups by exploiting browsing history leaks. Fortunately, the history sniffing technique required in their attack is no longer feasible in major browsers due to Baron's patch [6]. However, we find that our whack-a-mole attack above, and Like-jacking attacks in general, can still easily reveal the victim's real identity at the time of visit and compromise user anonymity in web surfing as follows.

Consider an attacker who is an admin for a Facebook page; the attacker crafts a separate malicious page which tricks users to click on his Like button. That page is notified when a victim clicks on the Like button via FB.Event.subscribe(), triggering the attacker's server to pull his fan list from Facebook and instantly identify the newly added fan. The attacker's server could then query the victim's profile via Facebook Graph API (and remove the victim fan to avoid suspicion). While we implemented this logic as a proof-of-concept and verified its effectiveness, we did not test it on real users.

In Section 7, we show our results on the effectiveness of all these attacks on Mechanical Turk users.

5 InContext Defense

As described in Section 1, the root cause of clickjacking is that an attacker application presents a sensitive UI element of a target application *out of context* to the user, and hence the user gets tricked to act out of context.

Enforcing context integrity for an application is essentially one aspect of application isolation, in addition to memory and other resource access. Namely, the context for a user's action in the application should be protected from manipulation by other applications. We believe it is an OS's (or a browser's) role to provide such cross-application (or cross-web-site) protection.

Section 1 introduced two dimensions of context integrity: visual and temporal. Enforcing visual integrity ensures that the user is presented with what she should see before an input action. Enforcing temporal integrity ensures that the user has enough time to comprehend what UI element they are interacting with.

We describe our design for each in turn.

5.1 Ensuring Visual Integrity

To ensure visual integrity at the time of a sensitive user action, the system needs to make the display of both the sensitive UI elements and the pointer feedback (such as cursors, touch feedback, or NUI input feedback) fully visible to the user. Only when both the former (*target display integrity*) and the latter (*pointer integrity*) are satisfied, the system *activates* sensitive UI elements and delivers user input to them.

5.1.1 Guaranteeing Target Display Integrity

Although it is possible to enforce the display integrity of all the UI elements of an application, doing so would make all the UI elements inactivated if any part of the UI is invisible. This would burden users to make the entire application UI unobstructed to carry out any interactions with the application. Such whole-application display integrity is often not necessary. For example, not all web pages of a web site contain sensitive operations and are susceptible to clickjacking. Since only applications know which UI elements require protection, we let web sites indicate which UI elements or web pages are *sensitive*. This is analogous to how HTML5 [43] and some browsers [32] (as well as earlier research on MashupOS [44]) allow web sites to label certain content as "sandboxed". The sandboxed content is isolated so that it cannot attack the embedding page. In contrast, the *sensitive* content is protected with context integrity for user actions, so that the embedding page cannot clickjack the sensitive content.

We considered several design alternatives for providing target display integrity, as follows.

Strawman 1: CSS Checking. A naïve approach is to let the browser check the computed CSS styles of elements, such as the position, size, `opacity` and `z-index`, and make sure the sensitive element is not overlaid by cross-origin elements. However, various techniques exist to bypass CSS and steal topmost display, such as using IE's `createPopup()` method [25] or Flash Player's Window
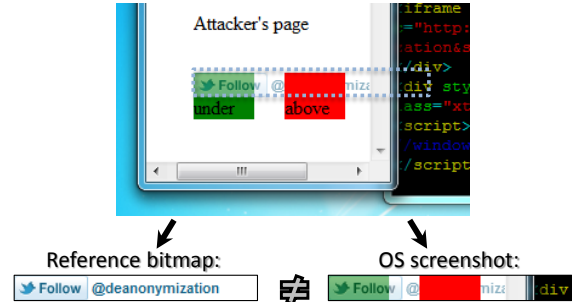

Figure 4: **Ensuring target element display integrity.** *Here, the attacker violates visual context of the Twitter Follow button by changing its opacity and obstructing it with two DIVs. InContext detects this during its bitmap comparison. Obstructions from other windows are also detected (e.g., the non-browser Vi window on the right).*

Mode [2]. Solely relying on CSS checking is not reliable and thus insufficient.

Strawman 2: Static reference bitmap. Another approach is to let a web site provide a static bitmap of its sensitive element as a reference, and let the browser make sure the rendered sensitive element matches the reference. Flash Player uses this approach for protecting its webcam access dialog (Section 3.2.1). However, different browsers may produce slightly differently rendered bitmaps from the same HTML code, and it would be too burdensome for developers to serve different reference bitmaps for different browsers. Furthermore, this approach fails when sensitive elements contain animated content, such as button `mouseover` effects, or dynamically generated content, such as the amount to pay in a checkout UI.

Our design. InContext enforces target display integrity by comparing the OS-level screenshot of the area that contains the sensitive element (what the user sees), and the bitmap of the sensitive element rendered in isolation at the time of user action. If these two bitmaps are not the same, then the user action is canceled and not delivered to the sensitive element. Figure 4 illustrates this process.

In the Likejacking attack example in Section 1, when a user clicks on the "claim your iPad" button, the transparent Facebook Like button is actually clicked, as the browser unconditionally delivered the click event to the Facebook Like button. With our defense, Facebook can label its Like button web page as "sensitive" in the corresponding HTTP response. The browser will then perform the following tasks before delivering each click event to the Like button. The browser first determines what the user sees at the position of the Like button on the screen by taking a screenshot of the browser window and cropping the sensitive element from the screenshot based on the element's position and dimenions known by the browser. The browser then determines what the

sensitive element should look like if rendered in isolation and uses this as a reference bitmap. To this end, the browser draws the sensitive element on a blank surface and extracts its bitmap. The browser then compares the cropped screenshot with the reference bitmap. A mismatch here means that the user does not fully see the Like button but her click targets the Like button. In this case, the browser detects a potential clickjacking offense and cancels the delivery of the click event. Instead, it triggers a new `oninvalidclick` event to give the application an opportunity to deal with such occasions.

This design is resilient to new visual spoofing attack vectors because it uses only the position and dimension information from the browser layout engine to determine what the user sees. This is much easier to get right than relying on other sophisticated logic (such as CSS) from the layout engine to determine what the user sees. By obtaining the reference bitmap at the time of the user action on a sensitive UI element, this design works well with dynamic aspects (such as animations or movies) in a sensitive UI element, unlike Strawman 2 above.

We also enforce that a host page cannot apply any CSS transforms [42] (such as zooming, rotating, etc.) that affect embedded sensitive elements; any such transformations will be ignored by InContext-capable browsers. This will prevent malicious zooming attacks [36], which change visual context via zoom. We also disallow any transparency inside the sensitive element itself. Although doing so may have a compatibility cost in terms of preventing legitimate blending effects of the sensitive element with the host page, we believe this is a necessary restriction, since otherwise attackers could violate visual context by inserting decoys that could show through the sensitive element.

Our bitmap comparison is similar to ClearClick (Section 3.2.1), with two crucial differences: (1) We use OS APIs to take a screenshot of the browser window, rather than relying on the browser to generate screenshots, making it more robust to rendering performed by Flash Player and other plug-ins, and (2) our approach is opt-in, eliminating false positives and obviating user prompts.

5.1.2 Guaranteeing Pointer Integrity

Without pointer integrity support, an attacker could spoof the real pointer. For example, an attack page may show a fake cursor to shift the user's attention from the real cursor and cause the user to act out of context by not looking at the destination of her action. To mitigate this, we must ensure that users see system-provided (rather than attacker-simulated) cursors and pay attention to the right place before interacting with a sensitive element.

For our design, we consider the following techniques, individually and in various combinations, to under-stand the tradeoff between their effectiveness of stopping pointer-spoofing attacks and intrusiveness to users. Some of the techniques limit the attackers' ability to carry out pointer-spoofing attacks; others draw attention to a particular place on the screen.

No cursor customization. Current browsers disallow cross-origin cursor customization. We further restrict this policy: when a sensitive element is present, InContext disables cursor customization on the host page (which embeds the sensitive element) and on all of the host's ancestors, so that a user will always see the system cursor in the areas surrounding the sensitive element.

Our opt-in design is better than completely disallowing cursor customization, because a web site may want to customize the pointer for its own UIs (i.e., same-origin customization). For example, a text editor may want to show different cursors depending on whether the user is editing text or selecting a menu item.

Screen freezing around sensitive element. Since humans typically pay more attention to animated objects than static ones [15], attackers could try to distract a user away from her actions with animations. To counter this, InContext "freezes" the screen (i.e., ignores rendering updates) around a sensitive UI element when the cursor enters the element.

Muting. Sound could also draw a user's attention away from her actions. For example, a voice may instruct the user to perform certain tasks, and loud noise could trigger a user to quickly look for a way to stop the noise. To stop sound distractions, we mute the speakers when a user interacts with sensitive elements.

Lightbox around sensitive element. Greyout (also called Lightbox) effects are commonly used for focusing the user's attention on a particular part of the screen (such as a popup dialog). In our system, we apply this effect by overlaying a dark mask on all rendered content *around* the sensitive UI element whenever the cursor is within that element's area. This causes the sensitive element to stand out visually.

The mask cannot be a static one. Otherwise, an attacker could use the same static mask in its application to dilute the attention-drawing effect of the mask. Instead, we use a randomly generated mask which consists of a random gray value at each pixel.

No programmatic cross-origin keyboard focus changes. To stop strokejacking attacks that steal keyboard focus (see Section 3.1.2), once the sensitive UI element acquires keyboard focus (e.g., for typing text in an input field), we disallow programmatic changes of keyboard focus by other origins.

Discussion. This list of techniques is by no means exhaustive. For example, sensitive elements could also draw the user's attention with splash animation effects on the cursor or the element [15].

Our goal was to come up with a representative set of techniques with different security and usability tradeoffs, and conduct user studies to evaluate their effectiveness as a design guide. We hope that this methodology can be adopted by browser vendors to evaluate a wider range of techniques with a larger-scale user study for production implementations.

5.2 Ensuring Temporal Integrity

Even with visual integrity, an attacker can still take a user's action out of context by compromising its temporal integrity, as described in Section 3.1.3. For example, a timing attack could bait the user with a "claim your free iPad" button and then switch in a sensitive UI element (with visual integrity) at the expected time of user click. The bait-and-switch attack is similar to time-of-check-to-time-of-use (TOCTTOU) race conditions in software programs. The only difference is that the race condition happens to a human rather than a program. To mitigate such TOCTTOU race conditions on users, we impose the following constraints for a user action on a sensitive UI element:

UI delay. We apply this existing technique (discussed in Section 3.2.2) to only deliver user actions to the sensitive element if the visual context has been the same for a minimal time period. For example, in the earlier bait-and-switch attack, the click on the sensitive UI element will not be delivered unless the sensitive element (together with the pointer integrity protection such as grey-out mask around the sensitive element) has been fully visible and stationary long enough. We evaluate trade-offs of a few delays in Section 7.3.

UI delay after pointer entry. The UI delay technique above is vulnerable to the whack-a-mole attack (Section 4.3) that combines pointer spoofing with rapid object clicking. A stronger variant on the UI delay is to impose the delay not after changes to visual context, but each time the pointer *enters* the sensitive element. Note that the plain UI delay may still be necessary, e.g., on touch devices which have no pointer.

Pointer re-entry on a newly visible sensitive element. In this novel technique, when a sensitive UI element first appears or is moved to a location where it will overlap with the current location of the pointer, an InContext-capable browser invalidates input events until the user explicitly moves the pointer from the outside of the sensitive element to the inside. Note that an alternate design of automatically moving the pointer outside the sensitive element could be misused by attackers to programmatically move the pointer, and thus we do not use it. Obviously, this defense only applies to devices and OSes that provide pointer feedback.

Padding area around sensitive element. The sensitive UI element's padding area (i.e., extra whitespace separat-

Sensitive Element	Dimensions	Click Delay	Memory Overhead
Facebook Like	90x20 px	12.04 ms	5.11 MB
Twitter Follow	200x20 px	13.54 ms	8.60 MB
Animated GIF (1.5 fps)	468x60 px	14.92 ms	7.90 MB
Google OAuth	450x275 px	24.78 ms	12.95 MB
PayPal Checkout	385x550 px	30.88 ms	15.74 MB

Table 1: **Performance of InContext.** *For each sensitive element, this table shows extra latency imposed on each click, as well as extra memory used.*

ing the host page from the embedded sensitive element) needs to be thick enough so that a user can clearly decide whether the pointer is on the sensitive element or on its embedding page. As well, this ensures that during rapid cursor movements, such as those in the whack-a-mole attack (Section 4.3), our pointer integrity defenses such as screen freezing are activated early enough. Sections 7.2 and 7.4 give a preliminary evaluation on some padding thickness values. The padding could be either enforced by the browser or implemented by the developer of the sensitive element; we have decided the latter is more appropriate to keep developers in control of their page layout.

5.3 Opt-in API

In our design, web sites must express which elements are sensitive to the browser. There are two options for the opt-in API: a JavaScript API and an HTTP response header. The JavaScript API's advantages include ability to detect client support for our defense as well as to handle `oninvalidclick` events raised when clickjacking is detected. On the other hand, the header approach is simpler as it doesn't require script modifications, and it does not need to deal with attacks that disable scripting on the sensitive element [37]. We note that bitmap comparison functions should not be directly exposed in JavaScript (and can only be triggered by user-initiated actions). Otherwise, they might be misused to probe pixels across origins using a transparent frame.

6 Prototype Implementation

We built a prototype of InContext using Internet Explorer 9's public COM interfaces. We implemented the pixel comparison between an OS screenshot and a sensitive element rendered on a blank surface to detect element visibility as described in Section 5.1.1, using the GDI `BitBlt` function to take desktop screenshots and using the MSHTML `IHTMLElementRender` interface to generate reference bitmaps.

To implement the UI delays, we reset the UI delay timer whenever the top-level window is focused, and whenever the computed position or size of the sensitive element has changed. We check these conditions whenever the sensitive element is repainted, *before* the actual

9

paint event; we detect paint events using IE binary behaviors [27] with the `IHTMLPainter::Draw` API. We also reset the UI delay timer whenever the sensitive element becomes fully visible (e.g., when an element obscuring it moves away) by using our visibility checking functions above. When the user clicks on the sensitive element, InContext checks the elapsed time since the last event that changed visual context.

Our prototype makes the granularity of sensitive elements to be HTML documents (this includes iframes); alternately, one may consider enabling protection for finer-grained elements such as DIVs. For the opt-in mechanism, we implemented the Javascript API of Section 5.3 using the `window.external` feature of IE.

Although our implementation is IE and Windows-specific, we believe these techniques should be feasible in other browsers and as well. For example, most platforms support a screenshot API, and we found an API similar to IE's `IHTMLElementRender` in Firefox to render reference bitmaps of an HTML element.

At this time, we did not implement the pointer integrity defenses, although we have evaluated their effects in Section 7.

Performance. To prove that InContext is practical, we evaluated our prototype on five real-world sensitive elements (see Table 1). For each element, we measured the memory usage and click processing time for loading a blank page that embeds each element in a freshly started browser, with and without InContext, averaging over ten runs. Our testing machine was equipped with Intel Xeon CPU W3530 @ 2.80 GHz and 6 GB of RAM.

Without additional effort on code optimization, we find that our average click processing delay is only 30 ms in the worst case. This delay is imposed only on clicks on sensitive elements, and should be imperceptible to most users. We find that the majority (61%) of the click delay is spent in the OS screenshot functions (averaging 11.65 ms). We believe these could be significantly optimized, but this is not our focus in this paper.

7 Experiments

7.1 Experimental design

In February of 2012 we posted a Human Interactive Task (HIT) at Amazon's Mechanical Turk to recruit prospective participants for our experiments. Participants were offered 25 cents to "follow the on-screen instructions and complete an interactive task" by visiting the web site at which we hosted our experiments. Participants were told the task would take roughly 60 seconds. Each task consisted of a unique combination of a simulated attack and, in some cases, a simulated defense. After each attack, we asked a series of follow-up questions. We then disclosed the existence of the attack and explained that since it was simulated, it could only result in clicking on harmless simulated functionality (e.g., a fake Like button).

We wanted participants to behave as they would if lured to a third-party web site with which they were previously unfamiliar. We hosted our experiments at a web site with a domain name unaffiliated with our research institution so as to ensure that participants' trust (or distrust) in our research institution would not cause them to behave in a more (or less) trusting manner.

For attacks targeting Flash Player and access to video cameras (webcams), we required that participants have Flash Player installed in their browser and have a webcam attached. We used a SWF file to verify that Flash Player was running and that a webcam was present. For attacks loading popup windows, we required that participants were not using IE or Opera browsers since our attack pages were not optimized for them.

We recruited a total of 3521 participants.[2] Participants were assigned uniformly and at random to one of 27 (between-subjects) treatment groups. There were 10 treatment groups for the cursor-spoofing attacks, 4 for the double-click attacks, and 13 for the whack-a-mole attacks. Recruiting for all treatments in parallel eliminated any possible confounding temporal factors that might result if different groups were recruited or performed tasks at different times. We present results for each of these three sets of attacks separately.

In our analysis, we excluded data from 370 participants who we identified (by worker IDs) have previously participated in this experiment or earlier versions of it. We also discarded data from 1087 participants who were assigned to treatment groups for whack-a-mole attacks that targeted Facebook's Like button but who could not be confirmed as being logged into Facebook (using the technique described in [8]). In Tables 2, 3 and 4, we report data collected from the remaining 2064 participants.

Except when stated otherwise, we use a two-tailed Fisher's Exact Test when testing whether differences between attack rates in different treatment groups are significant enough to indicate a difference in the general population. This test is similar to χ^2, but more conservative when comparing smaller sample sizes.

7.2 Cursor-spoofing attacks

In our first experiment, we test the efficacy of the cursor-spoofing attack page, described in Section 4.1 and illustrated in Figure 1, and of the pointer integrity defenses we proposed in Section 5.1.2. The results for each treatment group make up the rows of Table 2. The columns show the number of users that clicked on the "Skip ad" link (Skip), quit the task with no pay (Quit), clicked on

Treatment Group	Total	Timeout	Skip	Quit	Attack Success
1. Base control	68	26	35	3	4 (5%)
2. Persuasion control	73	65	0	2	6 (8%)
3. Attack	72	38	0	3	31 (43%)
4. No cursor styles	72	34	23	3	12 (16%)
5a. Freezing (M=0px)	70	52	0	7	11 (15%)
5b. Freezing (M=10px)	72	60	0	3	9 (12%)
5c. Freezing (M=20px)	72	63	0	6	3 (4%)
6. Muting + 5c	70	66	0	2	2 (2%)
7. Lightbox + 5c	71	66	0	3	2 (2%)
8. Lightbox + 6	71	60	0	8	3 (4%)

Table 2: **Results of the cursor-spoofing attack.** *Our attack tricked 43% of participants to click on a button that would grant webcam access. Several of our proposed defenses reduced the rate of clicking to the level expected if no attack had occurred.*

Figure 5: **Cursor-spoofing attack with lightbox defenses.** *The intensity of each pixel outside of the target element is darkened and randomized when the actual pointer hovers on the target element.*

webcam "Allow" button (Attack success), and those who watched the ad full video and were forwarded to the end of the task with no clicks (Timeout).

Control. We included a control group, Group 1, which contained an operational skip button, a Flash webcam access dialog, but no attack to trick the user into clicking the webcam access button while attempting to click the skip button. We included this group to determine the click rate that we would hope a defense could achieve in countering an attack. We anticipated that some users might click on the button to grant webcam access simply out of curiosity. In fact, four did. We were surprised that 26 of the 68 participants waited until the full 60 seconds of video completed, even though the "skip ad" button was available and had not been tampered with. In future studies, we may consider using a video that is longer, more annoying, and that does not come from a charity that users may feel guilty clicking through.

We added a second control, Group 2, in which we removed the "skip ad" link and instructed participants to click on the target "Allow" button to skip the video ad. This control represents one attempt to persuade users to grant access to the webcam without tricking them. As with Group 1, we could consider a defense successful it rendered attacks no more successful than using persuasion to convince users to allow access to the webcam.

Whereas 4 of 68 (5%) participants randomly assigned to the persuasion-free control treatment (Group 1) clicked on the "Allow" button, we observed that 6 of 73 (8%) participants assigned to the persuasion control treatment did so. However, the difference in the attack success rates of Group 1 and Group 2 were not significant, with a two-tailed Fisher's exact test yielding p=0.7464.

Attack. Participants in Group 3 were exposed to the simulated cursor spoofing attack, with no defenses to protect them. The attack succeeded against 31 of 72 participants (43%). The difference in the attack success rates between participants assigned to the non-persuasion control treat-
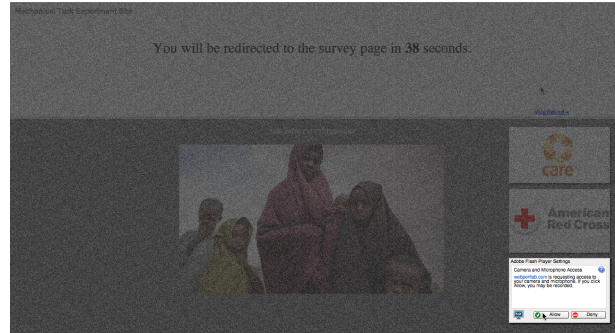
ment (Group 1) and the attack treatment (Group 3) is statistically significant (p<0.0001). The attack might have been even more successful had participants been given a more compelling motivation to skip the "skip this ad" link. Recall that only 51% of participants in the non-persuasion control treatment (Group 1) tried to skip the animation. If we assume the same percent of participants tried to skip the advertisement during the attack, then 84% of those participants who tried to skip the ad fell for the attack (43% of 51%).

Defenses. One straightforward defense against cursor-spoofing attacks is to disallow cursor customization. This would prevent the real cursor from being hidden, though the attack page could still draw a second, fake cursor. Some victims might focus on the wrong cursor and fall for the attack. In Group 4, we disallowed cursor customization and found that 12 of 72 (16%) participants still fell for the attack. This result, along with attacker's ability to draw multiple fake cursors and emphasize one that is not the real cursor, suggest this defense has limited effectiveness. Nevertheless, the defense does appear to make a dent in the problem, as there is a reduction in attack success rates from Group 3 (43%), without the defense, to Group 4 (16%), with the defense, and the difference between these two treatment groups was statistically significant (p=0.0009).

In Groups 5a-c, we deployed the freezing defense described in Section 5.1.2: when this defense triggers, all movement outside the protected region, including the video and fake cursor, is halted. This helps break the illusion of the fake cursor and draws the user's attention to the part of the screen on which there is still movement—that which contains the real cursor. The freezing effect will not help if users have already initiated a click before noticing it. We thus initiate the freeze when the cursor is within M pixels of the webcam dialog, for M of 0, 10, and 20 pixels. At M=20px (Group 5c), the attack success rate dropped to that of our non-persuasion control group,

Treatment Group	Total	Timeout	Quit	Attack Success
1. Attack	90	46	1	43 (47%)
2a. UI Delay (T_A=250ms)	91	89	0	2 (2%)
2b. UI Delay (T_A=500ms)	89	86	2	1 (1%)
3. Pointer re-entry	88	88	0	0 (0%)

Table 3: **Results of double-click attack.** 43 *of* 90 *participants fell for the attack that would grant access to their personal Google data. Two of our defenses stopped the attack completely.*

tricking only 3 of 72 (4%). Fewer participants assigned to the 20px-border-freezing defense of Group 5c fell for the attack (4%) than those in the cursor-customization-defense treatment of Group 4 (16%), and this difference was significant (p=0.0311).

Given the efficacy of the large-margin (20px) freezing defense in Group 5c, and the low rate of successful attacks on which to improve, our sample was far too small to detect any further benefits that might result from muting the speaker or freezing portions of the screen with a lightbox might provide. Augmenting the freezing defense to mute the computer's speaker (Group 6) yielded a similar attack success rate of 2 of 70 (2%) participants. Augmenting that defense again with a lightbox, greying over the frozen region as described in Section 5.1.2, (Groups 7 and 8) also resulted in attack success rates of 2-4%. The lightbox effect is a somewhat jarring user experience, and our experiments do not provide evidence that this user-experience cost is offset by a measurably superior defense. However, larger sample sizes or different attack variants may reveal benefits that our experiment was unable to uncover.

7.3 Double-click attacks

In our second experiment, we tested the efficacy of the double-click timing attack (described in Section 4.2 and shown in Figure 2) and the defenses proposed in Section 5.2. The attack attempts to trick the user into clicking on the "Allow Access" button of a Google OAuth window by moving it underneath the user's cursor after the first click of a double-click on a decoy button. If the "Allow" button is not clicked within two seconds, the attack times out without success (column Timeout). The results of each treatment group appear as rows of Table 3.

Attack. Of the 90 participants assigned to the treatment in which they were exposed to the simulated attack without any defense to protect them, the attack was successful against 43 of them (47%). If this had been a real attack, we could have accessed their GMail to read their personal messages or download their contacts. Furthermore, many of the users who were not successfully attacked escaped because the popup was not shown quickly enough. Indeed, the popup took more than 500ms to be displayed for 31 of 46 users who timed out on the attack (with 833ms average loading time for those users)—

likely greater than a typical user's double-click speed. The attack efficacy could likely be improved further by pre-loading the OAuth dialog in a *pop-under* window (by de-focusing the popup window) and refocusing the pop-under window between the two clicks; this would avoid popup creation cost during the attack.

Defenses. Two groups of participants were protected by simulating the UI delay defense described in Section 5.2—we treated clicks on the "Allow" button as invalid until after it has been fully visible for a threshold of T_A ms. We assigned a treatment group for two choices for T_A: 250ms (Group 2a), the mode of double-click intervals of participants in an early mouse experiment [29] in 1984, and 500ms (Group 2b), the default double-click interval in Windows (the time after which the second click would be counted as a second click, rather than the second half of a double-click) [26]. We observed that the delay of 250ms was effective, though it was not long enough for 2 out of 91 (2%) participants in Group 2a, who still fell for the attack. The difference in attack success rates between the attack treatment (Group 1) and the UI delay defense treatment for T_A=250ms (Group 2a) was significant (p<0.0001). Similarly, the 500ms delay stopped the attack for all but 1 of 89 (1%) participants in Group 2b.

We also simulated our *pointer re-entry* defense (Group 3), which invalidated UI events on the OAuth dialog until the cursor has explicitly transitioned from outside of the OAuth dialog to inside. This defense was 100% effective for 88 participants in Group 3. The difference in attack success rates between the attack treatment (Group 1) and the pointer re-entry defense treatment (Group 3) was significant (p<0.0001). While the attack success rate reduction from the delay defense (Groups 2a and 2b) to the pointer re-entry defense (Group 3) was not statistically significant, the pointer re-entry defense is preferable for other reasons; it does not constrain the timing with which users can click on buttons, and it cannot be gamed by attacks that might attempt to introduce delays—one can imagine an attack claiming to test the steadiness of a user's hand by asking him to move the mouse to a position, close his eyes, and press the button after five seconds.

7.4 Whack-a-mole attacks

Next, we tested the efficacy of the fast-paced button clicking attack, the *whack-a-mole* attack, described in Section 4.3 and shown in Figure 3. In attempt to increase the attack success rates, as a real attacker would do, we offered a $100 performance-based prize to keep users engaged in the game. In this experiment, we used the Facebook's Like button as the target element (except for Group 1b, where for the purposes of comparison, the Flash Player webcam settings dialog was also

Treatment Group	Total	Timeout	Quit	Attack Success	Attack Success (On 1st Mouseover)	Attack Success (Filter by Survey)
1a. Attack without clickjacking	84	1	0	83 (98%)	N/A	42/43 (97%)
1b. Attack without clickjacking (webcam)	71	1	1	69 (97%)	N/A	13/13 (100%)
2. Attack with timing	84	3	1	80 (95%)	80 (95%)	49/50 (98%)
3. Attack with cursor-spoofing	84	0	1	83 (98%)	78 (92%)	52/52 (100%)
4a. Combined defense (M=0px)	77	0	1	76 (98%)	42 (54%)	54/54 (100%)
4b. Combined defense (M=10px)	78	10	1	67 (85%)	27 (34%)	45/53 (84%)
4c. Combined defense (M=20px)	73	18	4	51 (69%)	12 (16%)	31/45 (68%)
5. Lightbox + 4c	73	21	0	52 (71%)	10 (13%)	24/35 (68%)
6a. Entry delay (T_E=250ms) + 4c	77	27	4	46 (59%)	6 (7%)	27/44 (61%)
6b. Entry delay (T_E=500ms) + 4c	73	25	3	45 (61%)	3 (4%)	31/45 (68%)
6c. Entry delay (T_E=1000ms) + 4c	71	25	1	45 (63%)	1 (1%)	25/38 (65%)
6d. Entry delay (T_E=500ms) + 4a	77	6	0	71 (92%)	16 (20%)	46/49 (93%)
7. Lightbox + 6b	73	19	0	54 (73%)	6 (8%)	34/46 (73%)

Table 4: **Results of the whack-a-mole attack.**
98% of participants were vulnerable to Likejacking de-anonymization under the attack that combined whack-a-mole with cursor-spoofing. Several defenses showed a dramatic drop in attack success rates, reducing them to as low as 1% when filtered by first mouseover events.

tested). We checked whether the participant was logged into Facebook [8] and excluded data from users that were not logged in. The results for each treatment group appear in the rows of Table 4. The "Timeout" column represents those participants who did not click on the target button within 10 seconds, and were thus considered to not have fallen for the attack.

We calculated the attack success rate with three different methods, presented in three separate columns. The first Attack Success column shows the total number of users that clicked on the Like button. However, after analyzing our logs, we realized that this metric is not necessarily accurate: many people appeared to notice the Like button and moved their mouse around it for several seconds before eventually deciding to click on it. For these users, it was not clickjacking that ultimately caused the attack, but rather it was the users' willingness to knowingly click on the Like button after noticing it (e.g., due to wanting to finish the game faster, or deciding that they did not mind clicking it, perhaps not understanding the consequences). For the purposes of evaluating our defense, we wanted to filter out these users: our defenses are only designed to stop users from clicking on UI elements *unknowingly*.

We used two different filters to try to isolate those victims who clicked on the Like button unknowingly. The first defined an attack to be successful if and only if the victim's cursor entered the Like button only once before the victim click. This *on first mouseover* filter excludes victims who are moving their mouse around the Like button and deliberating whether or not to click. The second filter uses responses from our post-task survey to exclude participants who stated that they noticed the Like button and clicked on it knowingly, shown in column "Attack Success (Filter by Survey)". We asked the participants the following questions, one at a time, revealing each question after the previous question was answered:

```
1. Did you see the Facebook Like button at any point
```

```
   in this task? <displayed an image of Like button>
2. (If No to 1) Would you approve if your Facebook
   wall showed that you like this page?
3. (If Yes to 1) Did you click on the Like button?
4. (If Yes to 3) Did you intend to click on the Like
   button?
```

We only included participants who either did not approve "liking" (No to 2), were not aware that they "liked" (No to 3) or did not intend to "like" (No to 4). This excludes victims who do not care about "liking" the attacker's page and who intentionally clicked on the Like button. We expected the two filters to yield similar results; however, as we describe later, the trust in our survey responses was reduced by indications that participants lied in their answers. Therefore, we rely on the *on first mouseover* column for evaluating and comparing our defenses.

Attacks. We assigned two treatment groups to a simulated whack-a-mole attack that did not employ clickjacking. The first (Group 1a) eventually were shown a Like button to click on whereas the second (Group 1b) were eventually shown the "allow" button in the Flash webcam access dialog. In the simulated attack, participants first had to click on a myriad of buttons, many of which were designed to habituate participants into ignoring the possibility that these buttons might have context outside their role in the game. These included buttons that contained the text "great," "awesome," and smiley face icons. On the attack iteration, the Like button simply appeared to be the next target object to press in the game. We hypothesized that users could be trained to ignore the semantics usually associated with a user interface element if it appeared within this game.

Though we had designed this attack, its efficacy surprised even us. The Like button version of Group 1a succeeded on 83 of 84 (98%) participants and the "allow" button of Group 1b succeeded on 69 of 71 (97%) participants. The differences between these two groups are not

statistically significant. The attacks were also so effective that, at these sample sizes, they left no room in which to find statistically significant improvements through the use of clickjacking.

In the whack-a-mole attack with timing (Group 2), the Like button is switched to cover one of the game buttons at a time chosen to anticipate the user's click. This attack was also effective, fooling 80 of 84 (95%) participants in Group 2. Next, we combined the timing technique with cursor spoofing that we also used in Section 7.2, so that the game is played with a fake cursor, with the attack (Group 3) succeeding on 83 of 84 (98%) participants.

Defenses. In Groups 4a-c, we combined the proposed defenses that were individually effective against the previous cursor-spoofing and the double-click attacks, including pointer re-entry, appearance delay of T_A=500ms, and display freezing with padding area size M=0px, 10px and 20px. We assumed that the attacker could be aware of our defenses; e.g., our attack compensated for the appearance delay by substituting the Like button roughly 500ms before the anticipated user click.

Using no padding area (M=0px), the attack succeeded on the first mouseover on 42 of 77 (54%) of the participants in Group 4a. The reduction in the first-mouseover-success rate from Group 3 (without defense) to 4a (with the M=0px combined defense) was statistically significant, with $p<0.0001$. So, while all of the participants in Group 4a eventually clicked on the Like button, the defense caused more users to move their mouse away from the Like button before clicking on it. Increasing the padding area to M=10px (Group 4b) further reduced the first-mouseover success rate to 27 of 78 (34%), and the maximum padding area tested (M=20px, Group 4c) resulted in a further reduction to 12 of 73 (16%). The reduction in the first-mouseover attack success rates between Groups 4a and 4b was statistically significant (p=0.0155), as was the reduction from Groups 4b to 4c (p=0.0151). We also noticed that adding a 10px padding area even reduced the unfiltered attack success rate from 76 of 77 (98%) in Group 4a to 67 of 78 (85%) in Group 4b, and a 20px padding area further reduced the unfiltered attack success rate to 51 of 73 (69%) in Group 4c. The reduction in the unfiltered attack success rates between Groups 4a and 4b was also statistically significant (p=0.0046), as was the reduction from Groups 4b to 4c (p=0.0191). Thus, larger padding areas provide noticeably better clickjacking protection. Participants assigned to Group 5 received the defense of 4c enhanced with a lightbox, which further decreased the first-mouseover attack effectiveness to 10 of 73 (13%). The difference in first-mouseover success rates between Group 4c and 5 was not statistically significant (p=0.8176).

Note that there is a large discrepancy comparing first-mouseover attack success to the survey-filtered attack success. After analyzing our event logs manually, we realized that many users answered our survey questions inaccurately. For example, some people told us that they didn't click on the Like button, and they wouldn't approve clicking on it, whereas the logs show that while their initial click was blocked by our defense, they continued moving the mouse around for several seconds before finally resolving to click the Like button. While these users' answers suggested that clickjacking protection should have stopped them, our defenses clearly had no chance of stopping these kinds of scenarios.

Participants assigned to Groups 6a-d were protected by the pointer-entry delay defense described in Section 5.2: if the user clicks within a duration of T_E ms of the pointer entering the target region, the click is invalid. In Groups 6a and 6b, we experiment with a pointer entry delay of T_E=250ms and T_E=500ms, respectively. We used an appearance delay of T_A=500ms and a padding area of M=20px as in Group 4c. In both cases, we observed that the addition of pointer entry delay was highly effective. Only 3 of 73 (4%) participants in Group 6b still clicked on the target button. We found a significant difference in attack success rate between Groups 4c and 6b (p=0.0264), indicating that the pointer entry delay helps stopping clickjacking attacks, compared to no pointer entry delays. We then test a more extreme pointer entry delay of T_E=1000ms, in which the appearance delay T_A must also be adjusted to no less than 1000ms. This was most successful in preventing clickjacking from succeeding: only 1 of 71 (1%) participants fell for the attack. We also tested the pointer entry delay T_E=500ms without a padding area (M=0px), which allowed 16 of 77 (20%) participants in Group 6d to fall for the attack. Note that the difference in first-mouseover success rates between Groups 6b and 6d was significant (p=0.0026). Again, our results suggest that attacks are much more effective when there is no padding area around the target. Finally, in Group 7 we tested the lightbox effect in addition to Group 6b. The attack succeeded on 6 of 73 (8%) participants in Group 7, in which the difference between Groups 6b and 7 was not statistically significant (p=0.4938).

Overall, we found that pointer entry delay was crucial in reducing the first-mouseover success rate, the part of the attack's efficacy that could potentially be addressed by a clickjacking defense. Thus, it is an important technique that should be included in a browser's clickjacking protection suite, alongside freezing with a sufficiently large padding area, and the pointer re-entry protection. The pointer entry delay subsumes, and may be used in place of, the appearance delay. The only exception would be for devices that have no pointer feedback; having an appearance delay could still prove useful against a whack-a-mole-like touch-based attack.

7.5 Ethics

The ethical elements of our study were reviewed as per our research institution's requirements. No participants were actually attacked in the course of our experiments; the images they were tricked to click appeared identical to sensitive third-party embedded content elements, but were actually harmless replicas. However, participants may have realized that they had been tricked and this discovery could potentially lead to anxiety. Thus, after the simulated attack we not only disclosed the attack but explained that it was simulated.

8 Conclusion

We have devised new clickjacking attack variants, which bypass existing defenses and cause more severe harm than previously known, such as compromising webcams, user data, and web surfing anonymity.

To defend against clickjacking in a fundamental way, we have proposed InContext, a web browser or OS mechanism to ensure that a user's action on a sensitive UI element is in context, having visual integrity and temporal integrity.

Our user studies on Amazon Mechanical Turk show that our attacks are highly effective with success rates ranging from 43% to 98%. Our InContext defense can be very effective for clickjacking attacks in which the use of clickjacking improves the attack effectiveness.

This paper made the following contributions:

- We provided a survey of existing clickjacking attacks and defenses.
- We conducted the first user study on the effectiveness of clickjacking attacks.
- We introduced the concept of *context integrity* and used it to define and characterize clickjacking attacks and their root causes.
- We designed, implemented, and evaluated InContext, a set of techniques to maintain context integrity and defeat clickjacking.

With all these results, we advocate browser vendors and client OS vendors to consider adopting InContext.

Acknowledgments

References

[1] F. Aboukhadijeh. HOW TO: Spy on the Webcams of Your Website Visitors. `http://www.feross.org/webcam-spy/`, 2011.

[2] Adobe. Flash OBJECT and EMBED tag attributes. `http://kb2.adobe.com/cps/127/tn_12701.html`, 2011.

[3] G. Aharonovsky. Malicious camera spying using ClickJacking. `http://blog.guya.net/2008/10/07/malicious-camera-spying-using-clickjacking/`, 2008.

[4] L. C. Aun. Clickjacking with pointer-events. `http://jsbin.com/imuca`.

[5] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[6] D. Baron. Preventing attacks on a user's history through CSS :visited selectors. `http://dbaron.org/mozilla/visited-privacy`, 2010.

[7] E. Bordi. Proof of Concept - CursorJacking (noScript). `http://static.vulnerability.fr/noscript-cursorjacking.html`.

[8] M. Cardwell. Abusing HTTP Status Codes to Expose Private Information. `https://grepular.com/Abusing_HTTP_Status_Codes_to_Expose_Private_Information`, 2011.

[9] J. Grossman. Clickjacking: Web pages can see and hear you. `http://jeremiahgrossman.blogspot.com/2008/10/clickjacking-web-pages-can-see-and-hear.html`, 2008.

[10] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), Apr. 2010.

[11] R. Hansen. Stealing mouse clicks for banner fraud. `http://ha.ckers.org/blog/20070116/stealing-mouse-clicks-for-banner-fraud/`, 2007.

[12] R. Hansen. Clickjacking details. `http://ha.ckers.org/blog/20081007/clickjacking-details/`, 2008.

[13] R. Hansen and J. Grossman. Clickjacking. `http://www.sectheory.com/clickjacking.htm`, 2008.

[14] B. Hill. Adaptive user interface randomization as an anti-clickjacking strategy. `http://www.thesecuritypractice.com/the_security_practice/papers/AdaptiveUserInterfaceRandomization.pdf`, May 2012.

[15] R. Hoffmann, P. Baudisch, and D. S. Weld. Evaluating visual cues for switching windows on large screens. In *Proceedings of the 26th annual SIGCHI conference on Human factors in computing systems*, 2008.

[16] L.-S. Huang and C. Jackson. Clickjacking attacks unresolved. `http://mayscript.com/blog/david/clickjacking-attacks-unresolved`, 2011.

[17] C. Jackson. Improving browser security policies. PhD thesis, Stanford University, 2009.

[18] K. Kotowicz. Exploiting the unexploitable XSS with clickjacking. `http://blog.kotowicz.net/2011/03/exploiting-unexploitable-xss-with.html`, 2011.

[19] K. Kotowicz. Filejacking: How to make a file server from your browser (with HTML5 of course). `http://blog.kotowicz.net/2011/04/how-to-`

make-file-server-from-your.html, 2011.

[20] K. Kotowicz. Cursorjacking again. http://blog.kotowicz.net/2012/01/cursorjacking-again.html, 2012.

[21] E. Lawrence. IE8 Security Part VII: ClickJacking Defenses. http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx, 2009.

[22] M. Mahemoff. Explaining the "Don't Click" Clickjacking Tweetbomb. http://softwareas.com/explaining-the-dont-click-clickjacking-tweetbomb, 2009.

[23] G. Maone. Hello ClearClick, Goodbye Clickjacking! http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/, 2008.

[24] G. Maone. Fancy Clickjacking, Tougher NoScript. http://hackademix.net/2011/07/11/fancy-clickjacking-tougher-noscript/, 2011.

[25] Microsoft. createPopup Method. http://msdn.microsoft.com/en-us/library/ms536392(v=vs.85).aspx.

[26] Microsoft. SetDoubleClickTime function. http://msdn.microsoft.com/en-us/library/windows/desktop/ms646263(v=vs.85).aspx.

[27] Microsoft. Implementing Binary DHTML Behaviors. http://msdn.microsoft.com/en-us/library/ie/aa744100(v=vs.85).aspx, 2012.

[28] M. Niemietz. UI Redressing: Attacks and Countermeasures Revisited. In *CONFidence*, 2011.

[29] L. A. Price. Studying the mouse for CAD systems. In *Proceedings of the 21st Design Automation Conference*, 1984.

[30] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE Symposium on Security and Privacy*, 2012.

[31] J. Ross, L. Irani, M. S. Silberman, A. Zaldivar, and B. Tomlinson. Who are the crowdworkers?: shifting demographics in mechanical turk. In *Proceedings of the 28th International Conference On Human Factors In Computing Systems*, 2010.

[32] J. Rossi. Defense in depth: Locking down mashups with HTML5 Sandbox. http://blogs.msdn.com/b/ie/archive/2011/07/14/defense-in-depth-locking-down-mash-ups-with-html5-sandbox.aspx?Redirected=true, 2011.

[33] J. Ruderman. Bug 162020 - pop up XPInstall/security dialog when user is about to click. https://bugzilla.mozilla.org/show_bug.cgi?id=162020, 2002.

[34] J. Ruderman. Race conditions in security dialogs. http://www.squarefree.com/2004/07/01/race-conditions-in-security-dialogs/, 2004.

[35] J. Ruderman. The Same Origin Policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2011.

[36] G. Rydstedt, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization. In *USENIX Workshop on Offensive Technologies*, 2010.

[37] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *Proceedings of the Web 2.0 Security and Privacy*, 2010.

[38] S. Sclafani. Clickjacking & OAuth. http://stephensclafani.com/2009/05/04/clickjacking-oauth/, 2009.

[39] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[40] P. Stone. Next generation clickjacking. In *Black Hat Europe*, 2010.

[41] E. Vela. About CSS Attacks. http://sirdarckcat.blogspot.com/2008/10/about-css-attacks.html, 2008.

[42] W3C. CSS 2D Transforms. http://www.w3.org/TR/css3-2d-transforms/, 2011.

[43] W3C. HTML5, 2012. http://www.w3.org/TR/html5/.

[44] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions in MashupOS. In *ACM Symposium on Operating System Principles*, 2007.

[45] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[46] Wikipedia. Likejacking. http://en.wikipedia.org/wiki/Clickjacking#Likejacking.

[47] C. Wisniewski. Facebook adds speed bump to slow down likejackers. http://nakedsecurity.sophos.com/2011/03/30/facebook-adds-speed-bump-to-slow-down-likejackers/, 2011.

[48] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *Proceedings of the 31th IEEE Symposium on Security and Privacy*, 2010.

[49] M. Zalewski. Browser security handbook. http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_(UI_redressing).

[50] M. Zalewski. Firefox focus stealing vulnerability (possibly other browsers). http://seclists.org/fulldisclosure/2007/Feb/226, 2007.

[51] M. Zalewski. [whatwg] Dealing with UI redress vulnerabilities inherent to the current web. http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-September/016284.html, 2008.

[52] M. Zalewski. The curse of inverse strokejacking. http://lcamtuf.blogspot.com/2010/06/curse-of-inverse-strokejacking.html, 2010.

[53] M. Zalewski. Minor browser UI nitpicking. http://seclists.org/fulldisclosure/2010/Dec/328, 2010.

[54] M. Zalewski. On designing UIs for non-robots. http://lcamtuf.blogspot.com/2010/08/on-designing-uis-for-non-robots.html, 2010.

[55] M. Zalewski. X-Frame-Options, or solving the wrong problem. http://lcamtuf.blogspot.com/2011/12/x-frame-options-or-solving-wrong.html, 2011.