ADAPTIVE USER INTERFACE RANDOMIZATION AS AN ANTI-CLICKJACKING STRATEGY

Brad Hill <bhill at paypal-inc.com>

Version 1.0, 18 May 2012

Abstract

Clickjacking, a subclass of "User Interface Redressing" attacks, is a threat against web applications arising from the combination of ambient authority and multiple browsing contexts available in many web user agent programs. Users can be tricked into clicking on obscured user interface elements of an application and in so doing initiate actions against their will, such as adding an attacker to a victim's social graph, promoting the attacker's content on a social network, or sending a payment to the attacker. Some technical countermeasures exist in web browsers, but offer incomplete protection or prohibit useful and legitimate constructs such as IFRAMEs in third-party browsing contexts. This paper describes a method for combining randomization of user interface elements with statistical analysis of first click success rates across a population to provide an effective and adaptive method of detecting and responding to clickjacking campaigns. Though not a general purpose solution to clickjacking, the method requires no modifications to existing web user agents and is applicable to many of the most widely deployed and commonly attacked use cases for which no other mitigations currently exist. The technique can also be effectively combined with client-side approaches to enhance the effectiveness of both.

Keywords: clickjacking, anti-clickjacking, UI redressing, UI randomization

1 INTRODUCTION

Clickjacking[1] is the most widely known and prominently exploited subclass of a larger set of client-side web application attacks known as "User Interface (UI) Redressing".[2] In a clickjacking attack, a user is tricked into clicking on a link and performing some authenticated action when they believe they are doing something else. UI redressing attacks generally, and clickjacking specifically, arise from a combination of properties common to web user agents (AKA web browsers): ambient authority and multiple browsing contexts.

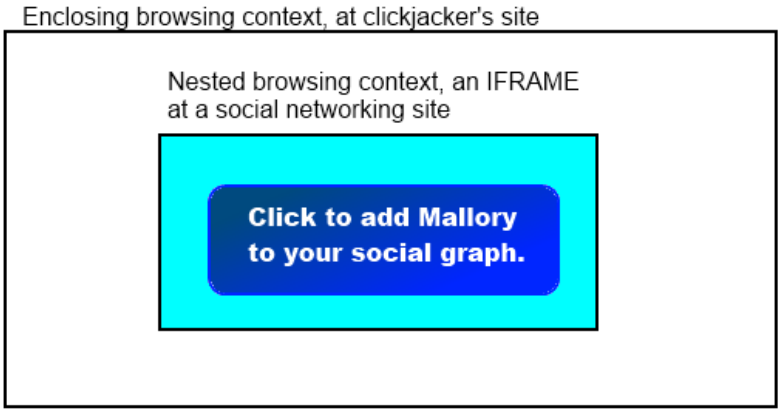
The first necessary property to enable clickjacking is "ambient authority". In a typical web browser, this means that a user's identity and privileges for a given web application (typically represented by

an HTTP cookie) are persistent and implicit in all interactions between the web user agent and that application, regardless of context. Once a user has logged in and received a session cookie it will be sent with all requests made from different tabs, browser windows, frames and often even in requests originating from unrelated applications. (until it expires) Although some modern user agents are providing mechanisms to isolate cookies between browsing contexts, in general the ambient authority of cookies is standard, expected and often desired by web application authors. It allows users to have contextual information (e.g. what friends have interacted with a given third party resource) and powerful interactions (e.g. adding a node to a social graph with a single click) available in "mash-up" type applications without needing to explicitly log-in and authorize every action individually.

Abuse of ambient authority by causing a user agent to send cross-origin requests is a form of Confused Deputy[3] attack known as Cross-Site Request Forgery. (CSRF)[4] CSRF is a well-known problem and can be defended against by including an un-guessable token explicitly with each request that originates from the same-origin application. This ensures that requests can be associated with a web application's own user interface, instead of relying only on the ambient authority of the user agent. Clickjacking attempts to circumvent this protection and re-gain the capabilities of CSRF by directly enlisting the human user, instead of the web browser, as the confused deputy.

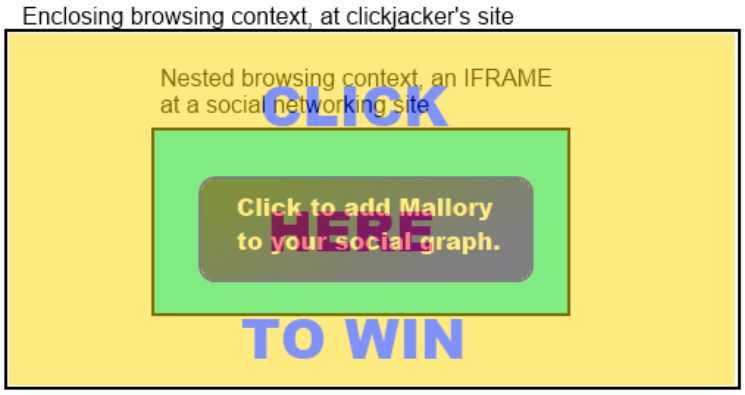
In a clickjacking attack, the attacker induces the user to interact with a browsing context containing content under the attacker's control, but actually sends the user's input to a different browsing context containing the web application under attack. For example, the attacker displays a game interface and uses it to overlay a social networking site in an IFRAME. The user believes he has clicked on an element of the game, but the click is actually delivered to a button on the social networking site. CSRF defenses are bypassed because the user's click is delivered to, and the action therefore initiated from, the browsing context of the genuine application. The user has thus unknowingly added the attacker to his social graph. A variety of techniques have been used to execute such attacks, including overlays, rapid movement or closing of the browsing context, and spoofing of a mouse cursor.[5]

A basic clickjacking attack is illustrated by the following nested browsing contexts. The user is logged into a social network, and visiting a site under the attacker's control. The attacker uses an IFRAME to create a nested browsing context to the social networking site, with a button that will add the attacker to the user's social graph:



The attacker overlays the nested browsing context so it appears to the user as:

But the overlay is set not to accept click events, which are delivered to the hidden interface below:



Although clickjacking attacks are often not taken seriously by web application authors, they remain a persistent and difficult-to-mitigate threat for application authors. Although the risk may be acceptable for many scenarios, it is quite serious for some of the most common use cases. A fraudulent payment can be reversed, but it is impossible for a user to claw-back the personal information exposed when they give an attacker access to their social network (e.g. by adding an identity thief as a friend), or reverse the reputational damage from being tricked into adding

objectionable content to such networks. (e.g. by clicking "like" or "+1" for pornographic content or hate speech)

2 PREVIOUS WORK

One widely adopted countermeasure to clickjacking is the "X-Frame-Options" HTTP header.[6] This allows a web resource to communicate to a user agent that it should not be rendered in a frame, or only in a frame of the same origin. This provides good protection for interfaces that are not intended to be displayed in a nested browsing context, but has several shortcomings. First, it does not provide a solution for user interface experiences that deliberately use nested browsing contexts. Social networking and payment sites will often use the Same Origin Policy isolation combined with ambient authority to provide a seamless, contextualized experience for users at a wide variety of third-party sites, without disclosing the user's identity to such sites. Second, sites may still be vulnerable to clickjacking attacks using browsing contexts other than frames, such as with "pop-under and close" techniques.[7]

The browser extension NoScript[8] provides an anti-clickjacking technology called ClearClick.[9] ClearClick compares screenshots of the composed view seen by a user with the browsing context to which a click is delivered, as if rendered alone. (David Lin-Shung Huang has done similar, unpublished work.) If comparison of these screenshots shows a discrepancy between these views, the user is warned and shown an un-obscured rendering of the browsing context to which their click was delivered in a browser-chrome pop-up.

Although a strong technique, it is subject to false positives, necessitating user interaction to warn and ask users to confirm when attacks are suspected – a message that non-technical users are unprepared to interpret. The screenshot comparison technique also does not solve an attack class known as the "phantom mouse cursor" – in which the genuine mouse cursor is visually reduced or made invisible, and a false cursor is rendered at an offset. In this case, the genuine user interface may be fully exposed at the time a user clicks. ClearClick incorporates checks to warn if the mouse cursor has been modified by CSS effects, but other variants on the technique can still be effective.

NoScript also has the disadvantage of being a separately-installed add-on currently only available on Firefox, where it has an install base of approximately 1% as of this writing. Web sites that may be victimized by clickjacking cannot rely on the availability of ClearClick to protect their user base.

The Web Application Security Working Group at the W3C has chartered work to produce a standard recommendation for browser technology to combat clickjacking, but that work has not progressed beyond exploratory concepts and any solution arrived at will take several years to diffuse through the general user agent population.

3 USER INTERFACE RANDOMIZATION

One of the limitations that a clickjacking attack must contend with is the Same Origin Policy. (SOP)[10] The SOP provides a limited security boundary between browsing contexts from different origins,

where an origin is defined by the set of the host, scheme and port from which content originated. Clickjacking attacks always occur cross-origin, so the restrictions of the SOP are always applicable. Most important of these restrictions for the purpose of the mitigation proposed here is that content from one origin is prevented from reading content from another origin.

The SOP read restriction means that an application mounting a clickjacking attack cannot directly observe the rendering of the browsing context it is attacking. To construct an attack, the attacker must be able to predict the layout of the target application on the victim's browser. Usually, this is quite predictable – every user receives the same basic layout for a given application, and the attacker can observe what user agent a victim is using and control the size and positioning of the target browsing context.

Because of this limitation, user interface randomization was one of the first ideas suggested as a countermeasure when clickjacking was first identified by the web security community.[11] It was rejected for a number of reasons:

- Randomization among a small number of locations still gives an attacker reasonable odds of success. Phishing attacks, for example, have remained a persistent and profitable attack even with success rates of less than 5%.
- Randomization among a large number of locations, or interactive randomization (button-chasing) provides a poor user experience.
- An attacker can induce the victim to send multiple clicks to the target application, exhausting the randomized possibilities.

The adaptive randomization technique described here uses back-end statistical analysis to overcome these limitations for certain classes of user interface today vulnerable to clickjacking.

The technique is most easily applicable to single-button interfaces, and the description will proceed with that assumption. It can be generalized to somewhat more complex interfaces, but is probably unsuitable for an application interface as complex as a mail client or word processor.

The first refinement to the basic idea of randomization of the placement of a button is that the button must be randomized within a space that does not contain other user interface elements, and the application must record "missed" clicks to possible locations, as well as clicks to the actual button.

Starting with this, a naïve algorithm could deactivate the genuine button after a missed click, or several. This prevents an attacker from sending multiple clicks to exhaust the possible locations for the button, but:

- Creates a poor user experience if the user accidentally misses a click when not being attacked
- Requires a very large space of possible targets to reduce the chances of success enough to deter the attack
- May not prevent the attacker from re-loading and trying again

3.1 "BUCKETIZATION"

The next refinement to make UI randomization useful is to record a population of user clicks and place them into groups or buckets for statistical analysis. For a payment system, a bucket might be the recipient account, containing aggregate data on all clicks by all users that cause a payment to flow to that account. For a social network, it might be the identity of an account linked to by a "Follow", "Like" or "+1" button. The "bucketizing" should be done to identify the beneficiary of a clickjacking campaign, and may need to be somewhat sophisticated to capture this concept. For example, for a payments processor, buckets might need to be created to represent the author of a work being purchased, rather than the merchant it is purchased from, to capture a clickjacking campaign attempting to generate fraudulent purchases for that author from multiple (innocent) merchants. A single click might contribute data to multiple buckets. For example, a "purchase" button might be linked to buckets for the merchant account, the product SKU, and an affiliate account that collects a commission.

Buckets are first-in, first-out collections of hit or miss data for a limited number of first clicks. Long-term historical data on a bucket might be used to assist in determining the natural first-click miss rate and standard deviation, but the size of the bucket must be adjusted to allow identification of an attack. If, for example, a bucket contained a window size in the millions, an attacker might be able to make many successful attacks, even at low accuracy, before the weight of these bad guesses became significant in the overall average for the bucket. Too small a window size, however, and the risk of false positives increases.

Once clicks can be grouped into buckets, a window size is chosen, and the natural missed first click rate is determined, it becomes relatively simple to identify a clickjacking campaign targeting a given bucket. For a user interface where a button appears on one of three locations, with the other two left blank, a bucket of only legitimate clicks will have a first click with something approaching 100% accuracy, while a bucket containing only clickjacking traffic will have a first click hit rate of only 33%. Even if only a limited percentage of traffic to a bucket is subject to clickjacking, the deviation from the baseline first click success rate will be quickly visible if the natural missed click rate is low. When a bucket's overall first click success rate drops below a threshold, (we will use two standard deviations from the mean missed click rate in our example) a response can be triggered.

4 EFFECTIVENESS

The baseline click accuracy will never be exactly 100%, and the natural miss rate must be determined by observation on an interface-by-interface basis. It will likely vary between user agents, between mouse and touch interfaces, and may vary naturally across buckets. (e.g. the "Like" button for a Parkinson's Disease support group might be expected to have a higher natural miss rate than other buckets) Overall, the natural missed click rate for a well-designed interface can be expected to be fairly low, and the technique is quite effective even with higher missed click rates.

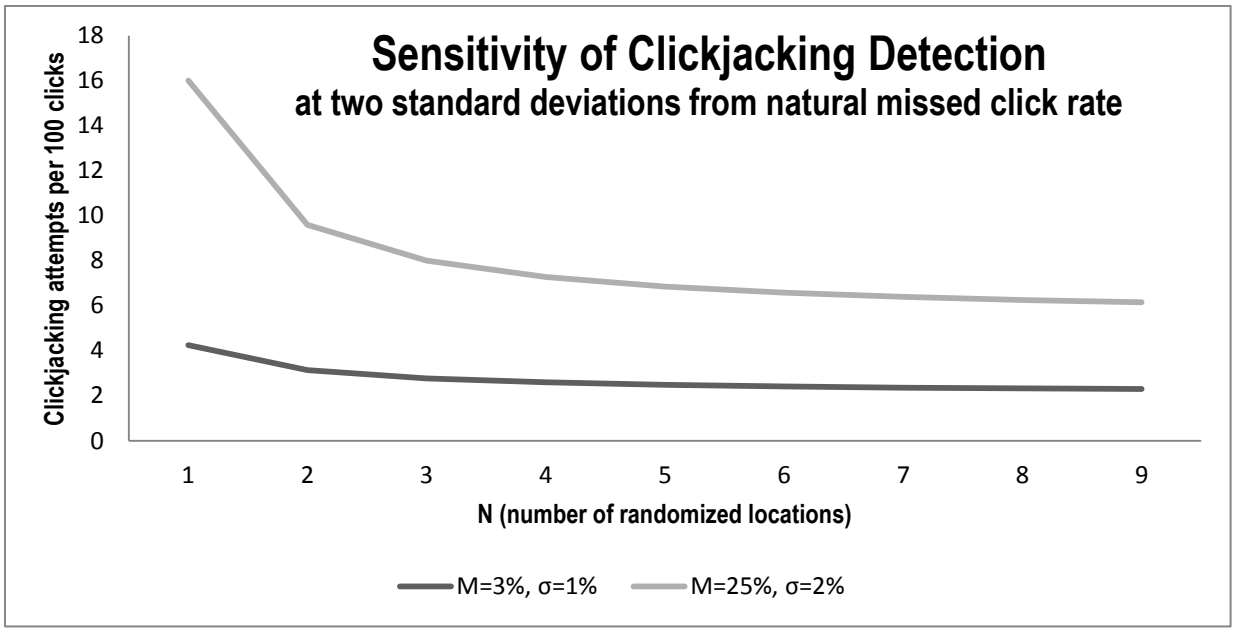
Let us say that we want to trigger a response when the missed click rate is more than two standard deviations (σ) above our observed normal. Call the natural miss rate as M, and the number of

randomized locations *N*. We can determine the number of clickjacking attempts *x* possible in a population of 100 clicks before the two standard deviation threshold is passed as follows:

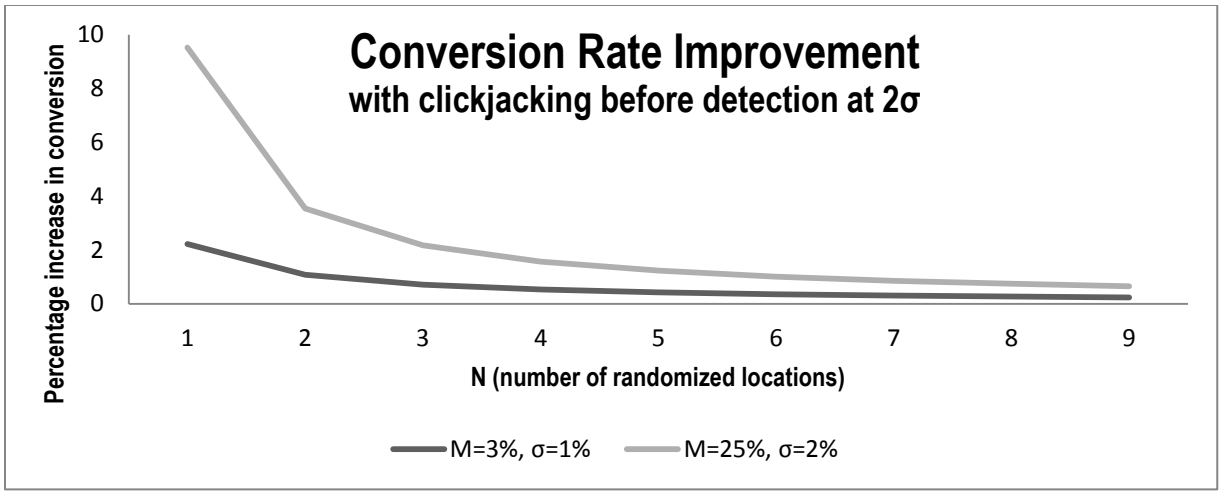
$$100(M + 2\sigma) = M(100 - x) + (x * (1 - 1/N))$$

With a natural missed click rate of 3% and a standard deviation of 1%, an attacker can inject just 4.25 clickjacking attempts per 100 total clicks before being detected when the UI is randomized among only 2 locations. This drops to 3.14 attempts in 100 for 3 locations, and 2.6 attempts at 5 locations.

With a much higher natural missed click rate of 25% and a standard deviation of 2%, the attacker can inject 16 clickjacking attempts per 100 clicks when *N*=2, dropping to 9.6 for *N*=3, 8 for *N*=4, and 7.25 at 5 locations.



We must also consider, however, that the expected success rate of these attempts also decreases as *N* increases. When we multiply the number of clickjacking attempts that can be made, undetected in the population, by their expected success rate, we see that an attacker in the first scenario, with *M*=3% and σ=1%, can only expect to increase their baseline conversion rate by a little more than 2% before detection when facing UI randomization among only two locations. The possible conversion rate increase drops to a little more than 1% at three locations and is around half a percent at five locations. With *M*=25% and σ=2%, the attacker can increase conversion by 9.5% with *N*=2, 3.5% with *N*=3, 2.2% with *N*=4 and 1.6% at *N*=5.



108 basis points of fraud (at *M*=3%, σ=1% and *N*=3) is still quite considerable if it were sustained across the entire population of transactions, but by limiting the possible profits and increasing the chances of detection, we can greatly reduce the motivation of the attacker to attempt fraud at all. An attacker without a legitimate click stream can mount very few clickjacking attempts before being detected, so for those attackers we have also raised the cost: they must generate 290 legitimate-seeming clicks for every successful attack to keep from being detected. If we can reduce the weight of multiple clicks from the same originating account in our bucket statistics, and if there is a cost to account creation or transactional friction, this may well make clickjacking unprofitable.

5 ADAPTIVE RESPONSE

For some applications it may be appropriate to simply suspend transactions to the beneficiary of a bucket that has suspiciously high missed click rates, especially if the target bucket has no legitimate click history. However, much like login attempt lockouts, care must be taken that this mechanism cannot be abused to cause a denial of service against a rival. Natural variation in data will also produce false positives.

For high value transactions at low volumes, it may be worthwhile to hold execution of transactions against buckets with high clickjacking suspicion until the transactions can be individually confirmed. Otherwise an automated response may be more appropriate.

Because clickjacking attacks are prevented from reading data in their target browsing contexts by the Same Origin Policy, adding an interstitial page that requires a user identification task to confirm the transaction, such as a CAPTCHA[12], may often foil the attack. These tasks need not be strong against automated analysis like a traditional CAPTCHA, because they are not directly visible to the attacker. The CAPTCHAs might themselves be framed by the attacking site in another User Interface Redressing attack, so tests should incorporate a logo as a watermark in the identification task, to help prevent users from being tricked into completing the task in a hostile context.

Opening a pop-up confirmation interface in a new browsing context with the X-Frame-Options header set to DENY can also provide a strong defense.

While these responses are generally considered undesirable user experiences that may negatively impact transaction completion rates, they need only be deployed when elevated first click miss rates indicate a clickjacking campaign may be in progress. Enabling and disabling these stepped-up protections can be a completely automated process.

If these experiences are still not desirable, a less obtrusive response could be to dynamically increase the set of possible randomized location for the button, and to disable transactions from any given user to that bucket after two or more missed clicks, greatly reducing the effectiveness of the campaign, if not completely preventing it.

6 Other UI Redressing Attacks

This approach does not address other forms of UI Redressing attacks, such as partial overlays with misleading context. For example, an attacker might expose a genuine "Pay" button across all of its randomized locations, but use an overlay to show the user false information about the payee and amount of the transaction.

In such cases, statistical analysis cannot be applied, but UI randomization may still be applied. If the button area and contextual information both utilized a background watermark that was randomized in size and orientation, the attacker would be unable to match it in their overlay. Unfortunately, this places the onus of action back on the user, to know the expected interface and recognize deviations, greatly limiting its effectiveness. In such cases, screenshot comparisons are again more appropriate, though it may be necessary to provide hints to such mechanisms about the critical context area for a given interface.

7 Weaknesses

This method will not prevent a one-off attack against a targeted user, it is ineffective where bucketization is not possible, or where it is cheap for an attacker to create many new buckets, because the data set is too small to identify attacks. When this is a serious concern, it may be necessary to delay executing clickjack-able actions or always apply an interstitial CAPTCHA for new buckets until they have sufficient data to be statistically meaningful.

The method of determining the natural missed click rate for a given bucket should also not be under the influence of attackers, or they may be able to bias the value. This means that determination of this value should be done under controlled experiment or statistically across a broad population of targets with similar characteristics, not on a bucket-by-bucket basis.

The approach cannot be applied where it is possible for the attacking browsing context to discern information about the layout or display of the browsing context it is attacking. Example attacks of this sort have been demonstrated in the past, using timing side-channels in the rendering of 3D overlays in WebGL, or calculated values for overlays based on SVG. The ability to read content in this manner is generally considered its own security flaw, a violation of the Same Origin Policy, and can lead to more serious attacks than clickjacking.

This approach is difficult to apply to complex or dense user interfaces where there is little room for randomized placement of user interface elements. It might be possible to randomize the (x,y) offset of an entire interface, and analyze data such as the average click distance from the center of UI elements, but the bucketization process to identify and isolate the possible beneficiaries of an attack is likely too difficult to attempt for a system as complex as a webmail interface, trading platform, or similar. Such applications might, at best, be profiled for an individual user over repeated interactions with the same application, so that an attempt to automate a complex action via clickjacking could be detected as anomalous. Such efforts are outside the scope of this paper.

7.1 The "Sleepy Frog" Attack

The most serious weakness of the technique is that the attacker can also supply multiple targets and use the genuine interface to enlist the user in making the correct choice. For example, consider a "Pay Now" button randomized among three locations, indicated by the dashed red lines:

The attacker overlays this interface with three identical images that are partially transparent:

And the composited picture to the user looks like:

The real button is only partially exposed by the frogs' transparent eyes, but the user is able to distinguish and click on the correct randomized location. Making the attack overlay partially transparent illustrates:

This is a quite serious weakness, as it is likely that an attacker can craft such an overlay for almost any possible button.

8 Combining with Screenshot Comparison Approaches

In the "Previous Work" section, we discussed screenshot comparison approaches to clickjacking prevention such as NoScript's ClearClick. This approach can easily detect and foil the "Sleepy Frog" attack.

As discussed previously, the weakness of screenshot based approaches is that they succumb to "phantom cursor" attacks that leave the real UI exposed while misdirecting the user's attention. Adaptive UI Randomization provides strong protections against this class of attacks.

Fortunately, the two approaches do not conflict in their implementation and execution. Screenshot approaches are applied at the user agent layer, and Adaptive UI Randomization are applied in the application and server back-end. Combining the two protections can give broad-spectrum coverage against all known vectors for clickjacking.

The back-end bucketizing and statistical analysis methodology can also be deployed to overcome other weaknesses of the screenshot approach: false positives, confusing user interaction, and the small install base of such solutions. A web application that is concerned it may be victimized by clickjacking could provide a feedback URI to a screenshot-based anti-clickjacking feature of a web browser. Upon detecting a likely clickjacking attempt, the plugin could simply connect back to the URI and report the suspected attack instead of intervening to prevent the click. If the URI included a unique transaction identifier that could be related to a set of buckets, the site can simply apply normal back-end anti-fraud measures when receiving such reports. The site can also use this data to learn the natural screenshot comparison false positive rate for its own interfaces and to refine the fraud risk score for buckets in a manner similar to that described for missed clicks. This allows

a small percentage of users with such a technology deployed to provide a sensitive detection tool that can be used to protect all users. Such an approach might prove very effective even without UI randomization.

9 CONCLUSION

By combining intelligent target identification and grouping with statistical analysis of both successful and missed first clicks, it is possible to accurately detect clickjacking campaigns with user interface elements randomized among only a very small set of possible locations. Although the technique is not generally applicable, it can be applied today, in the existing population of web user agents, to many of the most common clickjacking attack scenarios that are not protected by other measures: single-click actions in IFRAMES such as "Like", "+1", "Follow" and "Pay". Although randomization of the interface has the potential to create a poor user experience, considerable sensitivity can be obtained when randomizing within a mere 1x3 or 2x2 grid that requires minimal screen real estate.

Of particular advantage is that Adaptive UI Randomization can be effectively combined with screenshot comparison approaches such as ClearClick to remove the most important weakness of this approach: "phantom cursor" attacks. Combining the two approaches allows, for the first time, broad coverage of all classes of clickjacking attacks for many common use cases.

10 LICENSE

---

[1]Hanson, Robert and Grossman, Jeremiah, *Clickjacking*, Dec 2008, http://www.sectheory.com/clickjacking.htm

[2] Zalewski, Michal, *Dealing with UI redress vulnerabilities inherent to the current web*, Sep 2008, http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2008-September/016284.html

[3] Hardy, Norman, *The Confused Deputy*, Oct 1988, http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html See also: http://en.wikipedia.org/wiki/Confused_deputy_problem

[4] Burns, Jesse, *Cross Site Request Forgery,* 2005, 2007, http://www.isecpartners.com/files/CSRF_Paper.pdf

[5] Uhley, Peleus, *Clickjacking Threats*, March 2012, http://www.w3.org/Security/wiki/Clickjacking_Threats

[6] Lawrence, Eric, *IE 8 Security Part VII: Clickjacking Defenses*, Jan 2009, http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx

[7] Lin-Shung Huang, David, Jackson, Collin, *Clickjacking Attacks* Unresolved, July 2011. https://docs.google.com/document/pub?id=1hVcxPeCidZrM5acFH9ZoTYzg1D0VjkG3BDW_oUdn5qc

[8] Maone, Giorgio, http://noscript.net/

[9] Maone, Giorgio, *Hello ClearClick, Goodbye Clickjacking*, Oct 2008, http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/ and Maone, Giorgio, *ClearClick: Effective Client-Side Protection Against UI Redressing Attacks*, May 2012, http://noscript.net/downloads/ClearClick_WAS2012_rv2.pdf

[10] Barth, Adam, *RFC 6454, The Web Origin Concept*, Dec 2011, http://tools.ietf.org/html/rfc6454

[11] Corry, Bil, *Knowing Where to Click*, Oct 2008, http://lists.webappsec.org/pipermail/websecurity_lists.webappsec.org/2008-October/004484.html

[12] http://en.wikipedia.org/wiki/CAPTCHA