Security Against Clickjacking

An Implementation and Evaluation of a Chrome Extension

Zengyuan Liu
University of Minnesota
Minneapolis, MN
liux2310@umn.edu

Nathan He
University of Minnesota
Minneapolis, MN
hexxx503@umn.edu

Nick Khoury
University of Minnesota
Minneapolis, MN
khou0028@umn.edu

ABSTRACT

In this paper, we engage [7] on the basis of *implement and evaluate*. [7] proposes and designs a security mechanism to combat clickjacking, and shows how other defenses are not good enough. [7] never made their software available, or implemented it for Google Chrome, which is what this paper is adding.

As clickjacking becomes more threatening to Internet users, our implemented security mechanism aims to solve clickjacking protection by getting to the fundamental cause of the problem: a sensitive web element is taken out of context and given to the users to trick them into clicking on it by making it invisible or somehow hiding it. The Chrome security extension we implement works by giving website publishers more power by allowing them to flag some elements on their pages that they fear may be abused by attackers to steal identity of their users. The flagging of elements allows the Chrome security mechanism to give them special security attention. The end result is that the user can see what they are supposed to and events happen at the time they are supposed to while interacting with sensitive parts of websites. Unlike any other defense against clickjacking, this Chrome security extension is a fundamental scheme of protection, it will stop attacks that exist now, and there will be no loop hole for attackers to change their methods to get through after attacker learn how the mechanism works.

In evaluating effectiveness of the Chrome security mechanism for prevention of attacks, we use cognitive walkthroughs of different scenarios related to clickjacking attacks on the user, both with and without the Chrome security extension, to show that it is significantly effective against clickjacking.

CCS Concepts

•**Security and privacy** → *Spoofing attacks; Browser security; Information accountability and usage control; Social network security and privacy;* Intrusion/anomaly detection and malware mitigation; Software security engineering;

UMNSEC '16 Minneapolis, Minnesota USA
© 2016 ACM. ISBN 123-4567-24-567/08/06...$0.01
DOI: 10.475/123_4

Keywords

InContext; ClickJacking; Integrity;

1. INTRODUCTION

Any time a user on a personal computer has multiple processes under control of many principles of the operating system [21], and specifically, outside entities on the web through a browser, all the processes must share one resource, graphical output device. The user is now in danger of clickjacking [6]. This is a simple client side attack that can happen when an adversarial attacker tricks a user into clicking on an element purposely hidden or positioned maliciously to manipulate the original context in order to trigger events unknown to the user by clicking elements on the web.

One typical form of clickjacking is when the attacker makes a legitimate Facebook page and proceeds to use the Like button link provided by Facebook, sometimes called likejacking [23]. But the attacker manipulates the Like button to be invisible, and places it near or on top of an unrelated bait decoy to temp the victim user to click on. When the victim tries to click on the decoy, they are fooled into Liking the attacker's Facebook page.

Currently, methods of defense against this stop the attack only in a trivial way, meaning that it is possible for the attacker to come up with a new method or variant of clickjacking which the defense is incapable of stopping. For example, one a common defense is Framebusting [11] [17], which attackers defeat by nesting frames.

1.1 Design Goals

As many attempts to stop clickjacking have failed in the past due to superficial methods that did not meet the fundamental cause of the attack, we address certain issues that are what clickjacking is founded on in order to make our Chrome security extension.

These fundamental issues include: a way to decrease or stop false positives without requiring user interaction, and minimal intrusion on the web environment intended for users by legitimate websites that use graphical manipulation. In implementing InContext as a Google Chrome extension, we set out to achieve the following goals:

1.1.1 Unobtrusive

The defense design must allow embedded, framed and any other dressing of any sort of elements without interfering until there is an attack. Users cannot be expected to embrace a technology that they feel comes at such a great cost that they would rather not use it. Keeping the security mechanism unobtrusive is one of the most important factors in gaining users of our mechanism, as well as vendors to opt-in to participate with us, who will expect that their web site can keep a great appearance and feel, and only in critical moments of an attack, would anything be disabled on their site.

1.1.2 Prompt Free

There should be no confirmation needed from the user, which only annoys them and leads to disabling the mechanism, leaving the user vulnerable to an attacker who purposely triggers lots of prompting to trick the user into turning off their security. The security mechanism should create less work for the user, not more than they already have.

1.1.3 Optional

Web sites that do not participate with InContext by flagging their sensitive elements will not suffer any loss of function or decreased user experience. But the vendors who do opt-in to participation will be keeping their users out of harmful clickjacking situations, and making their web site more trustworthy.

1.1.4 Fundamental Solution

The security mechanism shall fundamentally stop clickjacking attacks. It will not be possible for attackers to make changes to attacks or new versions of the attack to bypass our security mechanism.

2. THREAT MODEL

Users browsing the web are clickjacked by the adversary who controls and uses a domain name, server, and web content, much like the common Internet attacker [8]. The attacker lures the victim onto their site by placing ads on other web sites, spam email, or something similar. When the victim is lured onto the attacker's page, there are invisible elements positioned in a way that the victim clicks the invisible element, thinking they are clicking something else.

The model does not include elements that are not out of context, but are misleading, such as a sensitive Facebook Like button, surrounded by content that is irrelevant. This is pure social engineering, where the sensitive element is not changed, only confusingly presented.

The threat model focuses on clickjackers using sensitive web GUI elements rendered in the victim's web browser, which an attacker manipulates to change the context and trick users. The attacker is out to steal the victim's identity, or money; fraudulently manipulate online voting outcomes, steal access to the web cam, and more.

3. BACKGROUND

The following section is a review of common clickjack attacks, where the victim is on the attacker's web page, which has a hidden button on a different domain than the attacker's, such as Facebook.

3.1 Known Attacks

We consider three methods attackers use to trick users into clicking on an element out of context. The three attack methods exploit integrity of: display, pointer, and time.

3.1.1 Exploiting The Display

Exploiting the integrity of the display will lead the victim of the attack to click on an element they do not see. The reason they click on that area with the unseen element can be due to clickbait, some tempting offer meant to be irresistible to the victim.

Hidden Elements.

The attacker uses HTML/CSS to hide the target element while allowing mouse events with it. The attack uses a fake element underneath the invisible element that victims are lured into clicking on. Also, the fake element may hide the target and let clicks fall through it.

Overlapping Elements.

The attacker blocks part of the target element with an `iframe` or popup window, so the victim sees false information but a genuine button to submit. For example, when the victim begins to transfer money, the "submit" button from the genuine bank site is visible, but the attacker covers the information with a popup, so the victim ends up sending money to the attacker.

3.1.2 Exploiting The Pointer

Exploiting the pointer will lead the victim of the attack to click on the wrong area because there is a spoofed pointer on their screen. The victim is tricked into thinking the fake pointer is their pointer, but really, it is maliciously distanced from the real pointer, so that the location of the real pointer does something the victim does not know about.

Cursorjacking.

The attacker shows a fake pointer to trick the victim into clicking on another area. This attack can be done with CSS or images with a picture of the cursor and setting the focus of the cursor near it.

Strokejacking.

Attacker uses a fake blinking keyboard input indicator to trick the victim to begin typing when the keyboard focus is set somewhere the victim does not know about, such as a hidden iframe. [24][26]

3.1.3 Exploiting Event Timing

An event timing exploit takes advantage of the time it takes the victim to comprehend that some element has changed, which causes a different, unwanted, event to happen. For example, a button on an attack page may say "double click" but actually uses two single click events, where the first causes an element to move over the button, and the second causes the moved element to be clicked.

This attack takes advantage of the time it takes a human to perceive changes visually from the moment of deciding to click on an element and the moment of their action or clicking on the element.

An attacker can make a game where the user is asked to shoot ducks appearing on the screen as quickly as possible, and in the middle of the game put a target button over the duck, tricking the user into clicking on it [1][2][27][28].

3.1.4 Attacker's Objective

Tweetbombs [13] and Likejacking [23] are two clickjack attacks that occurred in the past where victims were tricked into Tweeting and Liking whatever the attacker wanted.

Some of the attackers clickjacks would Friend or Follow the attacker, and repost the exploit onto the victims Facebook or Twitter page, making the attack a profitable virus that harvests Friends and Followers, and increases traffic.

Clickjacking can revel data about the victim if the OAuth [4] approval page button is clickjacked [18]. Other attacks exploit Flash Player's option menu for the webcam, tricking the victim into granting the attacker access to the victim's camera and microphone [1][2][3]. Attacks have also been known to forge votes [5], uploaded files [10], exposed location [27], and inject code by dragging and dropping an element [9][20].

3.2 Inferior Defenses

Defense mechanisms of the past and present that partially work, but all have problems and issues, either of exploitable weaknesses or user unfriendly features.

3.2.1 Visual Context

Prompting for Confirmation.

One method to combat clickjacking is to get a confirmation from the user whenever a blacklisted domain makes a request. This degrades user experience and attackers can easily switch to other domains. It can be defeated by using a timing attack where the user double clicks the target, where the first click summons the confirmation prompt and the second click grants permission to the prompt.

Random Layouts.

This defense method places sensitive elements in random positions each time the page is visited. This makes it harder for the attacker to put their exploitative frame in the correct position. This can be defeated by tricking the user to repeatedly click on the sensitive element until the position is known.

Banning Invisible Elements.

This defense forces all elements to be opaque at all times, on all pages. But this is detracting from the user experience, possibly ruining harmless sites.

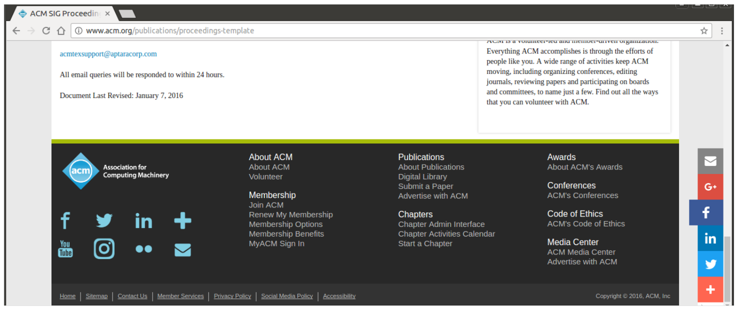
Frame Busters.

This defense blocks target elements from being in iframes, ensuring it is always the top level frame [17]. This can be done in JavaScript or using X Frame Options [11] and CSPframe ancestors [19]. This defense is not good because Facebook Like buttons are meant to be in frames on other sites. The defense can be defeated with popup windows.

Block Any Click Over Invisible Frames.

This defense blocks the user's clicks if the cursor is over a non opaque frame that has a different origin. Flash Player's option menu was protected with this method. But this defense does not protect users from other attacks.

The Firefox extension NoScript has a module ClearClick that uses this defense [14]. It compares a bitmap of the clicked element on any site to a bitmap of the same element rendered alone where its parent cannot make it invisible. This works against visual context exploits, but since it cannot distinguish any elements from each other, it checks everything the user clicks, and the result is a high rate of false



acmtexsupport@aptaracorp.com

All email queries will be responded to within 24 hours.

Document Last Revised: January 7, 2016

Everything ACM accomplishes is through the efforts of people like you. A wide range of activities keep ACM moving, including organizing conferences, editing journals, reviewing papers and participating on boards and committees, to name just a few. Find out all the ways that you can volunteer with ACM.

Figure 1: Screen shot of a web page with potential identity revealing links, which would be expected to be marked as sensitive by Google, Facebook, LinkedIn, and Twitter.

positives. Due to that, it must get user confirmation before the user can finish their intended action, which is annoying to users, and may cause them to disable the defense.

Flawed Ideology.

These inferior defenses are fundamentally crippled by their use of browser sourced bitmap creation, which fails to recognize the mouse pointer, or keyboard input cursor. Pointer integrity can still be exploited.

3.2.2 Context of Event Timing

Delay Interaction.

This defense delays any interaction from the user with elements on sites for a short time, just long enough for the user to comprehend and time attack clickjacking events that suddenly move an element at the moment the user decides to click it. However, it is in inconvenience to users and lowers the experience the user has with the site. Again, when users become annoyed with a delayed response, they may resort to disabling the defense. [25][22][15]

3.2.3 Access Control

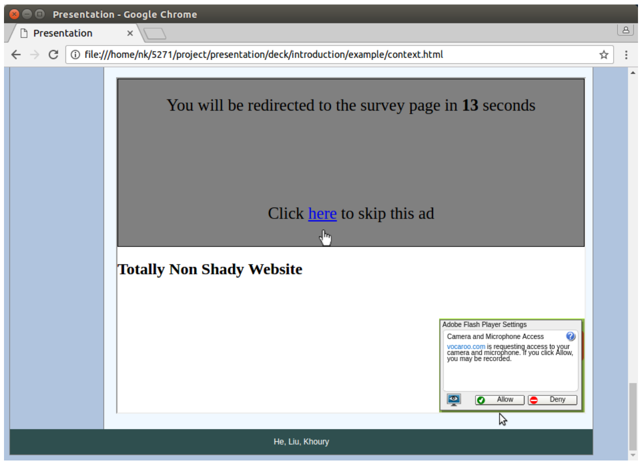
Access Control Gadgets allow individual applications to be allowed access to specific resources of the user, such as the microphone. This defense authenticates user actions to decide what resources an application should have access to. But, it fails to offer any pointer integrity. [16]

3.3 Superior Defense

InContext for Chrome will address all of the shortcomings of the inferior approaches. InContext for Chrome will feature: freedom from user confirmation prompts, pointer integrity, compatibility with elements meant to be framed on other sites, low rate of false positives, and delay free handling of timing attacks.

Lack of prompting for user confirmation offers users a secure path of least resistance, using the Chrome security extension is less work for the user because they are required to do nothing when there is no attack while using the mechanism, but would have to do something if they were attacked without the mechanism, like report fraud or have to explain an embarrassing social media post.

3.4 Motivation


Presentation - Google Chrome
Presentation
file:///home/nk/5271/project/presentation/deck/introduction/example/context.html
Allow
Deny
He, Liu, Khoury

You will be redirected to the survey page in **13** seconds

Click here to skip this ad

Totally Non Shady Website

Adobe Flash Player Settings

Camera and Microphone Access

vocaroo.com is requesting access to your camera and microphone. If you click Allow, you may be recorded.

Figure 2: Screen shot of a web page with a cursor spoofing clickjack attack, if the user skips the ad with the fake cursor, they will end up granting web cam access to the attacker.

As web users demand convenience of use and continue to live more complex lives on the web, this seemingly harmless trick, is actually a way to attack users of the web on the client side to do many things like steal identity information, cause passwords to be reset, or trick people into wiring money.

3.5 Contribution

Our contributions in this paper are implementing InContext, as designed by [7], as a new extension for Google Chrome, and evaluating this defense mechanism. We also evaluate the effectiveness of clickjack attacks, like [7], which include: cursor integrity, quickly clicking on elements, and timing attacks. The main objective is to keep the timing and pointer integrity intact, but we also maintain integrity of the display. Included in our consideration are novel methods of defeating the inferior defenses against clickjacking, as in [7].

4. STRONGER ATTACKS

Common clickjacking methods can be combined into new stronger attacks which defeat the inferior defenses. It is only a matter of time before the attacks become stronger, and the adversaries become more well funded, the stronger attacks show how clickjacking is a serious issue of security, and should be addressed with a solution that users of all skill levels can use to stay safe.

4.1 Spoofing the Cursor

To maliciously escalate privileges and take over the webcam, we use an attack site with CSS to make a fake cursor which tricks the user about the position of the real cursor. The attack page shows a popup ad and the only way to get back to the original page is to click “No, thanks.” The fake cursor is offset from the real cursor so that clicking the no thanks bait link will actually click a “Allow” button on an option menu below, requesting to access the user’s webcam.

4.2 Double Click Trick

To force a user to click on the attacker’s target link, we make a page that tricks users into double clicking an element, which triggers a timing attack. After the first click, a sensitive element is repositioned over the element the victim is double clicking, and as this happens in less than half of a second, the victim has no time to realize the change and the second click is a click on the attacker’s target.

This attack page is built by indicating through text that something must be double clicked, when really there are two single click events happening: the repositioning of an element, and clicking on that element.

To avoid double click, the detection of sensitive elements is important. For example, HTTPs has implemented the detection by using security flag. The security flag will only be sent to a HTTPs page. If the user sets the security tag, the transmission of the user cookie will go through an encrypted channel. To detect sensitive elements of the website, the function calls analyze the information on the website. To extract the sensitive information, two methods can be used including the element ID and tag name. Take element ID for example, ID is a case-sensitive string representing the unique ID of the element being sought. If a bank website using a HTTP not HTTPs, the user who wants to log into his online account will be required to type in the user name and passwords. There will be words explicitly indicating this requirement. By utilizing this situation, the ID will be something pointer point to this information. The function call will search the whole website to check whether there are some elements of the website match the words like ”user name” or ”password”. If yes, the function call will mark this frame as red (or trigger the alarm) which will notify the user of the potential risk he takes. Google now will list the non-HTTPs website who require passwords or username as insecure website.

Based on [7], even you have visual integrity, the attacker can still use the race condition exists at the user side. For example, the attacker can move the sensitive element to the region where the user click first. The speed is very fast so that the user can’t even notice it. The region the user click first is also considered as sensitive region identified by the similar method we mentioned in the above figure. Based on the reference bitmap of the website. If the identified region remains same for a certain long time, which also won’t affect the user experience. The time interval is $t1$. The time for the attacker to move the sensitive element is $t2$. $t1$ should be bigger than $t2$. If this can be satisfied, the action taken by the user will be accepted by the website. The other method or defense mechanism include moving the cursor to the safe region based on the sensitive elements detection and padding the sensitive region with thick background to make user realize the change if it is made to the sensitive region. To sum up, based on the accuracy of the sensitive detection, the defense mechanism has the capability to monitor the changes of sensitive element and make sure the action taken by the user is sent to web sever only after it concludes this action is safe on current sensitive region.

4.3 Duck Hunt Game

A game where the user tries to quickly shoot at ducks as they appear anywhere on the screen, by clicking on them with a fake cursor. The faster the user clicks on the ducks, the higher the point reward. During the game, after eight ducks are shot, the ninth duck will have a link appear lower on the screen positioned so that the real cursor clicks on it without the user's knowledge.

To steal the victim's identity, the attacker can make a Facebook page, and use the Like button as the target for the real cursor in the duck hunt game. When the victim clicks the button, the attacker is alerted by Facebook. `Event.subscribe()`, then the attacker can get the identity of the victim and look them up on Facebook, and remove the victim Liker to cover up the attack.

5. APPROACH OF DEFENSE

To defend against clickjacking at the root of the problem, the operating system should be used in monitoring communication between applications or web pages.

5.1 Integrity of Vision

For a user to see what they are truly clicking on at the moment they click a sensitive element, InContext will make the image of the sensitive element and the pointer visible at the operating system level. InContext will activate sensitive elements after checking that both of these things happen. Otherwise, no user input can reach them.

5.1.1 Check the Target Display

The CSS Myth.

The characteristics of the CSS can be described by the z-index, size and opacity. The important part is to make sure no sensitive element overlaps with the cross-origin elements which makes the web page affected by the attacker. These can be achieved by trying to get the topmost display by already-defined functions used in IE or Chrome. For our implementation, we focus on the Chrome extension. The security can not just build on the CSS checking because the user can be easily fooled by the attacker.

The Static Reference Bitmap Myth.

At the checking stage, the temporary bitmap rendered by the third party extension we used is compared to the reference bitmap which is secure. The variance of the temporary bitmaps provided by different third party software or extensions are big. This makes the implementation less practical. The time spent on extracting the bitmap and comparison process is slow which will greatly affect the user experience.

The Answer.

In the reference [7], they propose two improvements based on the two types of drawbacks talked above. They only use the position and dimension information extracted by the browser layer out engine. This is easier to be implemented and saves the storage space, which increase the effectiveness. They claimed this design works well with dynamic aspects. The second thing they did is to the enforcement that a host page can't apply any CSS transformations. This may include zooming, rotating, etc. The invisible elements inside the sensitive region are also disabled. The differences they achieved are they user OS APIs to take the screenshot to avoid the variance taken by different third party extensions and the opt-in feature eliminates false positives and freedom from user prompts. Additionally, for our implementation, the difference is, for some websites, we can transfer the bitmap to the plaintext. The information stored in the plaintext can also be used as the reference information for the comparison. The regions are labeled by different IDs. If the plaintext in certain region are changed, the action taken by the user will be disabled.

5.1.2 Check the Pointer

The following techniques of defense offer a good level of trade off between user experience and security. The pointer is checked in these ways to make sure it is not a spoofed pointer.

Default Cursor.

No custom cursors on any page hosting the sensitive element and all ancestors of the page. The user will see the system cursor when near sensitive elements.

This way, the user experience can still be complete, there is only a change when near sensitive elements. This is possible because the operator of the site opts in to the defense of InContext, and marks only certain elements as sensitive.

Freezing the Display.

When the cursor is near a sensitive element, there will be no updates allowed.

Mute Speakers.

A sudden audio noise may cause the user to excitedly try to turn it off. When the cursor is near sensitive elements, the speakers are disabled.

Borders.

A border of a light, randomly shaded grey color, is drawn around the sensitive element when the cursor is near it, and everything around it is lightly darkened, to make sure it is noticeable.

The shading is in randomly selected shades of grey to stop an attacker from reusing the effect to confuse users about what is and is not sensitive.

Keyboard Change of Focus.

The focus of the keyboard cannot be set outside of a sensitive element once it set into it, unless the user manually changes it. This will guard against strokejacking, when a sensitive element gets the focus of the keyboard, other applications cannot change it back and forth.

5.2 Integrity of Timing

To guard against bait and switch attacks that target users when they are rapidly clicking on elements, these methods offer security and a good user experience.

Selective Delay of Action.

If some visuals change near a sensitive element and the user goes to that element, the action they take will be delayed to make sure that they still want to click on that element.

Delay on Hover.

During rapid clicking, when the cursor enters the area of the sensitive element, there should be a delay to be sure the user has time to see what the sensitive element looks likes. This is necessary in case of a rapid click type game, where the element the user is clicking is moving and suddenly one of them is a sensitive element.

Element Jumps Under Pointer.
If a sensitive element is positioned to be under the pointer, that element will be blocked from taking any input until the user moves the cursor off of the sensitive element and back onto it.

Padding.
Sensitive elements are spaced away from all other elements, to give the defenses enough time to react.

5.3 Avoiding False Positives

As part of our implementation of the inContext defense mechanism we assume that there is some form of opt in process where they make the element as 'sensitive'. For example, if Facebook was to opt in to our defense mechanism then all Like, Share, etc. buttons that attackers would try to maliciously trick users to click would be marked by our extension as sensitive and we would known specifically to monitor possibly click jacking attack variants around those elements. This has the benefit of allowing our extension to have a very low false positive rate since we only monitor the elements that we know are marked as sensitive and could likely be targeted by a click jacking attack.

Without a low false positive rate, a defense mechanism becomes significantly less effective since many users end up either disabling the mechanism, or ignoring the warnings. Our security mechanism lets users and vendors team-up against adversaries in keeping letting the third party who owns the sensitive link optionally decide to participate in the defense, and mark specific elements as sensitive, we can ensure a low false positive rate. This is because we can be certain about whether or not a given element on any web page is sensitive or not.

6. IMPLEMENTATION

Our security mechanism implementation consists of an extension for Google Chrome as proposed by [7]. The extension is written in a Linux environment using JavaScript, and uses libraries including PhantomJS, JQuery, and ImageMagick.

We use ImageMagick to compare the bitmaps from the vendor and the screenshot. The vendor's bitmap is assumed to be available, because they choose to opt-in to using our mechanism and make bitmaps of sensitive elements available.

We used the PhantomJS screen capture capability to take a screenshot of the sensitive element, which included the functions `page.clipRect` and `page.render` in order to create an image only where the sensitive element is located on the screen.

When the cursor is near or over a sensitive element, we use JavaScript to darken the surrounding area, draw a red border around the sensitive element, and disable all cursor customization.

For maintaining visual integrity, we use ImageMagick in JavaScript to make a bitmap of the image of the sensitive element as the user sees it, as above, using PhantomJS. Next we compare it to the vendor's bitmap of the sensitive link, if they don't match, the context of the sensitive element has been compromised, and all interaction with the sensitive element is disabled.

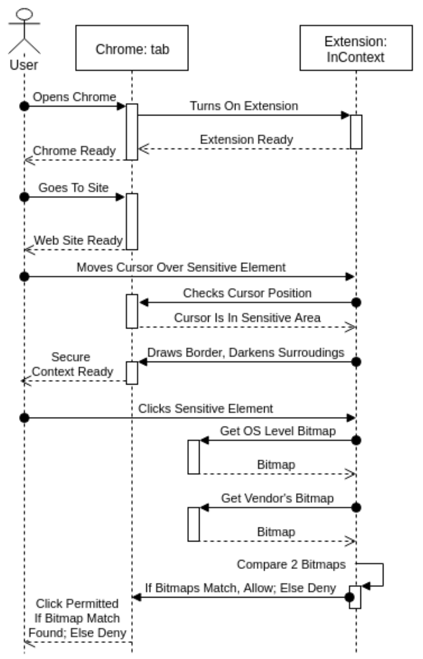


Figure 3: A system diagram showing the logical control flow as a user interacts with a sensitive element in the Chrome web browser. The mechanism will allow the interaction if the two bitmaps match, otherwise there must be a clickjacking attack and any interaction with the sensitive element is disabled.

We delay interaction on a sensitive element only if the sensitive element's appearance or position changed recently. We used JavaScript's `Object.observe` to keep track of whether sensitive elements have move within the last few seconds. If the sensitive element had recently moved, we impose a delay of several seconds where no interaction is allowed with the sensitive element.

6.1 Engineering Challenges

Due to limited resources, after the implementation of the extensions designed for Chrome, it is hard for us to evaluate the performance of the extension by querying the real user. We use cognitive walkthrough evaluation instead. The quality of the questions proposed during the evaluation walkthrough and the outcomes surprised us with good guidance and critical thinking. The cognitive walkthrough inspires us to come up with defense mechanism based on the essence of the attack. This greatly improve the efficiency of the proposed defense mechanism. In terms of the essence of the attack, proposing the stronger clickjacking attack is possible for us. This will in-turn to benefit our testing of defense mechanism. In order to avoid affecting the user experiences, we reset the page to the original configuration after the user realize the potential attacks. The time delaying control of the webpage loading process is dynamic. The way to reset to the original configuration requires the consideration from the users' perspectives. In the implementation of the security mechanism we experienced challenges with knowing the position of the sensitive element and its size, deciding the best way to mark elements as sensitive to reflect a realistic and real world option. The sensitive element detection is based on the OS-level bitmap extraction. The extraction varies based on the method used. The way of extracting stable or easy-to-implement bitmaps matters for the defense mechanism.

7. EVALUATION

7.1 Experiment Design

We do a cognitive walkthrough to evaluate the effectiveness of the extension in the wild as real users of any skill level and background of knowledge would use it. This approach is best because hiring testers to try out our extension is not a viable option for this project.

7.1.1 *Cognitive Walkthrough*

Our Users.

In conducting our cognitive walkthroughs we define several types of users that would be as representative as possible while showing us that they would be satisfied with our product.

1. Students from 15-25 years old, are profiled as spontaneously surfing from one site to the next without considering the domain they are taken to, only that it is some interesting looking link. The tasks done in the web browser of this group are generalized to entertainment, communication, social media, current events, and research for school studies.
2. Professional employees in their mid-twenties to mid-fifties, are profiled as more conservative in their spontaneity of web surfing, but still do surf the Internet with a web browser. Their tasks are generalized to communication, current events, shopping, social media, and physical home security applications that let them monitor security cameras in their homes over the internet.
3. Retired senior citizens over 55 years old, are profiled as unfamiliar with the concept of clickjacking and may not know what links are or even web sites, only that the browser is a window that displays a few resources they require, such as communication, the Google search engine, and current events.

User Tasks.

We choose specific tasks that we suspect will be most representative of tasks that all of our users will do on a regular or infrequent basis. Examples of tasks we decided are representative of what our users would do include: reading an article on a current event web site, updating status on a social media web site, and using the Google search engine to find a coffee shop. While doing each task we consider whether the user tries to achieve the correct outcome, whether they see there is a correct action to take, whether they associate the correct action with their desired result, and whether they realize the correct action is progress.

7.2 Scientific Control

The scientific control is the experiment where the user has no security mechanism, and visits a web site with a clickjack attack waiting. We watch to see how often the users become victims. If users tend to become victims more often in this experiment than in our experiments testing the security mechanism, then we can conclude that our mechanism provides a significant amount of security to our users.

7.3 Cognitive Walkthrough and Evaluation

We created realistic attacks on test web sites that are most typical and accurate representations of real attacks that our users would be subject to in completing their typical tasks.

7.3.1 *Invisible Overlay*

To test how effective our mechanism is at reducing clickjacking attacks that rely on making sensitive elements invisible and positioning them on top of clickbait, the user's task is to visit a social media web site with the attack set up. As the user visits the test attack page, they see a "free iPhone" button, which is overlaid with an invisible button that actually updates their status on the social media site to supporting a page affiliated with the attacker. If the user decides to click on the "free iPhone" button without our security mechanism, they become victim to the clickjacking attack. But if the user visits the page while using our security mechanism and decides to click the "free iPhone" button, the mechanism detects that the cursor is near the sensitive element, it darkens the area around the sensitive element even though it is invisible. Next the user still decides to click on the "free iPhone" button, as they are so enticed by this thought that they are overcome with emotion, and click on it. At this point the mechanism has disabled interaction with the sensitive element, which is invisible and fails to pass the visual integrity checks of the security mechanism. Since the disabled button is on top of the clickbait button, nothing happens when the user clicks.

This is a rare case where a part of a web page is disabled by our mechanism. This does not happen often because most elements on any given web page would not be expected to be marked as "sensitive" elements by the vendor. But for both the vendor and the user, disabling the sensitive element in this case is undoubtedly better than having a user become a victim of a clickjacking attack.

Finally the user moves their cursor away from the invisible sensitive element and the appearance of the web page returns to normal, un-darkening the area surrounding the sensitive element, and removing the border drawn around it.

For our cognitive walkthrough we chose Nick Khoury to simulate the victim user and Nathan He as the evaluator. The following set of questions then were asked to evaluate the effectiveness of our defense mechanism:

- Will the user try to achieve the correct outcome?

  We define the correct outcome is for the user not to click on the sensitive element in this case the 'free Iphone' button. It was noted that Nick did not click on the 'free Iphone' button.

- Will the user notice the correct action is available to them?

  By asking this question we evaluate how well the user was aware of the defense mechanism triggering and updating their visual context. In this case Nick noticed that the screen darkened.

- Will the user associate the correct action with the outcome they expect to achieve?

By asking this question we evaluate how well the user associated the update in visual context to being aware of a clickjacking attack attempt taking place and thus avoiding the attack. In the case of an invisible overlay Nick noticed the darkening and the red border but it was noticed that he did not immediately recognize that a click jacking attack had taken place. We observed from this that the defense mechanisms we were using for some users might seem to be too similar to the typical responsive animations that websites used. We concluded that in further iterations that a stronger emphasis on clearer communication to the user that changes to the screen were coming from our extension needed to be made.

- If the correct action is performed; will the user see that progress is being made?

  By asking this question we evaluate whether that in the case that the user successfully recognizes the click jacking attack due to the defense mechanism and takes the correct action of moving their mouse away from the sensitive element is there progress being made, i.e are they are more aware of the context of their browser than before. In this case we noted that in our initial iteration that when Nick moved his mouse away from the sensitive element that the changes persisted on the browser screen and the original context was never restored. While in some defense mechanism implementations this would be acceptable the overall theme of the inContext defense mechanism emphasizes non intrusiveness to the user. Our goal is to make the user aware of the true context of the browser screen, once that has been achieved we want to restore the original context of that screen back to the user. Due to this evaluation in further iterations we worked on implementing restoring back the original state of the web page back to the user.

7.3.2 Cursor Spoofing

To test our security mechanism's effectiveness in mitigating clickjack attacks that maliciously change the context of the cursor, the user task is to simulate executing our implemented defense mechanism against a Cursor Spoofing Attack. In our demo the victim user ends up on a page with the sensitive element being a notification asking for permission to access the webcam or microphone input from the user. When the user's mouse happens to move near the sensitive element a pop up ad appears at the top of the screen prompting the user with the option to 'skip' the ad. At the same time the attack will hide the cursor which is near the sensitive element and display a fake cursor near the ad which will mimic the user's actual cursor movement. The goal is to coordinate the positions of the fake cursor relative to the real cursor such that when the user clicks on 'skip' they will actually click to allow permission for the website to gain access to the webcam or microphone of the victim user. This is an example of maliciously altering the visual context of the victim user as described in the inContext paper.

Our Chrome Extension implements the defense mechanisms described in the inContext paper by disabling all cursor customization preventing the user's real cursor from being hidden in any case. We then monitor the sensitive element and when the victim user's mouse moves near it we dim the rest of the website and highlight the sensitive element with a red border ensuring visual context to the victim user.

For our Cognitive Walkthrough we chose Nick Khoury to simulate the victim user and Nathan He as the evaluator. We defined the task in a simple 1 step process by which Nick was to interact with the demo and trigger the clickjacking attack by moving his mouse near the sensitive element. When the attack triggered Nick briefly the state of the browser. The following set of questions then were asked to evaluate the effectiveness of our defense mechanism:

- Will the user try to achieve the correct outcome?

  In most of these cognitive walkthrough tasks as in this task the correct outcome is for the user to not click on the sensitive element in this case giving the website permission to access the user's webcam data. It was noted that Nick did not click on the webcam allow button and thus avoided being maliciously tricked into clicking on the sensitive element.

- Will the user notice the correct action is available to them?

  By asking this question we evaluate how well the user was aware of the defense mechanism triggering and updating their visual context. In this case an some interesting observations were made. In our first iteration we did not use any type of extra border around the sensitive element instead believing that dimming and fading the rest of the website around the sensitive element would be a significant enough of a change in the browser to notify the user. However in our first iteration of the cognitive walkthrough process due to the color scheme of the ad which was gray the effect of fading the non sensitive elements was not as effective as we predicted. In our initial evaluation Nick first noticed the ad at the top, and then noticed the sensitive element when ideally Nick would immediately notice the sensitive element. In our second iteration we added a red border that would appear along with the fading and Nick immediately noticed the sensitive element first and did not notice the ad at all. We concluded that combining various visual context changes was very effective and noted that it would reduce the chances of any one website's visual scheme of negating the effectiveness when using just one visual context change.

- Will the user associate the correct action with the outcome they expect to achieve?

  By asking this question we evaluate how well the user associated the update in visual context to being aware of a clickjacking attack taking place and thus avoiding the attack. In the case of cursor spoofing where the user is made immediately aware of the context of where their mouse is we found that our defense mechanism was very effective in communicating to the user that the correct outcome would be to move their mouse away from the sensitive element. From Nick's perspective once he realized the actual position of his mouse pointer he immediately recognized the attempt of obscuring the position of his mouse pointer and moved it away from the sensitive element.

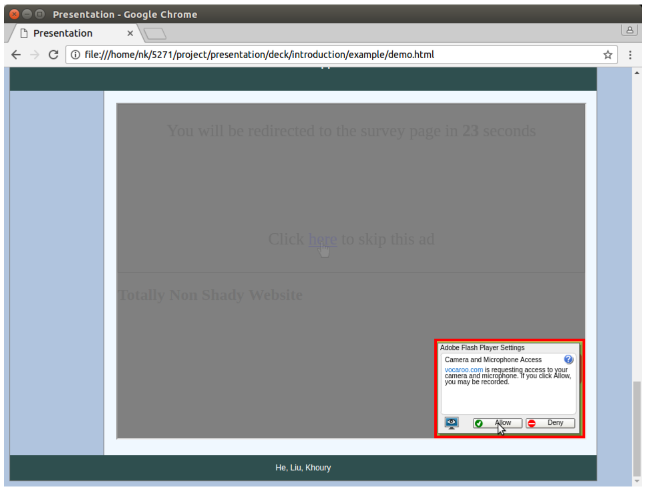

Presentation - Google Chrome
Presentation
file:///home/nk/5271/project/presentation/deck/introduction/example/demo.html
You will be redirected to the survey page in 23 seconds
Click here to skip this ad
Totally Non Shady Website
Adobe Flash Player Settings
Camera and Microphone Access
vocaroo.com is requesting access to your camera and microphone. If you click Allow, you may be recorded.
Allow
Deny
He, Liu, Khoury

Figure 4: Screen shot of the same web page with a cursor spoofing clickjack attack shown previously, but this time our Chrome security mechanism is being used.

- If the correct action is performed; will the user see that progress is being made

  By asking this question we evaluate whether that in the case that the user successfully recognizes the click jacking attack due to the defense mechanism and takes the correct action of moving their mouse away from the sensitive element are they more aware of the context of their browser then before. In this case we noted that when Nick moved his mouse away from the sensitive element the website returned back to its initial state however with the ad and fake cursor exposed. By this definition we could say yes progress was made as it was notable that a click jacking attack was attempted to Nick after the mouse moved away from the sensitive element. However we also noted that a more robust malicious design could possibly circumvent such changes by getting rid of the ad and fake cursor and not assume that the attack would work on the first attempt. We concluded that while our implementation o the inContext defense mechanism for our Chrome Extension was fairly effective in exposing the click jacking attack there is room for more robust exposition of click jacking cursor spoofing attack vectors.

7.3.3 *Double Click*

In our demo for a double click variant of a click jacking attack the victim user ends up on a page with a form of clickbait element that prompts the user to double click. This attack takes advantage of the user's familiarity with the act of double clicking on various elements perhaps on their own PC but as far as website functionality there is no real need to ask users to double click on any element. The sensitive element is positioned somewhere else on the screen. After the first click the sensitive element is then moved to the former position of the clickbait in an attempt to trick the user into clicking on it before they can react to the change in context of their screen. This is an example of altering not only the visual context of the victim user but also the temporal context, i.e. changing the context of the screen in a window of time too fast for the victim user to properly assess the updated context. This attack also circumvents the typical clickjacking defense of forcing layered sensitive elements to the top as the initial state of the sensitive element can be perfectly in view.

Our Chrome Extension implements the defense mechanisms described in the inContext paper by listening to click events on the browser screen and keeping track of double click events. When the user clicks on their browser we check to see if the user had clicked on a non sensitive portion of the screen. If true then we listen for another click in a short window of time, any click event that occurs in this window will register as a possible double click attack and triggers a visual update from our extension on the user's screen. We create a semi transparent div sized to each sensitive elements size and we move their position to be over each sensitive element intercepting any clicks from the possible victim user. After a window of time the intercepting div goes away resuming the original visual context

For our cognitive walkthrough we chose Zengyuan Liu to simulate the victim user and Nathan He as the evaluator. We defined the task in a simple 1 step process by which Zengyuan was to interact with the demo and double click the click bait element, in our case it was a button saying to 'Double Click Here!'. When the attack triggered Zengyuan responded to the state of the browser. The following set of questions then were asked to evaluate the effectiveness of our defense mechanism of the browser screen. The following set of questions then were asked to evaluate the effectiveness of our defense mechanism:

- Will the user try to achieve the correct outcome?

  We define the correct outcome for the user to not click on the sensitive element in this case clicking on a pseudo Facebook like button. It was noted that Zengyuan did not click on the Facebook like button due to the defense mechanism initially but chose to click on the like button anyway 'just to see what would happen.'

- Will the user notice the correct action is available to them?

  By asking this question we evaluate how well the user was aware of the defense mechanism triggering and updating their visual context. On our first iteration the div that was overlayed onto the sensitive element to intercept clicks merely had a red border around it to notify the user that the sensitive element was highlighted and that the extension was making a change to their screen. On our second iteration we added a background image composing of a warning sign, on both iterations Zengyuan successfully recognized the update in visual context that the defense mechanism triggered on his screen.

- Will the user associate the correct action with the outcome they expect to achieve?

  By asking this question we evaluate how well the user associated the update in visual context to being aware of a clickjacking attack taking place and thus avoiding the attack. In our initial iteration we started with the overlayed div displaying with a red border. While

Zengyuan noticed the update in visual context, in his feedback he said that it was not immediately apparent to him that this was the result of the defense mechanism. In our second iteration we displayed the overlayed div with a background image composed of a warning sign. Zengyuan noted that this was significantly more effective to him communicating the fact that the visual context update from our defense mechanism was triggering from a malicious attack of some kind. We noted that this could be made stronger by having the warning sign be similar in design or theme to our Chrome Extension product brand in order to strengthen the association between the user seeing the defense mechanism trigger and the user recognizing that a click jacking attack was occurring. In addition, the use a commonly themed warning symbol corresponding to our extension brand would strengthen the rest of our defense mechanisms across the board as the user would strengthen the association between the appearance of the brand warning image on their screen and the recognition that the defense mechanism from our extension was the cause.

- If the correct action is performed, will the user see that progress is being made?

  By asking this question we evaluate whether that in the case that the user successfully recognizes the click jacking attack due to the defense mechanism and takes the correct action of moving their mouse away from the sensitive element is there progress being made, *i.e* are they more aware of the context of their browser than before. In this case we noted that when Zengyuan noticed the defense mechanism triggering he chose to click on the Facebook like button anyway 'just to see what would happen'. In other defense mechanism implementations this would perhaps result in a negative outcome as part of our evaluation but the theme of the inContext defense is to solely present the true context of the screen to the user and to be as unobtrusive as possible by not taking away permanent functionality of the website. Zengyuan was aware of the visual context of what he was seeing on the website when he clicked on the Facebook like button so it does not negatively result our evaluation of our Chrome Extension.

7.4 Overall Summary of Results

Overall based on our cognitive walkthrough based evaluations, we found that the Chrome security extension is effective in mitigating clickjacking attacks. In all of our defined tasks the user noticed the defense mechanism from our Chrome Extension working. The Cognitive Walkthrough process in addition was a very helpful tool in aiding us in iterating through our extension. Some of the observations we made for example was to emphasize communicating to the user that the defense mechanism triggering was the result of our Chrome Extension and not the result of a reactive response from the website that they were visiting. In further iterations of our extension we included elements such as common themed warning image and an extension tutorial guide to show users what exactly our defense mechanism looked like. We also observed that it is important to integrate and mix defense mechanism implementations when possible. When multiple defense mechanisms are used it is more effective in getting the user to recognize a click jacking attack is taking place rather then relying on one discretely. Overall, the cognitive walkthroughs showed that our Chrome Extension had a basic effectiveness in getting the user to recognize and avoid the click jacking attack but showed that there was room for further robustness for future iterations.

8. CONCLUSIONS

We have implemented an extension for Google Chrome which is a security mechanism to protect Internet users from UI redress attacks, or clickjacking. As users visit wide varieties of web sites, and often expect that different sources on the Internet are able to interact with each other, with complete functionality, this type of security mechanism is necessary for users of all skill levels and technological backgrounds.

In this paper we reported on our implementation and evaluation of a defense mechanism proposed by [7] as a new extension for Google Chrome. This mechanism guards against the fundamental process of clickjacking, to make sure that user actions on sensitive elements are in proper context, keeping integrity of the display, timing of events, and the cursor. Our mechanism is the only one to secure users from all forms of clickjacking, in a user friendly way, that has no prompts, few delays, low false positives, and is easy to use.

By conducting experiments on the mechanism with cognitive walkthroughs, we found it to be effective in alerting users that they are under attack, and that they should take proper care to avoid the attack. We feel the future of people and the Internet will be better off if users and vendors can team up and make use of this security mechanism.

9. FUTURE WORK

One way to extend this mechanism to offer more protection is by using methods discussed in [12], using URLs to see when suspicious activity occurs.

10. ACKNOWLEDGMENTS

Thank you to our Professor, Nick Hopper and our TA, Nishad Triveldi, for providing us with inspiration, guidance, and feedback on this project.

11. REFERENCES

[1] F. Aboukhadijeh. How to: Spy on the webcams of your website visitors. 2011.

[2] G. Aharonovsky. Malicious camera spying using clickjacking. 2008.

[3] J. Grossman. Clickjacking: Web pages can see and hear you. 2008.

[4] E. Hammer-Lahav. The oauth 1.0 protocol. rfc 5849 (informational). 2010.

[5] R. Hansen. Stealing mouse clicks for banner fraud. 2007.

[6] R. Hansen and H. Grossman. Clickjacking. 2008.

[7] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 22–22, Berkeley, CA, USA, 2012. USENIX Association.

[8] C. Jackson. Improving browser security policies. *PhD thesis, Stanford University*, 2009.
[9] K. Kotowicz. Exploiting the unexploitable xss with clickjacking. 2011.
[10] K. Kotowicz. Filejacking: How to make a file server from your browser (with html5 of course). 2011.
[11] E. Lawrence. Ie8 security part vii: Clickjacking defences. 2009.
[12] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Learning to detect malicious urls. *ACM Trans. Intell. Syst. Technol.*, 2011.
[13] M. Mahemoff. Explaining the "don't click" clickjacking tweetbomb. 2009.
[14] G. Maone. Hello clearclick, goodbye clickjacking! 2008.
[15] G. Maone. Fancy clickjacking, tougher noscript. 2011.
[16] 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.
[17] 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.
[18] S. Sclafani. Clickjacking and oauth. 2009.
[19] 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.
[20] P. Stone. Next generation clickjacking. *In Black Hat Europe*, 2010.
[21] H. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions in mashupos. *In ACM Symposium on Operating System Principles*, 2007.
[22] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and V. H. The multi-principal os construction of the gazelle web browser. *In Proceedings of the 18thvConference on USENIX Security Symposium*, 2009.
[23] Wikipedia. Likejacking.
[24] M. Zalewski. Firefox focus stealing vulnerability (possibly other browsers). 2007.
[25] M. Zalewski. [whatwg] dealing with ui redress vulnerabilities inherent to the current web. 2008.
[26] M. Zalewski. The curse of inverse strokejacking. 2010.
[27] M. Zalewski. On designing uis for non-robots. 2010.
[28] M. Zalewski. X-frame-options, or solving the wrong problem. 2011.

APPENDIX

A. GLOSSARY

A.1 Clickjacking

Exploitation of the integrity of either the pointer, timing, or display, which changes the context of web elements, and tricks users to click on them.

A.2 Cognitive Walkthrough

A technique to evaluate the usability and measure the effectiveness of the extension, by completing different tasks that a typical end user would carry out, as if they were an end user.

A.3 False Positive

Happens when the extension wrongly detects a clickjack attack.

A.4 GUI

The Graphical User Interface.

A.5 InContext

A proposed security mechanism in [7] which offers contextual integrity on the client side of web browsing.

A.6 Pointer Integrity

The user understands where the pointer is while browsing the web, and the pointer is actually where the user believes it to be.

A.7 Site

A website on the Web.

A.8 User

The client who is potentially exposed to clickjacking while surfing the web with a Google Chrome browser on a personal computer.

A.9 Visual Context

What the user ought to see when they visit a site.

A.10 Web

The World Wide Web.