# Tricking Johnny into Granting Web Permissions

Mohammadreza Hazhirpasand
mohammadreza.hazhirpasand@inf.unibe.ch
University of Bern

Mohammad Ghafari
mohammad.ghafari@inf.unibe.ch
University of Bern

Oscar Nierstrasz
oscar.nierstrasz@inf.unibe.ch
University of Bern

## ABSTRACT

We studied the web permission API dialog box in popular mobile and desktop browsers, and found that it typically lacks measures to protect users from unwittingly granting web permission when clicking too fast.

We developed a game that exploits this issue, and tricks users into granting webcam permission. We conducted three experiments, each with 40 different participants, on both desktop and mobile browsers. The results indicate that in the absence of a prevention mechanism, we achieve a considerably high success rate in tricking 95% and 72% of participants on mobile and desktop browsers, respectively. Interestingly, we also tricked 47% of participants on a desktop browser where a prevention mechanism exists.

## CCS CONCEPTS

• **Security and privacy → Browser security**.

## KEYWORDS

Browser security, Web permission, UI security, Clickjacking

**ACM Reference Format:**
Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. 2020. Tricking Johnny into Granting Web Permissions. In *Evaluation and Assessment in Software Engineering (EASE 2020), April 15–17, 2020, Trondheim, Norway*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3383219.3383248

## 1 INTRODUCTION

In a clickjacking attack, an attacker tricks a user into clicking on webpage element that is hidden or disguised as another element. This can result in a situation in which the user takes an action unwittingly. For instance, attackers have used clickjacking attacks to trick users into liking a fan page on Facebook or re-tweeting a message on Twitter [2]. Several clickjacking attacks targeted the Adobe Flash Player's webcam access dialog to enable the victims' webcam and microphone [5]. Adobe finally fixed the issue by ensuring that the webcam access dialog is fully visible to users.


*EASE 2020, April 15–17, 2020, Trondheim, Norway*

ACM ISBN 978-1-4503-7731-7/20/04…$15.00
https://doi.org/10.1145/3383219.3383248

There are a number of ways in which clickjacking attacks can be prevented. The widely accepted approach is to use *frame busting* techniques. Frame busting refers to client-side code that is designed to prevent a given web page from being loaded in a sub-frame. Many JavaScript code snippets have been proposed to perform frame busting, though many of these have been found to be vulnerable later [8]. A more robust solution is to send a special HTTP header, *e.g.,* X-Frame-Options, which is supported by some browsers such as Mozilla Firefox and Google Chrome [7]. The X-Frame-Options header prohibits a website from being rendered within an iframe. However, some older versions of browsers do not support the special header that prevents Clickjacking attacks. Buchanan *et al.* analyzed Alexa's top one million sites and their results show that X-Frame-Options is implemented in only a small fraction (*i.e.,* 11.11%) of the websites [4].

In this paper, we discuss the lack of a preventive measure, *e.g.,* a delay period, in many browsers when the web permission dialog box pops up. This is a browser-dependent feature, and none of the aforementioned countermeasures could guarantee user safety. In order to highlight the importance of this issue, we investigate the following research question: *"How can the web permission dialog box be abused, and how effective are the existing preventive measures in browsers?"* To this end, we designed an experimental game, called "Furious Clicker," to evaluate whether or not users can be tricked into granting webcam permission by clicking on the *allow* button of the web permission API's dialog box. We asked 120 participants to take part in our web-based game experiment. We conducted three experiments, each with 40 different participants, on both desktop and mobile browsers. We used the mobile version of Google Chrome on Android, and the desktop version of Mozilla Firefox and Google Chrome on Mac OS. The results indicate that, in the absence of a prevention mechanism, we achieve a considerably high success rate in tricking 95% of participants on a mobile browser, and 72% on a desktop browser. We also tricked 47% of participants on a desktop browser where a prevention mechanism exists.

We conclude that without a preventive measure end users suffer from severe security implications. For instance, the Android version of the Google Chrome browser, with more than five billion downloads on Google Play, has no prevention mechanism. Nevertheless, the impact of any countermeasure, such as a delay when the permission dialog box appears, on the user experience needs to be taken into account.

The remainder of this paper is structured as follows. In section 2, we explain our motivation and various types of clickjacking attacks. In section 3 we introduce the Furious Clicker game. In section 4

**Table 1: Tested browsers and position of permission dialog box**

| Browser | Version | Platform | Position | Alert | Behavior | Prevention |
|---|---|---|---|---|---|---|
| Firefox | 68 | macOS - desktop | Top Left | Icon | Ask again | Yes |
| Chrome | 77 | macOS - desktop | Top Left | Icon | Never ask | Yes |
| Safari | 13 | macOS - desktop | Top Center | Icon | Ask again | No |
| Chrome | 78 | Android - mobile | Center | Notification | Never ask | No |
| Firefox | 68 | Android - mobile | Top | Notification | Ask again | No |
| Edge | 42 | Android - mobile | Center | Notification | Never ask | No |
| Dolphin | 8 | Android - mobile | Center | - | Never ask | No |
| Mint | 61 | Android - mobile | Bottom | - | Never ask | No |
| Samsung | 10 | Android - mobile | Bottom | - | Never ask | No |
| Mi Browser | 11 | Android - mobile | Bottom | - | Never ask | No |
| UC Browser | 12 | Android - mobile | Bottom | - | Never ask | No |
| Edge | 42 | Windows - desktop | Bottom | Icon | Ask again | No |
| Konqueror | 5 | Linux - desktop | Center | - | Ask again | No |
| Web | 3 | Linux - desktop | Top | - | Never ask | No |

we present a user experiment study, and discuss related work in section 5. We conclude this paper in section 6.

## 2 MOTIVATION

We present the web permission API, how it can be abused, and what factors are essential to consider in abusing the web permission API.

### 2.1 The web permission API

The web permission API offers a uniform way for websites to ask users' permission for critical features that require user consent, such as camera, clipboard, microphone, or geolocation. When a website requires a specific permission, a dialog box appears in the browser and the user can grant, deny, or dismiss the permission. Users can also revoke or grant the permission later in the browsers' settings. Nevertheless, for inexperienced users, it is challenging to find out where the permissions, which have been asked previously, are in browsers' settings.

The permission API behaves differently in different browsers. For example, a website cannot ask twice for a denied permission on the desktop version of Google Chrome, while it can do so after reloading on the desktop version of Mozilla Firefox.

### 2.2 Clickjacking attacks

There exist three main types of clickjacking attacks. In the first attack, *Jeopardizing target display integrity*, adversaries attempt to either set a sensitive UI element to be transparent and place an attractive decoy element beneath the main element, or they cover part of a sensitive UI element by overlaying a decoy element [12].[1]

In the second attack, *Jeopardizing pointer integrity*, adversaries show a fake cursor (pointer) instead of the real cursor to victims, and conceal the default cursor programmatically [3] [10].

In contrast to the two previous attack types, in the *Jeopardizing temporal integrity* attack, adversaries set a sensitive element to appear on top of a decoy element when users are busy clicking on the decoy element. The users observe the sensitive element, but are trapped into making an unwanted click due to having a very limited time to stop an action — humans need at least a few hundred milliseconds to react to a sudden visual change. Adversaries use this type of attacks in online games, *i.e.,* Adobe Flash webcam access, and security dialog bypass by captcha.[2] [3]

### 2.3 The problem

The usual way to keep users safe from *temporal integrity* attacks is to provide them with enough time to grasp any UI changes. For instance, to install a Chrome extension, users need to wait until the delay expires.

We found that almost none of the popular browsers take such a preventive measure when presenting the web permission API dialog box. The dialog box often appears in the area where websites present some content, *e.g.,* images or clickable elements, and therefore users might click on the *allow* button inadvertently.

We tested 14 browsers to collect data about where the permission dialog box appears, how each browser informs users that the webcam or microphone is in use, and how each browser behaves when the web permission dialog box is dismissed. The position of the dialog box depends on several factors. The first factor is that the buttons of the dialog box have different sizes in different browsers. For instance, in Safari, the buttons are narrower compared to those of Chrome, where the buttons are rectangular. Another factor, which is user-dependent, is that different types of bars below the address bar in browsers affect the position of the permission dialog box. For instance, the bookmark bar on the Chrome browser makes the dialog box appear at different coordinates. Table 1 shows the results of our observations. For instance, in the desktop version of Safari, version 13, the web permission dialog box appears in the top center of screen and informs users that the webcam is in use by placing a camera icon on the website's tab. In case a user clicks on the *deny* button of specific permission, an attacker can trigger the same permission under the same domain name more than once. In desktop browsers, the dialog box becomes visible usually on the

[1] https://feross.org/webcam-spy

[2] https://www.squarefree.com/2004/07/01/race-conditions-in-security-dialogs/

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=162020

top part of the browsers under the address bar. In all tested desktop browsers, there is a small blinking icon next to the website's icon, intended to catch the attention of users. In contrast, mobile browsers show a notification in the notification bar which clearly notifies users that the website is using your camera or microphone. Firefox, Konqueror, and Safari behave in such a way that websites can request specific permission more than once when users dismiss the permission for the first time. In contrast, the other browsers do not allow websites to ask for the same permission after being dismissed by users. Therefore, adversaries need to be careful in drafting their attack scenarios to precisely land a user's click on the *allow* button of the dialog box for such browsers. Only the desktop version of Firefox and Chrome accept user clicks with a short delay when a user clicks rapidly before the permission dialog box appears. The other tested browsers do not offer any preventive measures at the time of writing.

## 3 FURIOUS CLICKER

We now describe how we designed a game, called *Furious Clicker*,[4] to land a user click on the *allow* button of the web permission dialog box.

The game is designed in such a way as to persuade users to click quickly on an HTML element in order to complete an engaging task. The goal of Furious Clicker is to lift a basketball up and eventually drop it into a net, as shown in Figure 1. In the game, however, gravity is strong. To lift the basketball up, users need to defeat gravity by clicking very fast on a blue button, otherwise the ball will not go up far enough to fall into the net. The faster a user clicks on the button, the more the basketball goes up. We measure the time between two clicks of a user. If the time is very short (*e.g.*, less than 100 milliseconds), it conveys that the user is clicking fast and we raise the basketball more quickly.

In the beginning, the game asks the user to input a nickname, which is combined with a random number. In case the user clicks at a slow pace, the game encourages the user to click faster by displaying a message. The game issues two more messages depending on user's clicking speed, to persuade them to reach the highest speed. Eventually, when the ball reaches the net, the game displays a congratulatory message for completing the game successfully. On the desktop version of browsers, we place a basketball hoop on the right side of the screen to divert the user's attention to that side. This cannot be done in mobile browsers as the screen size is too small. Therefore, touch-based devices require a specific design, such as bigger icons and buttons, as users cannot click accurately on small HTML elements. The malicious blue button in the game, which is called the decoy element, must be placed in a precise position depending on the victim's user agent and platform, as discussed in subsection 2.3, otherwise, the attack will fail (see Figure 2).

In order to decide when to trigger the web permission dialog box, we established a clicking threshold to estimate the level of user engagement in the game. We iterated our initial test 20 times with different thresholds for the number of clicks. We determined that between 8 and 12 consecutive fast clicks indicate a high likelihood that the user will mistakenly click on the *allow* button. The duration

[4]https://www.crypto-explorer.com/clickjacking

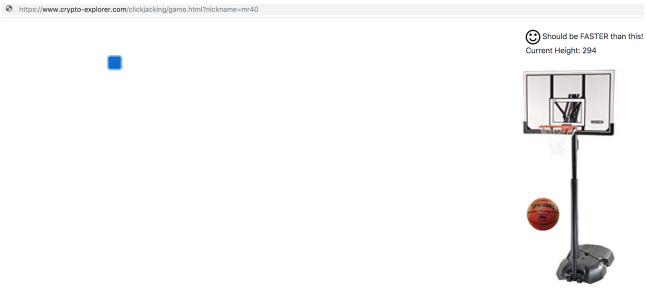


**Figure 1: The desktop version of Furious Clicker**

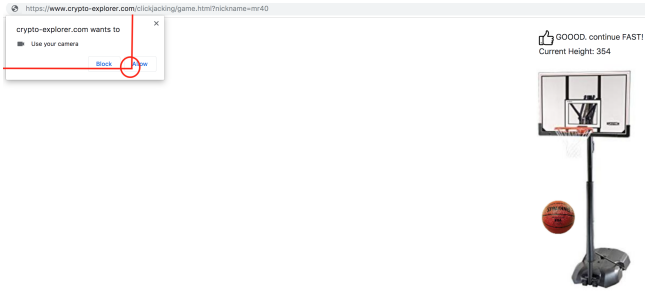


**Figure 2: The web permission dialog box's position in the game**

between each click needs to be less than 70 milliseconds, and the number of clicks should be more than 8 consecutive clicks.

Once a user reaches the fast clicking threshold, the game triggers the web permission. In case the user clicks on the *allow* button, the game takes a photo and silently sends it to the server. If the user stops clicking and chooses the *deny* button, the game will send a message to the server stating that the player is not tricked.

We examined different color schemes for the background color of Furious Clicker. To avoid distracting users, finally, we chose the white background color as it is close to the color of the web permission dialog box.

## 4 USER EXPERIMENT

We investigate how effective the Furious Clicker is in tricking users to click on the *allow* button when a browser presents a permission dialog box.

### 4.1 Method

We asked for interested students in twelve classes of a private educational technology and engineering institute to participate in our research study. We conducted three experiments with a total of 120 participants who willingly took part in our experiments without being paid. The participants were all studying practical courses related to engineering or computer science. They all had an academic background (*i.e.*, 67% Bachelor degree, and 33% Master degree). Only 22 had their degree in Computer Science. None of the participants had knowledge of or expertise in information security.

**Table 2: The three experiments for evaluating Furious Clicker**

| Browser | Version | Platform | Device | Prevention | # of Participants |
|---|---|---|---|---|---|
| Chrome | 72 | MacOS | Mouse | Yes - delay | 40 |
| Safari | 13 | MacOS | Mouse | No | 40 |
| Chrome | 78 | Android | Finger touch | No | 40 |

We designed three different experiments to assess how successful the Furious Clicker is in tricking users to grant the camera permission. Table 2 gives an overview of the three experiments. In particular, we designed one experiment to examine the impact of the existing preventive measure when clicking fast in the desktop version of Chrome; the second experiment tests the Safari browser which offers no preventive measure; and finally, we conduct an experiment with the mobile version of the Chrome browser, which also offers no preventive measures. In the first two experiments, participants are required to use a mouse, whereas in the last experiment users use the touch feature of the smartphone.

We divided the participants into three equal groups, each consisting of 40 people, and assigned each group to a different experiment. We first conducted an offline survey to understand the level of familiarity of participants with browsers and web permissions. We then presented the game to participants of each group and made sure that everyone understands how the game works. We stated that the goal of the test is to measure how fast people are able to click and evaluates the impact of such games on gamers in which fast clicking is required. We also observed how each participant engages in the game.

We interviewed each participant right after playing the game. We asked their opinion about the game and also what they observed while playing the game.

## 4.2 Results

We present the results of the three experiments in Figure 3. In the first experiment, 19 participants clicked on the *allow* button and did not realize what they clicked. Suspicions of all the remaining 21 participants arose as they observed the web permission dialog asking for camera access. This is due to the short delay implemented in Google Chrome. Of the users who were not tricked, 14 stopped playing the game as soon as they saw the permission dialog box pop up, and seven closed the dialog box, and continued playing after a short interruption.

In the second experiment, 29 participants clicked on the *allow* button and finished the game successfully. Seven contestants were not tricked because of their incorrect mouse coordinates during the fast click process. Their mouse pointer exited the region of the decoy button when the dialog box appeared. The rest of the not tricked participants stopped unexpectedly exactly when the game decided to show the web permission dialog box.

In the last experiment, we used the mobile version of Google Chrome on Android devices. In total, 38 participants were tricked into clicking on the *allow* button and only two of them suddenly stopped while clicking on the button.

We were interested to know how familiar participants are with web permissions. We therefore surveyed participants before conducting the experiments. To minimize the Hawthorne effect, we

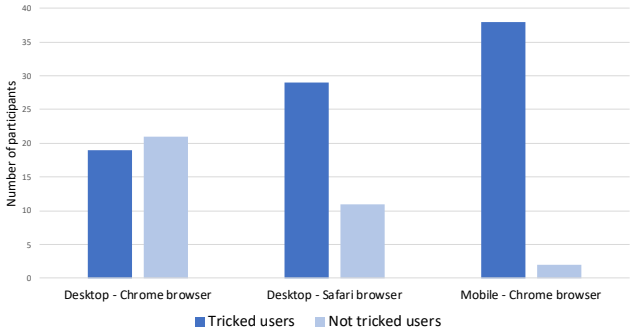


**Figure 3: The number of participants with different status in each experiment**

also populated the survey with some questions regarding other features of browsers, not relevant to web permissions, such as browser history, bookmarks, anonymous mode, and 3D games [6]. Almost 83% of all 120 participants had never played a game on browsers while others had played at least once. Participants were mainly familiar with the history feature of browsers (95%). More than three quarters (83%) of the participants had their customized bookmarks in their favorite browser. As the privacy of data has been emerging as a concern for web users, the anonymity feature of browsers had been used by 57% of the participants. We also asked if they had any experience in granting permission in the web environment. Participants did not know much about permissions and how they work in browsers as only 42% had some experience with web permissions. Finally, the least familiarity was for 3D games as only 10% of participants played such games in web settings.

At the end of each experiment, we asked the participants to express their judgment about the game. The 34 participants who were not tricked in all three experiments accurately realized why the button was located there and why they needed to click fast. In the first experiment, nine participants from the tricked users group realized that they clicked on the *allow* button but they were not familiar with the web permission dialog box. As a result, they could not figure out what the consequences might be of clicking on the *allow* button. After the second experiment, eight participants who were tricked into clicking on the *allow* button noticed a box appeared but could not guess what it could be. From the ones who were tricked in the smartphone's browser, only two participants had a weak doubt whether it was a web permission dialog box or not. Participants in the mobile browser experiment found that it was extremely difficult to avoid clicking on the *allow* button while clicking rapidly. Although the prevention technique in Google Chrome reduced the chance of fooling users by 52%, the preventive approach cannot prevent such attacks effectively.

### 4.3 Discussion

Interestingly, among all the participants, only eight people knew how the granted permission could be revoked from the desktop and mobile browser. This conveys that it is complicated for inexperienced users to revoke given permissions in browsers. The findings of this study suggest that browsers should list a website's granted/blocked permissions in an easy to access manner. In practice, browsers need to implement some preventive measures to decrease the likelihood of clickjacking attacks. As we observed, Google chrome has considerably decreased the impact of this attack by applying a very short delay on its desktop version. Browser vendors can employ the delay technique for the *allow* button. However, it might not be a sufficient defense mechanism against such attacks as we proved that almost half of our participants were tricked into clicking the *allow* button. To boost the prevention approach, we suggest invalidating the focus of the mouse pointer for a short period to allow users to perceive the instant UI changes. However, finding a tradeoff between a preventive measure and user experience is a challenging problem, which requires a dedicated study. For instance, a long UI delay could have a negative effect on user experience.

We reported the issue to Mozilla Firefox and Google Chrome and the two companies approved the validity of the problem. Fortunately Google Chrome in its newest version has patched this problem as follows: if the dialog box appears when the user is busy clicking, it does not accept any clicks until the user pauses for a second and clicks again. Although the desktop version of Google Chrome is armed with the new mitigation strategy, the mobile version of this browser has no prevention mechanism yet. We have contacted the remaining browser vendors and are awaiting their response.

## 5 RELATED WORK

Rydstedt *et al.* analyzed the top 500 websites to study if they implemented any preventive measures regarding clickjacking attacks [8]. They found that all the defense mechanisms used by the websites can be circumvented. To prevent clickjacking, they proposed a JavaScript-based mitigation method as a temporary measure until browsers fully support the X-FRAME-OPTIONS header. Five new clickjacking attacks were introduced by Akhawe *et al.* to circumvent current UI safety specifications proposed by W3C [1]. They obtained a success rate between 20% to 99% in challenging the limitations of current defenses against UI attacks. The proposed attacks exploit various aspects of human perception, for instance, adaptation, attention, and peripheral vision.

The ProClick framework, developed by Shahriar *et al.*, is designed to work as a proxy-level clickjacking detection framework [9]. Their framework works by examining the contents of requests and responses at the proxy level to detect clickjacking attacks. Shamsi *et al.* developed a clickjacking prevention tool, called Clicksafe [11]. The developed Firefox add-on intercepts users while clicking on clickable elements with redirection code and warns users by a popup displaying necessary information. In a study, Huang *et al.* proposed the InContext defense mechanism to enforce context integrity of user actions on sensitive UI elements [5]. Similarly, they developed a game in which they use a fake cursor and ask users to click rapidly on a button shown randomly in different locations in a browser. Once the users are engaged in the game, the game switches to a Facebook *Like* button at the real cursor's location, tricking the user into clicking on it. They carried out an experiment with 2 064 participants and achieved a success rate ranging from 43% to 98% in different scenarios. They concluded that InContext could assist participants against visual and temporal integrity clickjacking attacks.

The proposed detection tools and circumvention of defense mechanisms consider HTML elements, JavaScript code snippets, or specifications proposed by W3C. However, in this study, we target a browser feature, which is neither dependent on HTML elements nor JavaScript defense mechanisms.

## 6 CONCLUSION

We have presented Furious Clicker, a web-based game to trick users into granting web permissions. We tested two desktop browsers and one mobile browser in order to see how users react when the permission dialog box appears. A preliminary study showed that Furious Clicker is successful in tricking 72% of the 120 participants. Due to the rise of mobile phone usage nowadays, we believe that this issue has a grave impact on the security of mobile internet users. Browsers can prevent this issue by using the delay method for buttons or by invalidating the focus of the mouse pointer to interrupt clicking. However, finding a satisfactory tradeoff between a preventive measure and user experience is a challenging problem, which requires further study.

## 7 ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assistance" (SNSF project No. 200020-181973, Feb. 1, 2019 - April 30, 2022). We also thank CHOOSE, the Swiss Group for Original and Outside-the-box Software Engineering of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

## REFERENCES

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. 2014. Clickjacking Revisited: A Perceptual View of UI Security. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*.

[2] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. 2010. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 135–144.

[3] Eddy Bordi. 2010. Proof of concept-cursorjacking (noscript).

[4] William J Buchanan, Scott Helme, and Alan Woodward. 2017. Analysis of the adoption of security headers in HTTP. *IET Information Security* 12, 2 (2017), 118–126.

[5] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. 2012. Clickjacking: Attacks and defenses. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 413–428.

[6] Rob McCarney, James Warner, Steve Iliffe, Robbert Van Haselen, Mark Griffin, and Peter Fisher. 2007. The Hawthorne Effect: a randomised, controlled trial. *BMC medical research methodology* 7, 1 (2007), 30.

[7] Marcus Niemietz. 2011. Ui redressing: Attacks and countermeasures revisited. *in CONFidence* 2011 (2011).

[8] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. 2010. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web* 2, 6 (2010).

[9] Hossain Shahriar, Vamshee Krishna Devendran, and Hisham Haddad. 2013. ProClick: a framework for testing clickjacking attacks in web applications. In *Proceedings of the 6th International Conference on Security of Information and Networks*. ACM, 144–151.

[10] Hossain Shahriar, Hisham Haddad, and Vamshee Krishna Devendran. 2015. Request and Response Analysis Framework for Mitigating Clickjacking Attacks. *International Journal of Secure Software Engineering (IJSSE)* 6, 3 (2015), 1–25.

[11] Jawwad A Shamsi, Sufian Hameed, Waleed Rahman, Farooq Zuberi, Kaiser Altaf, and Ammar Amjad. 2014. Clicksafe: Providing security against clickjacking attacks. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. IEEE, 206–210.

[12] Paul Stone. 2010. Next generation clickjacking. *BlackHat Europe* (2010).