DOI: 10.21917/ijct.2016.0208

PREVENTING CLICK EVENT HIJACKING BY USER INTENTION INFERENCE

Kailas Patil
Center of Excellence in Research and Development, Vishwakarma Institute of Information Technology, India
E-mail: kailas.patil@viit.ac.in

Abstract

Web applications are getting more complex and dynamic. By exploiting layout and JavaScript features of a web page, attackers can create web page objects that hijack users' clicks. Such objects look like normal web page objects, but users' clicks on these objects lead to unexpected browser actions, such as visiting different URLs or sending out malicious requests. We call this type of attacks click event hijacking attacks. The Facebook Clickjacking attack is an example, which puts a transparent layer containing the victim web application on top of another web page that lures users to click. While users think they click on the underlying web page, they actually click in the victim web application, resulting in unauthorized actions to the web application. In this paper, we propose a solution to mitigate the problem of click event hijacking by inferring users' intentions. Our solution ClickGuard ensures that the browser's behavior after a click matches the user's original intention. The proposed solution is implemented as a Mozilla Firefox extension and evaluated its effectiveness against click event hijacking attacks.

Keywords:

Event Hijacking, Clickjacking, Pop-Up, UI Overlay

1. INTRODUCTION

Browsers have evolved dramatically from a program to display simple static web pages into an environment to run Rich Internet Applications (RIA), which often use a complex layout consisting of components or pages from different sources. Moreover, web applications also heavily rely on dynamic features such as those provided by JavaScript. By exploiting the complex features of web applications, attackers can create web page objects to hijack users' clicks. Hijacked clicks may lead to unexpected browser actions, such as visiting phishing websites, or sending malicious requests in web applications.

The Clickjacking attack [15][23][39] also known as UI redressing, is an example of such attacks. In this attack, attackers carefully craft overlapped layers on web pages to trick users into clicking web page objects without their consents. For example, in the Facebook attack [15], the malicious web page includes an invisible layer loaded with Facebook's page on top of a game web page. In this way, attackers trick users into clicking objects in the game, but the clicks actually occur on a button in the Facebook page, whose event handler in turn sends requests to share the malicious page with the victim's friends.

As another example, attackers may also use the onClick event of a hyperlink to redirect the browser to arbitrary pages. This type of attack is often used to launch phishing attacks or opening annoying pop-up advertisements. For example, a malicious web page can include a link to a bank website A but use the onClick event handler to redirect the browser to visit a phishing site B after users click on the link. By exploiting the cross-site scripting (XSS) [25], [35] vulnerability, attackers can attach onClick event handlers to links in trusted web sites to redirect users to phishing pages, which helps the phishing site to gain additional trust. If a user hovers the mouse pointer over the hyperlink to the site A, the browser's status bar still shows the URL to the site A. Although not as dangerous as Clickjacking, this type of JavaScript-based click redirection can annoy users or expose them to malicious sites hosting phishing pages or malware.

Although the actual techniques involved in these attacks may vary, they generally aim to make users' clicks trigger browser actions that users do not expect. We name this type of attacks as click event hijacking attacks, and we will describe more examples in section 2.

Researchers proposed solutions to detect the Clickjacking attack [40][41][42]. The NoScript [24] Mozilla Firefox extension uses a module called ClearClick to protect users against Clickjacking attacks. When a click happens on a web page with embedded elements that are partially obstructed or transparent, ClearClick suspends further actions triggered by the click and reveals the real click target to users. Balduzzi et al. [7] develop a Clickjacking detection system and perform a large-scale study of Clickjacking. Such solutions target specifically to the characteristics of Clickjacking.

We observe a common behavior in the general click event hijacking attack: users are lured to click on page objects that they will not click if they know the resulting browser behaviors. That is, based on the information presented to users, they believe the browser will perform certain actions when they click, but the actual browser behaviors are different. In other words, the actual browser behaviors do not match users' original intentions.

With this observation, we present a novel solution named ClickGuard to mitigate click event hijacking attacks. Its goal is to ensure that the web browser's behavior resulting from a click matches the intention of the user. When users make clicks, ClickGuard infers users' intentions. It then tracks the web browser's behaviors and ensures the resulting activities match the inferred intentions. ClickGuard is prototyped as a Mozilla Firefox extension. In our evaluation, it generated promising results by successfully preventing several types of click event hijacking attacks.

There are research efforts using user intentions to detect malicious behaviors in operating systems. BINDER [10] correlates user inputs to outbound network connections by the time delay in between, and network traffic that cannot be attributed to user inputs is considered suspicious. User intentions are also incorporated into access control policies [27], [31]. Our approach shares the same insight as the above cited works – user intentions are the most important indicators of the legitimacy of subsequent actions, but our approach uses more accurate user intention inference via analyzing browser internal details.

This paper makes the following contributions:

- We analyzed different types of click event hijacking attacks and summarized the key characteristics of them.
- We proposed an approach targeting all click event hijacking attacks by ensuring that browser behaviors match user intentions.
- We implemented a prototype of our approach in a Firefox extension and evaluated it with several attack examples, which showed that user intention based attack detection is effective.

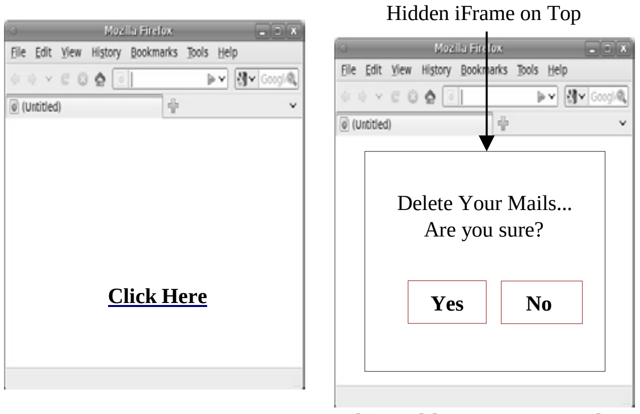
Paper organization: The rest of this paper is organized as follows: section 2 describes examples of attacks, section 3 explains the design of our approach, and then section 4 presents implementation details. Later we describe the evaluation for our solution in section 5. Related work is introduced in section 6. We discuss our limitations and future work in section 6, and we conclude in section 7.

2. EXAMPLES

In this section, we discuss examples of different types of click event hijacking attacks.

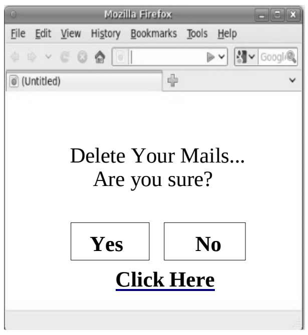
2.1 CLICKJACKING

In Clickjacking attacks [14], [23], attackers exploit the layout feature introduced by iFrames. Specifically, they load a victim web page into an iFrame on the top and make it transparent. Then they load a deceptive page in another iFrame at the bottom layer to attract users to click. The Fig.2 shows an example of a Clickjacking attack. The front page of http://example.com is loaded inside a transparent iFrame (zero opacity value to make it transparent). To lure users to click at a particular location of the page loaded inside the transparent iFrame, the attacker creates a link in the visible bottom layer, which is located exactly at the same position where the attacker wants users to click in the top layer. As shown in Fig.2, the attacker specifies the location of a link by setting its X and Y coordinates. When users try to click on the link, they actually click on the transparent layer of the iFrame loaded with the page from example.com. An illustration of such a Clickjacking attack is presented in Fig.1.



(a) User View of Web Page

(b) Hidden Content in the Web Page



(c) Actual Web Page Layout

Fig.1. Illustration of Clickjacking using transparent iFrame and overlay objects

```
<-- Page from www.Websitename.com -->
< html > ...
<iframe id="victim" src="http://example.com"
scrolling="no" width="600px" height="600px"
style="opacity: 0; position:absolute; left:10px;
top:10px;">
< /iframe > ...
<div style = "position:absolute; top:Ypx;left:Xpx;"> <a
href= "http://example.com">Click
Here</a>
< /div > ...
< /html >
```

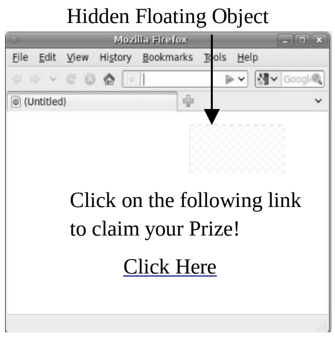
Fig.2. Clickjacking using transparent iFrame and overlay objects

2.2 FLOATING OBJECTS

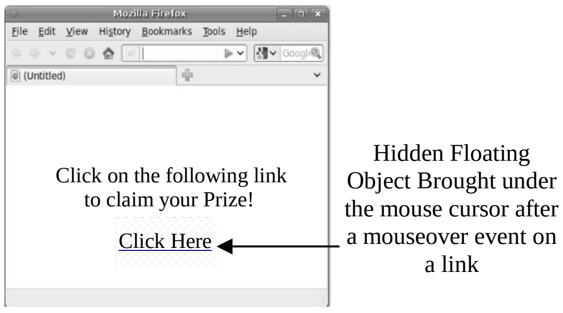
Alternatively, attackers can put malicious code inside a floating object, and automatically bring that object under the mouse pointer when users hover the mouse pointer over a particular link, which triggers the malicious code in the floating layer. The Fig.3 shows such an example, where float_layer is a JavaScript class defined by the web site that is hidden and floating around on a web page. The scenario of a floating object attack is illustrated in Fig.4.

```
<--Page from www.Websitename.com -->
    <float_layer id="layerk"
        onclick="document.location='http://www.malicio
        us.com';"
        style="position:absolute;width:2px;height:2px">
    < /float_layer >
    <script type="text/javascript"> function
        clickjack(evt) {
        mouseX=evt.pageX?evt.pageX:evt.clientX;
        mouseY=evt.pageY?evt.pageY:evt.clientY;
        document.getElementById('layerk').style.left
        =mouseX;
        document.getElementById('layerk').style.top
        =mouseY; }
    < /script >
    <a href="http://www.example.com"
        onmouseover="clickjack(event)"> Click here
        </a>
```

Fig.3. Floating object example


Hidden Floating Object
File Edit View History Bookmarks Tools Help
(Untitled)
Click on the following link to claim your Prize!
Click Here

(a) Before hovering the link


File Edit View History Bookmarks Tools Help
(Untitled)
Click on the following link to claim your Prize!
Click Here
Hidden Floating Object Brought under the mouse cursor after a mouseover event on a link

(b) After bringing the mouse on a link

Fig.4. Illustration of a floating object in a web page

The example shown here uses the onMouseover event. After onMouseover is triggered when users move the mouse cursor over the link, the float_layer object is moved to the point of the current mouse cursor, and when users click on the link, they click on the float_layer object. Therefore, the onClick event of the float_layer object is triggered.

Attackers are not limited to hijack onMouseover. They can also exploit onKeyup, onKeydown or other JavaScript event handlers to launch attacks. For a more complete list of exploitable JavaScript event handlers, we point readers to [13]. Attackers can also use the floating object concept to make the cursor follow the iFrame on which they want users to click [23].

2.3 POP-UP ON CLICK

By default, web browsers only allow pop-ups triggered by certain user interactions, such as clicks or double click. Therefore, attackers cannot create web pages those popup windows automatically. Instead, they have to generate pop-ups when users click on the page. The Fig.5 shows such an example. When users click on the link that displays its destination as http: //www.example.com, a pop-up window will be opened.

```
    <script Language="JavaScript"> function
       popUp() { window.open(
            "http://www.malicious.com",
            "malwin");
       }
    </script>
    <a href="http://www.example.com"
onclick="javascript:popUp()"> Click here </a>
```

Fig.5. Pop-up on click Example

This can be modified to open pop-up windows only when users click inside a particular area or on a particular object on the web page. In these scenarios, users' original intentions of making clicks are subverted into opening a pop-up window.

Our observation: From the examples above, we can see that the intrinsic property of click event hijacking is the mismatch of user intentions and actions taken by the browser, resulting from client-side scripting and complex page layouts. When users click somewhere on a web page, they believe the browser would perform common actions they expect, such as going to a target web page or submitting form data to the server. Careful users will also check more information provided by the browser before they click, such as the destination URL shown in the status bar when the mouse pointer is over the link. The status bar information is partially protected by web browsers that prevent JavaScript code from changing the status bar information. However, this protection does not prevent attacks discussed in this paper. Therefore, even if users check the destinations of the elements they click on and believe the click will result in expected browser actions, the browser may actually carry out a different action controlled by attackers.

3. DESIGN OF CLICKGUARD

The main idea of our solution is to ensure the browser's action after a click matches the user's original intention. To achieve this goal, we first need to know what users intend to do. Second, we need to know browser's behaviors resulting from the click, which can be intercepted as browser events. With user intentions and the corresponding browser behaviors, we can check whether they match, and report unmatched results to warn users of the potential threat.

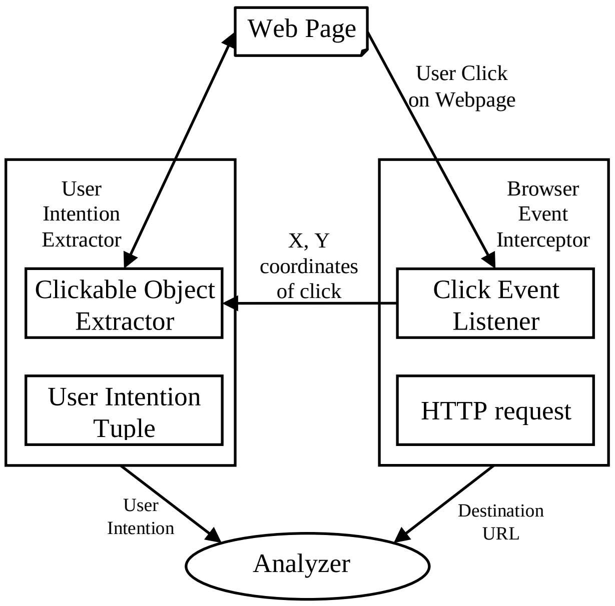


Fig.6. Component overview of ClickGuard

3.1 OVERVIEW OF PROPOSED APPROACH

The Fig.6 presents the component overview of ClickGuard. It has three main components: browser event interceptor, user intention extractor, and analyzer. The browser event interceptor intercepts important browser events, such as JavaScript events and HTTP request events. When a click event is intercepted, the user intention extractor infers the user intention of the click from web page objects in the browser window, and associates the inferred user intention with the click event. When the browser is about to perform important actions, the browser event interceptor finds the corresponding click event and its associated user intention, and asks the analyzer to match the extracted user intention with the suspicious browser event. If they do not match, the analyzer reports an attack and triggers actions specified by users, such as displaying an alert.

3.2 INTERCEPTING BROWSER EVENTS

This module intercepts two main types of browser events: input events and output events. Input events are those corresponding to user actions in web applications, such as user clicks, which reflect user intentions. Output events are events that can modify web applications' states or transfer data to external parties, such as HTTP requests.

The browser event interceptor intercepts input events by registering listeners for JavaScript events related to user actions. For example, when a user click happens on a web page, the onClick JavaScript event occurs and notifies the corresponding event listener. Then the event listener may invoke user-defined event handlers, which are JavaScript functions that have the access to event properties, such as, type, target, pageX, pageY, screenX, and screenY, etc. Type indicates the type of the event; target indicates the object to which the event is originally sent; pageX, pageY, screenX, and screenY represent the cursor location at the time the event occurs. Event handlers can perform tasks such as modifying the content of their web pages or sending out HTTP requests. The output events are HTTP requests, the main interface for web applications to communicate with the outside world. ClickGuard intercepts HTTP requests just before they are sent out.

When the intercepted output event is triggered, ClickGuard starts the detection process. It finds the input event that triggered this output event and invokes the analyzer. Next we will discuss how to infer user intentions from input events, and the details of correlating input events to output events are explained in section 3.4.

3.3 INFERRING USER INTENTIONS

After an input event is intercepted, ClickGuard infers the user intention for this event before the browser starts processing the event. This task must be performed immediately because the browser environment may change during the processing of the event, making users' original intentions harder to find.

Intuitively, if users click on a multi-layered region on a web page, the object on the visible layer is what they want to click; if they click in a single-layered region, the object clicked is what they want to click. Next, we use the object on which users want to click and its attributes to infer users' intentions.

In ClickGuard, the user intention is defined as a tuple <Target Object, Destination URL>, explained as follows:

- "Target Object" is a clickable object found under the mouse pointer when a click happens, which is considered the target object of the click. Clickable objects are either hyperlinks or button. Hyperlinks include HTML elements <a> and <link>. The appearance of hyperlinks depends on the HTML content embedded in these hyperlinks, such as text, images, etc. Other HTML elements such as <div>, <b> are considered as non-clickable objects and are not considered as user intentions, because they do not reveal any destination information to the user. We believe that it is not a good practice to use nonclickable objects as links, since this would confuse users in understanding the potential behaviors of web page objects.
- "Destination URL" is the URL of the inferred destination of the Target Object.

When a user clicks on the web page, ClickGuard stores the user intention tuple that is later fed to the analyzer component for matching the user intention with the actual browser behavior. To compose the user intention tuple, we need to obtain the Target Object as well as the Destination URL. The main challenge of this step is to deal with the complexity of page layout. Next we explain how ClickGuard finds the Target Object when there are multiple layers, and how the Destination URL is inferred from the Target Object.

3.3.1 Finding the Target Object from Multiple Layers:

If there are multiple layers on a web page formed using Frames or iFrames and there are multiple objects under the mouse pointer when the click happens, we find the Target Object based on the visibility of the objects under the mouse pointer. Our approach determines the visibility of layers and objects based on their opacity values. The opacity attribute [33] is used by the Cascading Style Sheets (CSS) to adjust the visibility of objects on a web page, where 0.0 denotes fully transparent and 1.0 indicates fully opaque. We consider an object as more related to the user intention if its opacity value is higher. Our approach works as follows:

3.3.2 Collect all Objects Under the Clicked Position:

We traverse the DOM (Document Object Model) [36] tree to examine the main web page and its iFrame and Frame elements to retrieve all objects under the mouse pointer at the time of user click.

3.3.3 Check Opacity Values:

If there are multiple layers in the web page and the layer clicked on is transparent, we check the opacity values of all objects collected in the previous step.

3.3.4 Choose the Object Representing User Intention:

Here we filter out objects with lower opacity values than our threshold. If none of the objects have an opacity value above our threshold, we record nil as the Target Object. If multiple clickable objects have the same opacity value higher than our threshold, we examine their vertical layer order and pick the object on the top as the Target Object.

3.4 OBTAINING THE DESTINATION URL FROM THE TARGET OBJECT

Now we describe the way we infer the Destination URL information from the Target Object. If the Target Object is a link created by an HTML [4] anchor element (<a>), then it is of type HTMLAnchorElement, and has an HREF attribute. In this case, the value of the HREF attribute is taken as the Destination URL.

For example, for Fig.2 and Fig.3, when the user clicks on the link, our interceptor stores the user intention information as <Target Object, http://example.com> and <Target Object, http://www.example.com>, respectively. If the Target Object is a submit button in an HTMLFormElement, it does not have the HREF attribute but its enclosing HTMLFormElement has an ACTION attribute. We take the value of the ACTION attribute as the Destination URL in the user intention tuple. If users click on blank space or a non-clickable object, then the Target Object is nil. We believe that there is no user intention to send a new HTTP request or open a pop-up window. In this case, the Destination URL is also nil.

3.5 CORRELATING OUTPUT EVENTS TO INPUT EVENTS

To match browser behaviors with user intentions, ClickGuard needs to correlate an output event to the input event (as described in section 3.2) that triggers it. For example, a single click on a web page may be followed by multiple HTTP requests: some of the requests result from the click, while others may be generated by other asynchronous events in the browser.

So for each browser action intercepted, we need to figure out whether it is triggered by an input event or not. If the HTTP request is not triggered by an input event, we should allow those requests, as web pages often send requests automatically to same origins or other domains to load resources (such as images, flash, etc.). If the HTTP request is triggered by an input event, we need to find the user intention information associated with it. In our approach we leverage the JavaScript call stack to find the association.

In Firefox, JavaScript-to-JavaScript function calls are implemented using a JavaScript call stack [5]. When a JavaScript event occurs, the JavaScript event handler is pushed onto the top of the JavaScript call stack, and the event handler executes the pre-defined JavaScript code inside that event handler. When the execution terminates, its frame is popped off from the JavaScript call stack. If a JavaScript event handler initiates an HTTP request, it does not terminate until the HTTP request is sent to the network. As a result, if there exist a JavaScript event handler in the JavaScript stack, it means this HTTP request results from that event handler. If the current JavaScript stack does not have any JavaScript event handler, then this HTTP request is not from client-side scripts and is allowed by ClickGuard. However, one exception is indirectly generated HTTP requests. For example, the click event handler can use the JavaScript setTimeout() or setInterval() functions to generate timed or deferred execution of specified code. In these cases, when the HTTP request is finally initialized, it is the timeout or interval event handler that is on the JavaScript call stack, not the original click event handler. Our current version of ClickGuard does not handle these cases, and we will discuss more on this issue in section 7.

3.6 DETECTING AND RESPONDING TO ATTACKS

Once ClickGuard correlates an output event to its corresponding input event, it retrieves the user intention associated with the input event, and activates the analyzer. The analyzer detects click event hijacking attacks by two inputs: the user intention and the intercepted output event.

We match user-intended destination against the target in the intercepted output event. We perform two checks between the user intention and the HTTP request output event using a policy similar to the same-origin policy (SOP) [38].

The first check is between the Destination URL value in the user intention and the destination URL of the HTTP request. The second check is between the URL of the enclosing web page of the Target Object in the user intention tuple and the URL of the web page that initiates the HTTP request. As we mentioned in section 3.3, the URL of the web page enclosing the Target Object is stored as one of its attributes. If either check fails, ClickGuard shows a security warning.

The first check ensures there is no JavaScript-based click event hijacking attacks and the second check is used to prevent layout-based attacks. As in the example of Fig.2, the attacker creates a visible link on the bottom layer to trick users into clicking on it. However, what is actually clicked is the invisible object in the layer above. In this case, the link in the bottom layer is inferred as the user intention by ClickGuard. An HTTP request is prepared for the object in the top layer, and its destination is matched against the URL extracted from the user intention. Since the link at the bottom layer will never be touched, the attacker can freely set its destination to that of the invisible object above it to bypass our first check. But this attack will be detected in our second check, as the attacker's web page has a different URL than the trusted one embedded in the iFrame.

For HTTP request output events, we use the following check on origins. We define the origin as a combination of the protocol, host name, and port number of a URL, the same as that in the same-origin-policy (SOP) [38] enforced by browsers. We require the two URLs to have the same origin, because naturally the actual destination should not have significant difference from the user intention.

3.7 INFERRING HOST RELATIONSHIPS BY COOKIE POLICY

The criterion above is strict, as it prevents requests from being sent to hosts in different sub-domains of the same domain, which is common in big web applications. For example, a website hosting videos may store videos on its sub-domains to reduce the load on the main web server. To prevent such unnecessary restrictions, we propose a method to automatically infer relationships among hosts within one domain.

HTTP, a stateless protocol, uses cookies to track user sessions, to authenticate users to web applications, or to remember custom preferences about users. The contents of cookies are name = value pairs [18]. To allow cookies to be sent to sub-domains, a web application sets domain = <domain-name> pair in its cookie. When the cookie is going to be sent to a website, the URL of that website is compared with the domain attribute of the cookie. If there is a tail match, then the cookie can be sent to that sub-domain. For example, if the domain attribute has the value .example.com, then it is allowed to send cookies to sub-domains such as first.example.com, second.first.example.com. However, it is not allowed to send cookies to example.first.com. The presence of a leading dot (.) in .example.com indicates it is a domain cookie; otherwise it is treated as a host cookie, which is sent back, during subsequent visits, only to the server that sets it. A domain cookie is sent back to any site in the same domain as the site that sets it.

The sub-domain policy in cookies indicates the trust among web servers within one domain. We leverage this information to handle sub-domain communications of JavaScript on legitimate websites to avoid false positives. If the origin check we perform fails, we check the cookie policy. Specifically, we check the domain attribute in the cookie. If it exists, we perform tail matching between fully qualified domain name of the HTTP request and the domain attribute value in the cookie. If it fails we report failure of the check; otherwise, the check passes. If the domain attribute is not set by the web application in its cookie, we also report failure of the check.

In summary, we relaxed our same origin checking criteria, and allowed access to sub-domains if the cookie policy allows cookies to be sent to them.

4. IMPLEMENTATION

ClickGuard is made as a prototype that works as an extension [21] of the Mozilla Firefox browser. We chose Firefox as the platform because it is the most popular open-source web browser. Our approach can be implemented by either modifying Firefox's source code or extending Firefox's functionality by its extension interfaces. We decided to implement ClickGuard as an extension since it is much easier to deploy and distribute. Meanwhile, the event interception and object access functionality needed by our approach are mostly supported by Firefox's extension interfaces. More on this issue is discussed in Section 7.

To intercept input events, ClickGuard intercepts JavaScript events such as click, and keypress. By calling the function: targetObject.addEventListener(eventType, listener, capt) in Mozilla Firefox, we can add an event listener on targetObject which can be an HTML document, window, etc [20]. The first parameter eventType can be click, keypress, etc. The second parameter, listener, is a JavaScript event listener function that will be invoked when the event occurs. The third parameter is important to us. If the third parameter is true, when the specified event occurs, our registered listener will be notified first before it is dispatched to other event targets. To get the genuine user intention we need to execute our listener before the execution of other event handlers. Therefore, when we register the event using the addEventListenner function, the third parameter must be set to true.

To intercept HTTP-related output events, ClickGuard intercepts the HTTP request event via the http-onmodify-request notification in Firefox. Firefox sends an http-on-modify-request notification to extensions after an HTTP request is prepared and before it is sent out to the network. Notifications are just like events or signals in other programming languages and frameworks.

We use the JavaScript call stack to look for correlations between input and output events. To get the JavaScript call stack, we use a Mozilla specific property stack of the Error object. It shows the functions called, their order and the arguments to them. Alternatively in Mozilla Firefox extension we can also use the Components.stack property of the nsIStackFrame interface.

The prototype of ClickGuard registers event listener functions for the click, mouseover, and keypress events. It is straightforward to include other JavaScript events, such as mousedown, keyup, keydown, etc [1].

In this implementation, we use 0.2 as our threshold value of opacity, since during our experiments layers with opacity values less than 0.2 were hardly visible. To decide the vertical layer order of objects, we use the CSS z-index attribute [34] of the layers to identify their vertical layer order. The z-index attribute specifies the stack level of the generated box in HTML rendering, and the layer at the top has the largest z-index value.

5. EVALUATION

We evaluated the prototype of ClickGuard on a computer with an Intel Core 2 Duo 2.33GHz CPU and 4GB RAM, running Ubuntu 9.10. We tested our solution in Mozilla Firefox.

5.1 EFFECTIVENESS

We evaluated the effectiveness of ClickGuard using several types of click event hijacking attacks. We created attack examples for each type of attacks discussed in section 2.

5.1.1 Floating Objects:

The Fig.3 is an example that floats an object and brings it under the mouse pointer on the onMouseover event, which subsequently triggers the onClick event handler on the floating object. It coordinates two events to achieve the goal of changing the DOM location on the fly. To detect this kind of attacks, we collected all clickable objects under the mouse pointer and recorded the Destination URL value as http://www.example.com in the user intention tuple. On the HTTP output event, we performed the first check between the URL of the HTTP request (malicious.com) and the Destination URL in the user intention (http://www.example.com). It failed because the host part in the Destination URL and the HTTP request were different. Therefore, we showed a security warning to the user.

5.1.2 Pop-up on Click:

On some carefully crafted websites, even a click in a white-space area triggers a pop-up window, which is annoying, and potentially insecure to users. Some of the most annoying pop-up window actions include windows that continually reopen themselves whenever users attempt to close them [19]. The Fig.5 is an example to open a pop-up window when a user click event occurs on a web page. Our prototype ClickGuard recorded the X and Y coordinates of the mouse click and retrieved all objects at that position. However, if a user click occurs in blank space or on a non-clickable object such as a piece of text on the web page, the user intention extractor returns nil. In the HTTP request event interceptor, we correlated HTTP requests with user clicks by examining the presence of click event listener in the JavaScript call stack. The first check failed in this case, because the user intention extractor returned nil. We showed a security warning to the user.

5.2 FALSE POSITIVE AND PERFORMANCE

In our system, a false positive is a normal page that is detected as malicious. To evaluate false positives generated by our system, we did arbitrary surfing among the top 180 sites from Alexa [6]. During the experiment, we did not observe any false positive. After we turned off the cookie policy check for relaxation, the origin check failed for 37 sites out of the 180 websites we arbitrarily surfed. The reason why the origin check failed for those 37 sites was that those websites were sending requests to their sub-domains and changing the HTTP addresses dynamically. However, our origin check passed for all those web sites when the cookie policy check was enabled.

We also measured the performance overhead of ClickGuard, incurred from intercepting at two types of events, JavaScript events (such as onClick, onMouseover, etc), and the http-on-modify-request notification event generated by Mozilla Firefox. We measured the wall clock time of a typical browsing session. The average overhead was around 3ms per session. Our solution does not cause noticeable slowdown to the interactivity of web applications.

6. RELATED WORK

6.1 POP-UP WINDOW BLOCKING

Using user clicks as indicators of user intentions, web browsers implement pop-up blockers to suppress unwanted pop-up windows. Pop-up blockers block pop-up windows that are created during page loading by JavaScript functions such as

- window.showHelp,
- window.showModalDialog,
- window.showModelessDialog,
- window.external.NavigateAndFind [2], [3], etc.

Web browsers block calls to window.open() if one of the following conditions is met: 1) global scripts executed during document loading request to open popup windows; 2) scripts executed as part of an onload event handler ask to open pop-up windows, or 3) scripts executed in setTimeout() or setInterval() try to open pop-up windows [3].

6.2 CLICKJACKING DEFENSE

ClearClick, the module in NoScript [24], protects users against Clickjacking. It performs same origin check between the URL of the web page loaded in the iFrame and the URL of the top-level document.

With ClearClick, NoScript is a good prevention for Clickjacking attacks. However, it does not provide a general solution for other attacks discussed in this paper, such as click redirection. Instead, users are expected to specify for each domain whether to allow JavaScript, which lacks the fine granularity to selectively allow/deny specific type or source of JavaScript code.

ClickIDS [7] introduces a solution to automatically detect Clickjacking attacks. ClickIDS registers for the onClick event listener, and retrieves the coordinates of the mouse pointer when clicks happen. Then it searches for objects at the same position. If there exists more than one object under the mouse pointer, ClickIDS generates an alarm. ClickIDS detects overlay-based attacks effectively. Compared to ClickIDS, our approach focuses more on studying the effect of user intentions on a broader class of attacks.

To defend Clickjacking, a web application can use a technique called “framekiller” to prevent itself from being loaded in an iFrame [37]. One piece of example code is shown below in Fig.7:

```
if ((top.location != self.location))
{
top.location = self.location.href;
}
```

Fig.7. Framekiller Example Code

On the server side, web application developers can protect their users against Clickjacking attacks by including similar framekiller JavaScript code in those web pages they do not want to be embedded inside frames by others web sites. However, this solution suffers from obvious disadvantages. First, it affects the functionality of web sites that utilize overlays or frames. Second, it is limited to Clickjacking attack, and does not prevent click redirection attacks. Third, it only works when JavaScript is not disabled by users.

Another direction to prevent Clickjacking attacks is to enhance web browsers. Microsoft released a Clickjacking prevention solution in Internet Explorer 8 (IE8), which detects and prevents overlays or frame-based attacks [8][12][16] [22][30]. IE8 introduces a new HTTP response header X-Frame-Options, which can be set to either Deny or SameOrigin [17], [38] by web sites. If it is set to Deny then IE8 prevents pages of that web site from being embedded into frames. If

X-Frame-Options is set to SameOrigin, then IE8 will prevent pages of that web site from being embedded into frames in web pages from a different origin. The solution implemented in IE8 only mitigates overlay or frame-based attacks, and requires an extra header in HTTP responses.

Content Security Policy (CSP) [11] [28][29][32] is another mechanism intending to mitigate web application vulnerabilities, but its primary focus is CrossSite Scripting (XSS). To mitigate Clickjacking attacks, CSP enables the site to specify which sources are valid for Frame and iFrame elements. It maintains a frame-ancestor list which indicates valid sources for Frame and iFrame tags. However, CSP addresses only overlay Clickjacking attacks. It cannot prevent click redirection attacks. In addition, each web site needs to maintain a frame-ancestor list in order to activate Clickjacking protection, whose size may grow rapidly as the number of sites allowed increases. Shah et al. [26] reported a large-scale measurement analysis of mobile browser SSL security warnings. The work reported inconsistency in modern mobile web browsers in implementing SSL security warnings. Other researcher efforts [40], [41], [42] proposed solutions to mitigate clickjacking attacks and identified limitations of web browsers in preventing clikcjacking attacks.

6.3 USER INTENTION INFERENCE

In the operating system environment, Cui et al. [10] propose an approach to detect malicious behaviors that are not intended by users. It correlates user inputs to outbound network connections by the time delay in between. Compared to this approach, the user intention inference in our approach is more detailed. Besides knowing the existence of a user action, we further find out the detailed target of the user action. Moreover, the correlation between output events and input events is more accurate in our approach because we use the dependency information inside browser internals. Shirley et al. [31] introduce a new approach to access control policies by incorporating user behaviors. They explore possible policies and their effectiveness in malware mitigation, and propose a mechanism to capture user actions and try to map those actions to user intentions.

7. LIMITATION AND FUTURE WORK

7.1 BROWSER EXTENSION VS. BROWSER INSTRUMENTATION

Our ClickGuard prototype is an extension to the Mozilla Firefox web browser, which facilitates convenient deployment and distribution. However, malicious Firefox extensions can affect the ClickGuard prototype. Therefore, ClickGuard assumes the browser environment (including extensions) is not affected by malicious programs. If implemented via browser modification, ClickGuard will be immune to threats from malicious extensions.

Moreover, as previously mentioned, the current prototype of ClickGuard cannot handle indirectly generated HTTP requests by JavaScript functions like setTimeout() and setInterval(). Default settings in web browsers disable the opening of a pop-up window by built-in JavaScript functions such as setTimeout() or setInterval(), but HTTP requests from these functions are still allowed.

The reason why our prototype lacks support in these scenarios is that the current Firefox extension interfaces [9] do not provide notifications for the setting of timeouts and intervals. These limitations are solved in our ongoing work through browser modification. Through timer object reference inside the Firefox browser, we can correlate the JavaScript code activated by a timer with the JavaScript code that sets the timer.

7.2 OBSTRUCTED UIS

Although ClickGuard is preventing overlay attacks, attackers can embed legitimate page inside an iFrame with opacity value set to 1, and obstruct part of the page using other objects. This may change the appearance of the victim application, such as swapping the labels of the "Yes" and "No" radio buttons, leading to incorrect actions by users.

This is a limitation of the current approach of ClickGuard. Although obstructed victim pages can be identified by users familiar with the interface, this remains a potential issue for first visits to websites. Extend the user intention to cover such attacks will be part of our future work.

8. CONCLUSION

Clickjacking attack encompasses multiple techniques to trick web users into clicking on web elements that lead to harmful actions. It is an example of a broader range of attacks of hijacking click events. This paper studied the user intentions to detect click event hijacking attacks. Based on intercepting browser events, the proposed approach detects the mismatch between user intentions and browser actions. For events related to user actions, the proposed approach infers the associated users' intentions; for events related to suspicious browser behavior, the proposed approach finds the corresponding user action and matches the browser behavior with the user's original intention. If they do not match, it reports an alert to the user. We prototyped the proposed approach in an extension to Mozilla Firefox, called ClickGuard, which generated promising results in our experiment.

REFERENCES

[1] Javascript Events, Available at: http://www.w3schools.com/js/js_events.asp

[2] Pop-up Blocker Settings, Available at: https://support.mozilla.org/en-US/kb/pop-blocker-settings-exceptions-troubleshooting.

[3] Pop-up window controls, Available at: https://addons.mozilla.org/en-US/firefox/addon/pop-up-control/

[4] Document Object Model, Available at: http://www.w3schools.com/xml/dom_intro.asp.

[5] Spider Monkey Internals, Available at: https://developer.mozilla.org/En/SpiderMonkey/Internals.

[6] Alexa Top sites, Available at:http://www.alexa.com/topsites.

[7] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti and Chrisopher Kruegel, "A Solution for the Automated Detection of Clickjacking Attacks", *Proceedings of 5th ACM Symposium on Information Computer and Communication Security* , pp. 135-144, 2010.

[8] Bugzilla, Available at: https://bugzilla.mozilla.org/show_bug.cgi?id=475530.

[9] Observer Notifications, Avaialble at: https://developer.mozilla.org/en/docs/Observer_Notifications.

[10] Weidong Cui, Randy H. Katz, and Wai Tian Tan, "Design and Implementation of an Extrusion-based Break-In Detector for Personal Computers", *Proceedings of 21st Conference on Computer Security Applications*, pp. 1-10, 2005.

[11] Dnyaneshwar K. Patil and Kailas Patil, "Client-Side Automated Sanitizer for Cross-Site Scripting Vulnerabilities", *International Journal of Computer Applications*, Vol. 121, No. 20, pp. 1-7, 2015.

[12] Xinshu Dong, Kailas Patil, Jian Mao and Zhenkai Liang, "A Comprehensive Client-Side Behavior Model for Diagnosing Attacks in Ajax Applications", *Proceedings of 18th International Conference on Engineering of Complex Computer Systems*, pp. 177-187, 2013.

[13] XSS (Cross Site Scripting) Cheat Sheet Calculator, Available at: http://ha.ckers.org/xsscalc.html.

[14] Clickjacking, Available at: http://www.sectheory.com/clickjacking.htm, Accessed on 2008.

[15] Facebook Hit by Clickjacking Attack, Available at: http://www.darkreading.com/attacks-breaches/facebook-hit-by-clickjacking-attack/d/d-id/1132670?

[16] IE8 Security Part VII: Clickjacking Defenses, Available at: http://blogs.msdn.com/ie/archive/2009/01/27/ ie8-security-part-vii-clickjacking-defenses.aspx.

[17] Collin Jackson, Andrew Bortz, Dan Boneh, and John C.Mitchell, "Protecting Browser State from Web Privacy

Attacks", *Proceedings of 15th ACM Conference on World Wide Web*, pp. 737-744, 2006.

[18] Http State Management Mechanism, Available at: http://www.ietf.org/rfc/rfc2109.txt.

[19] About the Pop-Up Blocker, Available at: http://msdn.microsoft.com/enus/library/ms537632(VS.85).aspx.

[20] Mozilla. Event.Addeventlistener, Available at: https://developer.mozilla.org/en/DOM/element.addEventListener.

[21] Mozilla. Extensions, Available at: https://developer.mozilla.org/En/Extensions, Accessed on 2009.

[22] What's new in Internet Explorer 8, Available at: http://msdn.microsoft.com/en-us/library/cc288472.aspx, Accessed on 2009.

[23] The Clickjacking Meets Xss: A State of Art, Available at: https://www.exploit-db.com/papers/12987/, Accessed on 2008.

[24] No Script, Available at: http://noscript.net, Accessed on 2009.

[25] Dnyaneshwar K Patil and Kailas Patil, "Automated Client-Side Sanitizer for Code Injection Attacks", *International Journal of Information Technology and Computer Science*, Vol. 8, No. 4, pp. 86-95, 2016.

[26] Ronak Shah and Kailas Patil. "Evaluating Effectiveness of Mobile Browser Security Warnings", *ICTACT Journal of Communication Technology*, Vol. 7, No. 3, pp. 1373-1378, 2016.

[27] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang and Xuxian Jiang, "Towards Fine-Grained Access Control in Javascript Contexts", *Proceedings of the 31st International Conference on Distributed Computing Systems*, pp. 720-729, 2011.

[28] Kailas Patil and Braun Frederik, "A Measurement Study of the Content Security Policy on Real-World Applications", *International Journal of Network Security*, Vol. 18, No. 2, pp. 383-392, 2016.

[29] Kailas Patil, Tanvi Vyas, Frederik Braun, Mark Goodwin and Zhenkai Liang, "Poster: UserCSP-User Specified Content Security Policies", *Proceedings of Symposium on Usable Privacy and Security*, pp. 1-2, 2013.

[30] Security and the Net. About IE8's Clickjacking Protection, Available at: http://securityandthe.net/2009/02/ 01/about-ie8s-clickjacking-protection/, Accessed on 2009.

[31] Jeffrey Shirley and David Evans, "The User is not the Enemy: Fighting Malware by Tracking User Intentions", *Proceedings of New Security Paradigms Workshop*, pp. 1-13, 2008.

[32] Content Security Policy, Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[33] Transparency: the 'Opacity' Property, Available at: http://www.w3.org/TR/css3-color/#transparency.

[34] W3C Recommendation, Available at: http://www.w3.org/TR/CSS21/visuren.html#propdef-z-index.

[35] Cross-Site Scripting, Available at: http://en.wikipedia.org/wiki/Cross-site scripting.

[36] Document Object Model, Available at: http://en.wikipedia.org/wiki/Document\ Object\ Model.

[37] Framekiller, Available at: http://en.wikipedia.org/wiki/Framekiller.

[38] Same Origin Policy, Available at: http://en.wikipedia.org/wiki/.

[39] Facebook Hit with Clickjacking Attack, http://www.darkreading.com/attacks-and-breaches/facebook-hit-with-clickjacking-attack/d/d-id/1089957.

[40] Sebastian Lekies et al., "On the Fragility and Limitations of Current Browser-Provided Clickjacking Protection Schemes", *Proceedings of 6th Workshop on Offensive Technologies*, pp. 1-6, 2012.

[41] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schechter, and Collin Jackson, "Clickjacking: Attacks and Defenses", *Proceedings of 21st Usenix Conference on Security Symposium*, pp. 1-16, 2012.

[42] J.A. Shamsi, S. Hameed, W. Rahman, F. Zuberi, K. Altaf and A. Amjad, "Clicksafe: Providing Security against Clickjacking Attacks", *Proceedings of 15th International Symposium on High-Assurance Systems Engineering*, pp. 206-210, 2014.