Mediums: Visual Integrity Preserving Framework*

Tongbo Luo, Xing Jin, and Wenliang Du
Dept. of Electrical Engineering & Computer Science, Syracuse University
Syracuse, New York, USA

ABSTRACT

The UI redressing attack and its variations have spread across several platforms, from web browsers to mobile systems. We study the fundamental problem underneath such attacks, and formulate a generic model called the *container threat model*. We believe that the attacks are caused by the system's failure to preserve visual integrity. From this angle, we study the existing countermeasures and propose a generic approach, Mediums framework, to develop a *Trusted Display Base* (TDB) to address this type of problems. We use the side channel to convey the lost visual information to users. From the access control perspective, we use the dynamic binding policy model to allow the server to enforce different restrictions based on different client-side scenarios.

Categories and Subject Descriptors

Security and Privacy [**Systems security**]: [Browser Security]

Keywords

Visual Integrity, Touchjacking, Web Container Model

1. INTRODUCTION

On May 31 2010, hundreds of thousands of Facebook users have fallen for a social-engineering trick which allowed a clickjacking worm to spread quickly over Facebook during that holiday weekend. The trick, which uses a clickjacking exploit, means that visiting users are tricked into "LIKING" a page without necessarily realizing that they are recommending it to all of their Facebook friends.

The phenomenon of such a proliferation of attacks without proper protections is hard to understand. Since the first bug report on the negative usage of iframe by [16], Clickjacking attacks with various forms have been proposed. They take advantage of transparent iframes. The similar technique has also been extended to the mobile platforms [11]. Although many countermeasures have been proposed to deal with this type of problems [1, 4, 8], we are more interested in knowing what fundamental flaw has caused such attacks, so we can develop countermeasures that directly address the fundamental flaw.

We believe that all the attacks discussed above are caused by the system's failure to preserve visual integrity, i.e., to ensure what users perceive is the same as what the system "see", so users' actions are based on the correct interpretation of the information presented to them. Based on this understanding, we further believe that an ideal solution is to preserve the visual integrity. This can be done using the techniques related to visualization. We will discuss this approach in the paper. However, we also point out that this ideal solution is not completely feasible, due to the limitation of the current technologies related to visualization. Therefore, it is essential for the system to identify whether the visual integrity is in danger; if it is, the system should restrict the access.

This is basically an access control problem. Namely, a system should adopt a good access control model to deal with the visual integrity problem. Much of the existing work selects an ad hoc access control to address a particular attack, making them inapplicable to the other variations. We believe that a generic approach should be adopted, so we can develop access control systems that can address the attacks at the fundamental level. Although we may not be able to develop one access control system that fits all the protection needs, the approach that we take should be applicable to all the variations of the visual integrity attacks.

The contribution of our work is the following:

1. Our paper is the first work to fomulate the container threat model, and try to deal with this attack vector on mobile device.

2. Our work is the first to use side channels to convey the lost visual information to users on mobile device. We have also developed a novel dynamic binding policy model to defeat the attacks on visual integrity on mobile device.

3. We have implemented our solutions in Android on WebView container, and our evaluation results are quite encouraging.

*The project was supported by the Google Research Award and the NSF Award No. 1017771.

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

2. MOTIVATION

In this section, we briefly review an attack vector called *Visual-hijacking* which is caused by the compromise of visual integrity. Visual-hijacking is a set of attacks that uses various visual techniques to trick users into unwittingly clicking on disguised user-interface (UI) elements on the screen, usually resulting in damage to the victim. We formulate the visual integrity problem into a common attack model named *Container-based Visual Attack Model*. We treat all the variations of the visual-hijacking attacks equally in this paper when we explain our solutions.

2.1 Existing Attacks Using Iframe

The most famous attack caused by the compromise of visual integrity was introduced by Robert Hansen and Jeremiah Grossman in 2008. The technique is called **Clickjacking** [3], which takes advantage of the CSS design specification "opacity". The attack uses multiple transparent or opaque layers to trick a user into clicking on a button or link in a page, while the user's actual intention is to click on a different page. Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of stylesheets, iframes, and text boxes, a user can be led to believe that they are typing in the password field on the pages associated with their email or bank accounts, but instead, they are typing in an invisible frame controlled by the attacker.

However, it is not always necessary to make elements invisible to compromise the visual integrity of a page. The **UI redressing** [15] attack is an example. The main idea of the UI redressing attack is to seamlessly merge two or more webpages, making them look like one, tricking users into perform an action that is different from the users' intentions. This user interface (UI) redressing method is especially useful when there are buttons with nonspecific text like "Download", "Click here" or "Exit". Another variant of clickjacking is to use JavaScript to make a small transparent iframe to follow the mouse cursor. For this attack, it is not important where a user moves his mouse, the click will always occur in the invisible iframe.

Many proof-of-concept attacks based on the clickjacking techniques have also been published. *Facebook Likejacking* [19] uses visible Facebook Like buttons to redress the contents, and thus tricks users of a website into posting a Facebook status update for a site that they did not necessarily like. *Twitter Tweetbomb* [12] uses the same technique to attack the Twitter network. Combining the invisible element technique with HTML5 File API, *Filejacking* [7] uses the invisible technique to get the user's uploaded private files. Flash Settings are also another victim to Clickjacking.

2.2 Existing Attacks on WebView

Similar attacks have been extended to the mobile platforms. Using iframes, the attacks on the mobile platforms are similar to those on the desktop platforms. However, on the mobile platform, similar attacks can be launched without iframes. **TapJacking** [18] is an example, and it occurs when a malicious application displays a fake user interface (via an Android component called Toast), to hide the real interface underneath. When users interact with this interface, the interaction events actually go to the real interface underneath (e.g. a phone dialer). Using this technique, an attacker can potentially trick a user into making purchases, making expensive phone calls, clicking on ads, granting permissions, or even deleting data from the phone.

Confused deputy attacks can also be applied to another type of web container, the **WebView**. The WebView technology packages the basic functionalities of browsers into a class. Similar to iframe, which allows one web application to be embeded in another potentially untrusted web application, WebView allows a web application to be embedded in a potentially untrusted Android application. For most cases, the owner of the Android application is not the same as the owner of the web application inside WebView. Technologies similar to WebView are adopted by various mobile platforms, including iOS and Windows Phone, although the corresponding classes are called different names. For the sake of simplicity, we only use the term WebView throughout this paper.

Attack the **Touchjacking** [11], which can be launched successfully to redirect user's touch-screen event to the target WebView, triggering actions on the web page inside the targeted WebView. Similar to clickjacking attacks, the attacker can develop a malicious Android application with multiple WebView instances embedded. The attacker puts a visible or invisible WebView instance above another instance to redress the webpage inside WebView, and redirects user's touch screen events. The attack works on all popular mobile platforms, including iOS, Android, and Windows Phone.

2.3 Miscellaneous Attacks

According to the security blogger [6], a new technique called **Cursorjacking** was demonstrated. It deceives users by using a custom cursor image, where the pointer was displayed with an offset, so the displayed cursor was shifted to the right from the actual mouse position. With clever positioning of page elements, attackers can direct user's clicks to the desired elements. Since our work only focuses on the mobile devices, Cursorjacking is not in our scope.

3. CONTAINER THREAT MODEL

We use a generic model called the *web container threat model* to model the attacks on visual integrity across different platforms. All the attacks described in the previous section take place under a similar scenario: The victim application is embedded in another application via the components provided by the system. These components are the essential part that makes the attack successful. We use the term **Container** to refer to these components in this paper. The application that holds the container is called *host* application, and the application loaded into the container is called *guest* application. For example, in the `iframe` container case, the main page is the host, and the pages loaded into the iframes are the guest. In the Android `WebView` container case, the Android application is the host, and the web page inside WebView is the guest.

It should be noted that for the iframe and WebView containers, even though the attacker has all the privileges of the host app [5], the integrity of the data in the containers is still preserved, because of the sandbox access control mechanisms provided by these system components. The users' credentials of the guest webpage will be stored inside the container, which is a part of the system (browser or mobile system). Namely, the host application cannot directly tamper with the contents in their containers. Although WebView does provide mechanisms in its current design to al-

low the host to tamper with the data in the container [10], those channels will soon be secured in future versions. The attacker only has the access to the UI-based APIs of the container for the layout purpose. Those APIs are designed for the general view-based UI objects in the system.

3.1 Weaken of Trusted Display Base

As we all know, security in any system must be built upon a solid Trusted Computing Base (TCB), and web security is no exception. Web applications rely on several TCB components to achieve security. In the container threat model, a secure container must serve as the TCB to allow web pages to be embedded in a untrusted host without compromising the data integrity. To achieve this goal, a well-designed container needs to enforce access control on exposed APIs that allow the host to interact with the container.

However, there is no access control enforced on the UI-based APIs exposed by the container. Through these APIs, the malicious host app can manipulate the display properties of the container and its inside contents. For example, the host application can set the position and size of the container; the alpha value of the contents in the container can also be decided by the host. Without access control on these UI-based APIs, there is no trusted computing base to ensure visual security. We call this kind of trusted computing base the **Trusted Display Base**(TDB). We will discuss why the weaken of TDB can lead to the compromise of visual integrity, and eventually lead to security breaches. We will explain how our Midiums framework rebuild solid TDB on container as well.

4. REBUILD TRUSTED DISPLAY BASE

As we discussed in the previous section, the weaken of Trusted Display Base (TDB) is due to the lack of access control on the UI-based APIs exposed by the container. As the result, visual information is lost and the visual integrity is compromised.

4.1 The Mediums Framework

In order to rebuild TDB, we propose a generic solution, the Mediums framework, to defend against the attacks on visual integrity. The Mediums framework consists of two solutions. In the first solution, we use side channels to convey the lost visual information to users. In the second solution, we know that sometimes the lost visual information cannot be completely conveyed to users, so we developed an enhanced access control model to complement the side channel solution.

Three key components in the design of Mediums framework are: Environment Monitor, Side-Channel Notifier and Dynamic Binding Engine. `Environment Monitor` is the module to intercept each UI event performed by the user before it reaches the rendering engine. This monitor analyzes the potential visual information lost at the place the UI event happened and returns the level of dangerous to the framework. Once Mediums framework receives the signal from Environment Monitor, it triggers `Side-Channel Notifier` and `Dynamic Binding Engine` to minimize the impact by notifying user the dangerous of visual integrity compromise through side-channels or dynamically binding the access control policy defined by the server. We will explain why these two approaches can successfully rebuild TDB later.

It is important to notice that Mediums focues on attacks under Web Container Threat Model. Mediums does not target on any specific container but is a more generic solution to deal with how to rebuild TDB to preserve the visual integrity of the webpage in the container. Only `EnvironmentMonitor` depends on the UI architecture of the platform (i.e. Android UI module in WebView case and browser rendering engine in iframe case). However, the design of `Side-ChannelNotifier` and `DynamicBindingEngine` is platform independent. Therefore, although we only implement and evaluate Medium framework for WebView case, it can also be applied to iframe case without changing the design.

4.2 Visualization Enhancement

As we just said, the fundamental problem that causes the compromise of visual integrity is the loss of visual information when the system conveys its information to the user. Therefore, the best solution is to enhance the communication channel to reduce the information loss, so what users learn is identical to what the system knows and Mediums framework builds the TDB.

Several solutions were proposed to permanently or temporarily disable the visualization features to prevent the information loss. For example, the *X-Frame-Option* HTTP header allows the guest web app to prevent the container from being invisible. However, those solutions solve the problem at the cost of user experience. Instead of banning these features, we propose to use side channels to make up for the lost information. We will describe some of the side channels that are suitable for this goal.

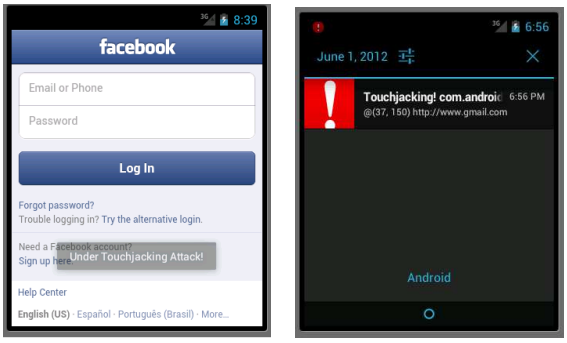
4.2.1 Mobile Device Sensors

Some side channels used by desktops/laptops may not be available for mobile devices. For example, there is no cursor on the screen for most mobile systems. However, most mobile devices have embedded sensors, such as accelerometer or vibrator; they can be used as side channels. In our implementation, we have chosen the **vibrator**, **speaker** and **flashlight** as our side changes. For example, when the user touches a display area that has overlapping WebViews, the system will vibrate the device; if the user touches on a transparent overlaid area, the device will beep. Those three types of sensors are only for the proof-of-concept purposes, and they can be extended to other types of sensors.

4.2.2 System UI

We can use the unique display features of mobile system as side channels. **Toast** mechanism in Android can be used: a toast notification is a message that pops up on the surface of the window; it only fills the amount of space required for the message and the user's current activity remains visible and interactive. The notification automatically fades in and out, and does not accept interaction events. If the Mediums framework detects that the user's current touch event is in an area with potential information loss, a toast message shown in Figure 1(a) will pop up. The status bar is another choice for side channels. An application can add an icon (with an optional message) to the system's **status bar**, which is normally located at the top of the screen. The color and content of the icon will alert users about a potential visual information lost. Users can read more details about the lost visual information by clicking on the status bar. Compared to the toast and sensor approach, the notification message is more persistent and stays there much longer

(see Figure 1(b)). It is also important to notice that those system UIs are triggered by the Mediums framework so that it is impossible for the attackers to block this side-channel.



(a) Toast Channel (b) Status Bar Channel

Figure 1: Side Channel on Mobile Devices

4.2.3 Security Concerns

There are several attacks that can be launched against our Mediums framework. First, attacker can intercept user's events before the Mediums framework gets the event. Malicious host applications can intercept the user events through the hook APIs exposed by the container. For example, by invoking the method *setOnKeyListener* of WebView, Android applications can register an event handler callback function, which will be triggered when a key is pressed in this WebView. To defeat this attack, we enforce the access control before the event hits the hook, guaranteeing that the monitor cannot be bypassed.

Moreover, to minimize the impact of unintended UI event, Mediums framework records whether users have performed click on each WebView instance or not. If framework detects that it is the first time for the user to click on the WebView instance with potential visual information lost, Mediums will discard this UI event and trigger the side-channel notifier to alert user. Users do not have to confirm that they indeed want to complete the action. If user believes the visual information lost is by design and wants to perform click on it, they just need to repeat the same action on the WebView isntance again. Mediums framework would not trigger the side-channel notifier since it has already saved user's choice on this WebView instance.

Second, attackers can attempt to block our side channels. Mediums framework will detect whether current side-channel is disabled by user or attacker, and automatically switch to other side-channel to notify user. For example, if the users turn off the speaker or ringtone, the framework needs to switch to the toast or status bar as the side channel which cannot be blocked by the attacker.

4.3 Dynamic Binding Framework

Even if we can use side channels to convey the lost dimension, users may still ignore them, as they are indeed different from an actual dimension. In these cases, a good system should be more intelligent in deciding whether it should allow users to conduct certain actions or not. Although a number of solutions have been proposed [8, 13], they seem to depend on ad hoc policies that can solve one type of problems, but it is difficult to be applied to other similar problems. The reason is that these solutions were not developed from the *access control angle*. Our framework allows us to treat the visual integrity problem as an access control problem, and can thus lead to a more generic solution.

Policy models decide when a click or touch action should be allowed or denied. Ideally, if the visual integrity is more likely to be compromised, the control on the access should be more restricted. There are two types of policy models: static binding model and dynamic binding model.

4.3.1 Static Binding Policy Model

In the static binding policy model, the access control policy is set when server constructs the webpage. The policy can be set and enforced by the client side or the server side, but they both suffer from the reliability and accuracy issues.

In the `client only` model, the client side sets and enforces the access control policy. Several work took this approach [13]. From the policy's reliability perspective, since the enforcer is at the client side, which is the place where all the access actions take place, by gathering all the environment information at the moment when the action happens, the enforcer can effectively enforce the access control policy. However, from the accuracy perspective, when the client sets the policy without the support from the server, they cannot take the contents in the container into consideration. This may lead to the granularity problem and affect the accuracy of the policies.

In the `server only` model, the policy is set and enforced by the web server. The widely adopted solution Framebuster [8] took this approach. Two major barriers make this access control policy either inaccurate or unreliable. Due to the lack of real-time environment information at the client side, when the server sets the policy, it is hard to predict the visualization environment when the user's action takes place. For the reliability issue, since the action happens at the client side, without the support from the specifically designed client framework, the servers do not have sufficient information to set the correct policy; this will reduce the reliability [17].

4.3.2 Dynamic Binding Policy Model

We propose to use a dynamic binding policy framework to solve the above problems. In this model, the server sets different access control policies for different client-side conditions. Although none of the existing works formally defined this policy model, some of the existing solutions, such as *X-Frame-Options* [14], take this approach. With the support of the browser that recognizes this new HTTP header, the web server can decide whether its pages can be loaded into the iframe or not. The recent project [4] proposed to allow the web application to use Sensitive-UI to mark the objects that do not want to be overlapped.

The limitations of those solutions are the following: The X-Frame-Options solution only deals with one situation, i.e., whether the page is loaded in the container or not. The Sensitive-UI solution only supports one action no matter what the client-side situation was. Moreover, the client-side environment can be dynamically changed. It is highly possible that the container does not overlap with others when the page was first loaded, but it overlaps with others when user performs click actions later. Since the server cannot predict the client-side situation when the access takes place, this framework should allow the web developers to define

policies that depend on the runtime conditions on the client side. Therefore, to support more accurate finer-grained access control policy in this model, we propose to use the *dynamic binding* framework.

Dynamic binding framework pre-defined several client-side scenarios that may cause visual information loss, and for each scenario, it sets actions to alleviate the loss. The web developers can associate the policy to the whole webpage or certain DOM objects based on the contents of the webpage. For example we can integrate the Contego [9] model to Mediums framework to enable the web developers to assign subset of privileges to specific DOM element of the webpage.

```
if (Senarios #1)        Allow Privilege Subset 1
else if (Senarios #2)   Allow Privilege Subset 2
else if (Senarios #3)   Allow no Privilege
if (Senarios #4)        Deny Privilege Subset 4
```

In WebView case, a concrete sample case is given in the following:

```
if (not in a WebView)
      Allow {Clickable, Attach-Cookie}
else if (embedded in a overlapping WebView)
      Allow {Clickable}
else if (embedded in an invisible WebView)
      Allow {}
```

5. IMPLEMENTATION

We have implementd the visualization enhancement using the side channel solution and the dynamic binding solution for the Android system (version 4.0.3). Figure 2 demonstrates the high-level architecture of our implementation.

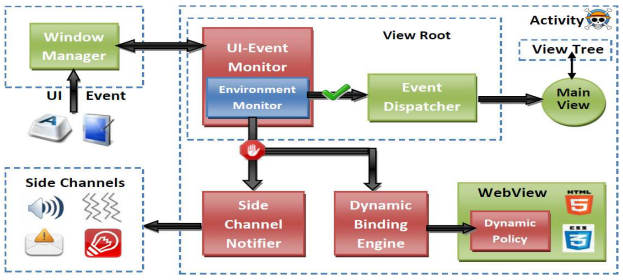


Figure 2: Mediums Framework Overview

The `UI-Event Monitor` located in the `RootView` object of each application intercepts every touch action performed by users, and invokes the `Environment Monitor`, which traverses the view tree of the application to detect whether there is a potential visual information loss or not. If there is a potential danger at the place where the touch action occurs, and the user has not been notified enough times, the framework will discard the event and trigger the protection mechanism to notify the users through side channels. Otherwise, the event will be dispatched to the target UI object.

5.1 UI-Event Monitor

The primary goal of the UI-event monitor is to intercept each UI event in the system and check the potential visual information loss before the event affects the application. To achieve this goal, The UI-event monitor needs to be placed in the event dispatching path, before the event reaches the application. Due to the page limitation, details of how the Android event handling mechanism works are not included in this paper; they can be found in the extended version of the paper.

5.2 Environment Monitor

The Environment Monitor is a module in the ViewRoot class to measure the danger level for the possible visual information loss. The module needs to extract the coordinate of the touch event from the event object, and traverses the view tree to find out all the views that contain this coordinate. Based on the predefined danger standard, the Environment Monitor will return the alert level. In our current implementation, we define the safe situation as the alert level 0; when a visible WebView instance overlaps with another UI object, the alert level is 1; when an invisible WebView instance is present but without overlapping with others, the alert level is 2; when an invisible WebView instance overlaps with others, the alert level is 3. The higher the alert level is, the more dangerous it is when the visual information is lost.

5.3 Side Channel Notifier

Once the UI-event monitor detects the potential visual information loss, it will check whether the user has been notified for a pre-defined number of times. If so, i.e., the user has been informed enough times, the notifier will not be triggered and the event will be dispatched. This means that the user has decided to accept the potential risk, and there is no need to continue "anonying" the user. Otherwise, side channel notification will be triggered. In our prototype, the alert level 1 will trigger the Vibrator; the alert level 2 will trigger the Vibrator and a Toast message; and the alert level 3 will trigger the Vibrator, a Toast message, and System Alert Bar.

5.4 Dynamic Binding Engine

The Dynamic Binding Engine will be triggered to dynamically bind the access control policy defined by the web application inside WebView. To use the Mediums prototype, web developers embed the dynamic policy in the HTTP headers and send to the WebView along with the webpage contents. In order to recognize the new dynamic binding policy header (i.e. the *DBPolicy* field), we need to modify the parser module to extract the value of DBPolicy field, and return the policy information to the WebView instance. WebView uses the WebKit rendering engine to parse and display web pages, and it is implemented as a native C++ library (*WebCore.so*). The class *WebUrlLoaderClient* in the WebKit library will fetch the response from the network driver; it then invokes the hook *didReceiveResponse*, and the code registered to the hook will begin parse the whole response. The Dynamic Binding Engine implements the code in this hook to retrieve the policy in the DBPolicy field.

Since policies are retrieved by WebKit, we need to find a way to return it to the WebView which is a Java class. The WebView Java package uses *BrowserFrame* class to represent a frame of a page, and WebKit library uses *WebFrame* class to represent the same concept. These two classes are binded togather through the JNI mechanism in Android. Therefore, the WebKit library can invoke the callback functions implemented in the C++ class *WebCoreFrameBridge* to return values from the native library to the Java framework. We add a new callback function called *jniSetPolicy* for the WebKit library to return the policy to the BrowserFrame instance. BrowserFrame will invoke the setPolicy function exposed by WebView class through the *WebViewCore* or *CallbackProxy*. Figure 3 shows the process.

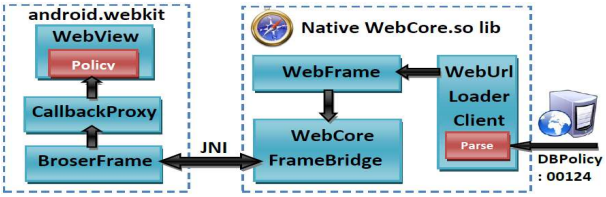


Figure 3: Dynamic Binding Engine

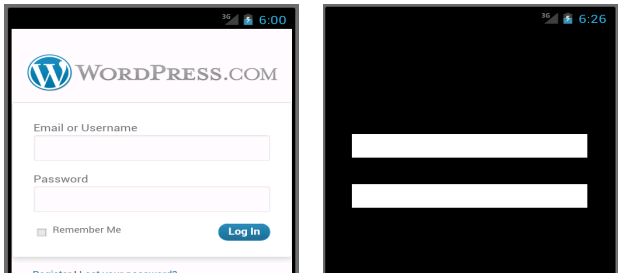
The dynamic policy should be stored at a secure place where cannot be tampered by malicious apps. We add a private field *policy* in the WebView class to store the dynamic policy set by the webpage. We also add a new *protected* methods *setPolicy* to allow the WebKit to set the dynamic policy. It is important to note that the setPolicy method is only accessible from the code within the android.webkit package in the Java framework. Therefore, malicious Android applications cannot invoke this method or directly change the value of private field *policy* in WebView class.

6. EVALUATION

We evaluated the Mediums framwork on the Android platform to demonstrate how our solution can effectively alleviate the visual hijacking attacks without sacrificing much user experience. The evaluation environment is Samsung Nexus S phone with Samsung Exynos 3110 processor, 512 MB Mobile DDR RAM and 4.0-inch screen.

6.1 Attack Scenarios

For our evaluation purpose, we wrote an Android application with various kinds of Touchjacking in it. To users, the main purpose of this application is to to conduct surveys, but behind the scene, the application tries to attack the user's online web account. We use two particular attacks, Keystroke Hijacking attack and Invisible WebView Touchjacking attack in our experiments.

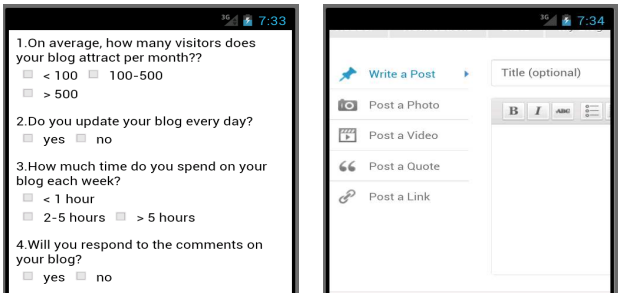


(a) What User Sees (b) UI on top

Figure 4: WebView overlapped with UI component

Figure 4 shows how was Keystroke Hijacking attack set up. In our app, we use a WebView to load the WordPress login page, and on top of it, we put two text input fields (native Android UI objects), each covering one text field on the web page. Therefore, the users see what is shown in Figure 4(a), but when they type the username and password, they actually type the information in those native UI objects (Figure 4(b)), which belong to the host Android application.

Figure 5 demonstrates how the Invisible WebView Touchjacking attack works. The WebView (Figure 5(a)) that loads a survey webpage is put underneath another transparent WebView. Figure 5(b) shows the transparent WebView (we intentionally make the picture non-transparant so readers can see it). What the users sees on the screen is a survey (Figure 5(a)), but when they select their choices, they actually click the "Write a Post" link on the transparent WordPress webpage.



(a) Visable WebView (b) Transparent WebView

Figure 5: Transparent WebView Overlapping

6.2 Evaluation of Visual Enhancement

6.2.1 *Experiment Setup*

We used two Samsung Nexus S phones to do user experience study. We installed the original system (Android 4.0.3) on one phone, and on the other phone, we installed the modified Android that has our Mediums framework. We designed two similar Android apps and both of them used the WebView component to load a survey web app in the WebView component. The topics and questions in the survey are different but share the similar layout. At certain pages in both survey apps, we overlapped both transparent WebView and visible native UI component to achieve the Touchjacking attack scenarios described above.

We randomly choose the 86 participants in different places such as library, street, restaurant and etc. The age of the participants ranges from 19 to 30. We also asked how much they knew about mobile security before the test and the results shows in the following subsections. We used survey app to distract participants' attention from our goal to test the side channel visualization. Before the test, we told participants that when they found something abnormal they can ask us, we would give some suggestions, since we did not want them to behave in a more (or less) trusting manner. Every participant was asked to finish the survey on both smartphones, and we collected participants' basic information such as sex, age, education level and major, etc. Even if the attacks were launched successfully during the evaluation, they would not cause real damage to the participant's account. We observe whether our framework can help users prevent the attacks or not.

Three major aspects can directly reflect the effectiveness of the side channel visualization solution, and we will design experiments to evaluate them. These aspects are formulated as the following questions:

- Can participants get the side channel signals generated by the Mediums Framework?
- Do participants have proper reactions to side channel signals?

Side Channel Usage	Receive Signal	Get Meaning	Perform Click	Attack Succeed
-	0	0	78	90.7%
T only	64	62	24	27.9%
V only	70	49	37	57.0%
V + T	77	75	11	12.8%
V + T + N	81	80	6	6.97%

V = Vibration Side Channel; T = Toast Side Channel;
N = Notification Bar Side Channel

Table 1: Survey Results Among 86 Participants

- Does the solution affect user experience?

6.2.2 User's Information Acquistion

In our evaluation, we used three side channels to convey the lost visual information: Vibration, Toast and Notification Bar. Among the 86 participants(Table 1), 81% participants noticed the vibration and 74% were aware of the toast. When we combined them together, 90% got the side channel signal. When we used the vibration, toast and alert bar together, the number becomes 94%. We also records the the reason why more participants miss the side-channel signals. This is because the viberation and toast only last for short period of time.

6.2.3 User's Reaction to Information

Another important factor that directly affects the success of our solution is whether the users is aware of the danger after they receive the side channel signals. The users' reactions to the signals may vary depending on their knowledge about the mobile security. After finishing two survey apps, we asked how much they knew about mobile security, such as the clickjacking and touchjacking attacks. On a scale of 1 to 5, 1 means knowing nothing and 5 means knowing much. Our results showed that the average rating was 1.76, which means most of participants know nothing or little about the clickjacking and touchjacking attacks. This way, we can test the effectiveness of our secure mechanism for people even without any knowledge. Table 1 shows the results we obtained. In the normal WebView without any Mediums framework, 8 participants chose not to click, because most of them know a lot about clickjacking and touchjacking, they thought it was not secure to perform actions on these apps, so they gave up on the survey. Among the 70 participants who noticed the vibration, only 49 (70%) chose not to click. Participants didn't connect the vibration to the potential danger because normal apps can vibrate too. Similarly, the toast approach has a lower success rate 27.9%, which is better than vibration, but some participants said that without vibration they did not notice the toast message. However, using vibration, toast and alert bar together is the most reliable way to alert users, which significantly dropped the touchjacking attack's success rate to 6.97%.

6.2.4 Usability of Solution

We also need to evaluate how the side channel signals affect user experience We also collected feedback on how annoyed the participants were when using apps in our framework. On a scale of 1 to 5, being 1 means "not at all" and being 5 means "very annoying". The average rating was 1.65, which is the acceptable level.

The overhead introduced by our framework to monitor each UI event and check environment is another factor that may affect user experience. We measured the overhead using 100 applications from the Android Market, the range of the overhead per touch event was from 0 to 6 milliseconds. The time basically comes from the view tree transversal, more precisely, it depends on the number of nodes in the view tree. The number of view objects in the applications that we tested ranges from 10 to 89.

6.3 Evaluation of Dynamic Binding

We tested the performance on the smartphones for four web applications (phpBB3, Collabtive, WordPress, and phpCalander) and shows the overhead introduced by Mediums in Figure 6. In this section, we evaluate the defense to the attacks mentioned in 6.1 by enforcing dynamic binding.

Place	Client-side Scenarios	Action Index	Actions
1st	not in WebView	0	Do Nothing
2nd	loaded in WebView	1	Remove From Screen
3rd	loaded in an overlapping WebView	2	Unclickable WebView
4th	loaded in an invisible WebView	3	Visible WebView
5th	loaded in an overlapping invisible WebView	4	Visible & Unclickable

Table 2: Mediums Scenarios and Action Definations

In order to prevent the Touchjacking, web developers set the policy header as **header("DBPolicy: 00124")** in the php file (Only 1 line of code need to be added). Each number of the DBPolicy value corresponds to one client-side situation defined by the Mediums framework. The value of each digit represents the action that needs to be taken if the client side satisfies the scenario. We use the definition in Table 2 to convert the policy to the following readable form:

```
if (not in a WebView)
      Do Nothing                                --> Take Action 0
else if (loaded in a WebView)
      Do Nothing                                --> Take Action 0
else if (loaded in an overlapping WebView)
      Set WebView Visibility to 'Gone'          --> Take Action 1
else if (loaded in an invisible WebView)
      Set WebView as Unclickable                --> Take Action 2
else if (loaded in an overlapping invisible WebView)
      Set WebView as Unclickable and Visible    --> Take Action 4
```

To defend against the keystroke hijacking attack (see Figure 4), WordPress developers can take the action to remove the WebView instance from the screen. Therefore, the 3rd digit of the DBPolicy value is set to 1. As results, if the webpage is subject to the keystroke hijacking attack, the dynamic binding engine detects the situation and enforces the dynamic policy. The WebView instance is removed from the screen, leaving only the overlapped UI objects depicted in Figure 4(b). Therefore, user can clearly know that they are under the attack and can stop.

To defend against the Invisible WebView Touchjacking attack depicted in Figure 5, developers set the 5th number of

the DBPolicy value to 4. This policy defines that if the WebView is transparent and is overlapping with other objects, WebView instance should be made unclickable and visible. Therefore, when the attack is launched, the screen will look like that in Figure 5(b), clearly showing the attack intent.

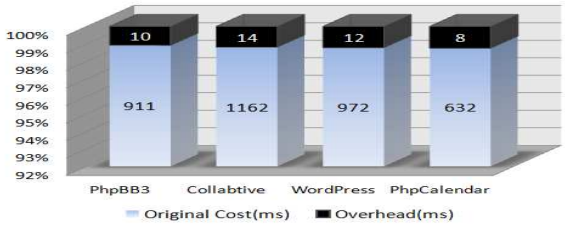


Figure 6: Dynamic Binding Performance Overhead

7. RELATED WORK

Section 2 has already described the work that is related to the attacks on visual integrity. We will not repeat them. We focus on discussing the existing solutions. We divide them into three categories.

Client-Side Solution: Some solutions purely depend on the client-side framework such as web browser. For example, by banning some particular features of the container, such as the transparent feature, web browser can alleviate the risk of the attacks. Some well-known projects include the ClearClick component in the NoScript [13] Firefox plug-in and Anti-Clickjacking component in the GuardedID project. All these solutions enhance the security by either temporarily or permanently banning features of container.

Server-Side Solution: Several solutions were proposed to modify the server-side code to defeat the attacks on visual integrity. No change to the client side is needed. One solution is to prevent web pages from being loaded into the container, and thus thwart the attacks. By embedding a piece of javascript code at the very beginning of the webpage, the webpage using Framebuster [8] can bust out from the iframe. However, this approach is not very reliable [17]. Another solution is to add unguessable secret to the URL of each web page, so the navigation can only start from certain trusted pages [2]. A third solution is to ask users to take additional actions, such as requiring the user to mark a checkbox, type in password, or solve a CAPTCHA, before clicking on the important button. These actions make it harder for clickjackers, as they now have to trick users into taking those actions. The last two solutions require significant changes on the server-side code.

Hybrid Solution: A hybrid solution is to let the server side set the policy on visual integrity, and depend on the browser to enforce the policy. Our dynamic binding approach takes a similar approach, but provides a finer granularity. We have already distinguished our work with some well-known projects in section 4.3.2.

8. SUMMARY

In this paper, we systematically study a class of UI redressing attacks, and we point out that the fundamental flaw of these attacks are the system's failure to preserve visual integrity. Based on this observation, we propose two solutions, a visualization method and a dynamic binding model. We implement our solutions in Android 4.0.3 system and our evaluation demonstrates encouraging results.

9. REFERENCES

[1] A. Chaitrali, S. Kapil, V. Arunabh, and P. Traynor. On the disparity of display security in mobile and traditional web browsers. In *SCS Technical Report*.

[2] T. Close. The confused deputy rides again! `http://waterken.sourceforge.net/clickjacking/`.

[3] R. Hansen. Clickjacking. `http://ha.ckers.org/blog/20080915/clickjacking/`.

[4] L. Huang, A., H. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, 2012.

[5] C. Jackson. *Improving browser security policies*. PhD thesis, Stanford, CA, USA, 2009. AAI3382749.

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

[7] K. Kotowicz. Filejacking: How to make a file server from your browser (with html5 of course), 2011.

[8] E. Lawrence. Ie8 security part vii: Clickjacking defenses. `http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx`.

[9] T. Luo and W. Du. Contego: capability-based access control for web browsers. In *Proceedings of the 4th international conference on Trust and trustworthy computing (TRUST 2011)*.

[10] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC 11.

[11] T. Luo, X. Jin, A. Ajai, and W. Du. Touchjacking attacks on web in android, ios, and windows phone. In *Proceedings of 5TH International Symposium on Foundations & Practice of Security (FPS 2012)*.

[12] M. Mahemoff. Explaining the "don't click" clickjacking tweetbomb. 2009.

[13] G. Maone. Hello clearclick, goodbye clickjacking! `http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/`.

[14] Mozilla Developer Network. The x-frame-options response header.

[15] M. Niemietz. Ui redressing: Attacks and countermeasures revisited. In *in CONFidence 2011*.

[16] J. Ruderman. Bug 154957 - iframe content background defaults to transparent., 2002.

[17] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*.

[18] G. Rydstedt, E.e Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization. In *in Usenix Workshop on Offensive Technologies (wOOt 2010)*.

[19] SophosLabs. Facebook worm - likejacking. 2010.