# UI Redressing Attacks on Android Devices

Marcus Niemietz, Jörg Schwenk
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{marcus.niemietz,joerg.schwenk}@rub.de

## ABSTRACT

In this paper, we describe novel high-impact user interface attacks on Android-based mobile devices, additionally focusing on showcasing the possible mitigation techniques for such attacks. We discuss which UI redressing attacks can be transferred from desktop- to mobile- browser field. Our main contribution is a demonstration of a browserless tap-jacking attack, which greatly enriches the impact of previous work on this matter. With this technique, one can perform unauthorized home screen navigation and attempt actions like (premium number) phone calls without having been granted appropriate privileges. To protect against this attack, we introduce a concept of a security layer that catches all tapjacking attempts before they can reach home screen/arbitrary applications.

## Keywords

Android, UI Redressing, Clickjacking, Tapjacking, Web Security; Browser Security; Mobile Security

## 1. INTRODUCTION

Upon Jesse Ruderman's publication of his observations concerning the behavior of transparent elements inside a Web browser in 2002 [1], the research on user interface (UI) redressing has begun. Ruderman's original idea was to load Web pages inside an iframe and perform interactions on its loaded elements without getting noticed. The latter was due to the fact that these elements remained invisible to the user. This particular potential security problem has been overlooked until 2008. At this time, Grossman and Hansen showed [2] that the Adobe Flash Player can be modified in a malicious way. More specifically, an attacker was able to automatically gain access to a camera or a microphone without any visible permission from a victim. The attack was then labeled clickjacking, as the attacker executes his ability to hijack clicks of a user in this scenario. Nowadays, *UI redressing* is a common name for the entire set of principally similar attacks and it includes clickjackingas a special case.

The early attack from 2008 is now filed under clickjacking subset and it can be referred to as classic clickjacking. Furthermore, this subclass consists of attacks like *cursorjacking, filejacking, tabnabbing*, and inter alia *tapjacking* [3]. Major commonalities shared between these concepts are related to attacker's capacity of controlling mouse clicks or similar events (for example touch gestures). In essence, all of these attacks need a Web browser to be executed. The case of UI redressing implies the same Web browser requirement. In contrast to clickjacking, UI redressing additionally gives an attacker the ability to hijack keystrokes or, alternatively, to bypass security mechanisms (such as same origin policy) implemented in a Web browser, all done via modified UI.

Considering the above given attack vectors on desktop-based Web browsers, we pose a question of whether they can be ported to smartphone-based systems. For this purpose, we focus on the Android operating system (combined with a touch display). The rationale behind this choice lays in the market share dominance of this setup, evident in late 2011 – a time when we have initiated our research. To clarify this point, let us name a study from Gartner in which the market share was over 52 % [4], making it double in size compared to last year. For the same reason, we will concentrate on the currently most widespread version 2.2 [5]. In addition, we will address Android 4.0.3 systems to underline the importance of our research for upcoming mobile devices.

In summary, this paper focuses on two points which were not raised in the academic research till now. Firstly, we will discuss currently present attacks and countermeasures for desktop-based Web browsers available for Android-based mobile devices. Next, we will show a UI redressing, precisely centering on the tapjacking attack technique which does not need a Web browser to execute. It operates in a way that a malicious application with only minimal rights proves to be enough to carry out successful attacks. This tapjacking technique allows us to make phone calls, install applications, and use the default browser for code injection attacks without any of the usually required permissions. A simple mechanism is the hijacking of some apparently harmless interactions on the victim's part. In this context, we will introduce a concept of a new security layer capable of mitigating this kind of attack in all modern Android-based operating systems.

## 2. RELATED WORK

This chapter is dedicated to an overview of the related work on desktop-based and browserless UI redressing techniques.

### 2.1 Desktop-based UI Redressing Techniques

As stated in 1, UI redressing supplies the attacker with an option to change the UI hijack events like pressing a key

or clicking with the mouse cursor. Many attacks belonging to UI redressing attacks' class have been surfacing since 2008 [6]. In an attempt to systemize them, we propose a following classification:

- Clickjacking
  - *Classic clickjacking [2].*;
  - *Likejacking [7] and sharejacking.* These are variations of classic clickjacking attack performed via Facebook's Like and Share buttons;
  - *Nested clickjacking [8].* Enables an attacker to perform attacks against social networks like Google+;
  - *Cursorjacking [9].* By changing the image of a mouse cursor one can hijack a click through a visible mouse cursor, unusually placed inside a manipulated one;
  - *Cookiejacking [10], filejacking [11].* Cookiejacking enables stealing cookies within the long-unpatched (pre-July 2011) versions of Internet Explorer while Filejacking uses a feature of Google Chrome for an unintentional upload substituting a download;
  - *Eventjacking, classjacking [6].* These attacks are based on HTML events and attributes fully controlled by an attacker;
  - *Tabnabbing [12].* It allows phishing by browser tab manipulations;
  - *Double clickjacking [13].* Unlike in classic clickjacking, two clicks to bypass typical clickjacking security mechanisms like a JavaScript-based frame buster are utilized;
  - *Combinations with CSRF, XSS, and CSS.*
- *Strokejacking.* A technique which facilitates keystrokes' hijacking.
- *Drag-and-drop operations [6].* Makes it possible for an attacker to bypass the restrictions of the same origin policy to e.g. perform invisible text injection attacks.
- *Content extraction [6].* A technique of stealing source code of a website reachable for a victim via the internet and intranet.
- *Event-recycling.* It supplies a possibility of bypassing mechanisms implemented in a Web browser, for example a pop-up blocker.
- *SVG masking [6].* Allows an attacker to show elements in another context with the help of Scalable Vector Graphics.

## 2.2 Browserless UI Redressing Attacks

For a while now, it has been quite clear that UI redressing and many other attacks related to the UI can be carried out against Web browsers executable on mobile devices. Nevertheless, a question still remains: Is it possible to perform browser-like attacks on mobile devices without using a Web browser or, at the very least, without using it directly? Carrying out an attack scenario which makes use of clickjacking provides a clear positive answer to the above inquiry.

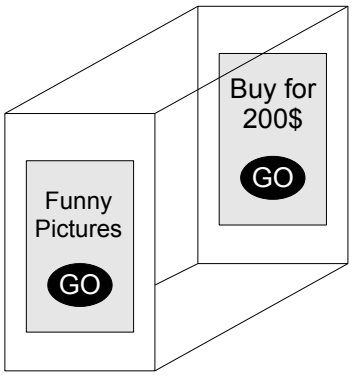


**Figure 1: The malicious application for the tapjacking attack in front of the attacked application is able to let touch gestures under the malicious application go directly to the target application.**

One of the first ideas tackling this attack scenario was published by David Richardson in 2010 [14]. He thought about how the Android trust model behaves when a dialog is opened through an application. Richardson theoretically suggested that an application was allowed to programmatically open a dialog but not to interact with it. This finding was not particularly surprising, since it is otherwise possible to do a privilege escalation attack. However, by using a *toast view*, one is able to show a quick little message to the user. A basic idea of the toast is that an application in question should be as unobtrusive as possible, while the main application is still being displayed; to give one example – this is the case of volume control.

Jack Mannino published a proof of concept of a tapjacking attack in 2011 [15]. The toast class was employed in his research. His method involved scaling of the usually small notification message to an entire display of the mobile device and subsequent usage of the default constant *LENGTH_LONG* to show the view or text notification for a long period of time. Notice, Android offers the option to use a long delay of 3.5 seconds and a short delay of 2 seconds [16]; the 3.5 seconds are not long enough and therefore the delay can be artificially extended [17] for such a kind of attack – for instance to 30 seconds.

The crucial point is that a touch gesture on such a message or notification will be passed through to the underlying application. Thus, an attacker can create a dummy button in the overlaying notification and place it over a button of the target application. In essence, it leads to the fact that the victim will make a touch gesture on the button of the attacked application without receiving any alert about what he or she has done. An attack scenario is shown in Figure 1. In Chapter 4, we extend the Manino's proof of concept attack, which was originally restricted to two target applications in the background, as we allow access to more applications and especially the home screen.

# 3. PORTING UI REDRESSING TO ANDROID DEVICES

To evaluate the impact of mobile-devices-targeted attacks listed in Chapter 2.1, we have examined the similarities between mobile and non-mobile systems in the UI redressing field.

*Classic clickjacking* usually requires a Web browser supporting frames. Furthermore, support for CSS, JavaScript, and HTML5 is needed to execute attacks like *classjacking* or *strokejacking*. This is exactly the setup witnessed in native Android Web browser at present. Most of the remaining attacks – *nested clickjacking, filejacking, tabnabbing, content extraction, event-recycling, and SVG masking* – need additional features, which can be found in desktop-based Web browsers like Firefox, Opera, or Chrome. In the past, this was a major obstacle as only native Web browser and some additional rather unknown Web browsers were available on smartphones. Nowadays, any Web browser one requires – like Firefox in case of an SVG masking – can be downloaded via Google Play.[1]

The following list enumerates the attacks which proved not transferable to Android devices during our research: cursorjacking, cookiejacking, double-clickjacking, and pop-up-blocker bypasses. Let us specify:

- *Cursorjacking.* Due to the nature of a mobile device with a touch display, there is usually no visible mouse cursor.
- *Cookiejacking.* This attack [10] does not work because Microsoft offers no Internet Explorer version for Android.
- *Double clickjacking and pop-up-blocker bypasses.* They cannot be applied for the reason that Android-based Web browsers have no native pop-up or pop-down windows, as it is the case in the desktop-based world.

All in all, there are many desktop-based attacks which can be adapted to Android-based mobile devices; more information on that matter can be found in work by Rydstedt et al. [3]. In the following, we will focus on a browserless attack related to classic clickjacking which can be carried out on Android-based mobile devices. Called tapjacking [3], this attack is part of the clickjacking subset.

# 4. NEW BROWSERLESS ATTACKS

*Tapjacking against security critical applications.* We found out that it is possible for an attacker to have access to many applications critical in terms of security considerations. Normally requiring an explicit permission from the victim, the attacks here are done by installing only one low-privileged malicious app. Probability of successful attack scenarios designed for mobile devices can be increased significantly with this tapjacking attack. In addition to the attack described by Jack Mannino, we are able to control and manipulate, among others, the following applications, regardless of having no rights to them granted:

- *Contact data manipulation.*: Violation of user privacy.
- *Native browser utilization.* Preparation step for further UI redressing attacks.
- *Touch gestures logging.* Performing attacks like keylogging.
- *Predefined phone calls.* Monetary damage by premium number calls.
- *Installing applications in the background.* Loader functionality for installing specialized malicious applications.

All of these attacks are using the same technique. Firstly, there is a visible attacker's application in form of a notification in the foreground – a code snippet from our touch gesture logger is given in Listing 1. Secondly, there is a target application in the background.

**Listing 1: In this code snippet of our touch gesture logging attack one can see that we are using at least one button with a – to clickjacking similar – opacity property for a full transparency. Beyond that we set inter alia a layout parameter to overlay all applications as well as the home screen.**

```
mButton = new Button(this);
// like the CSS opacity property
mButton.getBackground().setAlpha(0);
// needed for onTouch()
mButton.setOnTouchListener(this);
// Layout parameters with an overlay
WindowManager.LayoutParams params =
      new WindowManager.LayoutParams(
   WindowManager.LayoutParams.WRAP_CONTENT,
   WindowManager.LayoutParams.WRAP_CONTENT,
   WindowManager.LayoutParams.
      TYPE_SYSTEM_OVERLAY,
   WindowManager.LayoutParams.
      FLAG_WATCH_OUTSIDE_TOUCH,
   PixelFormat.TRANSLUCENT);
```

*Real life scenario.* By choosing the phone call attack as an example, one can construct the following real life scenario. The attacker offers an application in a form of a simple calculator as a (free) download. The victim uses Google Play to search and obtain such an application, which, due to its simplicity, has no access rights for e.g. doing phone calls and writing SMS.

Once the ill-disposed application is installed and opened, the victim performs regular actions, such as calculating a total amount of a bill. During this interaction, the malicious application displays a message with a confirmation button, placed at the required location on the screen – this can for example be a note that the victim should not press the displayed button, as we are demonstrating on the right side of our Figure 2. If the victim touches the confirmation button,

[1]Since March 2012, *Google Play* is a name used for the Android market [18].

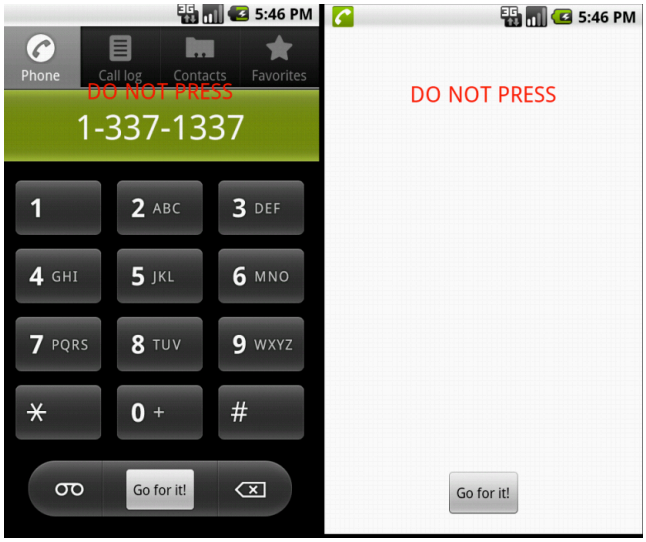


**Figure 2: On the left side, a malicious application with a transparent background is displayed as an illustration. On the right side the same application with a white background and an already initiated phone call can be observed.**

this 'tap' will be forwarded to the phone application, and a phone call with a predefined (premium) number will be initiated.

In Figure 2, a simple exemplary implementation is depicted. On the right side, the malicious application is given with a background colour set to white. Instead of this white background, a picture, a pocket calculator, or a simple game would typically be displayed in a real attack. On the left side, the background is changed to a transparent mode so that to visualize the effect of the tapjacking attack. The malicious button 'Go for it!' is placed exactly over the (normally invisible) call button of the phone call application. Thus, an attacker is able to open this particular application in the background with a predefined number and the only thing what he or she needs is a single touch gesture. Therefore, the victim is actually pressing a button situated below the malicious one.

The victim may notice that a phone call is performed because a green phone icon is displayed in the notification bar.

*Unauthorized home screen navigation.* A similar attack technique can be completed by using home screen, the rationale being that one can open and press the required buttons to initiate target applications. By using this attack technique there arises the question, which advantages are associated by tapjacking the home screen. The crucial point here is that there is a limited number of operations [19] like opening the phone call application by using *Intent.ACTION_DIAL* or displaying data to the user with *Intent.ACTION_VIEW*.

Summarized, the attack makes it possible to substantially extend the previously mentioned limited set of attacks. This works due to the reason that Android allows notifaction messages directly on the home screen when there is no application in the background and also in the case that there is an application in the background by explicitly using *Intent.ACTION_MAIN*. This operation leads to the possibility that one can perform *ACTION_MAIN* to launch the home screen, e.g. navigate in the Android menu, and open another application to change the system configuration – and thus reach an application, which is not intended to be reachable by the Android developers with a usage of *Intent.ACTION_*-operations.

Contrary to the previously mentioned attacks, a disadvantage here is that an attacker needs more touch gestures of a victim; the required touch gestures can be collected by using social engineering techniques, e.g. via a simple game which requires touch actions. Nevertheless, this attack gives us the possibility to open all vulnerable applications and it is a single point of attack for tapjacking, clearly making it very attractive target for criminal attackers.

*Responsible disclosure.* We have informed the Android team about this attack vector and are awaiting their response. Since system updates are a problem for Android smartphones and no general defense mechanisms exist in current Android systems, we have chosen not to publish the full source code of the attack. We will gladly provide it to certain parties on request.

# 5. MITIGATION TECHNIQUES

In this chapter, we focus on countermeasures against browser-based and browserless UI redressing attacks discussed in the previous sections. More importantly, we address their applicability in the context of the Android-based devices.

## 5.1 Browser-Based UI Redressing

*Frame Buster.* To be able to control the layout of the victim's website, the attacker usually loads the target Web page into an iframe. Standard countermeasures implemented in the attempts to shield a website from being framed are JavaScript-based frame busters. They usually consist of a conditional statement and a counter-action. The conditional statement verifies if the website implementing the frame buster is loaded in an iframe. In such case, the counter-action will be executed. Thus, if *attackers.org* wants to load *victim.org* in an iframe, *attackers.org* will be busted to *victim.org*.

In July 2010, Rydstedt et al. put forward a statistical inquiry into the usage of frame busters [20]. The researchers came to a conclusion that frame busters are the most widely used technique against classic clickjacking. Nonetheless, frames can be attacked in many different ways via *busting frame busting* attacks. A recommendation against classic clickjacking attack, as well as the frame busting attacks, was published by August Detlefsen in October 2010 [21]; Jason Li, Chris Schmidt, and Brendon Crawford optimized this countermeasure. This new kind of frame buster is displayed in Listing 2.

Detlefson et al. used Cascading Style Sheets to ensure that the whole body of the document will not be displayed to the user. After that, they used JavaScript to check if the page is being framed or not. If the latter is the case, the body of

the document will be displayed to the user. Otherwise, the frame will be busted.

Hence, since the execution of JavaScript code will be deactivated if there is a frame busting attack, the content of the Web page will then not be displayed. Because this countermeasure is purely JavaScript-based, it is applicable to modern mobile browsers.

After manually analyzing the Alexa Top 500 websites [22], we want to stress that this kind of JavaScript-based frame buster is one of the most attack-resistant countermeasures against the busting frame busting techniques published by Rydstedt et al. [20].

**Listing 2: A frame buster of August Detlefsen. The target is to prevent a website against being framed and to be resistant against busting frame busting attacks.**

```
<style id="antiClickjack">
  body{display:none !important;}
</style>
<script type="text/javascript">
  if (self === top) {
    var antiClickjack = document.
      getElementById("antiClickjack");
    antiClickjack.parentNode.
      removeChild(antiClickjack);
  } else {
    top.location = self.location;
  }
</script>
```

*X-Frame-Options and the CSP.* Using the HTTP header *X-Frame-Options* (XFO) provides another way of tackling framing attacks. Developed by Microsoft in 2008 and implemented since Internet Explorer (IE) 8 [23] this header's use forces a supported browser to check if a website should be loaded in a frame or not. In contrast to the JavaScript-based countermeasure, the frame will not be busted. Instead, a warning message inside the frame will be displayed.

Microsoft introduced three different values for the header in question. These are: *DENY*, *SAMEORIGIN*, and *ALLOW-FROM origin*. If one wants to forbid all Web pages from loading the protected Web page, then *DENY* should be used. To allow a Web page of the same domain to load the protected Web page, one should use *SAMEORIGIN*. The last value, which is only available in IE, gives an option of specifying a URL allowed to frame the protected Web page. In this case, the URL has to be replaced with *origin*.

A main disadvantage of the HTTP header is its limited support, restricted only to modern browsers such as Firefox ≥3.6.9, Opera ≥10.5, and the already mentioned IE ≥8. For this reason, most of the websites are using the JavaScript-based countermeasure. Our statistical query into this matter [6] included analyzing the TOP 100,000 Alexa websites and scanning the very first Web page of each domain. This resulted in a discovery that in the TOP 100 we only find three websites using *X-Frame-Options*. On top of that, there is a small total of 143 websites with such header in the TOP 100,000. Over 66 % used the value *SAMEORIGIN* to allow Web pages with the same domain to frame their Web page. Less than 34 % are using the value *DENY*.

The Content Security Policy (CSP) [24, 25] is a header which can be used for *X-Frame-Options* in a very similar way. Aside from the framing protection, one can also identify other targets, such as preventing data injection attacks or cross-site scripting. By using the directive *frame-ancestors*, one can specify a frame chain in a form of whitelist. In contrast to *X-Frame-Options*, *Content Security Policy* is experimentaly supported by Desktop-based browsers like Firefox ≥4, Chrome ≥13 and IE10. Sadly, compared to *X-Frame-Options*, the range of supporting browsers is even more restrictive. Furthermore, *frame-ancestors* is no longer specified in the W3C draft [26] as CSP will focus on sandboxing and source specification of style sheets, script files and similar issues.

As illistrated in Table 1, a test with various well-known mobile browsers has shown that they support *X-Frame-Options* without exception. In the case of the Content Security Policy only Firefox was able to protect a website with the old CSP (oCSP) – more precisely with the test case *X-Content-Security-Policy: allow 'self'; frame-ancestors 'none'*. For the sake of obtaining a full picture, we have verified if one of the mobile browsers is supporting the new CSP (nCSP) variant of the W3C draft. The latter is not even the case in WebKit-browsers, although it is supported in the tested desktop-based WebKit-browser Chrome 20.0.

| Browser | Engine | XFO | oCSP | nCSP |
|---|---|---|---|---|
| Android – 4.0.3 | WebKit | ✓ | × | × |
| Dolphin – 8.7.0 | WebKit | ✓ | × | × |
| Firefox – 4.0.3 | Gecko | ✓ | ✓ | × |
| Opera Mini – 7.0 | Presto | ✓ | × | × |
| Opera Mobile – 12.00 | Presto | ✓ | × | × |

**Table 1: This table gives an Overview of different Android mobile browsers and their support of X-Frame-Options as well as the old CSP (oCSP) and new – by the W3C draft specified – CSP (nCSP).**

## 5.2 Tapjacking Defense Mechanisms

*Android touch filter.* As described in Chapter 2.2, an attacker is able to let touch gestures pass through a malicious application. Considering this fact, it makes sense for an application to block touch gestures received whenever view's window is obscured. For that reason the Android developers have implemented such protection mechanisms. They can be used with a call of *setFilterTouchesWhenObscured()* or, alternatively, with the attribute *android:filterTouchesWhenObscured*. Unfortunately, these are not enabled by default and they are only available in some Android versions – specifically those higher than 2.2. To demonstrate the pressing urgency and potential harm caused by this matter, let us provide a Google Play 14-day period statistic from the 5th of March 2012 [5]. This measure indicates that Android ≥2.3 version constitutes less than 70 percent of a grand total.

The presence of the countermeasure proves that Android developers are aware of the security problem. Hence it is

possible to protect self-developed applications, such as those responsible for online banking transactions. At the same time, the (by default existing and attackable) home screen is totally unprotected and thus, at the very least, it leaves plenty ways for attacking most of the applications running on Android operating system by means of tapjacking.

The biggest limitation of this countermeasure is that only applications that are aware of tapjacking may protect themselves by using this filter. Note that even standard applications, like the phone application, do not use the Android touch filter. Given the slow system update cycles of mobile phone software, this will leave the majority of applications unprotected.

**Listing 3: This Listing depicts a code snippet of the TSL implementation approach. In our TSL application, we use a transparent button of a significant size with the touch filter. Consequently, it is assured that it is displayed over the whole screen.**

```
// position and width, height of the full
// screen defense button
params.gravity = Gravity.LEFT |
    Gravity.TOP;
params.width = LayoutParams.FILL_PARENT;
params.height = LayoutParams.FILL_PARENT;

// retrieve a WindowManager for accessing
// the system's window manager.
WindowManager wm = (WindowManager)
    getSystemService(WINDOW_SERVICE);

// adds a child view
wm.addView(mButton, params);
```

*Tapjacking Security Layer.* The fact remains that a non-negligible quota of Android users are prone to browserless UI redressing attacks described the in section 2.2. For that reason, we have developed an approach for a new security layer. However, we were not able to fully implement it with the existing environment, given our restricted access to the Android operating system. The implementation problems are caused by the limitations in the operating system; we were able to place an application in front of another application and not behind it. Android is open source and therefore it can implemented by e.g. the Android team into the kernel in the near future.

As it is shown in Figure 3, we assume that malicous and target applications are both open. We also presume that there is a layer between both applications[2]. From now on, this layer will be called *Tapjacking Security Layer* (TSL).

In our approach, the TSL opens automatically once a user fires an application. It is crucially important that it is always in the background and remains opened until the application in its forefront gets closed. Further, a touch gesture on the TSL will be blocked. This can be asured by using a large transparent button in a combination with the Android touch filter – a code snippet of our implementation is given in Listing 3.

[2]This scenario can be also applied to a layer between the malicious application and the home screen. Thus, we are able to protect a target application as well as the home screen.

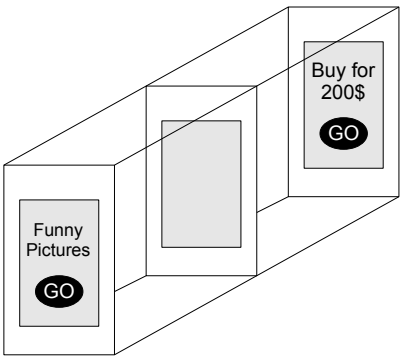


**Figure 3: The security layer, which is between the malicious application and the target application, protects a victim against browserless tapjacking attacks.**

All in all, this assures that no touch gesture on part of a victim will be unintentionally forwarded to another application. Therefore, classic clickjacking-related browserless attack scenarios can no longer be carried out. Besides, one can also say that the given in-built mitigation in later OS versions with the Android touch filter is no longer required This can be obtained through the use of TLS and not only fixed to the secured elements (by the developers of each individual application); nevertheless, we still need the touch filter feature for the TLS.

# 6. CONCLUSION AND OUTLOOK

In Chapter 2, we have described the exploits and vulnerabilities regarding UI attacks on Android devices and 3. As part of our general contribution, we have clarified which attacks can be carried out against Web browsers natively implemented in Android: Most of the existing attacks can be used with very little effort. Chapter 4 introduced a novel browserless UI redressing attack. The showcased scenario gives an attacker a powerful option of completing malicious actions vitally with the help of the home screen. Attempting phone calls without having any privileges for them should be pinpointed as one of the key examples.

Moving forward, we described important countermeasures against the previously presented attacks. We discussed in detail which browser-based UI redressing mitigation techniques exist and how they can be used; this includes JavaScript-based frame busters and the HTTP header *X-Frame-Options* as well as *X-Content-Security-Policy*.

Based on the ideas of the concepts highlighted above, we have introduced a new security layer against tapjacking attacks. Unfortunately, our invention cannot be fully implemented without the support from the Android developers. We think that this concept, which can be integrated into Android's kernel, will be a more reliable countermeasure than Android touch filter, which will only protect security-aware implementations.

In the outlook one shall wonder how UI redressing attacks are supposed to be mitigated in the future. Given the fact that HTML5 and CSS3 drafts are partially implemented in Web browsers, the field of attacks will continuously expand. Thus, it is probable that we will witness a long-lasting 'cat and mouse game' of fixing and breaking Web security features. Having taken a closer look at the browserless tapjacking attack, we must recommend that vendors of security software (especially those providing anti-virus products) urgently implement a functionality like our Tapjacking Security Layer and effectively gain better protection for currently available Android operating systems.

# 7. REFERENCES


[1] Ruderman, J.: Bug 154957 - iframe content background defaults to transparent. https://bugzilla.mozilla.org/show_bug.cgi?id=154957 (2002)

[2] Hansen, R., Grossman, J.: Clickjacking attack. http://www.sectheory.com/clickjacking.htm (2008)

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

[4] Gartner: Third quarter of 2011. http://www.gartner.com/it/page.jsp?id=1848514 (2011)

[5] developers, A.: Platform versions – current distribution. http://developer.android.com/resources/dashboard/platform-versions.html (2011, 2012)

[6] Niemietz, M.: Ui redressing: Attacks and countermeasures revisited. In: CONFidence 2011, http://data.proidea.org.pl/confidence/9edycja/materialy/prezentacje/MarcusNiemietz.pdf (2011)

[7] SophosLabs: Facebook worm - "likejacking". http://nakedsecurity.sophos.com/2010/05/31/facebook-likejacking-worm/ (2010)

[8] Lekies, S., Heiderich, M., Appelt, D., Holz, T., Johns, M.: On the fragility and limitations of current browser-provided clickjacking protection schemes. In: in Usenix Workshop on Offensive Technologies (wOOt 2012). (2012)

[9] Bordi, E.: Proof of concept - cursorjacking (noscript). http://static.vulnerability.fr/noscript-cursorjacking.html (2012)

[10] Valotta, R.: Cookiejacking. https://sites.google.com/site/tentacoloviola/ (2011)

[11] Kotowicz, K.: Filejacking: How to make a file server from your browser (with html5 of course). http://blog.kotowicz.net/2011/04/how-to-make-file-server-from-your.html (2011)

[12] Raskin, A.: Tabnabbing: A new type of phishing attack. http://www.azarask.in/blog/post/a-new-type-of-phishing-attack/ (2011)

[13] Huang, L.S., Jackson, C.: Clickjacking attacks unresolved. Technical report, Carnegie Mellon University (2011)

[14] Richardson, D.: Tapjacking. http://blog.mylookout.com/look-10-007-tapjacking/ (2010)

[15] Mannino, J.: Revisiting android tapjacking. http://blog.nvisiumsecurity.com/2011/05/revisiting-android-tapjacking.html (2011)

[16] developers, A.: Android notificationmanagerservice.java. http://source-android.frandroid.com/frameworks/base/services/java/com/android/server/NotificationManagerService.java (2011)

[17] Wei, J.: Indefinite toast hack. http://thinkandroid.wordpress.com/2010/02/19/indefinite-toast-hack/ (2010)

[18] Google: Introducing google play: All your entertainment, anywhere you go. http://googleblog.blogspot.de/2012/03/introducing-google-play-all-your.html (2012)

[19] developers, A.: Intent. http://developer.android.com/reference/android/content/Intent.html (2012)

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

[21] Detlefsen, A., Li, J., Schmidt, C., Crawford, B.: Clickjacking defense. https://www.codemagi.com/blog/post/194 (2010)

[22] Alexa: Alexa top 500 global sites. http://www.alexa.com/topsites (2012)

[23] Lawrence, E.: Internet explorer 8 security part vii: Clickjacking defenses. http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx (2009)

[24] Sterne, B.: Content security policy. http://people.mozilla.org/~bsterne/content-security-policy/ (2011)

[25] Consortium, W.W.W.: Content security policy – w3c working draft. http://www.w3.org/TR/CSP/ (2011)

[26] Consortium, W.W.W.: Content security policy 1.1. https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html (2012)