Sidewinder Targeted Attack against Android in The Golden Age of Ad Libraries

Tao Wei, Yulong Zhang, Hui Xue, Min Zheng, Chuangang Ren, and Dawn Song

FireEye, Inc.

{*tao.wei, yulong.zhang, hui.xue, min.zheng, chuangang.ren, dawn.song*}*@fireeye.com*

1 Introduction

By 2014, the number of Android users has grown to 1.1 billion and the number of Android devices has reached 1.9 billion [1]. At the same time, enterprises are also embracing Android based Bring Your Own Device (BYOD) solutions. For example, in Intel's BYOD program, there are over 20,000 Android devices across over 800 combinations of Android versions and hardware configurations [2].

Although Google Play has little malware, there are many vulnerabilities in Android apps and the Android system itself. Aggressive ad libraries also leak the user's private information. By combining altogether, an attacker can conduct more targeted attacks, which we call "Sidewinder Targeted Attacks". In this paper, we explain the security risks from such attacks, in which an attacker can intercept private information like GPS location uploaded from ad libraries and use that information to precisely locate targeted areas such as a CEO's office or some specific conference rooms. When the target is identified, "Sidewinder Targeted Attack" exploits popular vulnerabilities in ad libraries, such as Javascript-binding-over-HTTP or dynamic-loading-over-HTTP, etc.

It is a well-known challenge for an attacker to call Android services from injected native code which doesn't have Android application context. We explain how attackers can invoke Android services for tasks such as taking photos, calling phone numbers, sending SMS, reading/writing the clipboard, etc. Furthermore, the attackers can exploit several Android vulnerabilities to get valuable private information or to launch more advanced attacks.

Finally, we show that this threat is not only real but also prevalent due to the popularity of Android ad libraries. We hope that this paper will kick start the conversation on how to better protect the security and privacy in third-party libraries and how to further harden the Android security framework in the future.

2 Sidewinder Targeted Attack Overview

To understand the security risks brought by a Sidewinder targeted attack, we first explain one possible attack mechanism, which is similar to that of Sidewinder missiles, as illustrated in Figure 1. The attacker can hijack the network that the targeted victim resides in. Like those infrared homing system, the attacker then seeks for "emission" from ad libraries

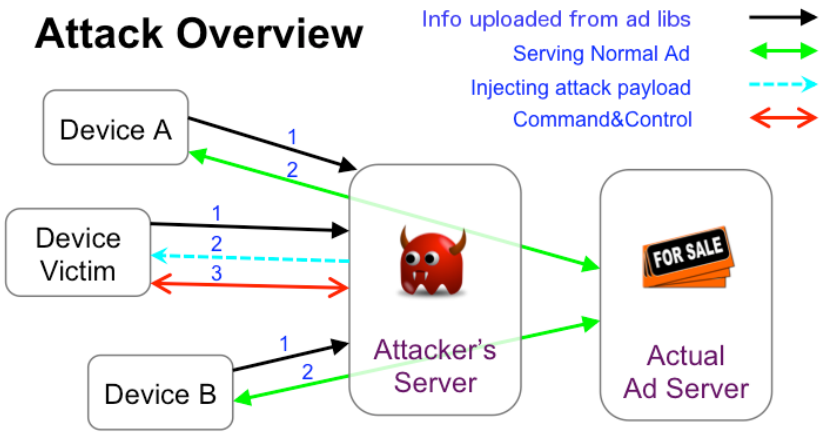


Figure 1: Illustration of the Sidewinder Targeted Attack Scenario.

Table 1: Outline of the Sidewinder Targeted Attack through Vulnerable Ad Libraries

API Level	$\leq$ API 16		$>$ API 16
Attack Vector	JBOH and DLOH		JS Sidedoor
Attacks	(*w/ Android Context*) Clipboard manipulation Launcher settings modification Proxy modification Taking pictures Audio & video recording Stealthy app installation	(*w/o Android Context*) Local files uploading Root exploit & Code injection Implanting bootkit Sending SMS Making phone calls	Abusing privileged interfaces

running on the target device to track and lock on it. Once the target was locked on, the attacker can launch advanced persistent attacks. To minimize the chance of being detected, the attacker can choose to take action on important targets only, ignoring all other devices.

In Section 3 and Section 4, we discuss attacking("warhead") and targeting("homing") components in detail and show how a combination of these components can launch powerful and precise attacks towards the target devices.

Table 1 proposes different attacks that an attacker can launch remotely on target devices through vulnerable ad libraries. Figure 2 shows a proof-of-concept attack control interface. This attack targets at one of the ad libraries described in this paper. The security risks become obvious by looking at what the attacker can do with this control interface. The left panel enables the attacker to give command to the victim's device, including uploading local files, taking pictures, recording audio/video, manipulating the clipboard, sending SMS, dialing numbers, implanting bootkit, or installing the attacker's apps uploaded to Google Play, etc. The right panel lists all information stolen from the victim's device. In this screenshot, the victim's installed app list, clipboard, a photo taken from the back camera, an audio clip, and a video clip have been uploaded, with the GPS location intercepted from the ad library. The panel also pins down the GPS location of the victim's device onto a Google Map widget.

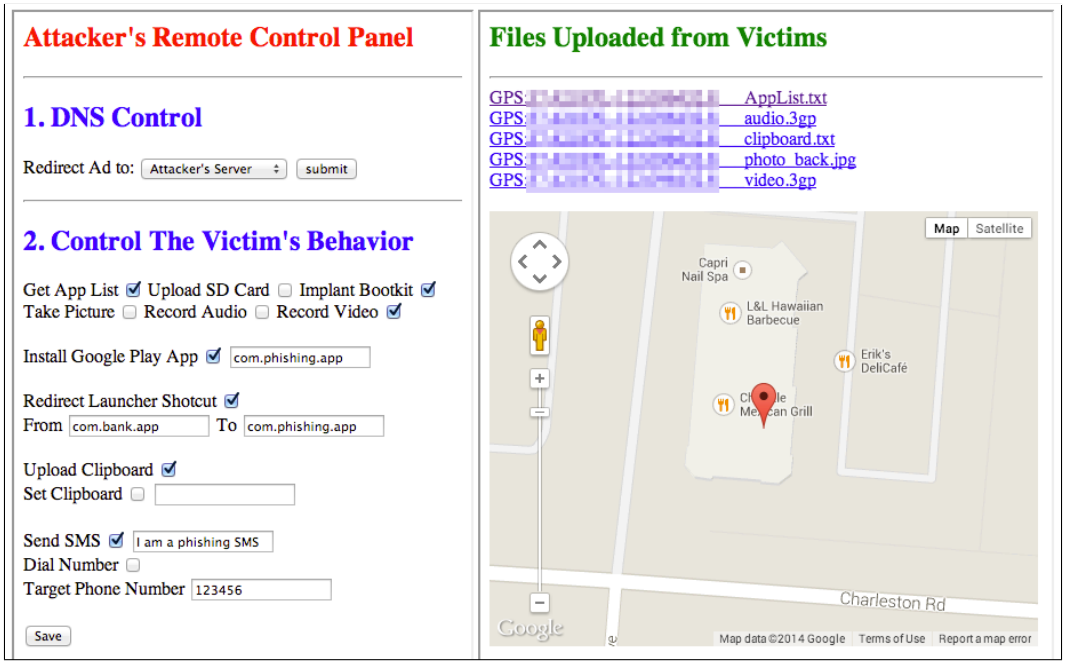


Figure 2: The control pannel of the attacker, and the files uploaded from the victim.

Based on this precise position information, it is trivial to identify individual or groups of "VIP" targets by which offices they are in.

3 Warhead: Attacking Vulnerabilities of Android

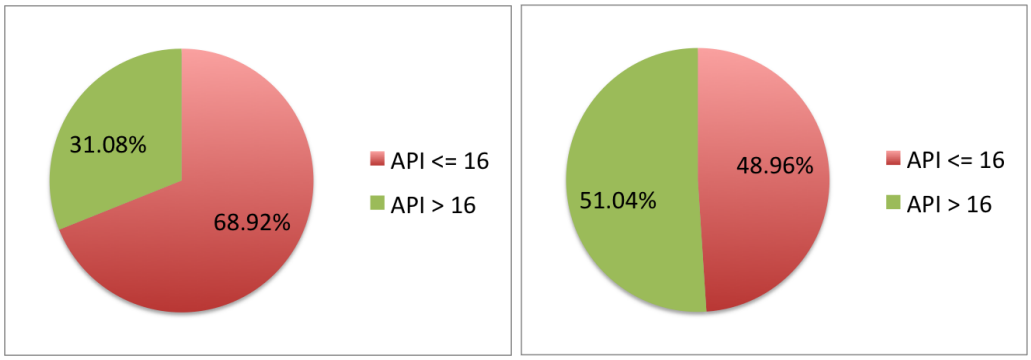
3.1 Piercing The Armor

In this section, we explain the risks of remote attacks on the Android devices in more details.

3.1.1 Attacking JavaScript Binding over HTTP (JBOH)

Android uses the JavaScript binding method `addJavascriptInterface` to enable JavaScript code running inside a WebView to access the app's Java methods (also known as the Javascript bridge). However, it is widely known that this feature, if not used carefully, presents a potential security risk when running on Android API 16 (Android 4.1) or below. As noted by Google: "Use of this method in a WebView containing untrusted content could allow an attacker to manipulate the host application in unintended ways, executing Java code with the permissions of the host application." [3]

In particular, if an app running on Android API 16 or below uses the JavaScript binding method `addJavascriptInterface` and loads the content in the WebView over HTTP, then an attacker over the network could hijack the HTTP traffic, e.g., through WiFi or DNS hijacking, to inject malicious content into the WebView and to control the host application. Listing 1 is a sample Javascript snippet to execute shell command.



(a) Statistics by app number (b) Statistics by app download count

Figure 3: Target SDK statistics of Popular Google Play apps

Listing 1: Sample Javascript snippet to execute shell command.

```
jsObj.getClass().forName("java.lang.Runtime")
    .getMethod("getRuntime",null).invoke(null,null).exec(cmd);
```

We call this the JavaScript-Binding-Over-HTTP (JS-Binding-Over-HTTP) vulnerability [4]. This applies to insecure HTTPS channels as well. If an app containing such vulnerability has sensitive Android permissions such as access to the camera, then a remote attacker could exploit this vulnerability to perform sensitive tasks such as taking photos or record video in this case, over the Internet, without the user's consent. Based on the official data in June 2014 [5], ~60% of Android devices are still running API≤16.

Note that those API>16 platforms are not necessarily secure. If the app is targeting at a lower API level, Android will still run it with the lower API level for compatibility reasons. Figure 3 shows the targeted API of popular Google Play apps, each of which has over 50,000 downloads. We can see that a large portion of apps are targeting at API≤16.

3.1.2 Attacking Annotated JavaScript Binding Interfaces

Starting with Android 4.2 (API>16), Google introduced the `@JavascriptInterface` annotation [6] to explicitly designate and restrict which public Java methods in the app were accessible from JavaScript running inside a WebView. However, if an ad library uses the `@JavascriptInterface` annotation to expose security-sensitive interfaces, and uses HTTP to load content in the WebView, it is vulnerable to attacks where an attacker over the network could inject malicious content into the WebView to misuse the interfaces exposed through the JS binding annotation. We call these exposed JS binding annotation interfaces JS sidedoors.

For example, we found a list of sensitive Javascript Interfaces that are publicly exposed from a real world ad library: `createCalendarEvent`, `makeCall`, `postToSocial`, `sendMail`, `sendSMS`, `takeCameraPicture`, `getGalleryImage`, `registerMicListener`, etc. Given that this ad library loads ads using HTTP, if the host app has the corresponding permissions (e.g. CALL_PHONE), attackers over the network can abuse these interfaces to do malicious things (e.g. utilizing the `makeCall` interface to dial phone numbers without the user's consent).

3.1.3 Security Issues with Dex Loading over HTTP (DLOH)

Similar to JBOH, Dex loading over HTTP or insecure HTTPS (DLOH) is another serious issue raised by ad libraries. If the attackers can hijack the communication channels and inject malicious dex files, they can then control the behaviors of the victim apps.

3.2 Detonation without Android Context

After getting local access, the attacker can upload private and sensitive files from the victim's device, or modify files that the host app can write to, e.g., the directory of the host app and SD Card with FAT file system.

In order to launch more sophisticated attacks such as sending SMS or taking pictures, the attackers may seek to use Java reflection to call other APIs from the Javascript bridge. As far as we know, sending SMS is easy to achieve via this method. However, some other operations require Android context [7] or registering Java callbacks. Android context provides an interface to the global information about an app's environment. Many Android functionalities, especially remote call invocations, are encapsulated in the context. We discuss attacks requiring context later in Section 3.3. In this Section, we explain attacks that don't need Android context and the security risks.

3.2.1 Root Exploits and Code Injection

One direct threat posed by JBOH is to use the JBOH shell (Listing 1), to download executables and use them to root the device. Commercial one-touch root apps claim that they can root more than 1,000 brands (>20,000 models) [8]. `towelroot` [9], which exploits a bug found recently in Linux kernel, claims that it can root most new devices released before June 2014. Thus, as long as an attacker can get the JBOH shell, he or she has the tools to obtain root on most Android phone models.

Even if the attackers can't obtain root, they can try to use ptrace [10] to control the host app. Although only processes with root privilege can ptrace others, child processes are able to ptrace their parents. Because the shell launched from the Javascript bridge is a child process of the host app, it can ptrace the host app's process. Note that only apps with `android:debuggable` set as "true" in the manifest can be ptraced, which limits its adoption.

3.2.2 Sending SMS And Dialing Numbers without User Consent

Sending SMS does not require context or user interaction. A simple call does the job, as shown in Listing 2.

Listing 2: Sending SMS without user consent.

```
SmsManager.getDefault().sendTextMessage(phoneNumber, null, message, null,
    null);
```

In order to make calls from the Javascript bridge without user consent, we can invoke the telephony service to dial numbers directly via binder, as shown in Listing 3, where `phone` is the remote Android telephony service and the number `2` represents the second remote call. `s16` is the type marker represents "16 bit string", and `packageName` is the host app's package name, where we can obtain from the information posted from the ad libraries. The sequence number of the remote calls can be found in the corresponding Android Interface

Definition Language (AIDL) files [11]. Many other Android services can be invoked in the same way, including sending SMS.

Listing 3: Dial numbers without user consent.

```
Runtime.getRuntime().exec("service call phone 2 s16 " + packageName + " s16 "
    + phoneNumber);
```

3.3 Detonation with Android Context

As mentioned before, it will be way more convenient if we can directly obtain the Android context via the Javascript bridge. Code in Listing 4, for example, is one of those easy ways to get context from anywhere of the application.

Listing 4: Sample code to obtain context.

```
// We omit all try-catch statements and other unimportant code in this paper.
public Context getContext() {
    final Class<?> activityThreadClass = Class
        .forName("android.app.ActivityThread");
    final Method method = activityThreadClass
        .getMethod("currentApplication");
    return (Application) method.invoke(null, (Object[]) null);
}
```

Operations like taking pictures and recording videos need to register Java callbacks. The attackers either need to boot a Java VM from the Javascript bridge, or to inject code into the host app's Java VM.

Fortunately, Android Runtime offers us another way to load JNI code into the host app using `Runtime.load()`. As shown in Listing 5, an attacker can load executables compiled from JNI code. Once loaded, the code can obtain context as described in Listing 4, or further call `DexClassLoaderload` [12] to inject new classes from the attackers' dex files to register callbacks to take picutures/record videos.

Listing 5: Sample Javascript snippet to load JNI binary into the host app's Java VM.

```
jsObj.getClass().forName("java.lang.Runtime")
    .getMethod("getRuntime",null).invoke(null,null).load(binaryPath);
```

There are also other ways to obtain Android context, like reflecting to the private static context variable of WebView [13]. However, without Java VM instances, it's difficult to take pictures and record videos. After our submission to Blackhat in April 2014, we noticed that MWR was also concurrently and independently working on this issue. They published a similar mechanism in June 2014 [14].

3.3.1 Clipboard Monitoring And Tampering

With the Android context, an attacker can monitor or tamper the clipboard. Android users may perform copy-paste on important text content. For example, there are many popular password management apps in Google Play, enabling the users to click-and-copy the passwords and paste them into login forms. Malicious apps can steal the passwords, if they can read the contents on clipboard. Android has no permissions restricting apps from

accessing the global clipboard. Any UID has the capability to manipulate clipboard via the API calls in Listing 6:

Listing 6: API calls to peek into/tamper the clipboard.

```
ClipboardManager.getText()
ClipboardManager.hasPrimaryClip()
ClipboardManager.setText()
ClipboardManager.setPrimaryClip()
ClipboardManager.hasText()
ClipboardManager.addPrimaryClipChangedListener()
ClipboardManager.getPrimaryClip()
```

Using these APIs, the attackers can monitor changes to clipboard and transfer the clipboard contents to some remote server. They can also alter the clipboard content to achieve phishing goals. For example, the user may copy a link he or she is about to visit and the background malicious service can change the link to a phishing site. We have notified Google about this issue.

3.3.2 Launcher Settings Modification

Android Open Source Project (AOSP) classifies Android permissions into several protection levels: "normal", "dangerous", "system", "signature" and "development" [15, 16, 17]. Dangerous permissions "may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities". In contrast, normal permissions are automatically granted at installation, "without asking for the user's explicit approval (though the user always has the option to review these permissions before installing)" [15]. If an app requests both dangerous permissions and normal permissions, Android only displays the dangerous permissions by default. If an app requests only normal permissions, Android doesn't display any permission to the user.

We have found that certain "normal" permissions have dangerous security impacts [18]. For example, the attackers can manipulate Android home screen icons using two normal permissions: launcher READ_SETTINGS and WRITE_SETTINGS permissions. These two permissions enable an app to query, insert, delete, or modify the whole configuration settings of the Launcher, including the icon insertion or modification.

As a proof of concept attack scenario, a malicious app with these two permissions can query/insert/alter the system icon settings and modify legitimate icons of some security-sensitive apps, such as banking apps, to a phishing website.

After our notification, Google has patched this vulnerability in Android 4.4.3 and has released the patch to its OEM partners. However, based on the official data [5], more than 90% of the Android devices are still using Android versions below 4.4.3.

3.3.3 Proxy Modification

With the CHANGE_WIFI_STATE permission, Android processes can change the proxy settings of the WIFI networks (not necessarily the current connected one). To do this, the attacker can use the remote calls exposed by `WifiManager` to obtain the `WifiConfiguration` objects, then create new `proxySettings` to replace to corresponding field. Note that the `proxySettings` field is a private Java field, which is not intended to be accessed by other

processes. Unfortunately, the flexible and powerful Java reflection mechanism (especially the `forName(), getField(), setAccessible()` calls) exposes such kind of components to the attackers for arbitrary read or write operations.

3.3.4 Taking Pictures And Recording Audio/Video without User Interaction

Android audio recording via the `MediaRecorder` APIs does not need user interaction or consent, which makes it easy to record sound in the background.

On the contrary, taking pictures and recording videos are more challenging. First, this requires registering Java callbacks. Second, Android obliges that "Preview must be started before you can take a picture", according to the official guidance [19]. It seems that taking pictures and recording videos without user notification is impossible. However, security largely depends on the correct implementation. It is hard to enforce a flawless implementation. On some of the popular phones (model anonymized for security consideration), `startPreview()` is indeed required to take pictures/record videos. But you do not really have to create a view before calling `startPreview()`. It's highly possible that on these devices `takePicture()` fails to check whether a view has been presented to the user. Fortunately, we have never witnessed a case where the MediaRecorder can shoot videos without calling `setPreviewDisplay`. But we were able to create and register a dummy `SurfaceView` to the `WindowManager` on the fly. With this trick, taking photos and videos become possible even on the devices with proper checking for an existing preview.

3.3.5 Stealthy App Installation by Abusing Credentials

With both the GET_ACCOUNTS and the USE_CREDENTIALS permissions, Android processes can get secret tokens of services (e.g. Google services) from the `AccountManager` and use them to authenticate to these services [20]. We have verified that Android apps with these two permissions are able to authenticate themselves using the user's Google account, to access Google Play store and send app installation requests. Attackers through the Javascript bridge can utilize this mechanism to install any apps of attacker's choice (e.g. a phishing app the attacker publishes) to any devices registered in user's account in the background without user consent. Combined with the launcher modification attack introduced in Section 3.3.2, the attackers can redirect other app icons (e.g. bank or email app icons) to the phishing app installed stealthily and steal the user's login credential inputs.

4 Targeting Victims Based on Ad Traffic

In this section, we explain the risks of victims' devices being tracked and targeted through ad traffic.

4.1 Communication Channels Prone to Hijack

It is well known that communication via HTTP is prone to hijacking and data tampering. Though ad libraries may not have the incentive to abuse users' private and sensitive data, this is not the case with the attackers eavesdropping or hijacking the HTTP traffics. Switching to HTTPS may not solve this issue, since the security of HTTPS relies on the implementation and it is difficult to guaranttee a totally flawless implementation. For example, there are cases where the developer forgot (or intentionally ignored) checking the server's

certificate [21]. We find that some of the most popular ad libraries (see Table 3) do have this issue. We successfully launched Man-in-the-middle (MITM) attacks and intercepted the data uploaded to the remote server. Note that even if the ad libraries have a correct and rigorous implementation, the SSL library itself may contain serious vulnerabilities that can be exploited by MITM attacks [22, 23].

4.2 Information Leakage from Ad Libraries

Almost every ad library uploads local information from Android devices. Based on our observations, they do so mostly for purposes such as checking for platform compatibility and user interest targeting. The information uploaded the most include IMEI, Android version, manufacturer, Android ID, device specification, carrier information, host app information, installed app list, etc. Table 3 lists the info uploaded from the top 5 popular ad libraries.

Listing 7 is a captured packet posted to the remote ad server by one of the ad libraries. It is captured from a popular Google Play app. From this packet we can tell the device's screen density (d-device-screen-density), screen size (d-device-screen-size), host app's package name (u-appBId), host app's name (u-appDNM)[1], host app's version (u-appVer), user agent (h-user-agent), localization (d-localization), mobile network type (d-netType), screen orientation (d-orientation), and GPS location (u-latlong-accu). The most important information is the GPS location, where the victim's latitude, longitude and the location precision are shown. It is totally reasonable for an ad to obtain these information to improve the ad serving experience. However, with this information, an attacker can precisely locate the victim and acquire the victim device's specification.

Listing 7: Packet Cpatured from A Real World Ad Library

```
requestactivity=AdRequest&d-device-screen-density=1.5&d-device-screen-size=320X533&
u-appBId=com.example.app&u-appDNM=Example&u-appVer=1.2&h-user-agent=Mozilla
%2F5.0+%28Linux%3B+U%3B+Android+4.1.2%3B+en-us%3B+sdk+Build%2FMASTER%
29+AppleWebKit%2F534.30+%28KHTML%2C+like+Gecko%29+Version%2F4.0+Mobile+Safari
%2F534.30&d-localization=en_us&d-netType=umts&d-orientation=1&u-latlong-accu=
37.410835%2C-121.920514%2C8
```

4.3 Large-scale Monitoring and Precise Hijacking

In order to locate victims effectively, an attacker needs to monitor large-scale network traffic which contains such private information. Unfortunately, several well-known attacks can be used to achieve large-scale monitoring, including DNS hijacking, BGP Hijacking ,and ARP hijacking in IDC.

In this paper, DNS hijacking is to subvert the resolution of Domain Name System (DNS) queries through modifying the behavior of DNS servers so that they serve fake DNS information. DNS hijacking is actively used in many situations, including both legal or malicious usage, such as traffic management, phishing or censorship. Many DNS servers, including the ones from Google and Godaddy, were compromised successfully by attackers [24]. By DNS hijacking, attackers can effectively access all the traffic to ad servers.

BGP hijacking is to take over groups of IP addresses by corrupting Internet routing tables through breaking BGP sessions or injecting fake BGP information. In this way, attackers can monitor all traffic to specific IPs. In history there were many BGP hijacking

[1]We anonymized the host app's name and package name to hide the identity.

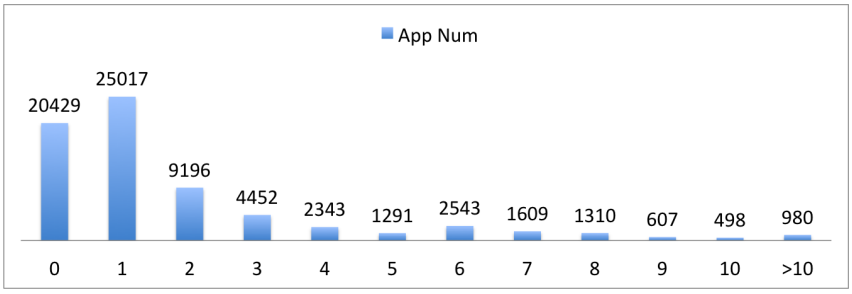


Figure 4: Number of ad libraries included in Google Play apps (with more than 50,000 downloads).

attacks that affected YouTube, DNS root servers, Yahoo, and many other important Internet services [25].

ARP hijacking (or spoofing) in IDC [26] is to hijack the traffic to the ad server in the IDC where the ad server locates through fake ARP packets. Attackers may rent servers close to the target servers, and use fake ARP packets to direct all the traffic to the target servers to go through the hijacking servers first. In this way, the attacker can monitor and hijack the traffic. ARP hijacking is a well-known approach used in network attacks.

Based on the large-scale traffic intercepted from the above methods, attackers can identify potential victims based on information leakage such as GPS location described in Section 4.2. After that, they can inject exploits only into the targeted traffic to launch further attacks on the victims. Attackers can let all other irrelevant network traffic pass without being modified to keep a low profile.

5 Targetable and Exploitable Google Play Apps

We use FireEye Mobile Threat Prevention (MTP) engine analyzed all of the ∼73,000 popular apps from Google Play with more than 50,000 downloads, and identified 93 ad libraries. The detailed ad library inclusion statistics are shown in Figure 4. 71% of the apps contain at least one ad library, 35% have at least two ad libraries, and 22.25% include at lease three ad libraries. The largest ad inclusion number is 35. Since Google is usually pretty cautious about the security of the products it directly controls, we exclude Google Ad in the following discussion. For security considerations, in this paper we anonymize the names of the other 92 ad libraries, using Ad_1, Ad_2, ..., Ad_{92} to refer to them, where the subscripts represent the rankings of how many apps include the ad libraries. The top 5 popular ad libraries' inclusion and download statistics are listed in Table 2.

We analyzed the 92 ad libraries found in the popular Google Play apps, and summarized the communication channel vulnerabilities in Table 3. Combined with the uploaded information column we can learn what kind of data the attackers can obtain.

57 of the 92 ad libraries in the popular Google Play apps have the JBOH issue. Specifically, 4 out of the top 5 ad libraries are subject to this problem (shown in Table 2). 7 out of the 92 ad libraries are prone to DLOH attacks. Particularly, some versions of Ad_5 in Table 3 have this problem. The affected Google Play apps number and the accumulated download counts are listed in Table 4.

Table 2: The inclusion statistics of the top 5 Android ad libraries excluding Google Ad. Their JBOH statistics are also listed (discussed in Section 3.1.1).

Ad Library	Number of Apps	JBOH Apps	Total Downloads	JBOH Downloads
Ad_1	9,702	2,802	8,781M	2,348M
Ad_2	8,856	4,204	7,865M	4,754M
Ad_3	8,818	2,117	8,499M	1,611M
Ad_4	5,519	1,112	4,687M	617M
Ad_5	5,170	0	4,519M	0

Table 3: The uploaded data, communication channel vulnerabilities, and JBOH/DLOH details of the top 5 ad libraries.

Ad Library	Uploaded Info	Protocol	SSL Vuln	JBOH	DLOH
Ad_1	IMEI/device id, device model, Android version, location	HTTP/HTTPS	✓	✓	✗
Ad_2	device specification, Android version, host app info, location	HTTP	✗	✓	✗
Ad_3	IMEI/device id, device model, Android version, device manufacturer, carrier info, location, ip	HTTP	✗	✓	✗
Ad_4	IMEI/device id, device model, device specification, Android version	HTTP	✗	✓	✗
Ad_5	IMEI/device id, device model, device specification, Android version, country, launguage	HTTPS	✓	✗	✓

Table 4: Assessment statistics of Google Play apps (downloads ≥50,000) that are vulnerable to the Sidewinder Targeted Attack. **Type I** apps are those subject to JBOH or DLOH attacks; **Type II** apps are those not only JBOH/DLOH exploitable but also have the LOCATION leakage (thus vulnerable to the Sidewinder Targeted Attack). Note that an app is counted into the total statistics if it is subject to *any of the attacks*, including uploading files and root exploits.

Subject to attack type	**Type I #**	**Type I Downloads**	**Type II #**	**Type II Downloads**
Code injection via ptrace	2,055	444M	272	67M
Send SMS	349	340M	229	254M
Make phone calls	572	399M	426	324M
Launcher modification	111	95M	81	37M
Proxy modification	644	792M	419	378M
Record audio	1,097	1,408M	654	621M
Take pictures/record videos	1,141	1,380M	622	665M
Install apps stealthily	351	552M	197	332M
Total(incl. root exploits)	**16,579**	**11,706M**	**4,201**	**3,207M**

6 Conclusion

In the current golden age of Android ad libraries, Sidewinder Targeted Attack can target victims using ad libraries' info leakage and exploit other vulnerabilities of ad libraries to get valuable sensitive information. Millions of users are still under the threat of Sidewinder Targeted Attacks. The first thing we need to do is to improve the security and privacy protection of the ad libraries. For example, we encourage the ad libraries to use HTTPS with proper SSL certificate validation, and to properly encrypt the network traffic. They need to also be cautious about what kind of privileged interfaces should be exposed to the ad providers, just in case of the interfaces being controled by malicious ads or attackers hijacking the communication channels.

Meanwhile, Android itself needs to further harden the security framework as well. This is way more difficult than it sounds, because:

1. Android is a complex system. Any sub-component's vulnerability may impact the security of the whole system. Fragmentation makes the situation even worse.
2. The trade-off between usability, performance and security always matters, and the market demand determines that security often comes last. Many Android developers do not even understand how to program securely (as shown in the JBOH issue).
3. Many security patches are not back-ported to old versions of Android (like the launcher settings problem described in Section 3.3.2), even though the old versions are still widely used.
4. There is always information asymmetry in the development chain. For example, it usually takes 6 months for the vendors to apply the security patches after Google releases them.

Albeit challenging, we hope that the work can kick start the conversation on how to better improve the security and privacy protection in third-party libraries and how to further harden the Android security framework in the future.

References

[1] Ranjit Atwal, Lillian Tay, Roberta Cozza, Tuong Huy Nguyen, Tracy Tsai, Annette Zimmermann, and CK Lu. Forecast: Pcs, ultramobiles and mobile phones, worldwide, 2010-2017, 4q13 update. *Gartner*, 2013.

[2] Rob Evered, Steve Watson, Paul Dockter, and Derek Harkin. Android devices in a byod environment. *Intel White Paper*, 2013.

[3] `http://developer.android.com/reference/android/webkit/WebView.html#addJavascriptInterface(java.lang.Object,%20java.lang.String)`.

[4] `http://www.fireeye.com/blog/technical/2014/01/js-binding-over-http-vulnerability-and-javascript-sidedoor.html`.

[5] `https://developer.android.com/about/dashboards/index.html`.

[6] `http://developer.android.com/reference/android/webkit/JavascriptInterface.html`.

[7] http://developer.android.com/reference/android/content/Context.html.

[8] http://shuaji.360.cn/root/.

[9] http://towelroot.com/.

[10] http://linux.die.net/man/2/ptrace.

[11] http://developer.android.com/guide/components/aidl.html.

[12] http://developer.android.com/reference/dalvik/system/DexClassLoader.html.

[13] http://www.weibo.com/p/1001603724694418249344?utm_source=weibolife.

[14] https://labs.mwrinfosecurity.com/blog/2014/06/12/putting-javascript-bridges-into-android-context.

[15] http://developer.android.com/guide/topics/manifest/permission-element.html.

[16] https://android.googlesource.com/platform/frameworks/base/+/master/core/res/AndroidManifest.xml.

[17] https://android.googlesource.com/platform/packages/apps/Launcher2/+/master/AndroidManifest.xml.

[18] http://www.fireeye.com/blog/technical/2014/04/occupy_your_icons_silently_on_android.html.

[19] http://developer.android.com/reference/android/hardware/Camera.html.

[20] http://seclists.org/bugtraq/2014/Mar/52.

[21] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[22] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0224.

[23] http://www.fireeye.com/blog/technical/2014/04/if-an-android-has-a-heart-does-it-bleed.html.

[24] https://isc.sans.edu/diary/Domaincontrol+(GoDaddy)+Nameservers+DNS+Poisoning+/5146.

[25] http://www.networkworld.com/article/2272520/lan-wan/six-worst-internet-routing-attacks.html.

[26] http://en.wikipedia.org/wiki/ARP_spoofing.