The Lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface

Daniel R. Thomas[1], Alastair R. Beresford[1], Thomas Coudray[2], Tom Sutcliffe[2], and Adrian Taylor[2]

[1] Computer Laboratory, University of Cambridge, Cambridge, UK
`firstname.lastname@cl.cam.ac.uk`
[2] Bromium, Cambridge, United Kingdom
`thomas.coudray.fr@gmail.com`, `tom.sutcliffe@bromium.com`, `adrian@bromium.com`

Abstract. We examine the lifetime of API vulnerabilities on Android and propose an exponential decay model of the uptake of updates after the release of a fix. We apply our model to a case study of the JavaScript-to-Java interface vulnerability. This vulnerability allows untrusted JavaScript in a WebView to break out of the JavaScript sandbox allowing remote code execution on Android phones; this can often then be further exploited to gain root access. While this vulnerability was first publicly disclosed in December 2012, we predict that the fix will not have been deployed to 95% of devices until December 2017, 5.17 years after the release of the fix. We show how this vulnerability is exploitable in many apps and the role that ad-libraries have in making this flaw so widespread.

Keywords: API security, Android, WebView, security updates, ad-libraries, JavaScript, Java, vulnerabilities, network attacker, RCE

1 Introduction

The Android ecosystem today is a complex network of competing and collaborating companies. In addition to the main OS developer (Google) there are at least 176 additional open source projects whose code is used in the platform. There are also many manufacturers and network operators who customise Android for their devices and networks. For example, 20 300 study participants in the Device Analyzer project [17] use devices built by 298 distinct manufacturers and networks run by 1 440 different operators.

In this landscape, fixing security flaws is hard since it often involves the collaboration of open source developers, Google, the device manufacturers, the network operators and the user (who needs to approve the installation of updates). In this paper we explore Application Programming Interface (API) vulnerabilities in Android and quantify the rate at which these flaws are fixed on real devices. Such vulnerabilities often represent a security protocol failure, in the sense that the API designer had a particular protocol or API call sequence in mind, and the attacker repurposes those API elements to break the intended security model.

Fixing API vulnerabilities, like fixing deployed protocols, is often hard: fixes may require changes to the API, which breaks backwards compatibility. In our analysis we find that an exponential decay function provides a good model for predicting the rate of fixes for API vulnerabilities in Android. Unfortunately, the rate of decay is low: it takes nearly a year for half of the Android devices using the Google Play Store to update to a new version of Android. In other words, it takes a long time to move from the domain of security fiction (a new release is available which has fixed the vulnerability) to fact (devices are now secure). This is explored further in Section 2.

In order to ground our approach we have included a case study in Section 3 to investigate the timeline for fixing one API vulnerability in Android. We have selected the JavaScript-to-Java interface vulnerability for this purpose as it is particularly serious and affects all versions of Android prior to the release of version 4.2. The fixing release was first available in October 2012 and as such we now have sufficient data to quantify the speed at which updates have propagated.

2 API vulnerabilities in Android

At the beginning of 2015 Android had revised its API twenty one times since version one was released with the first Android handset in 2008. We have manually collected the monthly statistics published by Google which record the proportion of devices using particular API versions when they connect to the Google Play Store since December 2009. These statistics[3] are plotted in Figure 1. The API version distribution shows a clear trend in which older API versions are slowly replaced by newer ones.

In order to quantify the lifecycle of a particular API version we recalculate the API version data in two ways. Firstly, in order to understand the speed of adoption of a particular version of the API, we are interested in the number of days since release rather than specific calendar dates. Secondly, we are interested in the proportion of devices which have *not* upgraded to a particular API version or any successor. For example, when a new API version is first released, no devices could have already been updated to it and therefore the proportion which have not upgraded is one. As devices begin to upgrade to the new API version (or any subsequent release), the proportion not upgraded tends to zero.

We have replotted data from Figure 1 in Figure 2 to show the proportion of devices not upgraded to a particular version of Android against days since the API version was first released. These data show that all API version upgrades follow a similar trend: a slow initial number of upgrades in the first 250 days, then widespread adoption between 250 and 1000 days, followed by a slow adoption of the new API version by the remaining devices.

Visually, these data appear to have an exponential decay as it tends to zero. We therefore decided to model this as $f(t)$, a combination of an exponential

[3] We have made these available [13]

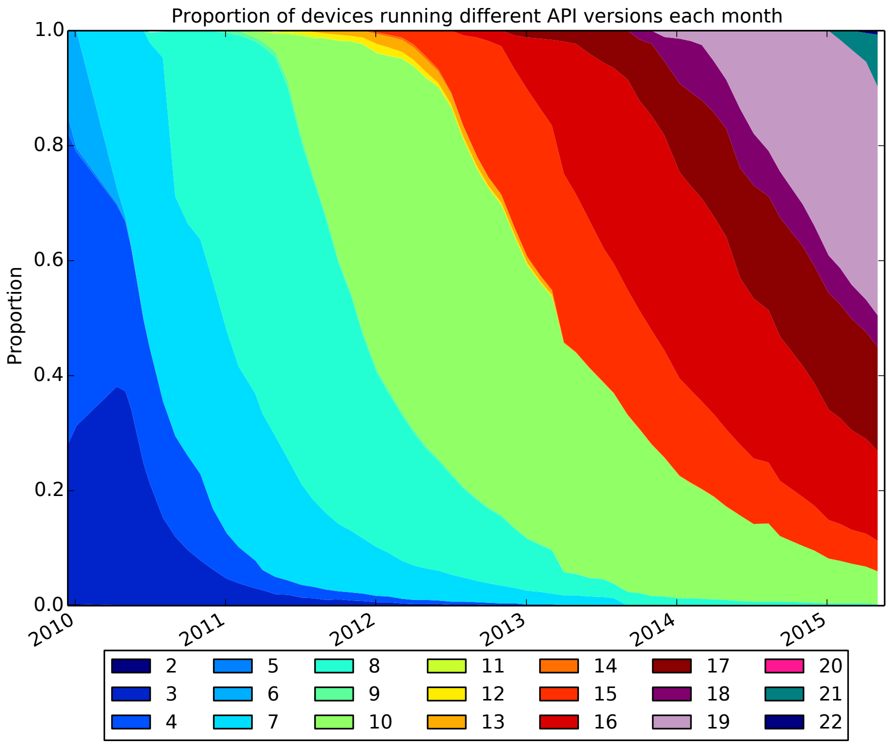

Fig. 1. Proportion of devices running different API versions

function together with a delay t_0 which offsets the start time:

$$f(t) = \begin{cases} 1.0 & \text{if } t < t_0 \\ e^{-\text{decay}(t-t_0)} & \text{otherwise} \end{cases} \tag{1}$$

Fitting $f(t)$ to these, we get a Root-Mean-Squared-Error (RMSE) of 0.182 with the parameters $t_0 = 80.7$ days, decay $= 0.0026$ days^{-1} across all API versions. An RMSE of 0.182 compares favourably with a standard polynomial fit (3 degree polynomial fit gave an RMSE of 0.182) or a spline fit (RMSE of 0.182) and gives a meaningful model of behaviour rather than a generic curve.

From this fit, the number of days from the release of a new version of Android until 50% of devices are running that version or higher is 347 (0.95 years) and full deployment to 95% of devices takes 1 230 days (3.37 years). The same analysis using the Device Analyzer data on OS versions in use gives 324 days (0.886 years) and 1 120 days (3.06 years) respectively which is faster but not by much.

Hence if a security vulnerability is fixed through the release of a particular API version it will be 1 230 days (3.37 years) after that until the fix is fully deployed.

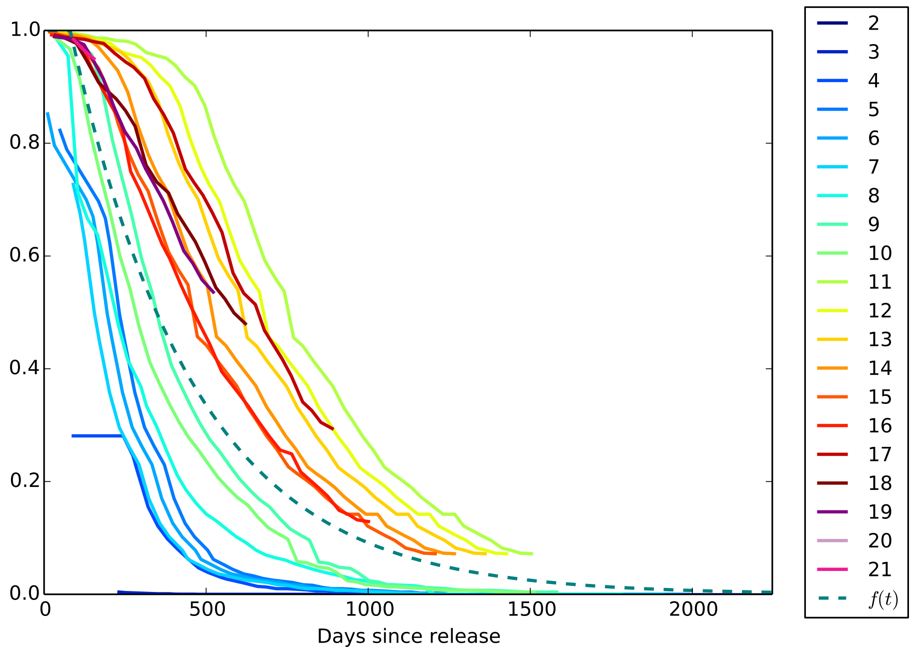

Fig. 2. Proportion of devices not updated to particular versions of Android or any later version. The best fit $f(t)$ is an exponential decay function.

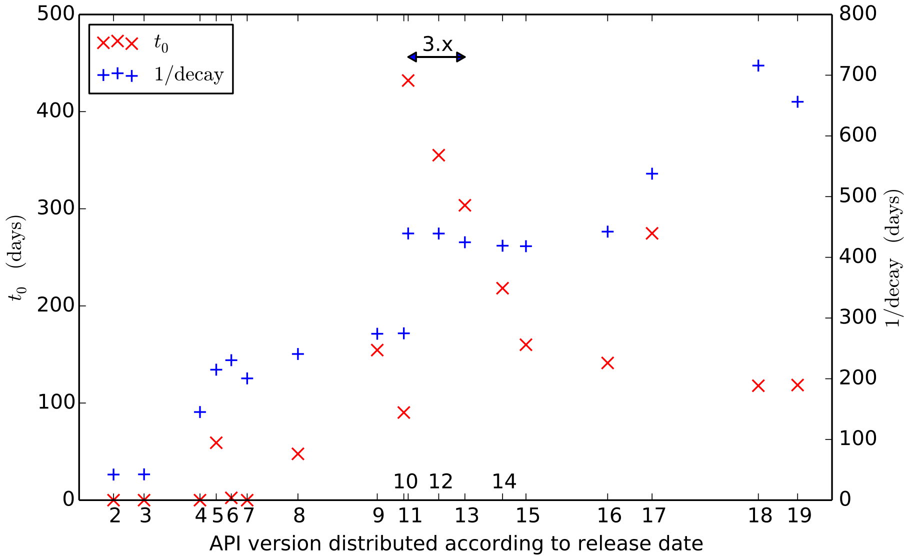

Fig. 3. Fitted parameters for different API versions.

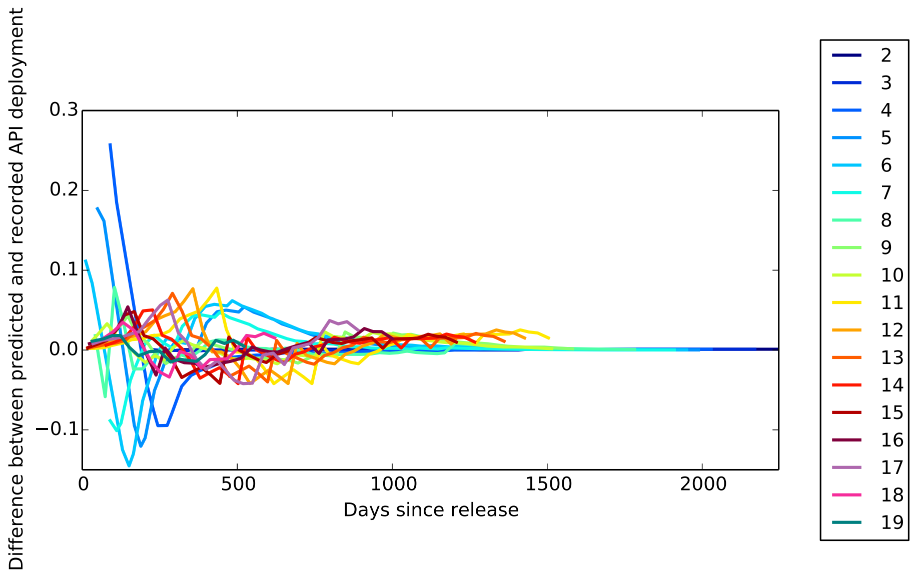

Fig. 4. Difference between predicted behaviour and recorded behaviour.

Unfortunately, while this is a good predictor of average behaviour, individual API versions are systematically different from each other. Hence, we took the fit parameters from the global analysis and used them to seed a fit for each API version. This gave us the parameters in Figure 3 with t_0 and 1/decay plotted as this means that larger values for both are worse. API versions 11, 12 and 13 were for Android 3.x which never saw widespread deployment because they targeted tablets and were not available for use on phones. Discounting those values, Figure 3 shows a trend of updates taking longer over time as t_0 increases and 1/decay increases. This implies that the Android ecosystem is getting worse at distributing updates.

The differences between the predictions and recorded reality is shown in Figure 4. It shows how the difference between our prediction and recorded behaviour oscillates around 0 with some systematic errors early on due to the simple model of $f(t)$. The errors are mostly less than 10% and fall over time.

3 Case study: The JavaScript-to-Java interface vulnerability

The Android *WebView* provides a developer with a web browser UI component that can be controlled programmatically by a hosting app, including rendering dynamic HTML content driven by JavaScript. To allow convenient interaction between the WebView and the hosting app, a Java object instance can be bound to a JavaScript variable name, allowing the JavaScript code to call any public

```
<script>
    Android.getClass()
        .forName('java.lang.Runtime')
        .getMethod('getRuntime',null)
        .invoke(null,null).exec(['id']);
</script>
```

Fig. 5. JavaScript attack, assuming *Android* is the JavaScript alias for the exposed Java object.

methods on the Java object. Prior to Android 4.2, the exposed public methods included those inherited from parent classes, including the `getClass()` method of `java.lang.Object`. This permitted the execution of arbitrary Java code from the JavaScript running inside the WebView. For example, Java reflection controlled from JavaScript can be used to execute Linux programs such as `id`, as shown in Figure 5. This is a security vulnerability (CVE-2012-6636) which can be used to remotely run malicious code in the context of an app using the JavaScript-to-Java interface vulnerability and from there exploit other vulnerabilities to gain root privileges on devices and spread as an Android worm.

The attack is comprised of the following steps.

1. Content for WebViews in apps is commonly from untrusted sources or over an unauthenticated HTTP connection, so therefore an active attacker controlling the network (strategies for doing this are discussed in Section 3.1) can inject a malicious JavaScript payload into the HTTP stream, which is then executed inside the JavaScript sandbox.
2. The malicious JavaScript can then use the JavaScript-to-Java interface vulnerability to break out of the JavaScript sandbox into the app context.
3. Then the malicious code can often use other known vulnerabilities to break out of the app sandbox and gain root privileges on the device. We know that, on average, approximately 88% of Android devices are vulnerable to at least one known root vulnerability [14].
4. Once an attacker has root on a device, he can use ARP spoofing or ICMP redirect attacks to reroute local traffic through the device and inject malicious JavaScript into any HTTP traffic, thereby starting step (1) above on a new device. Thus the attack has the potential to act as an an Android worm.

Google has modified the Android API or its implementation twice in an attempt to fix the JavaScript-to-Java interface vulnerability. In the first, the function of the JavaScript-to-Java interface was modified in Android 4.2 to ensure that only public methods with the annotation `@JavaScriptInterface` could be called from within JavaScript for new apps. This change only prevents the attack if *both* the phone is running Android 4.2 or greater *and* the app has been compiled with a recent version of the Android framework with a target API Level of 17 or newer. In the second, for devices using the WebView based on Google

Chrome for Android version 33.0.0.0 or later (included in Android 4.4.3 and later), access to the `getClass` method on injected Java objects was blocked.[4] This prevents the most obvious JavaScript-to-Java interface attacks by preventing direct access to the Java runtime. An attacker must instead find a different route through the public methods on the injected Java objects, which may not always exist and is certainly much harder.

Any app with a WebView containint a JavaScript-to-Java interface is potentially vulnerable to this attack. We label an app which uses JavaScript-to-Java interface *always vulnerable* if it contains a target API level of 16 or older, since such an app is vulnerable when run on any version of Android less than 4.4.3; and *vulnerable only on outdated devices* if the app has a target API Level of 17 or newer, since such an app is vulnerable only if running on a device running Android 4.1.x or older.

3.1 Threat model

There are several different scenarios in which an attacker could inject malicious JavaScript to exploit the JavaScript-to-Java interface vulnerability.

1. An attacker could control the original server that supplied 'legitimate' HTML either through compromising it or by using some other means (such as buying ads) to supply the malicious JavaScript.
2. They could control a node on the path from the original server allowing them to inject malicious JavaScript into the HTTP traffic.
3. An attacker could control traffic passing through the device's local network and inject malicious JavaScript. This could be achieved by either running a public WiFi network, or compromising an existing network using ARP spoofing or ICMP redirect attacks to redirect all traffic via a machine under their control.

Level 1 attacks can be mitigated by better system security and input validation at the original server. Level 2 and Level 3 attacks can be mitigated by apps using HTTPS with proper validation of certificates [3] (for example using pinning [2]) to prevent an attacker from being able to inject malicious JavaScript. Level 3 attacks can also be mitigated through the use of a secure VPN to a trustworthy network and by better security on the local network (protection against ARP spoofing, ICMP redirect attacks and independently encrypted connections to the router).

[4]

`https://codereview.chromium.org/213693005/patch/20001/30001` committed as 261801 or afae5d83d66c1d041a1fa433fbb087c5cc604b67 or e55966f4c3773a24fe46f9bab60ab3a3fc19abaf

3.2 Sources of vulnerability

To investigate the severity of this vulnerability we need data on which apps use the JavaScript-to-Java interface and where they use it. We analysed 102 174 APK files from the Google Play Store collected on 2014-03-10 and between 2014-05-10 and 2014-05-15. We found that 21.8% (22 295) of apps were always vulnerable, 15.3% (15 666) were vulnerable only on outdated devices, 62.2% (63 533) were not vulnerable and 0.67% (680) could not be analysed due to failures of our static analyser. These results are presented in Table 1 and show that most apps are not vulnerable, but that more apps are always vulnerable than are vulnerable only on outdated devices.

The static analysis was performed by decompiling the APKs using `apktool` and extracting the target API version from the Android Manifest. Apps using JavaScript-to-Java interface were detected by string matching for '`addJavascriptInterface`' in the decompiled .smali files.

Of the 38 000 vulnerable apps, 12 600 were in the Device Analyzer data [17], those which are not in the Device Analyzer data are unlikely to be widely used, since they were not installed on any of the 20 300 devices in Device Analyzer data.

In the following analysis, values are given ± their standard deviation. We found that always vulnerable apps were started 0.6 ± 0.0 times a day between the disclosure of the vulnerability and the start of our APK file collection, with 8.34 ± 0.67 such apps installed.

We found that apps vulnerable only on outdated devices were started 0.78 ± 0.10 times a day between the disclosure of the vulnerability and the start of our APK file collection, with 7.27 ± 0.71 such apps installed.

Hence on an outdated device vulnerable apps were started 1.38 ± 0.11 times a day with 15.6 ± 0.9 vulnerable apps installed. Due to static analysis failures and the fact that not all the apps are observed by Device Analyzer, these rates are likely to be underestimates. It is also possible that the Device Analyzer data could be biased towards users with more apps than is typical, which might cause this figure to be an overestimate.

Classification	Percentage	Count
Always vulnerable	21.8	22 295
Vulnerable only on outdated devices	15.3	15 666
Not vulnerable	62.2	63 533
Unscanable	0.67	680

Table 1. Percentage of the 102 174 apps analysed which fell in each category

Ad-libraries We tested a couple of dozen of the apps we had identified as always vulnerable by MITMing them and injecting JavaScript which exploited the JavaScript-to-Java interface vulnerability. We found that 50% of these were actually exploitable. There are several reasons why we might not have been able to exploit the vulnerability: we did not activate the vulnerable activity, HTTPS was used for the request or the vulnerable code in the app was not exercised during our testing.

Inspecting the vulnerable HTTP requests revealed that ad-libraries were the the usual reason for vulnerable requests. We performed further static analysis on the APK files by reverse engineering the ten most downloaded apps that detect ads and constructing a list of pattern matches for different ad-libraries. The distribution of different ad-libraries is shown in Figure 6.

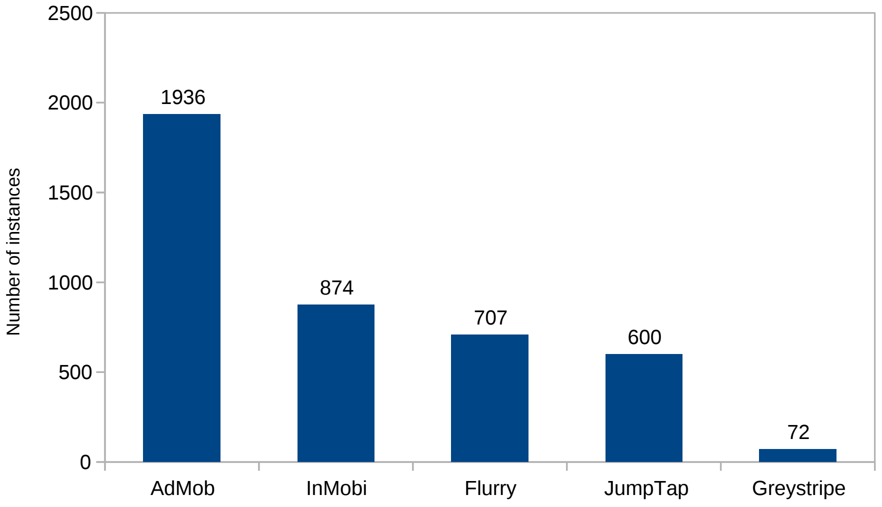

Fig. 6. Number of instances of each ad-library within the always vulnerable apps. Not all of these frameworks necessarily introduce the vulnerability and some versions of them may have been fixed, this plot just shows the distribution of the ad-libraries within the always vulnerable apps.

3.3 Lifetime of the vulnerability

The vulnerability was first publicly recorded in December 2012 [1].The proportion of devices that contacted the Google Play Store and are secure for apps vulnerable only on outdated devices are shown in blue in Figure 7. In summary, in April 2015 70.6% of devices were running a version of Android that protects users from apps vulnerable only on outdated devices.

This vulnerability will cease to be problematic when all Android devices run API version 17 or later *and* all apps which use JavaScript-to-Java interface target API version 17 or later. Using our model for $f(t)$ from Equation 1 and knowledge that API version 17 was released in October 2012 we expect 95% of

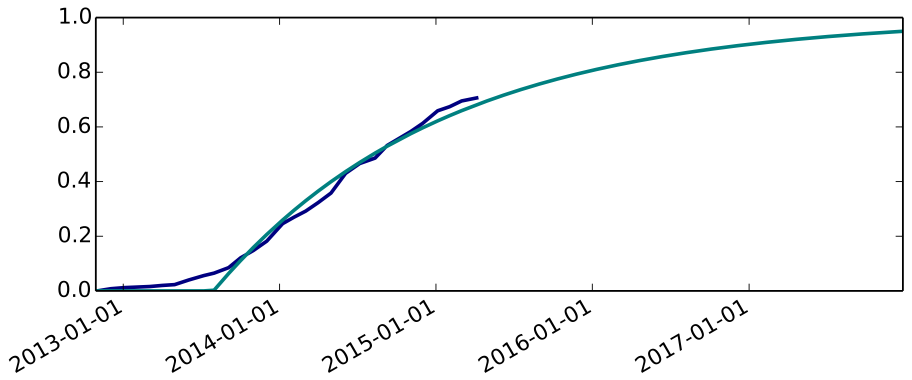

Fig. 7. Proportion of fixed devices with these data from Google Play given in blue and above it our prediction in green

all Android devices to be secure for apps vulnerable only on outdated devices by December 2017. This prediction is shown in green in Figure 7. We do not have visibility into the way apps' target API versions on Android change over time and therefore it is harder to understand whether always vulnerable apps will continue to represent a significant risk after almost all Android devices support API version 17 or later.

3.4 Solutions

There are various strategies which could have been adopted to more rapidly mitigate this vulnerability. Android could have broken API compatibility and backported the fix to all versions of Android, however other work has shown that security updates are deployed slowly on Android [14]. Android could refuse to load JavaScript over HTTP and require HTTPS (with properly verified certificates) or use local storage which would make MITM attacks injecting malicious JavaScript much harder. Part of the problem is that the libraries (particularly ad-libraries) which developers bundle inside their apps target old API versions and developers need to update their dependencies to versions that target higher API versions.

If Android had a more comprehensive package management system, which handled dependencies then apps could be loosely coupled with their ad-libraries and the libraries could be updated to fixed versions without the app developers having to re-release every app that used it. Alternatively, to maintain backwards compatibility while fixing the vulnerability, apps could be automatically rewritten to fix the vulnerability.

Users could use a VPN to tunnel all their traffic back to a trusted network so that MITMs on local networks (such as open wifi access points) would not be able to mount this attack, but this would not protect against attackers on the network path to the ad-server, or malicious ad-servers.

The fix included in Android 4.4.3 discussed earlier in Section 3, where access to the `getClass` method is blocked substantially, mitigates this vulnerability. Language solutions such as Joe-E could be used to enforce security properties, including preventing JavaScript from executing arbitrary code [8]. Such a solution would need to avoid legacy interfaces (which the JavaScript-to-Java interface vulnerability itself illustrates the danger of) allowing this protection to be bypassed [18].

4 Related work

The JavaScript-to-Java interface vulnerability has been investigated before. It was demonstrated by MWR Labs [6] who showed how it could be used to run the Android security assessment framework `drozer`, which can be used to run a remote shell on a compromised Android device. The strategies we used for statically analysing Dalvik bytecode to discover use of JavaScript-to-Java interface have also been used previously [19].

Attacks have been published against WebView [7] including those relating to the JavaScript-to-Java interface and vulnerabilities caused by the violation of the origin-based access control policy in hybrid apps [4].

There have been investigations of the behaviour of ad-libraries on Android. Stevens et. al. demonstrated how attacks could be mounted on JavaScript-to-Java interface used by ad-libraries, but without realising the significance of `getClass` [12]. However, unlike the vulnerability we have discussed, these attacks continue to work on fixed devices even for apps vulnerable only on outdated devices. Grace et. al. have shown that ad-libraries require excessive permissions and expose users to additional risks [5], which are further compounded by the JavaScript-to-Java interface vulnerability.

To counteract the problems caused by ad-libraries being packaged within an app, and thereby inheriting all their permissions, there have been proposals to separate out the ad-libraries into separate processes by altering Android to provide an API [10] and then automatically rewriting apps to use such an API [11]. This improves security, particularly if it means that the ad-libraries can be updated independently of the apps, but it does not otherwise help if an attack on JavaScript-to-Java interface can be followed up with a root exploit.

We scanned 102 174 apps but the PlayDrone crawler was able to analyse over 1 100 000 apps, and found a similar but more comprehensive distribution of usage of different ad-libraries [16].

Nappa et. al. have analysed the decay of vulnerabilities affecting client applications on Windows [9], this uses similar data to our analysis but collects this from information provided by hosts directly rather than using published summary data.

5 Conclusion

In this paper we proposed the exponential decay model for Android API vulnerabilities and we explored one case study: the JavaScript-to-Java interface vulnerability. By applying our model to our case study we find that for apps which are vulnerable only on outdated devices, 95% of all Android devices will be protected from the JavaScript-to-Java interface vulnerability by December 2017, 5.17 ± 1.23 years after the release of the fix. It is not known whether always vulnerable apps will continue to present a security risk and therefore it is unclear whether Android users will be safe from this vulnerability after this date.

Acknowledgements

This work was supported by a Google focussed research award and the EPSRC Standard Research Grant EP/P505445/1. Some of the raw data and source code is available [15], the analysed APKs are not included as we do not have distribution rights for them. Thanks to Robert N. M. Watson for his insight and useful feedback.

References

[1] Neil Bergman. *Abusing WebView JavaScript Bridges.* 12/2012. URL: `http://d3adend.org/blog/?p=314` (visited on 2015-01-09).

[2] Jeremy Clark and Paul C. van Oorschot. "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements". In: *IEEE Symposium on Security and Privacy* (2013), pp. 511–525. DOI: `10.1109/SP.2013.41`.

[3] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. "Why Eve and Mallory love Android: an analysis of Android SSL (in)security". In: *CCS*. ACM, 2012, pp. 50–61. ISBN: 9781450316514. DOI: `10.1145/2382196.2382205`.

[4] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks". In: *Network and Distributed System Security Symposium (NDSS)* (2014). DOI: `10.14722/ndss.2014.23323`.

[5] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. "Unsafe exposure analysis of mobile in-app advertisements". In: *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)* (2012), pp. 101–112. DOI: `10.1145/2185448.2185464`.

[6] MWR Labs. *WebView addJavascriptInterface Remote Code Execution.* 2013. URL: `https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/` (visited on 2014-12-19).

[7] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. "Attacks on WebView in the Android System". In: *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. Orlando, Florida, USA: ACM, 2011, pp. 343–352. ISBN: 9781450306720. DOI: `10.1145/2076732.2076781`.

[8] Adrian Mettler, Daniel Wagner, and Tyler Close. "Joe-E: A security-oriented subset of Java". In: *Network and Distributed System Security Symposium (NDSS)*. 2010.

[9] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. "The Attack of the Clones : A Study of the Impact of Shared Code on Vulnerability Patching". In: *IEEE Symposium on Security and Privacy* (2015), pp. 692–708. DOI: 10.1109/SP.2015.48.

[10] Paul Pearce, Adrienne Porter Felt, and David Wagner. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *ACM Symposium on Information, Computer and Communication Security (ASIACCS)* (2012). DOI: 10.1145/2414456.2414498.

[11] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. "AdSplit: Separating smartphone advertising from applications". In: *Proceedings of the 21st USENIX conference on Security symposium* (2012), p. 28. arXiv: 1202.4030.

[12] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. "Investigating user privacy in Android ad libraries". In: *IEEE Mobile Security Technologies (MoST)*. 2012.

[13] Daniel R. Thomas. *Historic Google Play Dashboard*. 2015. URL: http://androidvulnerabilities.org/play/historicplaydashboard.

[14] Daniel R. Thomas and Alastair R. Beresford. *AndroidVulnerabilities.org*. URL: http://androidvulnerabilities.org/.

[15] Daniel R. Thomas, Thomas Coudray, and Tom Sutcliffe. *Supporting data for: "The Lifetime of Android API vulnerabilities: case study on the JavaScript-to-Java interface"*. 05/2015. URL: https://www.repository.cam.ac.uk/handle/1810/247976 (visited on 2015-05-26).

[16] Nicolas Viennot, Edward Garcia, and Jason Nieh. "A measurement study of Google Play". In: *SIGMETRICS* (2014). DOI: 10.1145/2591971.2592003.

[17] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. "Device Analyzer: Large-scale mobile data collection". In: *Sigmetrics, Big Data Workshop*. Pittsburgh, PA: ACM, 06/2013. DOI: 10.1145/2627534.2627553.

[18] David Wagner and Dean Tribble. *A Security Analysis of the Combex DarpaBrowser Architecture*. 2002. URL: http://combexin.temp.veriohosting.com/papers/darpa-review/security-review.pdf (visited on 2012-03-08).

[19] Erik Ramsgaard Wognsen and Henrik Sø ndberg Karlsen. "Static Analysis of Dalvik Bytecode and Reflection in Android". In: *Master's thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark* (2012).