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] Adrian Taylor[2]

[1]Computer Laboratory, University of Cambridge, United Kingdom
firstname.lastname@cl.cam.ac.uk

[2]Bromium, Cambridge, United Kingdom
thomas.coudray.fr@gmail.com, tom.sutcliffe@bromium.com,
adrian@bromium.com

23rd Security Protocols Workshop

UNIVERSITY OF CAMBRIDGE

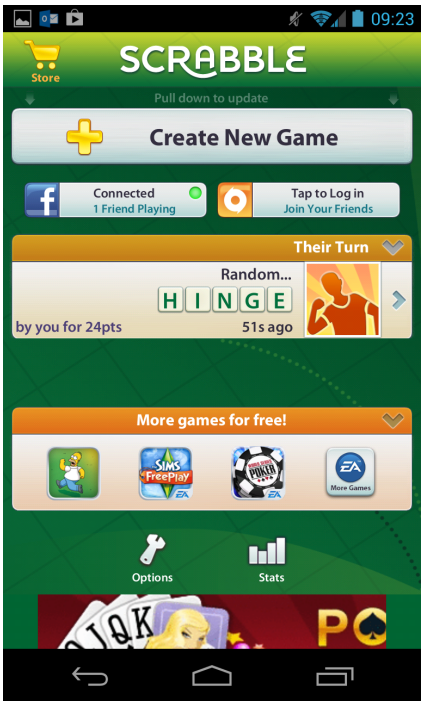
Daniel: 5017 A1EC 0B29 08E3 CF64 7CCD 5514 35D5 D749 33D9
Alastair: 9217 482D D647 8641 44BA 10D8 83F4 9FBF 1144 D9B3

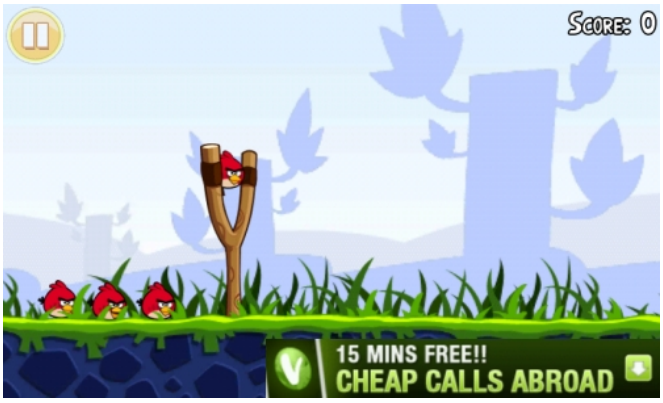
We often see the two sided upgrade problem with protocols like TLS where upgrades are deployed very slowly with substantial impacts on security.

APIs are another kind of protocol.

We look at API upgrades in Android with a case study of a remote code execution API vulnerability.

Android apps display ads in WebViews





WebViews display HTML/CSS and **JavaScript**.

Some apps are just a wrapper around a WebView which points at their website

JavaScript communicates with Java

- Collect information for ads
- Provide interactivity
- Sit down in a coffee shop and open angry birds, now your phone is compromised and infecting other phones.

```
/** Show a toast from the web page */
public void showToast(String toast) {
    Toast.makeText(context, toast, LENGTH).show();
}
```


Location, read contacts, installed appsTake a photo, record audio, open an app

The JavaScript-to-Java interface vulnerability

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

JavaScript attack, assuming *android* is the JavaScript alias for the exposed Java object.

- Apps use WebViews to display HTML fetched over HTTP.
- Bridge from JavaScript-to-Java exposes *all* public methods.
- Android worm.

Android apps use WebViews to display HTML content like adverts fetched over HTTP
To interact with the app and to get data for the advertisers there is a bridge from JavaScript to Java. Originally all public methods, including the `.getClass()` inherited from `java.lang.Object` were then accessible.
This means that JavaScript can use reflection to execute arbitrary code in the context of the app.
Since 87.7% of Android devices are exposed to known critical vulnerabilities this means that this code can then escalate its privileges.
Once code has root it can use attacks like ARP spoofing, ICMP redirect and exploit vulnerabilities in DHCP to spread to other Android devices on the same network.

Two approaches to fixes

1. Change the API and require apps to recompile (2012)

	target API < 17	target API ≥ 17
device API < 17	Vulnerable	Vulnerable
device API ≥ 17	Vulnerable	Safe

```
/** Show a toast from the web page */
@JavascriptInterface
public void showToast(String toast) {
    Toast.makeText(context, toast, LENGTH).show();
}
```


Change the API to require apps to annotate methods which should be accessible, but fallback to the old API for apps which have not made this change for compatibility.
How effective is this first approach at fixing the vulnerability?
70% in the bottom 2/3 of the table, 60% in the left half of the table

Two approaches to fixes

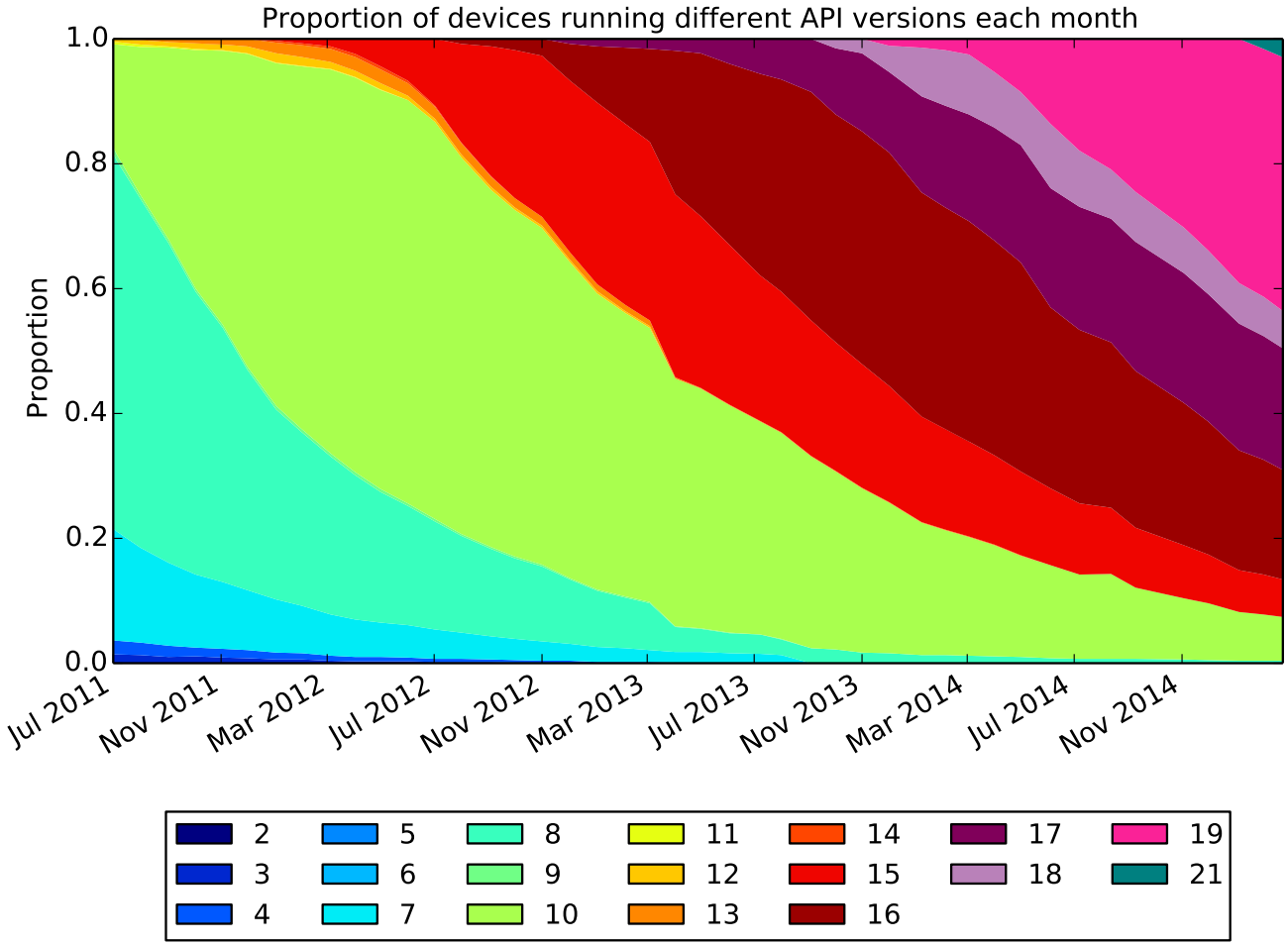
1. Change the API and require apps to recompile (2012)
2. Block calls to `.getClass` (2014)

	target API < 17	target API $\geq$ 17
API < 17	Vulnerable	Vulnerable
API $\geq$ 17 and OS < 4.4.3	Vulnerable	Safe
OS > 4.4.3	Safe(ish)	Safe

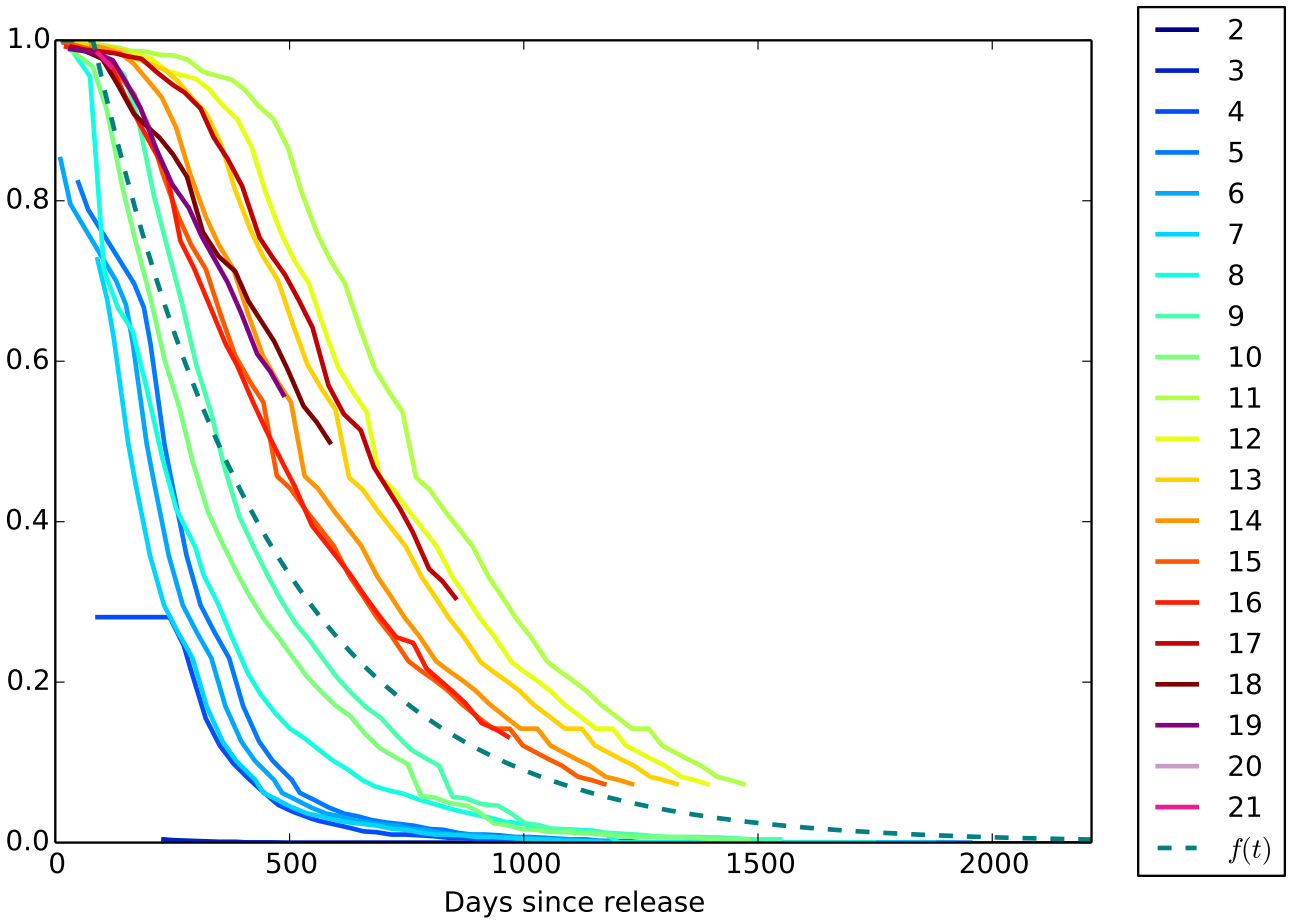
In the Chrome WebView included in Android 4.4.3 calls to `.getClass` from JavaScript are blocked which blocks the most obvious attack path, this still requires an OS upgrade but is a single sided fix.

How effective is this first approach at fixing the vulnerability?

70% in the bottom 2/3 of the table, 60% in the left half of the table



Google provides data on the distribution of API versions for most months since December 2009.
They publish this so that developers know what proportion of user devices support different features.
I have collated this by going through old blog posts and visiting the site each month.
We could also plot this differently to compare how API versions get deployed.



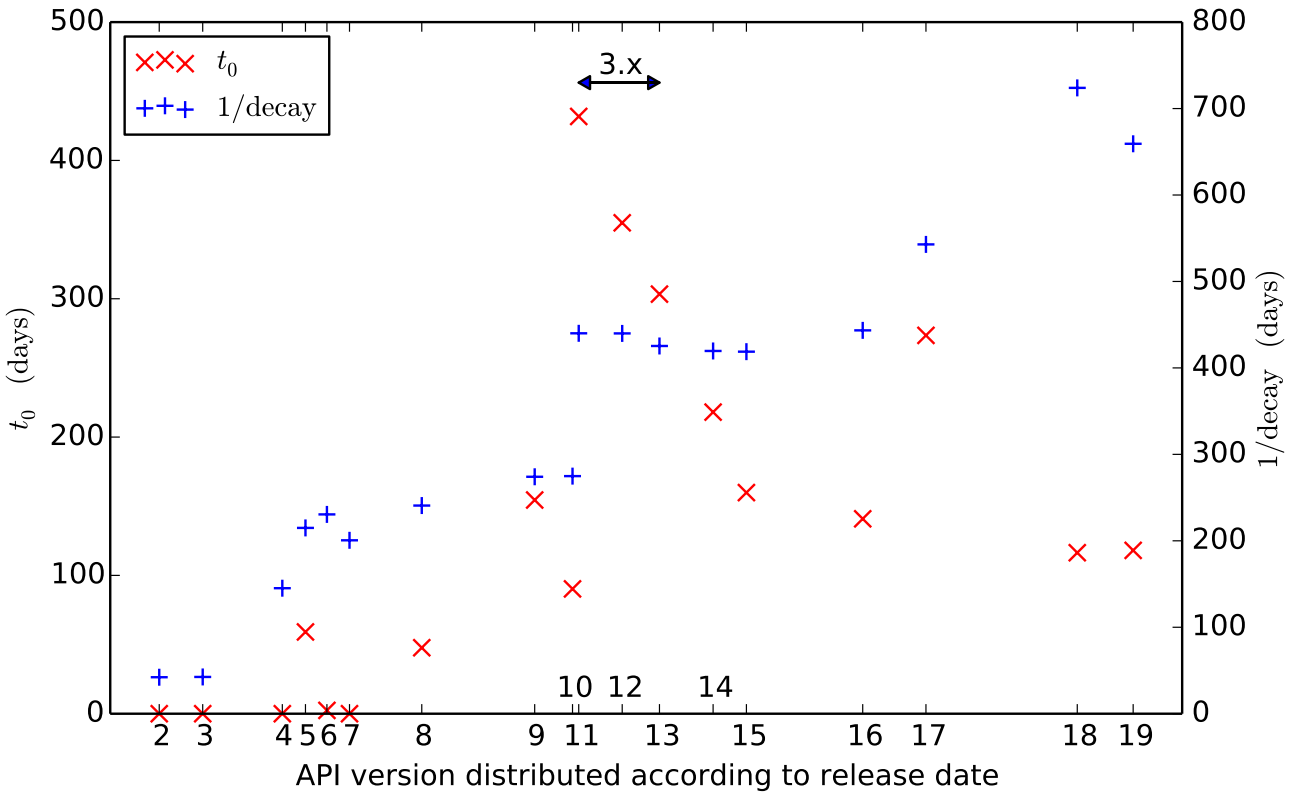
By taking the proportion of devices not upgraded to a particular API version and normalising the dates to be days since release we can see how the behaviour of the change in proportion of not upgraded devices changes with different API versions.
This shows similar looking curves, they look like exponential decay curves with a slow start.
So we fitted a simple curve to them shown here $f(x)$
This is reasonable because at first the new version is not deployed as it must be tested first and there are many independent parties deploying the updates.

Meaningful curve fit

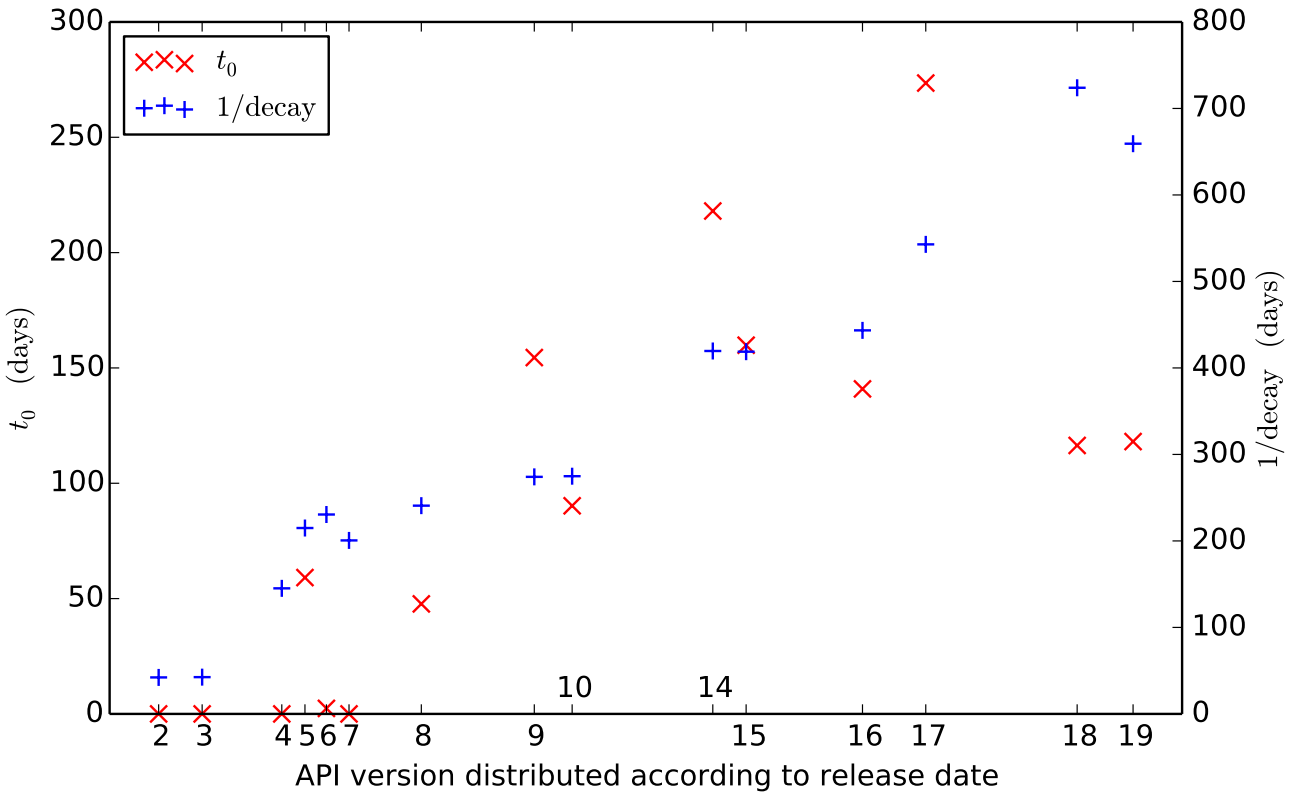
$f(t)$: a combination of an exponential function together with a delay t_0 which offsets the start time:

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

We use this simple curve fit to give us meaningful parameters when we fit it to the data.
The delay t_0 before anything happens and the rate of deployment when it does.
While this gives us a good fit in general for the average curve, there are actually big differences between the different API versions and this has changed over time.



This graph shows the parameters fitted to the curve to each API version.
The API versions are distributed according to their release date to show how close they were chronologically.
Android versions 3.x targeted tablets and did not see widespread deployment and so are outliers on the plot.
On this plot higher values for t_0 and for 1/decay are bad as they indicate slower deployment.
The graph shows a trend of increasing delays and decreasing rates of deployment: it is getting worse not better.



So we exclude Android 3.x.
On this plot higher values for t_0 and for 1/decay are bad as they indicate slower deployment.
The graph shows a trend of increasing delays and decreasing rates of deployment: it is getting worse not better.
We looked at what this means for API version 17. Despite being released in 2012, we don't expect it to be fully deployed until 2018!

Apps are vulnerable (and have not upgraded)

- We scanned 102 174 apps.
- 59% of apps which could be vulnerable had not upgraded their target API version.
- On an outdated device vulnerable apps were started 1.38 ± 0.11 times a day.
- On an up to date device vulnerable apps were started 0.6 ± 0.0 times a day.

We scanned 102 174 apps and found which ones used the JavaScript-to-Java interface and whether they were vulnerable. Having not upgraded, they are always vulnerable.

Two sided upgrade means that even on an updated device users are exposed to this vulnerability every other day

Conclusion

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

Two sided fixes are hard for API vulnerabilities, even when there is one coordinating party (Google) who has a strong influence on both sides. Fixing it takes 5.2 ± 1.2 years.

- Daniel R. Thomas drt24@cam.ac.uk
  5017 A1EC 0B29 08E3 CF64 7CCD 5514 35D5 D749 33D9
- Alastair R. Beresford arb33@cam.ac.uk
  9217 482D D647 8641 44BA 10D8 83F4 9FBF 1144 D9B3

- Install Device Analyzer for Android
  https://deviceanalyzer.cl.cam.ac.uk/

It is not really surprising that two sided fixes to API vulnerabilities are hard, but you might think that Google was better placed with Android to solve the problem. 5.2 ± 1.2 years is a long time.