Android architecture: attacking the weak points

Steve Mansfield-Devine

Steve Mansfield-Devine, editor, *Network Security*

It may not just be its popularity that has made Android a target for attackers and cyber-criminals. It's arguable that the very nature of the platform lends itself to manipulation and subversion. There are purely technical issues, such as the way communications between apps are handled. And there are problems with the ecosystem – not least how the OS gets updated and potential issues in the future with advertising.

Much of this stems from Google's desire to create a platform that is relatively open – relative, that is, to Apple's 'walled garden' for iOS. By eschewing rigid control, Google has created an environment that is highly attractive to many developers, app vendors and users (especially those of the geek persuasion). But the openness and flexibility have also introduced weak points that are being mercilessly exploited. In this second article in our series of three on Android security, we'll look more closely at the platform.

Fragmentation issues

Many of the exploits we see on Android work only on older versions of the operating system – but that's not a problem for the bad guys. The fact is that it's a notoriously fragmented user base. Developers occasionally complain about the complicating factors such as a wide variety of screen resolutions and sizes. These problems stem from the large number of vendors producing devices to no particular hardware standards. But, from a security perspective, the biggest issue is the range of OS versions in current use.

Many of the trojanised apps that Google has been forced to remove from its Play marketplace – and which continue to pop up in less rigorously policed third-party app stores – exploited vulnerabilities that had been fixed in the versions of Android that were already current when the malware was released. In fact, the malicious apps would only work on OS versions that were (by tech standards) considerably out of date. Yet because manufacturers were still pushing out products with those old versions, and because such a large percentage of users never get around to upgrading, the malware was able to be highly effective. The cyber-criminals simply don't need to care that they haven't found exploits for the current OS. To some people, this might sound depressingly similar to the way so many users have held onto insufficiently patched installations of Windows XP while ignoring the charms of Vista and Windows 7.

While Jelly Bean (version 4.1) is the most current version, only 1.8% of devices are using it (as on 1 Oct 2012).[1] Indeed, only a minority (23.7%) of device manufacturers have made it as far as Ice Cream Sandwich (4.0). Google officially unveiled Jelly Bean in late June 2012, with the firm's own device, the Nexus 7, being the first to run it. But the company is not in control of how updates are pushed out to most users – that's in the hands of handset vendors and/or mobile service operators. Samsung started its Jelly Bean roll-out in Europe – starting with Poland – at the end of September 2012. But this is for a limited range of devices and even then would not reach users whose mobile operators have chosen, for one reason or another, not to promulgate the update. Other vendors are also rolling out the update, but again to limited ranges of devices and in a very sporadic fashion. Many devices will not be able to run the new version of the OS.

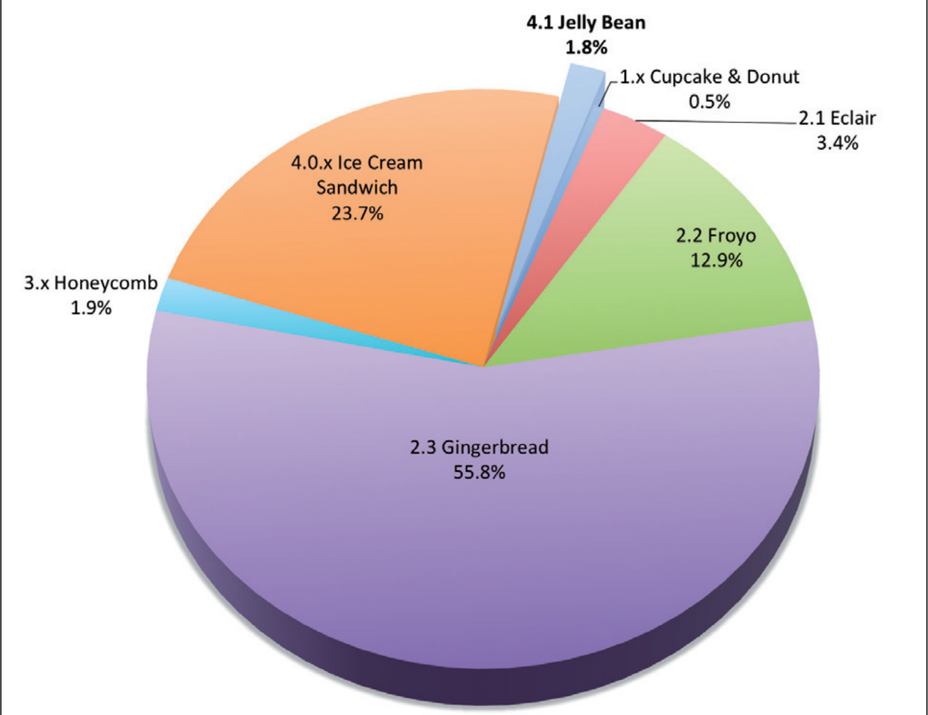
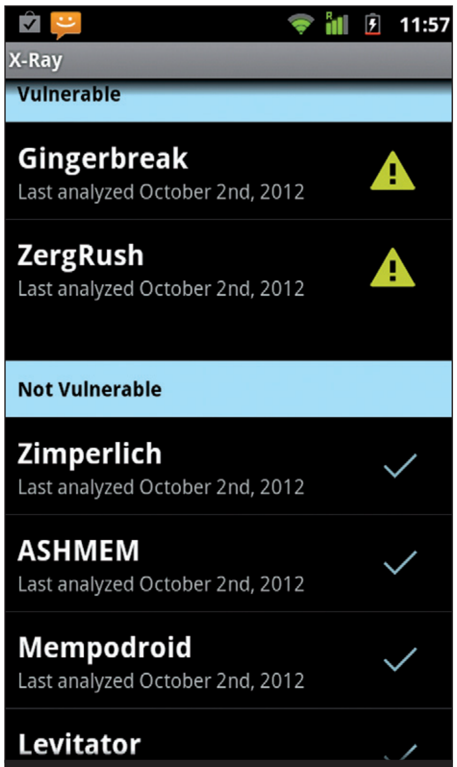


Figure 1: Android versions in use, as of 1 Oct 2012. Source: Google.



X-Ray

Vulnerable

Gingerbreak
Last analyzed October 2nd, 2012

ZergRush
Last analyzed October 2nd, 2012

Not Vulnerable

Zimperlich
Last analyzed October 2nd, 2012

ASHMEM
Last analyzed October 2nd, 2012

Mempodroid
Last analyzed October 2nd, 2012

Levitator

Figure 2: X-Ray from Duo Security on a device running Android 2.3 – showing it is vulnerable to two known exploits.

Multiple points of delay

There are multiple points at which an OS upgrade can be delayed or prevented. For example, as this article was being written, UK mobile operator O2 announced that customers with certain Sony Xperia handsets – the Neo, Arc and Ray models – would not be updated to Android 4.0 ICS because of performance issues. "These issues were present on three separate versions of the Android 4.0 software we tested and are caused by the software having more advanced hardware requirements than previous versions," the company said. "Because the software affects the phone's performance in this way and because you can't revert back to an earlier version of Android without having your phone completely restored, we have decided not to approve the update." Users of these handsets with this operator are now faced with the prospect of having to root their phones if they want to upgrade, in order to bypass the operator. And that's to get to a level of Android that is still one version old.

The problem isn't confined to updating to new versions of the OS: vendors and mobile operators are also somewhat slow and inconsistent when it comes to pushing out patches. Duo Security recently launched the X-Ray, which scans devices for known vulnerabilities, and early reports from devices reporting back suggests that half of Android devices have unpatched flaws that can be readily exploited.[2]

Security firm Lookout did research that suggested that the time taken for 50% of Android users to patch their handsets was eight to 10 months. This 'half-life' varied from one vulnerability to the next: half of users had patched against the Exploit rootkit within 42 weeks of the patch being available, while it was 'only' 30 weeks for WebKit NaN (CVE-2010-1807).[3] In fact, a significant proportion of users are more likely to buy a new device before they upgrade to a new version of the OS or even patch.

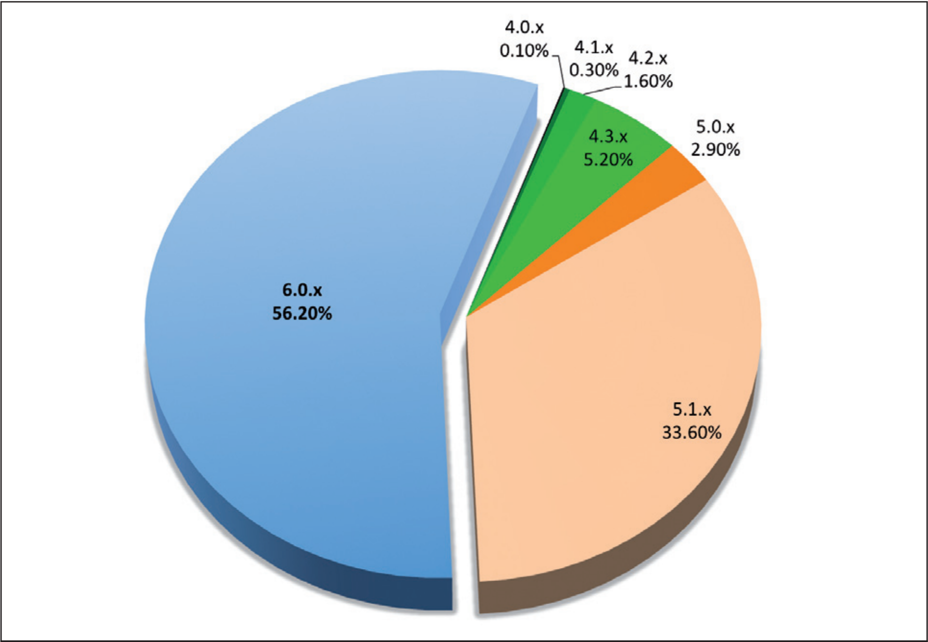


Figure 3: Versions of Apple's iOS currently in use. Source: David Smith.

Stark contrast with iOS

All of this contrasts starkly with iOS. In June 2012, at a developers conference, Apple claimed that over 80% of iPhone and iPad users were running iOS 5, which was then the most up-to-date version of the mobile OS. And that's actually a much lower figure than some other organisations were seeing. David Smith, who produces a number of iOS apps, publishes stats on his blog showing the iOS versions of people downloading his software.[4] As of 2 Oct 2012, more than 90% were using iOS 5 or iOS 6. And these figures mask the fact that many users of older devices, such as the iPhone 3G, couldn't adopt the new version. So the figure for users of later devices is probably much higher.

"Android is particularly popular among geeks. This is a user base that is not afraid to bypass restrictions and security features"

The recently launched iOS 6 saw an even more rapid uptake than iOS 5. Apple claimed that by the end of the weekend on which it was released, 100 million devices were running the new OS.[5] One app developer reported that 60% of traffic to its servers was coming from iOS6 devices within a week of the launch. Virtually all of the rest was from the most up-to-date version of iOS 5 and the tiny amount of other traffic was largely from older versions of iOS 5.

The rapid uptake is partly due to the enticement of new features in the latest version. Also, iTunes is configured by default to check for updates and encourages users to patch. This is much easier to do from within Apple's 'walled garden' in which all users employ common software. However, not everyone uses iTunes for updates: with iOS 5, Apple adopted Over The Air (OTA) patching, making it even easier and more likely that users will update. That said, research by mobile security firm Mobilisafe claimed that, while the OS might be up to date, 56% of the iOS devices it saw were running out-of-date firmware.[6]

Root users

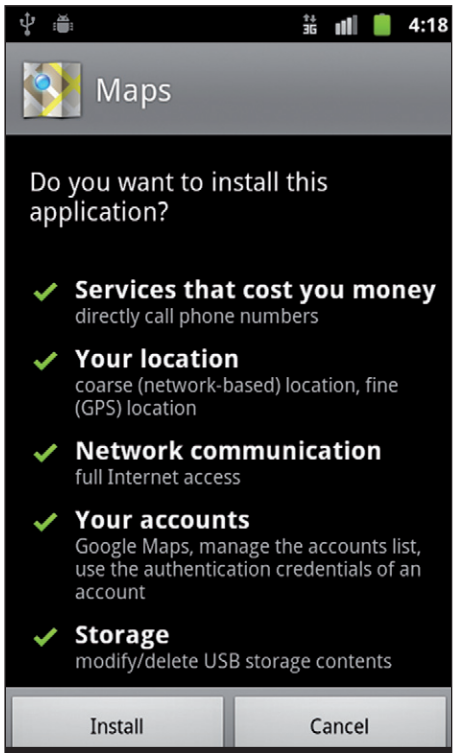
While the majority of infections happen as a result of users downloading trojanised apps – often compromised or fake versions of genuine and sometimes well-known titles – another key vector is the trojanised ROM.

Android is particularly popular among geeks. This is a user base that is not afraid to bypass restrictions and security features to gain full access to the platform. (For the sake of balance, it's worth pointing out that Apple iPhone users are also quite keen to 'jailbreak' their systems in order to use alternative sources of apps.) Such users will frequently 'root' their devices, allowing superuser access to the operating system. This also opens the devices to the risk of malicious software being able to escalate privileges more easily.

Re-ROMing is also common. Many Android devices are delivered with ROM-based software in which the vendor (often the mobile network operator) has added its own, heavily branded user interface, additional software and so on. This is regarded as intrusive 'bloatware' by many users who prefer the unadorned Android OS. Downloading new ROM software, which is then written into non-volatile Flash memory, is therefore very common. But little of this software comes from verifiable or trusted sources and it is difficult for users to be sure the ROM software has not been compromised.

Permissions model

Android's Linux roots are evident in the permissions model it uses to provide core security features.[7] Every app on an Android device runs as a separate 'user' account – for example, 'app_1', 'app_2' etc – with a unique User ID (UID) and Group ID (GID). (This contrasts with the iOS model in which all apps share the same UID but are sandboxed.) Each app may also belong to other groups depending on the permissions granted by the user – the app requests these permissions the first time it is run, although many users will simply 'click through' this step. This is especially true of the Internet permission, as most people will simply assume the app needs this for updating, access to online services and so on.


Maps

Do you want to install this application?

✓ **Services that cost you money**
directly call phone numbers

✓ **Your location**
coarse (network-based) location, fine (GPS) location

✓ **Network communication**
full Internet access

✓ **Your accounts**
Google Maps, manage the accounts list, use the authentication credentials of an account

✓ **Storage**
modify/delete USB storage contents

Figure 4: An app (in this case, Maps) requesting permissions from the user the first time it is run.

"Any malicious code will only have the permissions of the exploited app. Escalating privileges requires exploiting the kernel, which is a much tougher proposition"

This is the main 'sandboxing' method. For the most part, then, any given app is unable to access the files of the other apps because it doesn't have the necessary read, write or execute permissions. There are a few exceptions in which processes run with shared UIDs, but these are very limited. This provides application resource isolation – each app has its own directories for data, preference settings, caches and databases. Each app also has a manifest file – AndroidManifest.xml – that defines configuration and security settings. It is possible for an app to write data to its own directory but set the file permissions to 'world readable'. This subverts the sandboxing model and is regarded as bad practice, but it does happen, particularly with less experienced developers.

With apps running native code, there's always the potential for memory corruption vulnerabilities, such as buffer overflows. But any malicious code that gets run by exploiting such a vulnerability will only have the permissions of the exploited app. Escalating privileges – ie, getting root – requires exploiting the kernel, which is a much tougher proposition. The kernel and main system files – including libraries, application runtime, application framework and system applications – are kept in a separate, read-only partition.

Apps have access to only a limited number of system resources by default. Some capabilities deliberately have no API – for example, manipulating the SIM card. Other functions, that do have APIs, require that the user gives permission for an app to access them. These include:

- Camera
- Location data (GPS)
- Bluetooth
- Telephony
- SMS/MMS
- Network connections

The app also needs to declare its requirement for these capabilities in its manifest.

However, any app can see verbose system info, regardless of permissions assigned to it. This includes a list of installed apps, platform/device info and device identity info (IMEI, IMSI, phone number). This is information that could be used by a malicious app to profile a device in order to work out if it has, for example, exploitable software, or for tracking the user.

Any app can also read the contents of any SD card present because world-readable permissions are automatically given to any file written to the card. Developers, unaware of this, often use the SD card for data storage. A malicious app could upload photos, videos,

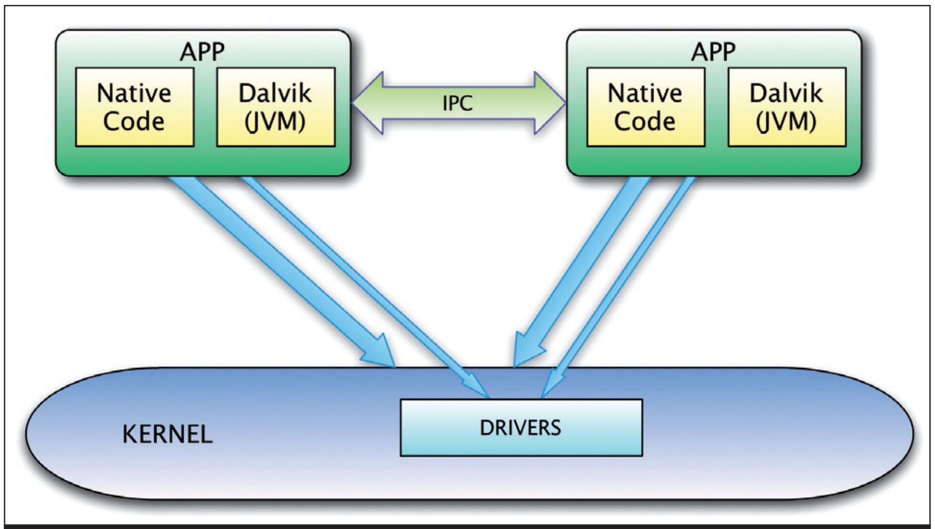
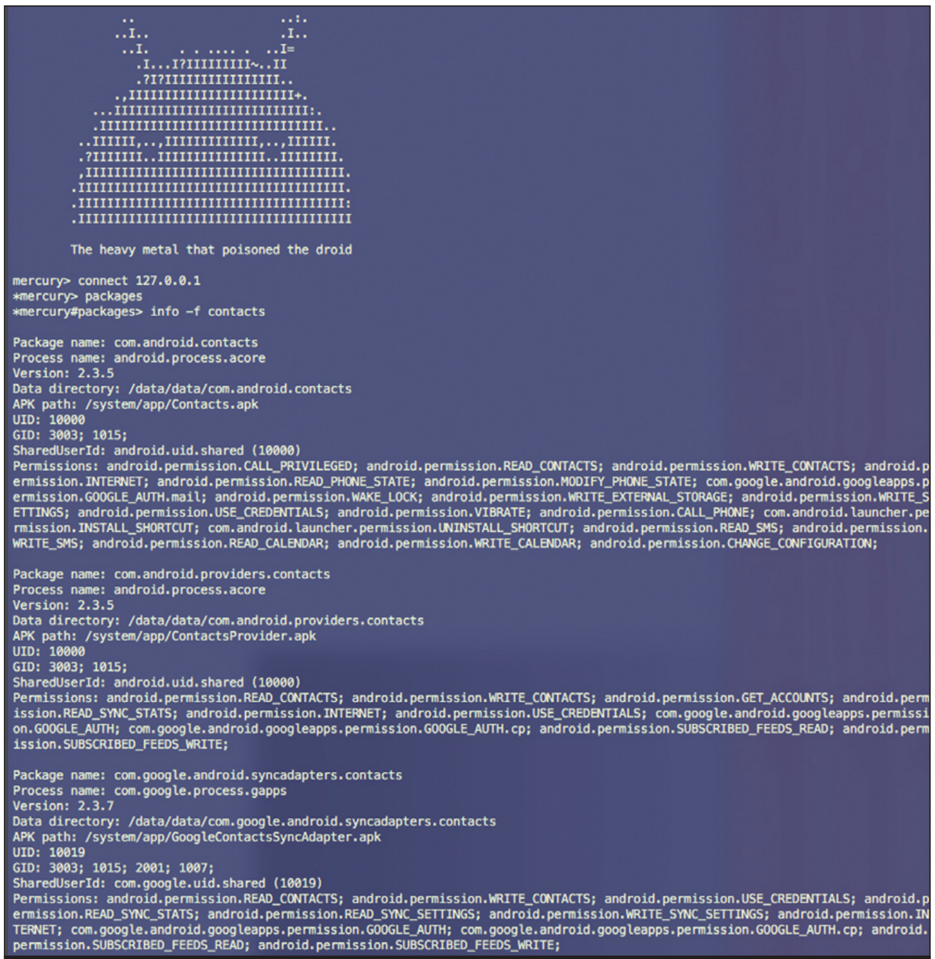


Figure 5: The Android security model, in which apps exchange information using the Inter-Process Communication (IPC) mechanism.

documents and so on – all it needs is Internet permission.

During development, apps are set to be debuggable. Many developers forget to set 'debuggable=false' in the manifest when releasing the app. This allows a malicious app to pose as a debugger and get the app to do things it shouldn't. By opening the @jdwp-control socket, from an unprivileged context, any debuggable apps will connect. Research by Tyrone Erasmus of MWR Labs, of whom more in a moment, suggested that 5% of the apps available in Google Play have the debugging switch still enabled.



```
                 ..                    ..:.
                ..I..                  .I..
                 ..I.    . . .... .  ..I=
                   .I...I?IIIIIIIII~..II
                   .?I?IIIIIIIIIIIIIII..
                ..,IIIIIIIIIIIIIIIIIIIIIII+.
             ...IIIIIIIIIIIIIIIIIIIIIIIIIIII:.
             .IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII..
           ..IIIIII,..,IIIIIIIIIIIII,..,IIIIII.
           .?IIIIIII..IIIIIIIIIIIIIII..IIIIIIII.
           ,IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII.
          .IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII.
          .IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII:
          .IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII

          The heavy metal that poisoned the droid

mercury> connect 127.0.0.1
*mercury> packages
*mercury#packages> info -f contacts

Package name: com.android.contacts
Process name: android.process.acore
Version: 2.3.5
Data directory: /data/data/com.android.contacts
APK path: /system/app/Contacts.apk
UID: 10000
GID: 3003; 1015;
SharedUserId: android.uid.shared (10000)
Permissions: android.permission.CALL_PRIVILEGED; android.permission.READ_CONTACTS; android.permission.WRITE_CONTACTS; android.p
ermission.INTERNET; android.permission.READ_PHONE_STATE; android.permission.MODIFY_PHONE_STATE; com.google.android.googleapps.p
ermission.GOOGLE_AUTH.mail; android.permission.WAKE_LOCK; android.permission.WRITE_EXTERNAL_STORAGE; android.permission.WRITE_S
ETTINGS; android.permission.USE_CREDENTIALS; android.permission.VIBRATE; android.permission.CALL_PHONE; com.android.launcher.pe
rmission.INSTALL_SHORTCUT; com.android.launcher.permission.UNINSTALL_SHORTCUT; android.permission.READ_SMS; android.permission.
WRITE_SMS; android.permission.READ_CALENDAR; android.permission.WRITE_CALENDAR; android.permission.CHANGE_CONFIGURATION;

Package name: com.android.providers.contacts
Process name: android.process.acore
Version: 2.3.5
Data directory: /data/data/com.android.providers.contacts
APK path: /system/app/ContactsProvider.apk
UID: 10000
GID: 3003; 1015;
SharedUserId: android.uid.shared (10000)
Permissions: android.permission.READ_CONTACTS; android.permission.WRITE_CONTACTS; android.permission.GET_ACCOUNTS; android.perm
ission.READ_SYNC_STATS; android.permission.INTERNET; android.permission.USE_CREDENTIALS; com.google.android.googleapps.permissi
on.GOOGLE_AUTH; com.google.android.googleapps.permission.GOOGLE_AUTH.cp; android.permission.SUBSCRIBED_FEEDS_READ; android.perm
ission.SUBSCRIBED_FEEDS_WRITE;

Package name: com.google.android.syncadapters.contacts
Process name: com.google.process.gapps
Version: 2.3.7
Data directory: /data/data/com.google.android.syncadapters.contacts
APK path: /system/app/GoogleContactsSyncAdapter.apk
UID: 10019
GID: 3003; 1015; 2001; 1007;
SharedUserId: com.google.uid.shared (10019)
Permissions: android.permission.READ_CONTACTS; android.permission.WRITE_CONTACTS; android.permission.USE_CREDENTIALS; android.p
ermission.READ_SYNC_STATS; android.permission.READ_SYNC_SETTINGS; android.permission.WRITE_SYNC_SETTINGS; android.permission.IN
TERNET; com.google.android.googleapps.permission.GOOGLE_AUTH; com.google.android.googleapps.permission.GOOGLE_AUTH.cp; android.
permission.SUBSCRIBED_FEEDS_READ; android.permission.SUBSCRIBED_FEEDS_WRITE;
```

Figure 6: The Mercury framework. This shows the app providing basic information on apps that have the word 'contacts' in their name.

Communicating apps

Unlike multiple users on a normal Unix system, there are many cases in which one app needs to talk to another – for example, when an email app needs access to an address book. This capability is provided by the Inter-Process Communication (IPC) mechanism, which has four endpoints:

- Binder – for remote procedure calls using a custom Linux driver.
- Services – for running background processes. These can provide interfaces to other apps using the Binder mechanism.
- Intents – a message object that alerts the system to resources it requires and also for notifications.
- ContentProviders – which provide access to data. Often these run using SQLite, but even where they don't, the content providers commonly present an interface that supports standard SQL queries.

The availability of these endpoints is defined in the app's manifest, through the use of the export flag (eg, export=true) or the use of <intent-filter>. One problem sometimes inadvertently run into by developers is that the content provider endpoint is exported by default. If this shouldn't be available to other apps this has to be explicitly stated by setting the export flag for that endpoint to false in the manifest. It's very easy to forget (or not know about) this requirement, with the result that the endpoint is made available without the developer ever realising it.

Erasmus at MWR Labs has shown how the IPC mechanism can be plundered and exploited. MWR has developed a discovery framework, Mercury, that provides a means to analyse apps for potential data leaks and vulnerabilities, which Erasmus presented at Black Hat Europe 2012.[8] The chief reason for following this line of work, he said at the presentation, is that Android has, "very interesting attack surfaces. There's quite a lot of attack surface in the way that applications communicate with each other across the sandbox using IPC. Also, because it's good old Linux there's always the chance of priv[ilege] escalating up to root."

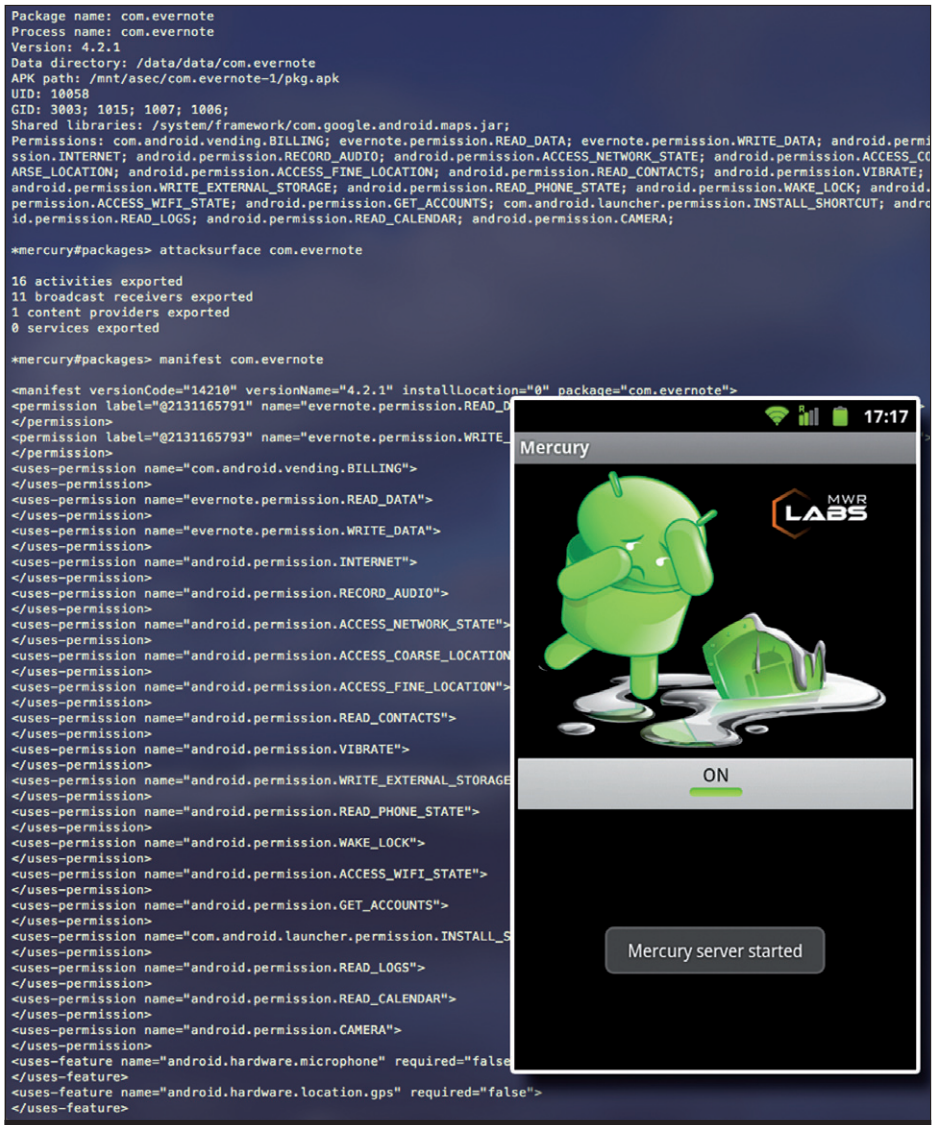
Cross-application exploitation

Erasmus' key interest is in cross-application exploitation – finding out what one app can do to another without needing special privileges. If a piece of malware can access data from another app, for example, the only permission it might need is the ability to access the Internet, in order to upload the stolen data to a Command and Control (C&C) server. As we've mentioned, users will often give apps Internet access without thinking. The Mercury framework needs only Internet permissions and can be seen as a kind of malware proof-of-concept tool.

"Android is unique," said Erasmus in his presentation. "Developers have not, in my opinion, got a full grasp on it yet … it's ripe picking for anyone looking for vulnerabilities."

In July 2011, Erasmus found a vulnerability in Dropbox and wrote a proof-of-concept exploit that caused Dropbox to upload its login credential to its Public folder, which is world-accessible. He was motivated to write Mercury because of frustration with existing approaches. Analysing code for exploitation often uses a static approach. For example, the manifest file can be extracted from apps using standard Android SDK tools. This allows you to see entry points into code. You can then decompile code to at least bytecode or even recompiled source code to understand what's running behind the entry points. That allows you to write attack code – which then needs to be tested and amended. This is a very slow process.

The dynamic approach is to use code running on the device to analyse flaws and attempt exploits on the fly. This is how Mercury works. It makes use of reusable modules because many apps share common attack vectors. It runs as a server on the Android device and you connect to it using a Python-based client on a PC. It very quickly provides info on any app, including the permissions it was given when installed. It's also possible to run filters – eg, to get a quick list of all apps that have the 'install packages' permission.


```
Package name: com.evernote
Process name: com.evernote
Version: 4.2.1
Data directory: /data/data/com.evernote
APK path: /mnt/asec/com.evernote-1/pkg.apk
UID: 10058
GID: 3003; 1015; 1007; 1006;
Shared libraries: /system/framework/com.google.android.maps.jar;
Permissions: com.android.vending.BILLING; evernote.permission.READ_DATA; evernote.permission.WRITE_DATA; android.permi
ssion.INTERNET; android.permission.RECORD_AUDIO; android.permission.ACCESS_NETWORK_STATE; android.permission.ACCESS_CO
ARSE_LOCATION; android.permission.ACCESS_FINE_LOCATION; android.permission.READ_CONTACTS; android.permission.VIBRATE;
android.permission.WRITE_EXTERNAL_STORAGE; android.permission.READ_PHONE_STATE; android.permission.WAKE_LOCK; android.
permission.ACCESS_WIFI_STATE; android.permission.GET_ACCOUNTS; com.android.launcher.permission.INSTALL_SHORTCUT; andro
id.permission.READ_LOGS; android.permission.READ_CALENDAR; android.permission.CAMERA;

*mercury#packages> attacksurface com.evernote

16 activities exported
11 broadcast receivers exported
1 content providers exported
0 services exported

*mercury#packages> manifest com.evernote

<manifest versionCode="14210" versionName="4.2.1" installLocation="0" package="com.evernote">
<permission label="@2131165791" name="evernote.permission.READ_D
</permission>
<permission label="@2131165793" name="evernote.permission.WRITE_
</permission>
<uses-permission name="com.android.vending.BILLING">
</uses-permission>
<uses-permission name="evernote.permission.READ_DATA">
</uses-permission>
<uses-permission name="evernote.permission.WRITE_DATA">
</uses-permission>
<uses-permission name="android.permission.INTERNET">
</uses-permission>
<uses-permission name="android.permission.RECORD_AUDIO">
</uses-permission>
<uses-permission name="android.permission.ACCESS_NETWORK_STATE">
</uses-permission>
<uses-permission name="android.permission.ACCESS_COARSE_LOCATION
</uses-permission>
<uses-permission name="android.permission.ACCESS_FINE_LOCATION">
</uses-permission>
<uses-permission name="android.permission.READ_CONTACTS">
</uses-permission>
<uses-permission name="android.permission.VIBRATE">
</uses-permission>
<uses-permission name="android.permission.WRITE_EXTERNAL_STORAGE
</uses-permission>
<uses-permission name="android.permission.READ_PHONE_STATE">
</uses-permission>
<uses-permission name="android.permission.WAKE_LOCK">
</uses-permission>
<uses-permission name="android.permission.ACCESS_WIFI_STATE">
</uses-permission>
<uses-permission name="android.permission.GET_ACCOUNTS">
</uses-permission>
<uses-permission name="com.android.launcher.permission.INSTALL_S
</uses-permission>
<uses-permission name="android.permission.READ_LOGS">
</uses-permission>
<uses-permission name="android.permission.READ_CALENDAR">
</uses-permission>
<uses-permission name="android.permission.CAMERA">
</uses-permission>
<uses-feature name="android.hardware.microphone" required="false
</uses-feature>
<uses-feature name="android.hardware.location.gps" required="false">
</uses-feature>
```

Figure 7: The Mercury framework showing information about an app (in this case Evernote), including permissions, UID and GID, endpoints exported and the contents of the manifest file. Inset: the Mercury server app running on an Android device.

During his demonstration of Mercury, Erasmus used simple commands to find apps with content provider endpoints requiring no (null) permissions to read or write. The tool also allows you to find apps that leak passwords (sometimes hard-coded online service passwords in the manifest), personal data or messages. For example, an app called IM, which was provided with a device that Erasmus tested, leaks instant messages from Google Talk, Windows Live Messenger and Yahoo Messenger by allowing you to read from the app's database without needing any privileges. The SocialHub app similarly leaks messages from social networking services, including Facebook, MySpace, Twitter and LinkedIn.

Mercury has a tool to search for content URIs used by apps, allowing a researcher to dig deeper and pull content from the app database. It can do this because the app binaries are readable from an unprivileged context. Another area of vulnerability is logs: by dumping log information you can sometimes find data containing information about emails, SMS messages, phone calls and so on, both sent and received. In his presentation, Erasmus also showed how Mercury can be used to mount SQL injection attacks against apps' databases: in one instance, he dumped the contents of SMS messages. Normally, SMS requires the android.permission.READ_SMS permission. But by querying other authorities in the same package as the SMS database you may be able to find other content providers in the same package that do not require permissions.

If a single database is shared by multiple content providers, you can go in via a provider with null permissions and use SQL injection.

The storage of various settings is another weak spot: Erasmus was able to pull from Settings Storage the SSID and WPA2 password of a portable wifi hotspot.

"While Android theoretically has a strong permissions model, it is frequently undermined by how developers produce their apps"

With no special permissions, using the Mercury framework (which has no higher privileges than any other app) Erasmus was able to show how he could obtain: email address and password; email contents; contents of SMS messages; IM messages and contacts list; social networking messages; call logs; notes; current city; portable wifi hotspot credentials; and contents of the SD card. With this information, it's possible to build a very detailed user profile, as well as stealing private and important information.

Most of this info is gleaned from third-party apps, although many are very common as they're installed by default by OEMs. Also, they show how, while Android theoretically has a strong permissions model, it is frequently undermined by how developers produce their apps, with content providers not requiring permissions. In fact, with nothing more than the Internet permission, Erasmus showed how it was easy to open a shell to remote location. He used the ever-popular Netcat for the demonstration.

Advertising channels

For all its success on the desktop, advertising hasn't yet been used as a major cybercrime vector on Android, but some believe it's ripe for the plucking. The lack of activity may be down to simple financial considerations. We saw in the previous article that cyber-criminals are guided by a number of factors when mounting a malware campaign, which include: the need to remain anonymous; the low cost of exploitation; and a large number of targets. To this, you can add the ability to script.

Although app developers can choose from a number of advertising services when it comes to choosing what to embed in their software, by far the biggest two are those run by Apple and Google for their own platforms. Given that the size of these ad networks relates directly to the number of targets, if ad-based malware is going to really become an issue on mobile platforms it would probably need to involve Apple's iAd or Google's Admob service. The Mobile Exploit Intelligence Project (MEIP) run by Dan Guido of Trail of Bits and Mike Arpaia at iSEC Partners compared these as potential malware channels and the result is instructive.

Joining Apple's iAd service requires an up-front payment of $300,000 and Apple insists on verifying your identity, so that is two strikes against it as far as cyber-criminals are concerned. As it's HTML5-based, iAd is scriptable, though. By comparison, joining Admob simply requires the filling in of an online form, with no checks, and a payment of $50. Ads are simple image or text, so there's no scope to insert JavaScript – the 'exploit' would be a simple link to malicious content, probably ruling out the automatic compromise of systems, although if the device's owner has enabled app installation from 'Unknown sources' (not uncommon) a simple URL can result in the automatic downloading of an app. On the whole, Admob looks highly attractive to cyber-criminals while iAd is a definite non-runner.

Number of targets

The final issue, then, is the available number of targets. And this is why we haven't seen much in the way of malicious advertising on mobile platforms. We've already seen that web browsing doesn't happen all that much on smartphones. And when websites serve special versions of themselves for mobile users, they generally omit the ads because of the limited screen space of mobile devices. Finally, it's a known fact that mobile users simply don't click advertising links all that much.

So mobile advertising simply isn't all that appealing as a malware vector – yet. The MEIP noted one campaign – GGTracker, which was used for SMS fraud. It was released mid-2011 but hasn't been repeated since, so it's possible it didn't work well.

"We think [advertising] is mildly attractive and it may be revisited in the future but only if the incentives change," said MEIP's Guido.

If it does change, one issue that could become a serious concern is that ads are most commonly delivered using third-party ad libraries. And developers may not be all that scrupulous in checking the trustworthiness of the libraries they use to monetise their apps. Already, some libraries are the cause of privacy concerns. An analysis by a team at NC State found that, of 100,000 apps in Google Play they tested, 48,139 tracked the device's GPS location and 4,190 passed this information to advertisers, 18,575 captured and shared the IMEI number of the device, and 4,047 captured the user's phone number.[9] Of even greater concern, 297 apps ran ad code that could, itself, execute code downloaded over the Internet. This means that even perfectly innocent apps could be a conduit for malware.

BitDefender recently noted a large increase in "aggressive adware" on Android. By itself, adware – where the user is bombarded with pop-ups, often to encourage them to visit (potentially malicious) websites – doesn't count as a compromise of the device. BitDefender reckons as many as 90% of the free apps in Google Play contain adware. But a very large proportion of them – around 75% of free apps – fall into BitDefender's 'aggressive' category where the app may cause configuration changes to the device and push notifications in such a way that performance can degrade. One of the most common pieces of adware is the game Ant Smasher, which BitDefender says has been downloaded more than 50 million times.

App development

Software development is where security starts, and much could be done to improve the Android security landscape by improving the education of developers and implementing tighter controls during the application development lifecycle.

When we turn to Android software development, there's more fragmentation to be found. Development for Android is mostly done in Java, with perhaps some C (on iOS, the main language is Objective-C). But some apps, especially web-based ones, might also be developed with the usual mix of HTML, CSS, JavaScript and Action Script. Further scripting is possible with Python, Perl, JRuby, Lua, BeanShell, JavaScript, Tcl and shell scripts. And it's even possible to develop in Visual Basic or C#. With such a diversity of languages and platforms, it's hard to establish the kinds of procedures and standards that lead to more secure software.

"He found an astonishing level of flaws, some of which leaked critical information and others which suggested potential for exploitation"

Poor developer practices, some of them probably attributable to simple laziness, are at the root of many problems. For example, a developer may decide to have an app ask for permissions it doesn't need 'just in case' – perhaps to cut down on the amount of testing required or to allow for enhancements or upgrades later. The problem of excessive permissions is very common – more than 42% of Android apps request device access permissions they don't actually need, according to Korean firm AhnLab. Around 39% of apps demand unnecessary location information access and 33% want personal information access.

Researcher Simon Roses of Vulnex, talking at Black Hat Europe 2012, revealed details of the 100 or so apps he had examined, all available in Google Play.[10] He analysed each app for only about an hour but found an astonishing level of flaws, some of which leaked critical information and others which suggested potential for exploitation, and most of them due to poor coding. For example, in a credential manager app, he found the master password stored in clear text in an xml preferences file. He also founds apps – including finance, social networking and FTP software – with debugging features enabled, which can result, for example, in potentially exploitable information being written to logs. Google's Bouncer system, which is supposed to weed out dubious or faulty apps in Play, doesn't seem to notice or flag this.

Some apps Roses examined were vulnerable to code injection. OWASP has produced data validation libraries for Java, Objective-C and so on in its Enterprise Security ESAPI programme, so there's no excuse for errors such as not correctly validating user input.[11] Roses also found that many apps collect unnecessary amounts of information, often requesting permissions they don't need and sometimes then storing that information in insecure ways. He also noted that a lot of apps use third-party libraries, which is to be expected – but some of the libraries are somewhat obscure and developers can't be sure what vulnerabilities they may contain or what the quality of the code may be. There's also a question mark over the suitability of some of the libraries – he found one finance app that uses Facebook libraries. This could lead to baking in unnecessary vulnerabilities and attack surfaces.

Enterprise apps

Most of the apps that researchers analyse are publicly available ones, typically from Google Play. But many organisations produce their own apps, intended either for internal use, or sometimes for use by suppliers, partners or customers.

Veracode, which analyses software for security issues, has done some research in this area. Its findings aren't about the kinds of flaws being exploited (although it does test against things like the OWASP Top 10 or CWE/SANS top 25 vulnerabilities), but more about those that may lead to the exploits of tomorrow. For Android, it found that the key areas where problems arose were cryptographic issues (44%), Carriage Return/Line Feed (CRLF) injection (28%) and information leakage (10%). This is based on about 100 apps.

"It looks like Android developers don't understand how to use the crypto APIs well on the platform," said Chris Wysopal, Veracode's CTO presenting the findings at Black Hat Europe 2012. "And they're also baking in a lot of static crypto keys, which is definitely a bad idea."

Top problems were:

- Insufficient entropy (61% of apps tested), meaning no-one's using secure random number generation: "They just don't get the concept, or something like that," said Wysopal.
- The use of hard-coded crypto keys (42%): "That means that if you have possession of the binary, you have possession of the key." This is something Veracode sees a lot in Java apps.
- Information exposure through sent data (39%). This might be something like using a phone number or device ID as tokens sent in the clear.
- Information exposure through error messages (6%).

It would help if users were a little more cautious, too. Just as with desktop machines, unused or little-used software presents an unnecessary security risk. And research by Roses suggests that the average smartphone user has 65 apps installed but uses only around 15 in any given week.

About the author

Steve Mansfield-Devine is a freelance journalist specialising in infosecurity. He is the editor of Network Security *and also its sister publication* Computer Fraud & Security.

Coming next…

In the final part of this series, next month, we'll be taking a closer look at Android malware – what kinds of exploit exist and how they are delivered. We'll examine app distribution channels, and how the availability of third-party app stores has contributed to the malware problem. And we'll look at some of the tools and developments that may help to make Android safer.

References

1. 'Platform versions'. Google. Accessed 2 Oct 2012. http://developer.android.com/about/dashboards/index.html.
2. Oberheide, Jon. 'Early results from X-Ray: over 50% of Android devices are vulnerable'. The Duo Bulletin, 12 Sep 2012. Accessed Oct 2012. https://blog.duosecurity.com/2012/09/early-results-from-x-ray-over-50-of-android-devices-are-vulnerable/.
3. Wyatt, Tim. 'Inside the Android security patch lifecycle'. Lookout blog, 4 Aug 2011. Accessed Oct 2012. https://blog.lookout.com/blog/2011/08/04/inside-the-android-security-patch-lifecycle/.
4. Smith, David. 'iOS Version Stats'. Accessed 2 Oct 2012. david-smith.org.iosversionstats.
5. Friedman, Lex. 'Apple: Five million iPhone 5 sales, 100 million iOS 6'. Macworld, 24 Sep 2012. Accessed Oct 2012. www.macworld.com/article/2010503/apple-five-million-iphone-5-sales-100-million-ios-6-upgrades.html.
6. 'Research shows majority of Apple iOS devices running outdated firmware'. InfoSecurity, 7 June 2012. Accessed Aug 2012. www.infosecurity-magazine.com/view/26194/research-shows-majority-of-apple-ios-devices-running-outdated-firmware/
7. 'Android Security Overview'. Open Source Project. Accessed Aug 2012. http://source.android.com/tech/security/index.html#android-security-program-overview.
8. Mercury. MWR Labs. Accessed Oct 2012. http://labs.mwrinfosecurity.com/tools/2012/03/16/mercury/.
9. Talbot, David. 'Android ads could attacks, study warns'. Technology Review, 19 Mar 2012. Accessed Aug 2012. www.technologyreview.com/news/427274/android-ads-could-attack-study-warns/.
10. Vulnex. Accessed Oct 2012. www.vulnex.com.
11. 'Enterprise Security API'. OWASP. Accessed Oct 2012. https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API.

Cleaning up dirty disks in the cloud

Michael Jordon, Context Information Security

Michael Jordon

There must be times when cloud service providers rue the day that the term 'cloud' was chosen to describe their technologies. 'Cloud computing' sounds light and fresh, a world away from clunky old mainframes, but it also conjures up negative images for corporate users, making one think of something windblown, flimsy and vague – while also somehow opaque and mysterious.

Many people have grappled with the challenge of assessing cloud security. This is a particularly pressing issue for businesses in sectors such as financial services, which are most likely to attract the attention of malicious attackers with sophisticated tools and extensive resources at their disposal.

In 2011, Context published research following its assessment of the security offered by a number of cloud service providers.[1] This established that some providers exposed clients' data to a risk of compromise as a result of serious flaws in the implementation of their technologies. Context has been working with those providers to resolve the security issues identified and to establish best practice in securing similar cloud environments.

The research focused on services providing Infrastructure as a Service (IaaS), which uses virtualisation to provide computing resources as Virtual Private Servers (VPSs). These are the equivalent of separate dedicated physical servers, but share computing resources with other VPS nodes. Virtualisation allows multiple VPS nodes to be hosted on a single physical machine.

"An attacker could purchase a cloud node from a provider which also serves a target organisation, then could start looking for a way to launch an attack on the target's node"

The conclusion of the report was that the major security improvement required was a more complete separation between nodes. In a traditional dedicated hosted environment an attacker needs to break through the outer firewall, then work their way through web server, then application server and so on.

In the cloud, by contrast, all systems within the virtualised network reside next to each other. An attacker could purchase a cloud node from a provider which also serves a target organisation, then could start looking for a way to launch an attack on the target's node – present on the same physical machine as the node purchased by the attacker.

The Context research reviewed separation of hard disks, memory, networks, hypervisors (the node operating systems providing an abstraction interface between the physical hardware and the virtual nodes) and remote management. It discovered that some providers failed to separate the nodes through the shared