

Chapter 2

A Survey of Android Malware

In this chapter, we present a survey of existing Android malware. Particularly, with more than one year effort, we have collected a large dataset of existing Android malware. Based on this dataset, we are able to systematically characterize existing Android malware from various aspects, including their installation, activation methods, and malicious payloads.

2.1 Malware Dataset

When the very first Android malware, i.e., the *FakePlayer* malware [28], was discovered in August 2010, we realized the importance of collecting them for systematic examination. Specifically, to that end, we take two main approaches to actively collect Android malware samples. The first one is to obtain relevant information of new Android malware by following up with any Android malware announcements, threat reports, and event blog contents from existing mobile anti-virus companies and active researchers [8, 13, 16, 21, 31, 32] as exhaustively as possible and then diligently requesting malware samples from them. The second one is to crawl malware samples directly from existing Android marketplaces, including both third-party and official Android Market.¹

With more than one year effort (i.e., from August 2010 to October 2011), we have successfully collected 1260 Android malware samples in 49 malware families. In Fig. 2.1, we show the list of the 49 Android malware families in our dataset along with the time when each particular malware family is discovered. If we take a look at the Android malware history [23] from the very first Android malware *FakePlayer* in August 2010 to recent ones in October 2011, it spans slightly more than one year with around 52 Android malware families reported. Our dataset so far has 1260 samples in 49 different malware families, indicating a very decent coverage of existing Android malware.

¹ Android Market is now part of Google Play.

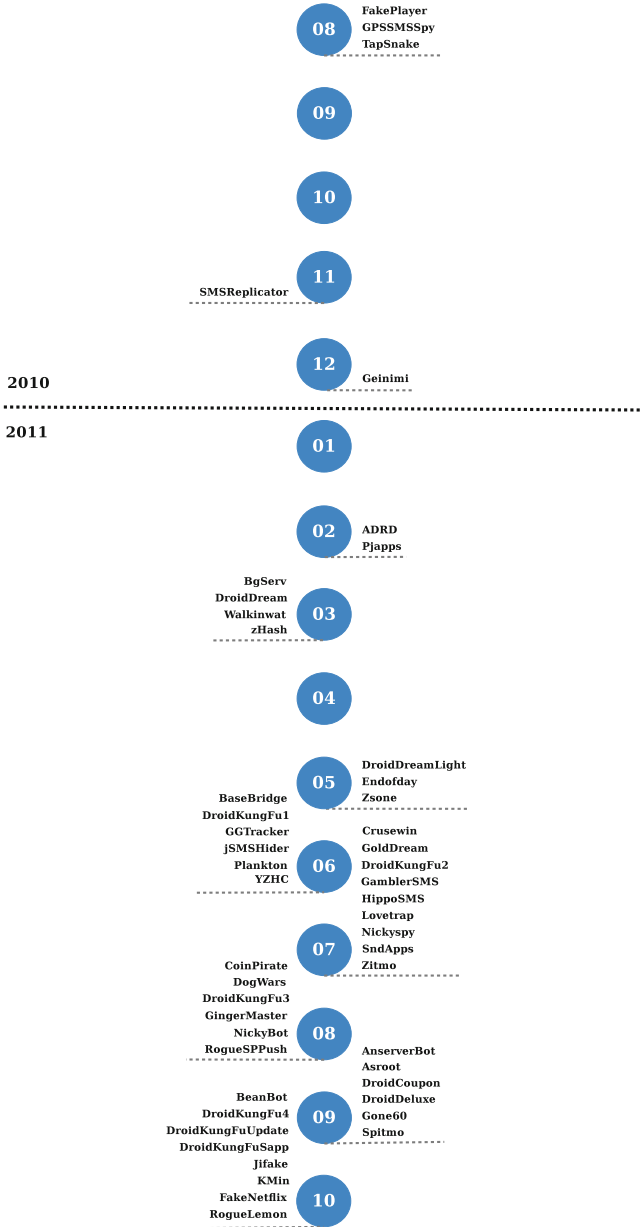


Fig. 2.1 The timeline of 49 Android malware families in our study

To engage the research community and better our defense, we released this dataset in May 2012 to community under the name *Android Malware Genome Project*. Immediately following the release, we received numerous requests and have so far

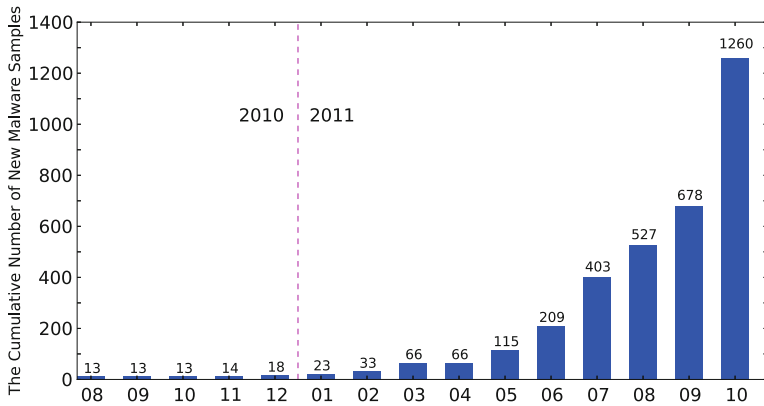


Fig. 2.2 The cumulative growth of new malware samples in our collection

shared our dataset with more than 160 universities, research labs and companies from five continents (except Antarctica) across the world. We have the reason to believe that the earlier efforts of sample collection, analysis, and sharing are useful to the community and therefore are motivated to continue to do so.

In order to characterize Android malware growth in the wild, we show the monthly cumulative growth of Android malware in Fig. 2.2. The figure clearly indicates that starting summer 2011, Android malware has experienced an exponential growth. The outbreaks of several major Android malware families, such as *DroidKungFu* (June, 2011) and *AnserverBot* (September, 2011), greatly contribute to the trend: among these 1260 samples in our dataset, 37.5 % of them are related to *DroidKungFu* [29] and its variants; 14.8 % are *AnserverBot* [27]. Most of these malicious apps are still actively evolving and we will have a detailed study of them in Chap. 3.

2.2 Malware Characterization

Based on the dataset, we next present a systematic characterization of existing Android malware. By doing so, we can possibly understand how they infect mobile users and what kinds of damage that might be caused. Also we will compare the list of permissions often requested by malware with popular benign apps (e.g., from official Android Market) and illustrate key differences between the two.

2.2.1 Malware Installation

To infect mobile users, malicious apps typically lure users into downloading and installing them. By manually analyzing them, we categorize their attack techniques into the following ones: *repackaging*, *update attack*, and *drive-by download*.

Repackaging is one of most widely-adopted techniques in Android malware, which basically works by first downloading popular benign apps, repackaging them with additional malicious payloads, and then uploading repackaged ones to various Android marketplaces. Update attack does not directly inject malicious payloads into benign apps. Instead, the malicious payloads are disguised as the “updated” version of legitimate apps. Drive-by download is similar to traditional web-based attack that is launched to redirect users to download malware, e.g., by using aggressive in-app advertisement or malicious QR code.

2.2.1.1 Repackaging

As mentioned earlier, repackaging is one of the most common techniques malware authors use to piggyback malicious payloads into popular apps. In essence, malware authors may locate and download popular apps, disassemble them, enclose malicious payloads, re-assemble and then submit the new apps to official and/or alternative Android markets. Users could be vulnerable by being enticed to download and install these infected apps.

To quantify the use of repackaging technique among our dataset, we take the following approach: if a sample shares the same package name with an app in the official Android Market, we then download the official app and manually compare the difference, which typically contains the malicious payload added by malware authors. If the original app is not available, we choose to disassemble the malware sample and manually determine whether the malicious payload is a natural part of the main functionality of the host app. If not, it is considered as repackaged app.

In total, among the 1260 malware samples, 1083 of them (or 86.0 %) are repackaged. By further classifying them based on each individual family (Table 2.1), we find that within the total 49 families in our collection, 25 of them infect users by these repackaged apps while 25 of them are standalone apps, which are designed to be spyware in the first place. One malware family, i.e., *GoldDream*, utilizes both for its infection. Among these repackaged apps, we find that malware authors have chosen a variety of apps for repackaging, including paid apps, popular game apps, powerful utility apps (including security updates), as well as porn-related apps. For instance, one *AnserverBot* malware sample repackaged a paid app *com.camelgames.mxmotor* available on the official Android Market and injected its malicious payload. Another *BgServ* [6] malware sample repackaged the security tool released by Google to remove *DroidDream* from infected phones.

Possibly due to the attempt to hide piggybacked malicious payloads, malware authors tend to use the class-file names which look legitimate and benign. For example, *AnserverBot* malware uses a package name *com.sec.android.provider.drm* for its payload, which looks like a module that provides legitimate DRM functionality. The first version of *DroidKungFu* chooses to use *com.google.ssearch* to disguise as the Google search module and its follow-up versions use *com.google.update* to pretend to be an official Google update.

Table 2.1 An overview of existing Android malware (Part I: installation and activation)

	Installation			Activation								
	Repackaging	Update	Drive-by download	Standalone	BOOT	SMS	NET	CALL	USB	PKG	BATT	SYS
ADRD	✓				✓		✓	✓				
AnserverBot	✓	✓			✓	✓	✓		✓		✓	✓
Asroot				✓								
BaseBridge	✓	✓			✓	✓	✓	✓			✓	✓
BeanBot	✓				✓	✓						
BgServ	✓				✓	✓						
CoinPirate	✓				✓	✓						
Crusewin			✓		✓	✓						
Dog Wars	✓											
DroidCoupon	✓				✓		✓	✓		✓		
DroidDeluxe			✓									
DroidDream												
DroidDreamLight	✓											
DroidKungFu1	✓				✓			✓			✓	✓
DroidKungFu2	✓				✓					✓	✓	✓
DroidKungFu3	✓				✓					✓	✓	✓
DroidKungFu4	✓				✓					✓	✓	✓
DroidKungFuSapp	✓				✓					✓	✓	✓
DroidKungFuUpdate	✓	✓										
DroidKungFuUpdate	✓				✓							
Endofday	✓					✓						
FakeNetflix				✓								
FakePlayer			✓									
GamblerSMS			✓									
Geinimi	✓				✓							
						✓						

(Continued)

Table 2.1 (Continued)

	Installation				Activation							
	Repackaging	Update	Drive-by download	Standalone	BOOT	SMS	NET	CALL	USB	PKG	BATT	SYS
GGTracker			✓	✓	✓	✓					✓	
GingerMaster	✓				✓							
GoldDream	✓			✓	✓	✓		✓				
Gone60				✓								
GPSSMSSpy				✓		✓						
HippoSMS	✓				✓	✓						
Jifake	✓											
jSMShider	✓		✓							✓		
KMin				✓	✓							
Lovetrap				✓	✓	✓						
NickyBot				✓	✓	✓						
Nickyspy				✓	✓							
Pjapps	✓				✓	✓						✓
Plankton		✓		✓								
RogueLemon				✓		✓						
RogueSPPush				✓		✓						
SMSReplicator				✓		✓						
SndApps				✓								
Spitmo			✓	✓	✓	✓		✓				
TapSnake				✓								
Walkinwat				✓								
YZHC				✓								
zHash				✓								
Zitmo			✓			✓						
Zzone	✓			✓		✓						

It is interesting to note that one malware family—*jSMShider*—uses a publicly available private key (serial number: *b3998086d056cffa*) that is distributed in the Android Open Source Project (AOSP). The current Android security model allows the apps signed with the same platform key of the phone firmware to request the permissions which are otherwise not available to normal third-party apps. One such permission includes the installation of additional apps without user intervention. Unfortunately, a few (earlier) popular custom firmware images were signed by the default key distributed in AOSP. As a result, the *jSMShider*-infected apps may obtain privileged permissions to perform dangerous operations (installing another app which can send SMS messages to premium-rate numbers) without user’s awareness.

2.2.1.2 Update Attack

The first technique typically piggybacks the entire malicious payloads into host apps, which could potentially expose their presence. The second technique makes it difficult for detection. Specifically, it may still repackage popular apps. But instead of enclosing the payload as a whole, it only includes an update component that will fetch or download the malicious payloads at runtime. As a result, static scanning of host apps may fail to capture the malicious payloads. In our dataset, there are four malware families, i.e., *BaseBridge*, *DroidKungFuUpdate*, *AnserverBot*, and *Plankton*, that adopt this attack (Table 2.1).

The *BaseBridge* malware has a number of variants. While some embed root exploits that allow for silent installation of additional apps without user intervention, we here focus on other variants that use the update attacks without root exploits. Specifically, when a *BaseBridge*-infected app runs, it will check whether an update dialogue needs to be displayed. If yes, by essentially saying that a new version is available, the user will be offered to install the updated version (Fig. 2.3a) (The new version is actually stored in the host app as a resource or asset file). If the user accepts, an “updated” version with the malicious payload will then be installed (Fig. 2.3b). Because the malicious payload is in the “updated” app, *not* the original app itself, it is more stealthy than the first technique that directly includes the entire malicious payload in the first place.

The *DroidKungFuUpdate* malware is similar to *BaseBridge*. But instead of carrying or enclosing the “updated” version inside the original app, it chooses to remotely download a new version from network. Moreover, it takes a stealthy route by notifying the users through a third-party library [35] that provides the (legitimate) notification functionality. (Note the functionality is similar to the automatic notification from the Google’s Cloud to Device Messaging framework.) Once downloaded, the “updated” version turns out to be the *DroidKungFu3* malware. By leveraging the service provided by legitimate library to download Android malware, it becomes stealthy and hard to detect.

The previous two update attacks require user approval to download and install new versions. Others such as *AnserverBot* and *Plankton* advance the update attack by stealthily upgrading certain components in the host apps *not* the entire app. As a

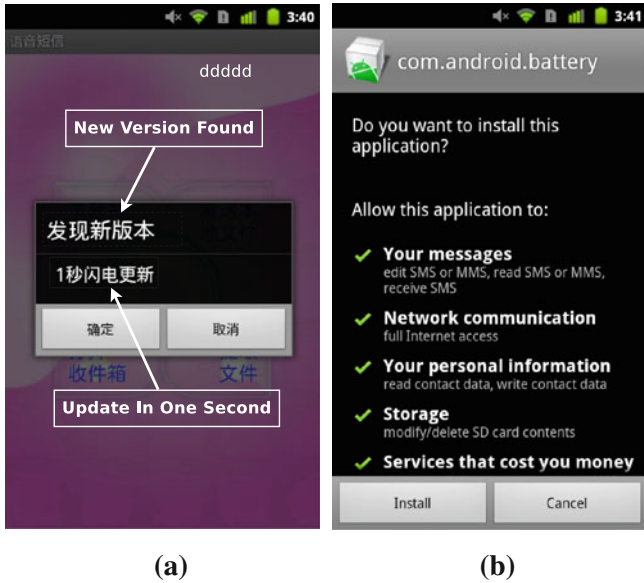


Fig. 2.3 An update attack from *BaseBridge*. **a** The update dialog. **b** Installation of a new version

result, it does not require user approval. In particular, *Plankton* directly fetches and runs a *JAR* file maintained in a remote server while *AnserverBot* retrieves a public (encrypted) blog entry, which contains the actual payloads for update! Apparently, the stealthy nature of these update attacks poses significant challenges for their detection. We will analyze these two malware families in Chap. 3.

2.2.1.3 Drive-by Download

The third technique applies the traditional drive-by-download attacks to mobile space. Though they are not directly exploiting mobile browser vulnerabilities, they are essentially enticing users to download “interesting” or “feature-rich” apps. In our collection, we have four such malware families, i.e., *GGTracker* [14], *Jifake* [18], *Spitmo* [12] and *ZitMo* [37]. The last two are designed to steal user’s sensitive banking information.

The *GGTracker* malware starts from in-app advertisements. In particular, when a user clicks a special advertisement link, it will redirect the user to a malicious website, which claims to be analyzing the battery usage of user’s phone and will redirect the user to a fake Android Market to download an app (for the purpose of improving battery efficiency). Unfortunately, the downloaded app is not one that focuses on improving the efficiency of battery, but a malware that will subscribe to a premium-rate service without user’s knowledge.

Similarly, the *Jifake* malware is downloaded when users are redirected to a malicious website. However, it is not using in-app advertisements to attract and redirect users. Instead, it uses a malicious QR code [24], which when scanned will redirect the user to another URL containing the *Jifake* malware. This malware itself is the repackaged mobile ICQ client, which sends several SMS messages to a premium-rate number. While QR code-based malware propagation has been warned earlier [34], this is the first time that this attack actually occurred in the wild.

The last two *Spitmo* and *ZitMo* are ported versions of nefarious PC malware, i.e., *SpyEye* and *Zeus*. They work in a similar manner: when a user is doing online banking with a comprised PC, the user will be redirected to download a particular smartphone app, which is claimed to better protect online banking activities. However, the downloaded app is actually a malware, which can collect and send mTANs (a credential for online banking) or SMS messages to a remote server. These two malware families rely on the comprised desktop browsers to launch the attack. Though it may seem hard to infect real users, the fact that they can steal sensitive bank information raises serious alerts to users.

2.2.1.4 Others

We have so far presented three main social engineering-based techniques that have been used in existing Android malware. Next, we examine the rest samples that do not fall in the above three categories. In particular, our dataset has 1083 repackaged apps, which leaves 177 standalone apps. We therefore look into those standalone apps and organize them into the following four groups.

The first group is considered spyware as claimed by themselves—they intend to be installed to victim's phones on purpose. That probably explains why attackers have no motivations or the need to lure victim for installation. *GPSSMSSpy* is an example that listens to SMS-based commands to record and upload the victim's current location.

The second group includes those fake apps that masquerade as the legitimate apps but stealthily perform malicious actions, such as stealing users' credentials or sending background SMS messages. *FakeNetflix* is an example that steals a user's Netflix account and password. Note that it is not a repackaged version of Netflix app but instead disguises to be *the* Netflix app with the same user interface. *FakePlayer* is another example that masquerades as a movie player but does not provide the advertised functionality at all. All it does is to send SMS messages to premium-rate numbers without user awareness.

The third group contains apps that also intentionally include malicious functionality (e.g., sending unauthorized SMS messages or subscribing to some value-added services automatically). But the difference from the second group is that they are not fake ones. Instead, they can provide the functionality they claimed. But unknown to users, they also include certain malicious functionality. For example, one *RogueSPPush* sample is an astrology app. But it will automatically subscribe

to premium-rate services by intentionally hiding and automatically replying to subscription-confirming SMS messages.

The last group includes those apps that rely on the root privilege to function well. However, without asking the user to grant the root privilege to these apps, they leverage known root exploits to escape from the built-in security sandbox. Though these apps may not clearly demonstrate malicious intents, the fact of using root exploits without user permission seems cross the line. Examples in this group include *Asroot* and *DroidDeluxe*.

2.2.2 Activation

Next, we examine the system-wide Android events of interest to existing Android malware. By registering for the related system-wide events, Android malware can rely on the built-in support of automated event notification and callbacks on Android to flexibly trigger or launch its payloads. For simplicity, we abbreviate some frequently-used Android events in Table 2.2 (and use them in Table 2.1).

Among all available system events, *BOOT_COMPLETED* is the most interested one to existing Android malware. This is not surprising as this particular event will be triggered when the system finishes its booting process—a perfect timing for malware to kick off its background services. By listening to this event, the malware can start

Table 2.2 The (abbreviated) Android events/actions of interest to existing malware

Abbreviation	Events	Abbreviation	Events
BOOT (Boot Completed)	BOOT_COMPLETED	SMS (SMS/MMS)	SMS_RECEIVED WAP_PUSH_RECEIVED
CALL (Phone Events)	PHONE_STATE NEW_OUTGOING _CALL	USB (USB Storage)	UMS_CONNECTED UMS_DISCONNECTED
PKG (Package)	PACKAGE_ADDED PACKAGE_REMOVED PACKAGE_CHANGED PACKAGE_REPLACED PACKAGE_RESTARTED PACKAGE_INSTALL	BATT (Power/ Battery)	ACTION_POWER_CONNECTED ACTION_POWER _DISCONNECTED BATTERY_LOW BATTERY_OKAY BATTERY_CHANGED_ACTION
SYS (System Events)	USER_PRESENT INPUT_METHOD _CHANGED SIG_STR SIM_FULL	NET (Network)	CONNECTIVITY_CHANGE PICK_WIFI_WORK

itself without user's intervention. In our dataset, 29 (with 83.3 % of the samples) malware families listen to this event.

The *SMS_RECEIVED* comes second with 21 malware families interested in it. This event will be broadcasted to the whole system when a new SMS message is being received. By listening to this event, the malware can be keen in intercepting or responding to particular incoming SMS messages. As an example, *Zsone* listens to this *SMS_RECEIVED* event and intercepts or removes all SMS message from particular originating numbers such as "10086" and "10010". The *RogueSPPush* listens to this event to automatically hide and reply to incoming premium-rate service subscription SMS message. In fact, the malware can even discard this *SMS_RECEIVED* event and stop it from further spreading in the system by calling *abortBroadcast()* function. As a result, other apps (including system SMS messaging app) do not even know the arrival of this new SMS message.

During our analysis, we also find that certain malware registers for a variety of events. For example, *AnserverBot* registers for callbacks from 10 different events while *BaseBridge* is interested in 9 different events. The registration of a large number of events is expected to allow the malware to reliably or quickly launch the carried payloads.

In addition, we also observe some malware samples directly hijack the entry activity of the host apps, which will be triggered when the user clicks the app icon on the home screen or an intent with action *ACTION_MAIN* is received by the app. The hijacking of the entry activity allows the malware to immediately bootstrap its service before starting the host app's primary activity. For example, *DroidDream* replaces the original entry activity with its own *com.android.root.main* so that it can gain control even before the original activity *com.codingcaveman.SoloTrial.SplashActivity* is launched. Some malware may also hijack certain UI interaction events (e.g., button clicking). An example is the *Zsone* malware that invokes its own SMS sending code inside the *onClick()* function of the host app.

2.2.3 Malicious Payloads

As existing Android malware can be largely characterized by their carried payloads, we also survey our dataset and partition the payload functionalities into four different categories: *privilege escalation*, *remote control*, *financial charges*, and *personal information stealing*.

2.2.3.1 Privilege Escalation

The Android platform is a complicated system that consists of not only the Linux kernel, but also the entire Android framework with more than 90 open-source libraries,

Table 2.3 The list of platform-level root exploits and their uses in existing Android malware

Vulnerable program	Root exploit	Release date	Malware with the exploit
Linux kernel	Asroot [7]	2009/08/16	Asroot
init (≤ 2.2)	Exploit [5]	2010/07/15	DroidDream, zHash, DroidKungFu[1235]
adbd ($\leq 2.2.1$)	RATC [9]	2010/08/21	DroidDream, BaseBridge
zygote ($\leq 2.2.1$)	Zimperlich [38]	2011/02/24	DroidKungFu [1235], DroidDeluxe DroidCoupon
ashmem ($\leq 2.2.1$)	KillingInThe NameOf [3]	2011/01/06	–
vold ($\leq 2.3.3$)	GingerBreak [36]	2011/04/21	GingerMaster
libsysutils ($\leq 2.3.6$)	zergRush [26]	2011/10/10	–

including WebKit, SQLite, and OpenSSL. The complexity naturally introduces software vulnerabilities that can be potentially exploited for privilege escalation. In Table 2.3, we show the list of known Android platform-level vulnerabilities that can be exploited for privilege exploitations. Inside the table, we also show the list of Android malware that actively exploit these vulnerabilities to facilitate the execution of their payloads.

Overall, there are a small number of platform-level vulnerabilities that are being actively exploited in the wild. The top three exploits are *exploit*, *RATC* (or *RageAgainstTheCage*), and *Zimperlich*. We point out that if the *RATC* exploit is launched within a running app, it is effectively exploiting the bug in the *zygote* daemon, *not* the intended *adbd* daemon, thus behaving as the *Zimperlich* exploit. Considering the similar nature of these two vulnerabilities, we use *RATC* to represent both of them.

From our analysis, one alarming result is that among 1260 samples in our dataset, 463 of them (36.7 %) embed at least one root exploit (Table 2.4). In terms of the popularity of each individual exploit, there are 389, 440, 4, and 8 samples that contain *exploit*, *RATC*, *GingerBreak*, and *Asroot*, respectively. Also, it is not uncommon for a malware to have two or more root exploits to maximize its chances for successful exploitations on multiple platform versions. (In our dataset, there are 378 samples with more than one root exploit.)

2.2.3.2 Remote Control

During our analysis to examine the remote control functionality among the malware payloads, we are surprised to note that 1172 samples (93.0 %) communicate with remote servers or turn the infected phones into bots for remote control. Specifically, there are 1171 samples that use the HTTP-based web traffic to communicate with remote servers and receive bot commands from their C&C servers.

Table 2.4 An overview of existing Android malware (Part II: malicious payloads)

	Privilege escalation			Remote control		Financial charges			Personal information stealing			
	Exploit	RATC	Ginger break	Asroot	NET	SMS	Phone call	SMS	Block	SMS	Phone number	User account
ADRD					✓			✓ [†]				
AnserverBot					✓							
Asroot				✓			✓	✓ [†]	✓			
BaseBridge		✓			✓		✓	✓ [†]	✓		✓	
BeanBot					✓		✓	✓ [†]	✓		✓	
BgServ					✓		✓	✓ [†]	✓			
CoinPirate					✓		✓	✓ [†]	✓	✓		
Crusewin					✓		✓	✓	✓	✓		
DogWars								✓				
DroidCoupon					✓							
DroidDeluxe		✓										
DroidDream		✓			✓							
DroidDreamLight	✓				✓							✓
DroidKungFu1	✓	✓			✓						✓	
DroidKungFu2	✓	✓			✓						✓	
DroidKungFu3	✓	✓			✓						✓	
DroidKungFu4					✓							
DroidKungFu5					✓						✓	
DroidKungFuUpdate	✓	✓			✓							
Endofday					✓			✓			✓	
FakeNetflix												✓
FakePlayer								✓ [‡]				
GamblerSMS										✓		
Geinimi					✓		✓	✓ [†]	✓	✓	✓	

(Continued)

Table 2.4 (Continued)

	Privilege escalation				Remote control		Financial charges			Personal information stealing		
	Exploit	RATC	Ginger break	Asroot	NET	SMS	Phone call	SMS	Block SMS	SMS	Phone number	User account
GGTracker								✓ [‡]	✓	✓		
GingerMaster			✓		✓					✓	✓	
GoldDream					✓		✓			✓	✓	
Gone60										✓		
GPSSMSspy								✓				
HippoSMS								✓ [‡]	✓			
Jifake								✓ [‡]				
jSMShider					✓			✓ [‡]	✓			
KMin					✓			✓ [‡]	✓		✓	
Lovetrap								✓ [‡]	✓			
NickyBot						✓		✓		✓		
Nickyspy					✓			✓		✓		
Pjapps					✓			✓ [‡]	✓		✓	
Plankton					✓							
RogueLemon					✓			✓ [‡]	✓	✓		
RogueSPush					✓			✓ [‡]	✓	✓		
SMSReplicator								✓		✓		
SndApps												✓
Spitmo								✓ [‡]	✓	✓	✓	
TapSnake												
Walkinwat								✓				
YZHC					✓			✓ [‡]	✓		✓	
zHash												
Zitmo								✓		✓		
Zzone	✓							✓ [‡]	✓			

We also observe that some malware families attempt to be stealthy by encrypting the URLs of remote C&C servers as well as their communication with C&C servers. For example, *Pjapps* develops its own encoding scheme to encrypt the C&C server addresses. One of its samples encodes its C&C server *mobilemeego91.com* into *2maodb3ialke8mdeme3gkos9glicaofm*. *DroidKungFu3* employs the standard AES encryption scheme and uses the key *Fuck_sExy-aLL!Pw* to hide its C&C servers. *Geinimi* similarly applies DES encryption scheme (with the key *0x01020304050607-08*) to encrypt its communication to the remote C&C server.

During our study, we also find that most C&C servers are registered in domains controlled by attackers themselves. However, we also identify cases where the C&C servers are hosted in public clouds. For instance, the *Plankton* spyware dynamically fetches and runs its payload from a server hosted in the Amazon cloud. Most recently, attackers are even turning to public blog servers as their C&C servers. *AnserverBot* is one example that uses two popular public blog services, i.e., *Sina* and *Baidu*, as its C&C servers to retrieve the latest payloads and new C&C URLs (Chap. 3).

2.2.3.3 Financial Charge

Beside privilege escalation and remote control, we also look into the motivations behind malware infection. In particular, we study whether malware will intentionally cause financial charges to infected users.

One profitable way for attackers is to surreptitiously subscribe to (attacker-controlled) premium-rate services, such as by sending SMS messages. On Android, there is a permission-guarded function *sendTextMessage* that allows for sending an SMS message in the background without user's awareness. We are able to confirm this type of attacks targeting users in Russia, United States, and China. The very first Android malware *FakePlayer* sends SMS message "798657" to multiple premium-rate numbers in Russia. *GGTracker* automatically signs up the infected user to premium services in US without user's knowledge. *Zsone* sends SMS messages to premium-rate numbers in China without user's consent. In total, there are 55 samples (4.4 %) falling in 7 different families (tagged with ‡ in Table 2.4) that send SMS messages to the premium-rate numbers hardcoded in the infected apps.

Moreover, some malware choose *not* to hard-code premium-rate numbers. Instead, they leverage the flexible remote control to push down the numbers at runtime. In our dataset, there are 13 such malware families (tagged with † in Table 2.4). Apparently, these malware families are stealthier than earlier ones because the destination number will not be known by simply analyzing the infected apps.

In our analysis, we also observe that by automatically subscribing to premium-rate services, these malware families need to reply to certain SMS messages. This may due to the second-confirmation policy required in some countries such as China. Specifically, to sign up a premium-rate service, the user must reply to a confirming SMS message sent from the service provider to finalize or activate the service subscription. To avoid users from being notified, they will take care of replying to these confirming messages by themselves. As an example, *RogueSPPush* will

automatically reply “Y” to such incoming messages in the background; *GGTracker* will reply “YES” to one premium number, 99735, to activate the subscribed service. Similarly, to prevent users from knowing subsequent billing-related messages, they choose to filter these SMS messages as well. This behavior is present in a number of malware, including *Zsone*, *RogueSPPush*, and *GGTracker*.

Besides these premium-rate numbers, some malware also leverage the same functionality by sending SMS messages to other phone numbers. Though less serious than previous ones, they still result in certain financial charges especially when the user does not have an unlimited messaging plan. For example, *DogWars* sends SMS messages to all the contacts in the phone without user’s awareness. Other malware may also make background phone calls. With the same remote control capability, the destination number can be provided from a remote C&C server, as shown in *Geinimi*.

2.2.3.4 Information Collection

In addition to the above payloads, we also find that malware are actively harvesting various information on the infected phones, including SMS messages, phone numbers as well as user accounts. In particular, there are 13 malware families (138 samples) in our dataset that collect SMS messages, 15 families (563 samples) gather phone numbers, and 3 families (43 samples) obtain and upload the information about user accounts. For example, *SndApps* collects users’ email addresses and sends them to a remote server. *FakeNetflix* gathers users’ Netflix accounts and passwords by providing a fake but seeming identical Netflix UI.

We consider the collection of users’ SMS messages is a highly suspicious behavior. The user credential may be included in SMS messages. For example, both *Zitmo* (the *Zeus* version on Android) and *Spitmo* (the *SpyEpy* version on Android) attempt to intercept SMS verification messages and then upload them to a remote server. If successful, the attacker may use them to generate fraudulent transactions on behalf of infected users.

2.2.4 Permission Usage

For Android apps without root exploits, their capabilities are strictly constrained by the permissions users grant to them. Therefore, it will be interesting to compare top permissions requested by these malicious apps in the dataset with top permissions requested by benign ones. To this end, we have randomly chosen 1260 top free apps downloaded from the official Android Market in the first week of October, 2011. The results are shown in Fig. 2.4.

Based on the comparison, Android permissions such as *INTERNET*, *READ_PHONE_STATE*, *ACCESS_NETWORK_STATE*, and *WRITE_EXTERNAL_STORAGE* are widely requested in both malicious and benign apps. The first two are typically needed to allow for the embedded ad libraries to function properly.

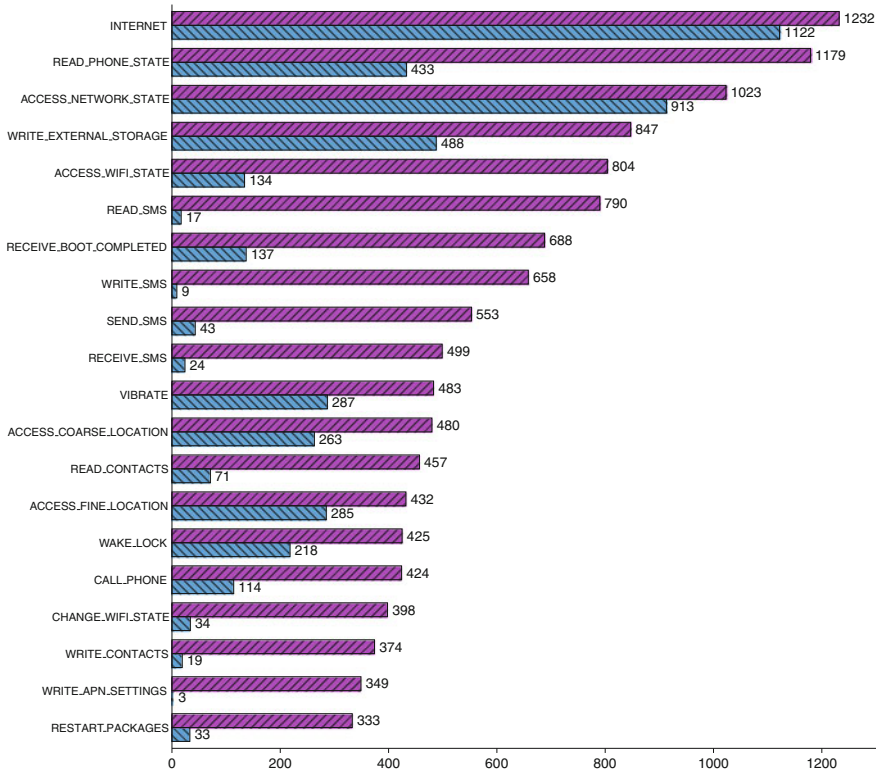




Fig. 2.4 The comparison of top 20 requested permissions by malicious apps (the  bar) and benign apps (the  bar)

But malicious apps clearly tend to request more frequently on the SMS-related permissions, such as *READ_SMS*, *WRITE_SMS*, *RECEIVE_SMS*, and *SEND_SMS*. Specifically, there are 790 samples (62.7 %) in our dataset that request the *READ_SMS* permission, while 17 benign apps (or 1.3 %) request this permission. These results are consistent with the fact that 28 malware families in our dataset (or 45.3 % of the samples) that have the SMS-related malicious functionality.

Also, we observe 688 malware samples request the *RECEIVE_BOOT_COMPLETED* permission. This number is five times of that in benign apps (137 samples). This could be due to the fact that malware is more likely to run background services without user's intervention. Note that there are 398 malware samples requesting *CHANGE_WIFI_STATE* permission, which is an order of magnitude higher than that in benign apps (34 samples). That is mainly because the *Exploidy* root exploit requires certain hot plug events such as changing the WIFI state, which is related to this permission.

Finally, we notice that malicious apps tend to request more permissions than benign ones. In our dataset, the average number of permissions requested by

malicious apps is 11 while the average number requested by benign apps is 4. Among the top 20 permissions, 9 of them are requested by malicious apps on average while 3 of them on average are requested by benign apps.

Android Malware

Jiang, X.; Zhou, Y.

2013, XI, 44 p. 9 illus., Softcover

ISBN: 978-1-4614-7393-0