Security Analysis Of Android OS

Jeet Mohapatra, Mark Wang, Mike Wang

1. Introduction

The primary motivation for the project was the news about Wikileaks releasing a trove of hacking tools employed by the CIA. This includes tools and techniques which can hack phones. However, this information was later redacted, and so after investigating the vault released by Wikileaks, no information or knowledge was gained. Instead, other sources were used to research and learn about attacks.

Android is a very robust system with a dedicated team to identify and patch problems. As a result, many of the attacks described within the paper do not work on modern versions of Android.

2. Security Policy of Android

We can define the Android security through two different roles. For one role, we have the Android App Developers and for the other role, we have the users. First let us list some assumptions for each role.

Android App Developers:

- Has certain non-dangerous permissions automatically granted to them
- Can communicate with other apps with own app
- Can ask for more permissions over phone
- Has access to the Android API

Users:

- Chooses which programs to download
- Can grant permissions for other apps
- Assumes Android provides basic protection from attacks

We make the assumption that most of the attacks on the Android either involves some sort of malicious code execution or network attack. For malicious code, the most common way of inserting it will be through an app. There are certain security mechanisms on the phone that help prevent malicious code from hijacking the phone. Here are some following mechanism:

Security Mechanisms:

- Application Sandbox: Each app runs on its own process and is monitored by the kernel. This prevents an app from accessing sensitive information or performing actions without permissions
- Permissions: Android will notify the user if an app wants to use protected permission and prevent the app from using corresponding feature without permission
- Code signature: Every app must be signed by the developer, or the code will be rejected before installation
- Root Access: Only a small number of applications and kernel have root access. Most apps and user are not granted access
- Address Space Random Layout (ASLR): Helps prevent attacks by rearranging key data areas of a process

3. Methodology/Setup

We developed all of our proofs of concepts with Android Studio. We tested all of our code on emulated devices, since many of the attacks have been patched and only work for older versions of Android. For Cloak and Dagger, we tested on an emulated Nexus 5 with Android 7.0 installed. For One Class to Rule Them All, we tested on a Nexus 5 with Android 4.4 installed. For Stagefright, we tested on a Nexus 5 with Nexus 5.0 installed.

4. Types of Attack

4.1. Cloak and Dagger

One of the key security mechanism for Android is the permission system. The Cloak and Dagger vulnerability Wang, Savage, and Voelker (2011) is based on two permissions:

- $SYSTEM\_ALERT\_WINDOW$ - This allows an app to draw overlays on top of other apps.
- $BIND\_ACCESSIBILITY\_SERVICE$ This grants an app the ability to discover UI widgets displayed on the screen, query the content of these widgets, and interact with them programmatically. It exists to make Android devices more accessible to users with disabilities.

Although there are a lot of attacks based on these permissions alone, these attacks cannot accomplish too much because of the following security meachanisms in place.

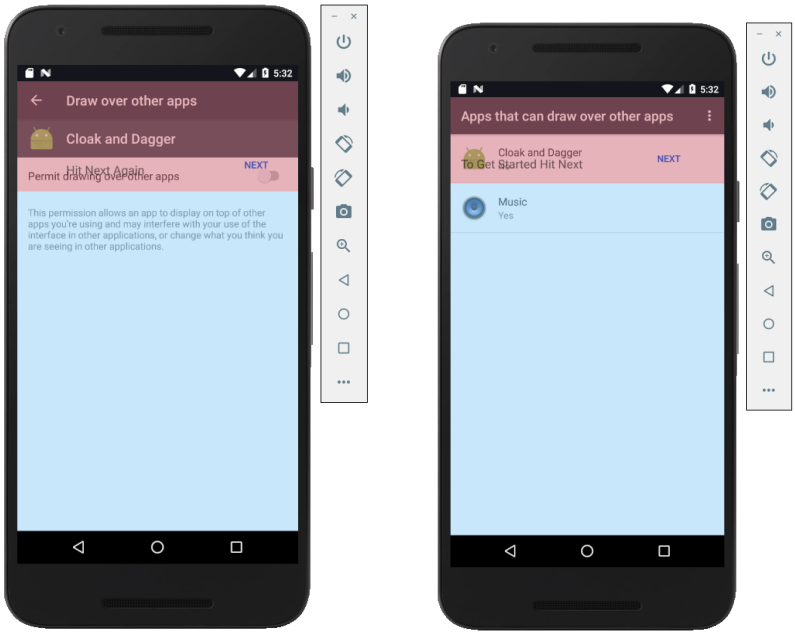
Security Mechanisms

- The SYSTEM ALERT WINDOW permission allows an app to create custom views and widgets on top of any another app. However, the system is designed so that the following constraint always holds: if an overlay is marked as pass through (that is, it will not capture clicks), the app that created the overlay will not know when the user clicks on it (however, since the overlay is pass through, the click will possibly reach what is below); instead, if the overlay is created to be clickable, the app that created it will be notified when a click occurs, and it will also have access to its exact coordinates. However, the system is designed so that an overlay cannot propagate the click to the underlying app.
- An accessibility service app has access, by design, to the content displayed on the screen by the apps the user is interacting with. Although the accessibility service does not have access to passwords, it does have privacy-related implications. Thus, in Android, the service needs to be manually enabled by the user: after pressing on the enable switch, the system shows to the user an informative popup and she needs to acknowledge it by pressing on the OK button.
- Given the security implications of the accessibility service, the Android OS has a security mechanism in place that aims at guaranteeing that other apps cannot interfere during the approval process (i.e., when the user is clicking on the OK button). This defense has been introduced only recently, after a security researcher showed that it was possible to cover the OK button and the popup itself with an opaque, passthrough overlay: while the user is convinced to interact with the app-generated overlay, she is actually authorizing the accessibility service permission by unknowingly pressing OK.
- To maximize the usefulness of accessibility service apps, they are given access to the content displayed on the screen. However, for security reasons, they are not given access to highly private information, such as password. This is implemented by stripping out the content of EditText widgets known to contain passwords.

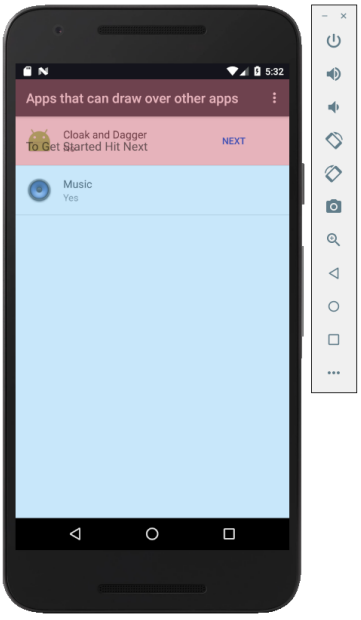
Given these security measures, any attack that uses only one of these permissions is either useless in terms of leaking information or easily detectable. As if one tries to gain information by overlays using $SYSTEM\_ALERT\_WINDOW$ permission they cannot replicate the normal response. Similarly if an app uses $BIND\_ACCESSIBILIY\_SERVICE$ permission to interact with UI, they cannot prevent the user from finding out about the resulting event. However, combining these two permissions provides a way around these security mechanisms.

4.1.1. Attack Implementation

To start the attack, we use a toast overlay attack. We do this by asking the user for the permission and then overlaying an opaque screen on top of the permission window to confuse the user into thinking that they are responding to some other message. The images below describe this attack:



(a) Overlay Attack (Click-jacking)



(b) Keyboard Cloak And Dagger

After gaining the permissions our creates a transparent keyboard overlay (translucent here for demonstration) on the users virtual keypad (using the $SYSTEM\_ALERT\_WINDOW$ permission) to get the information about the key-presses and then use the $BIND\_ACCESSIBILITY\_SERVICE$ permission to interact with the UI of the original app the user was trying to use, to have the user get the normal experience. This makes this attack especially hard to detect. Using this, the app could steal account passwords, bank PINs and also monitor users messages and emails.

4.1.2. Fixes

This vulnerability gave android a very hard time. The vulnerability was first discovered in August of 2016 but was not fixed till the launch of Android 7.1.2 in April 2017. The reason behind this being that the $SYSTEM\_ALERT\_WINDOW$ permission cannot given special status. Popular apps like Facebook messenger use this to create the messenger bubbles which lot of Android Users want. In order to keep the user experience good, Android did not provide a hasty patch. However, in Oreo a lot of new mechanisms were placed to protect against this attack. Switching off the functionality of background apps, timing applications to allow for detection (as cloak and dagger is slower than the normal functioning of the app). A lot other similar changes make oreo invulnerable to cloak and dagger now.

4.2. One Class to Rule them All

4.2.1. Mechanism

One Class to Rule them All Peles and Hay (2015) and related attacks work by attacking the inter-app communication channels of Android. Apps communicate with one another through a "Bundle" object. This "Bundle" object may be used in order to send a string or even an arbitrary object. However usually, the object must implement the 'Serializable' interface.

One previous attack that took advantage of the weakness took advantage of the fact that a victim app will not check if an object from a "Bundle" is serializable before de-serializing it. This in of itself poses no problem since this means that no information is transmitted from the "Bundle" object. However, when the object is freed by the Garbage Collector, the Garbage Collector will call the "finalize" method of the "Bundle" object, regardless of whether is was serializable or not. This "finalize" method could access the native code of the app and could be controlled by an adversary.

One Class to Rule them All use classes such as OpenSSLX09Certificate and Binder Proxy. These classes have a "finalize" method such that when they are destroyed, they decrement the address of a pointer within the class. For example, for BinderProxyHorn (2014), the finalize method looks like this. Here is some code for a proof of concept attack involving BinderProxy Lavi and Markus (2015):

```
static void android_os_BinderProxy_destroy(JNIEnv* env, jobject obj) {
    IBinder* b = (IBinder*)
            env->GetIntField(obj, gBinderProxyOffsets.mObject);
    DeathRecipientList* drl = (DeathRecipientList*)
            env->GetIntField(obj, gBinderProxyOffsets.mOrgue);
    LOGDEATH("Destroying BinderProxy %p: binder=%p drl=%p\n", obj, b, drl);
    env->SetIntField(obj, gBinderProxyOffsets.mObject, 0);
    env->SetIntField(obj, gBinderProxyOffsets.mOrgue, 0);
    drl->decStrong((void*)javaObjectForIBinder);
    b->decStrong((void*)javaObjectForIBinder);
    IPCThreadState::self()->flushCommands();
}
```

In the above code, we see that decStrong() takes as input drl, which in turn is determined by the mOrgue pointer of the BinderProxy class. We can quickly see that this pointer is attacker controllable.

4.2.2. Attacks

One example of an attack possible by One Class to Rule them All is to change the pointer of a callback function to an adversary determined location. This causes the adversary to have the power to run arbitrary code on the operating system. Another example is the the code can cause the phone to crash by continuously decrementing a value in memory. Once the value becomes negative, this can cause a segmentation fault.

4.2.3. Fixes

This attack no longer works on newer version of Android. The reason why is that Android now checks whether an object is serializable before deserializing it.

4.3. StageFright

Stagefright is a group of bugs that affects versions of android. The names comes from the library that Stagefright affects (an android core component called "Stagefright"), which is involved in playing some multimedia formats such as MP4.

4.3.1. Mechanism

The bug Be'er (2016) that we focus on for this project affects an imperfect patch meant to address some earlier vulnerabilities with the Stagefright library.

This is the code with the incorrect patch applied.

```
case FOURCC('t', 'x', '3', 'g'):
{
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
        size = 0;
    }
    if (SIZE_MAX - chunk_size <= size) { // <---- attempt to prevent overflow
        return ERROR_MALFORMED;
    }
    uint8_t *buffer = new uint8_t[size + chunk_size]; // <---- size + chunk_size
        can lead to integer overflow
    if (size > 0) {
        memcpy(buffer, data, size);               // <---- buffer overflow
    }
    if ((size_t)(mDataSource->readAt(*offset, buffer + size, chunk_size))
            < chunk_size) {
        delete[] buffer;
        buffer = NULL;
        return ERROR_IO;
    }
    mLastTrack->meta->setData(
            kKeyTextFormatData, 0, buffer, size + chunk_size);
    delete[] buffer;
    *offset += chunk_size;
    break;
```

```
}
```

Specifically, the bug is an integer underflow error. Since the chunk_size is an uint_64t, it can be larger than SIZE_MAX, which is a 32 bit integer. This can cause the write to be a buffer overflow for improperly formatted MP4 files.

4.3.2. Attacks

There are many possible attack vectors using this vulnerability. The simplest one, which we demonstrate in our proof of concept, is creating a crash in the system by overwriting enough bytes. A more interesting attack is remote code execution.

The RCE attack PoC that we will describe in this paper is Metaphor, an attack developed by Northbit. In this attack, the attacker controls a server that serves MP4 files to clients. When the server detects that the user is vulnerable to the Stagefright exploit, it first serves the client a small crash MP4 (this MP4 files crashes the media server on the android device and restarts it).

Afterwards, the attack server will create leak MP4 files. These leak MP4 files are specific to the type of device that is downloading them (the device information, such as the device ID, can be requested by the attack server). The leak happens because when a device downloads a track, it sends metadata of the track back for the server to verify. By heap grooming (allocating and deallocating chunks of data), the vulnerable buffer can be reliably placed before the metadata table. While most of the values of the metadata are not pointers, the duration field (the largest field in the metadata table) is. Therefore, by overwriting the metadata table, it is possible to leak around 30 bits of data from a memory location.

These MP4 files are looking for a particular ELFI header. One feature that many android phones have is ASLR (address space layout randomization), which randomizes the locations of the modules in the phone on startup. However, the randomization only shifts the modules within 256 pages from the base location, and this shift is constant for all modules. By downloading enough MP4 files, the client would eventually leak the ELFI header which identifies a particular location of a library. Therefore, attacker is able to find what the ASLR shift is and what physical address all of the modules are (the relative positions for all modules are fixed for a specific type of device).

This leads to the final stage of the attack, remote code execution. The attack is able

to inject code using the buffer overflow. The attack can also overwrite the vtable (mDataSource), therefore executing the injected code. Since the location of all of the gadgets are known (ASLR shift is known), it becomes trivial to create a chain of gadget calls to execute malicious code.

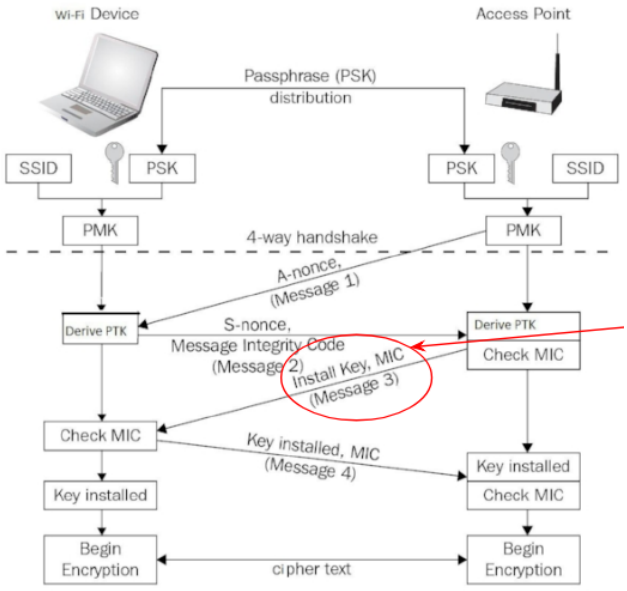
4.3.3. Fixes

This bug has been patched (we just need to modify the code to perform the buffer overflow check correctly). However, it is also suggested to disable auto-fetching for MMS and Google Hangouts, since it was possible the MP4 to be auto-fetched without the user even noticing. This may help to prevent other attacks using the Stagefright library.

4.4. KRACK

Key Reinstallation Attacks (KRACK) Vanhoef and Piessens (2017) is an attack of the WPA and WPA2 Internet protocol. In particular, it targets the 4-way handshake that is used in order to set a encryption key.

4.4.1. Mechanism



(c) WPA2 Diagram

In the WPA2 protocol, there is a 4-way handshake in order to set up the encryption key; in particular, the encryption key is installed after message three of the handshake. However, it is possible for the Access Point to replay the third message several times, if for example the

client does not acknowledge the first responses (from packet losses). Every time the message is replayed, the same encryption key is installed and the incremental transmit packet number (nonce) and receive replay counter are reset. However, since the nonce is being reused, packets could be replayed, decrypted, or forged.

Even worse, for the android system, when it receives message three of the handshake multiple times, it would install a cleared encryption key (an all-zero encryption key). It becomes trivial to intercept and manipulate the packets.

4.4.2. Attacks

We were not able to implement this attack, since we do not have the necessary hardware (our computers only have one wireless network interface, so we were not able to use a wireless network interface to create a rogue channel). However, hypothetically, if we can force someone to install an all-zero encryption key, we would be able to tamper with and intercept sent packets at will.

4.4.3. Fixes

The best fix to patch this vulnerability is to use WPA3 rather than WPA2.

5. Conclusion

Our initial goal was to look for attacks that could be used to get confidential information about the user or overhear the user's conversations. Going over the various attacks on the Android OS we see that all these attacks can quite easily leak confidential information stored on the phone. For most of these attacks there is no easy way to protect yourself from these attacks. As of now, over 90% of users have Android version lower than or equal to 7.1.2 and are thus still vulnerable to Cloak and Dagger. Most users are also still vulnerable to KRAck as they use wpa2 for wifi authentication. The best possible practice is keep your device up-to-date with the latest patch, be careful which apps you download and install (even apps on the Google playstore may not be secure), and keep track of what permissions you give to apps.

References

Be'er, H., 2016. Metaphor: A (real) real-life stagefright exploit.

Horn, J., 2014. Cve-2014-7911: Android¡ 5.0 privilege escalation using objectinput-stream.

Lavi, Y., Markus, N., 2015. A deep dive analysis of android system service vulnerability and exploitation.

Peles, O., Hay, R., 2015. One class to rule them all: 0-day deserialization vulnerabilities in android. WOOT 15, 5–5.

Vanhoef, M., Piessens, F., 2017. Key reinstallation attacks: Forcing nonce reuse in wpa2. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 1313–1328.

Wang, D. Y., Savage, S., Voelker, G. M., 2011. Cloak and dagger: dynamics of web search cloaking. In: *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, pp. 477–490.