CERTIFI-GATE: FRONT DOOR ACCESS TO PWNING MILLIONS OF ANDROID DEVICES

Ohad Bobrov
Avi Bashan

We are more mobile than we've ever been, so it's really no surprise that smartphones and tablets aren't our second screens, they're our first. These devices move massive amounts of data around the clock and around the world, and while some data may be trivial, the increasing trend is that most of it isn't.

We use mobile devices to manage everything from our health records and banking information to confidential work documents and sensitive plans with little worry over security or privacy. That's because we trust everyone from manufacturers to network provides to keep us safe. Whether data is at rest on a device or in flight through the cloud, we're confident that technology exists to shield us from danger.

However, as our research team recently discovered, we shouldn't be quite so trusting.

mRATs vs mRSTs

Mobile Remote Access Trojans (mRATs) provide unauthorized, stealth access to mobile devices. An attacker can exploit mRATs to exfiltrate sensitive information from devices such as location, contacts, photos, screen capture, and even recordings of nearby sounds. Known mRAT players include HackingTeam, mSpy, and SpyBubble.

Check Point analyzes mRATs on a daily basis to find similar traits in these apps which can then be used for detection and classification purposes. Our analysis has identified many common traits between different malware families, among these are app installation, screen access, and user input simulation. These traits are considered privileged and require an exploit to gain access to them.

While analyzing and classifying mRATs, our research team found some apps share common traits with Mobile Remote Support Tools (mRSTs) but do not use an exploit to obtain privileged permissions. These particular apps were actually part of the mRST family. mRSTs are used by organizations like IT departments, mobile service providers, and OEMs to provide remote support for smartphones and tablets.

July 18, 2015

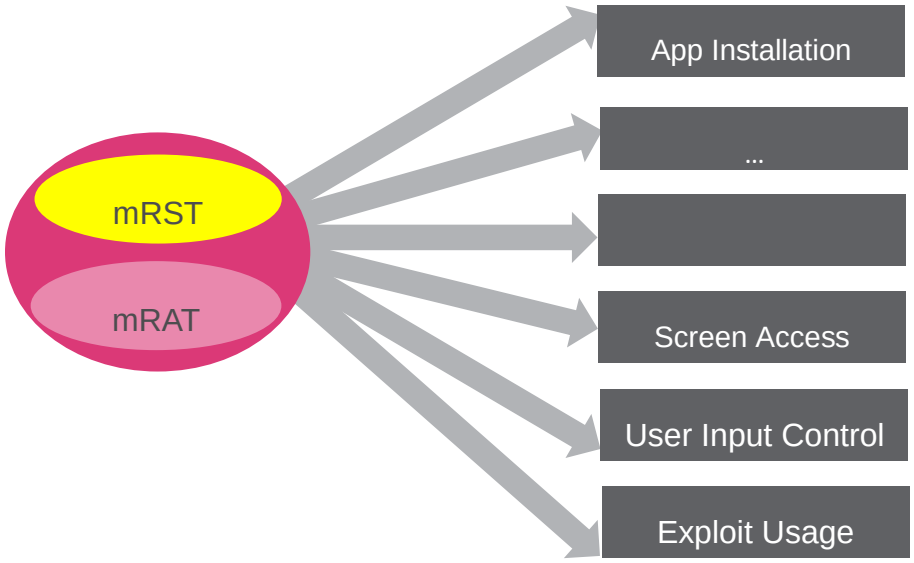

Figure 1: mRAT and mSRT shared traits

Mobile Remote Support Tools Architecture

Android's Permission Model 101

On Android, a modern operating system based on sandboxing principles, each app is installed in a sandboxed environment that restricts it from accessing any type of data. To gain access to resources such as an Internet connection or device data, the app must declare which permissions it needs. These permissions are declared in the AndroidManifest.xml file.

At the time of app installation, the user is presented with a required permission list (in Google Play these permissions are grouped and simplified). The user can then decide whether or not to accept the requested permissions that are granted using a `take it or leave it` approach. An app's permissions cannot be changed after installation. Instead, a new version must be installed in order to obtain more or different permissions.

Android has a set of 'privileged' permissions[1] that can be obtained by privileged system apps only. These system apps can be one of the following:

- Apps signed by the OEM.
- Apps located under `/system/priv-app` folder

[1] http://developer.android.com/reference/android/Manifest.permission.html

Remote Support Tools Permissions

To provide remote support capabilities, a Remote Support Tool (mRST) needs to provide as much data as possible from the device to the support screen remotely controlling the device. To do this, the mRST must obtain permissions granted to system apps only. Among those permissions are:

Permissions	Action
INSTALL_PACKAGES	App installation
READ_FRAME_BUFFER ACCESS_SURFACE_FLINGER	Screen access
INJECT_EVENTS	Inject user events

Table 1 - Privilged Permisisons Example

Most third-party vendors take a similar approach to resolving this problem when designing mRST tool architecture. The remote control app is split into separate apps:

- **The Main App:** An app signed by the vendor, and which contains most of the business logic, server connections, user interface, etc.
- **The Plugin:** An app signed by the OEM, this is an exported service that supplies an interface over Binder[2]. The plugin can obtain privileged permissions (because of the OEM signature), effectively enabling the main app to access privileged APIs without being signed by the OEM. The third-party vendor creates multiple copies of the same plugin signed by a separate OEM.

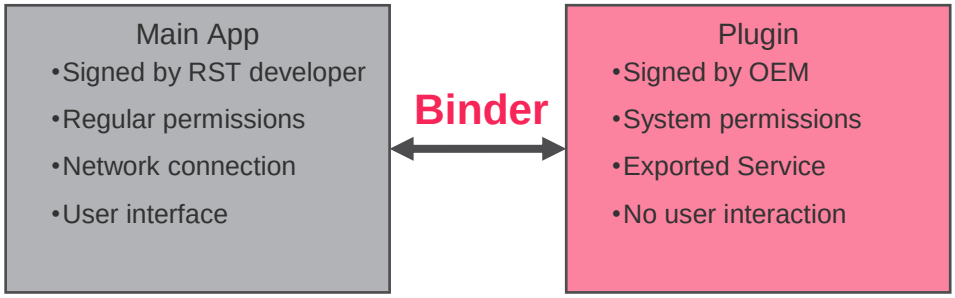

Figure 2: Relationship between the main app and the plugin

[2] Android's IPC Mechanism - http://developer.android.com/reference/android/os/Binder.html

Issues With Current mRST Architectures

Implementing mRST solutions has a few issues:

- **The OEM must sign the plugin.**
  This means the job of validating that code is secure and doesn't expose the system to any kind of vulnerability is passed to the OEM. Not all OEMs have the proper security teams to check third-party code. Furthermore, signing of an app by OEM can delay an update release in case vulnerability is discovered.
- **Certificate revocation.**
  Android doesn't have a revocation mechanism, meaning an app that has system privileges and that is vulnerable could be scraped by an attacker and used later to attack a victim through a phishing scam.
- **Simplified permissions.**
  The view of app permissions is simplified in the Google Play Store, but Google Play discards the *privileged permissions* required by the plugin. Therefore, it looks as if the plugin doesn't require any permission to be installed.
- **The plugin is an exported service.**
  An app that is only used as an exporter service doesn't have an icon in the launcher. Most users don't know how to look into more advanced screens on Android to remove the plugin later.
- **Lack of verification functions.**
  Android doesn't supply any method to verify the identity of an app sending data over the Binder (Android's IPC mechanism). This means every vendor needs to develop its own technique for verifying the identity of the source app. Delegating this to a third-party may lead to implementation flaws.

TECHNICAL RESEARCH

The research analyzed various mRSTs to discover the method by which they gain privileged permissions. Once the method was discovered, a second phase was performed to find any vulnerability that an attacker could exploit to take unauthorized remote control of devices. During the research, our team found certificate verification vulnerabilities in mRSTs from the following vendors:

- TeamViewer
- RSupport
- AnySupport
- CommuniTake

TeamViewer

TeamViewer Quick Support is an mRST with over 5 million downloads in Google Play. This app follows the main app / plugin architecture described previously. TeamViewer has collaborated with a number of OEMs including Samsung, LG, HTC, and Lenovo.

The TeamViewer main app communicates with the plugin over the Binder. When the main app initializes the plugin, the plugin loads the certificate of the caller and verifies that the serial number of the certificate equals a hardcoded serial number.

To understand how the certificates serial number is set, we reviewed RFC 2459 – Internet X.509 Public Key Infrastructure. The RFC states that the certificate's serial number is defined by the root CA.

In Android, each developer generates its own self-signed certificate to sign an app. This enables the developer to decide the certificate's serial number.

An attacker can exploit this and generate a certificate with a serial number that will match the plugin's required hardcoded serial number. The attacker can then create an app signed with this certificate and that interacts with the plugin. Now, the app can bypass the plugin verification mechanism and obtain full access to the device.

```
.method static constructor <clinit>()V
          .registers 2
00000000  new-instance          v0, BigInteger
00000004  const-string          v1, "1287658381"
00000008  invoke-direct         BigInteger-><init>(String)V, v0, v1
0000000E  sput-object           v0, TVAddonService->serialNum_v:BigInteger
00000012  return-void
.end method

.method private checkCallerCertSerialMatch__v(String)Z
          .registers 4
00000000  invoke-virtual        TVAddonService->getApplicationContext()Context, p0
00000006  move-result-object    v0
00000008  invoke-static         certMgr_v->return_caller_cert__v(String, Context)X509Certificate, p1, v0
0000000E  move-result-object    v0
00000010  invoke-virtual        X509Certificate->getSerialNumber()BigInteger, v0
00000016  move-result-object    v0
00000018  sget-object           v1, TVAddonService->serialNum_v:BigInteger
0000001C  invoke-virtual        BigInteger->equals(Object)Z, v0, v1
00000022  move-result           v0
00000024  if-eqz                v0, :2C
:28
00000028  const/4               v0, 1
:2A
0000002A  return                v0
:2C
0000002C  const-string          v0, "TVAddonService"
00000030  const-string          v1, "checkSignature(): serial mismatch - onBind will fail"
00000034  invoke-static         Logging->a(String, String)V, v0, v1
0000003A  const/4               v0, 0
0000003C  goto                  :2A
.end method
```

Figure 2: Vulnerable code in the TeamViewer plugin

RSupport

RSupport is an mRST that provides remote control and support for corporate environments and end-users. It has a wide set of products including Mobizen and OneTouch Support. The app has over 10 million downloads in Google Play. Moreover, vulnerable plugin is supplied as a pre-bundled app in some of the LG devices. These products are also built using the same technique we described earlier, a main app signed by RSupport and a plugin signed by multiple OEMs.

The research team found a flaw in RSupport's verification mechanism used in the plugin. When the main app initializes the plugin, the plugin loads the signature of the main app. The plugin then verifies that a hash of the certificate contained in the signature equals a predefined hash code.

Figure 3: Vulnerable code in the RSupport plugin

Android is open source, so the hashCode function for the Signature can be reviewed.

```
@Override
public int hashCode() {
    if (mHaveHashCode) {
        return mHashCode;
    }
    mHashCode = Arrays.hashCode(mSignature);
    mHaveHashCode = true;
    return mHashCode;
}
```

```
public static int hashCode(byte[] array) {
    if (array == null) {
        return 0;
    }
    int hashCode = 1;
    for (byte element : array) {
        // the hash code value for byte value is its integer value
        hashCode = 31 * hashCode + element;
    }
    return hashCode;
}
```

Figure 4: hashCode implementation

The signature's object hashCode implementation uses Array's object hashCode function. The described function returns a signed integer based on sum and multiply algorithm. Using an integer as a hash entry is not considered secure as an integer can only have 2^{32} which roughly amounts to only 4 billion possibilities.

This means the resulting hash can be calculated with some computing effort[3]. Once an attacker can generate a certificate that its hash code will match with the hardcoded hash code, the attacker could sign a malicious app that will communicate with the plugin and exploit its privileged permissions to take over the device.

CommuniTake: Attacking plugin architecture from a different angle

Our research team also tried to attack the plugins in other ways, such as attacking the main app to take control of the OEM-signed plugin's logic. CommuniTake is another mRST vendor that supplies an mRST product called RemoteCare. During the research, it was discovered that the main app allows changing settings through an SMS message.

One of the commands can modify the subdomain of the CnC server. For example, the message can replace it from 'foo.communitake.com' to 'bar.communtiake.com'. Allowing a change of the CnC server without authentication is problematic and could allow a denial of service attack on the client.

However, our team saw something more interesting. Looking deeper we saw that the subdomain change code does not sanitize the content of the SMS message properly allowing an attacker to insert a '/' message as part of the subdomain. For example, changing foo.communtiake.com to 'www.evil.com/.communtiake.com'.

This enables an attacker to change completely the CnC server address to an attacker-controlled CnC server. This means that we could take control of the plugin and the device only using an SMS message.

[3] The 3 things you should know about hashCode - http://eclipsesource.com/blogs/2012/09/04/the-3-things-you-should-know-about-hashcode/

CONCLUSION

Our team's research demonstrates how the Android ecosystem architecture is flawed. These flaws continue to expose sensitive information on devices, including both personal and enterprise content.

In order to support advanced usages such as remote support, vendors and OEMs abuse Android's privileged permissions mechanism. OEMs sign third party apps with their certificate to let it obtain privileged permissions. This means that third party code that doesn't go over scrutinized code review can gain access to sensitive system resources.

The problem is further intensified because vulnerable apps cannot be completely revoked. Even after a fixed version is releaed, an attacker could use the old version to get control of the device.

This issue affects hundreds of millions of Android devices, as most popular OEMs have collaborated with these vendors.

Contributing researchers

The Check Point research team extends its thanks to Pavel Berengoltz, Daniel Brodie, Andrey Pokovnichenko, and Denis Voznyuk for their individual contribution to this research.