Here was the resolution:

There turned out to be a **problem with my laptop's ADB RSA key**. I'm assuming my G3 was rejecting a bad key and disallowing my laptop to connect to it via ADB.  
  
ADB saves a key file in one of multiple places on a Windows computer, the first is in the location where adb.exe is (**C:\android**), the second is in the user's profile (**C:\Users\*\*username\**\.android**), the third place is in the Windows system files (**C:\Windows\System32\config\systemprofile\.android**), the file is simply named "**adbkey**" with no extension. If there is no key file when ADB runs, it will **generate one automatically**.  
  
Mine was **located in my user folder** (**C:\Users\*\*username\**\.android\**). All I had to do was **delete the adbkey file** (there was also a file named "adbkey.pub" which I deleted as well), **restart the adb server** in command prompt (adb start-server) and **plug my phone in**. I instantly received the RSA Fingerprint Key window on my G3 allowing connection between the two devices. Then typing "**adb devices**" returned my phone's serial number followed by "**device**" showing it was **available**. I went back to the C:\Users\*\*username\**\.android\ folder and sure enought there were **new** "adbkey" and "adbkey.pub" files.  
  
Problem finally fixed!  
  
Here are some VERY useful links which helped me come to this fix:  
[How ADB enables a secure connection](http://nelenkov.blogspot.com/2013/02/secure-usb-debugging-in-android-422.html#!/2013/02/secure-usb-debugging-in-android-422.html)  
[Reconstructing ADB's RSA key file](http://android.stackexchange.com/questions/71761/how-to-regenerate-create-adbkey-and-adbkey-pub-from-the-command-line)

刪除後須重啟adb server

Adb kill-server

來源:

The allow/deny debugging functionality, along with starting/stopping the adbd daemon, is exposed as public methods of the UsbDeviceManager system service.

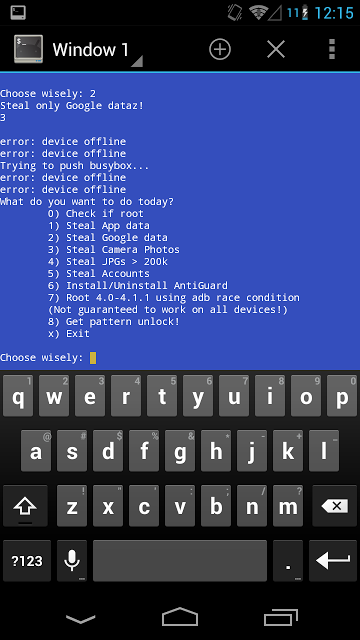
We've described the ADB authentication protocol in some detail, but haven't said much about the actual keys used in the process. Those are 2048-bit RSA keys and are generated by the local ADB server. They are typically stored in $HOME/.android as adbkey and adbkey.pub. On Windows that usually translates to %USERPOFILE%\.android, but keys might end up in C:\Windows\System32\config\systemprofile\.android in some cases (see issue [49465](http://code.google.com/p/android/issues/detail?id=49465)). The default key directory can be overridden by setting the ANDROID\_SDK\_HOME environment variable. If the ADB\_VENDOR\_KEYS environment variable is set, the directory it points to is also searched for keys. If no keys are found in any of the above locations, a new key pair is generated and saved. On the device, keys are stored in the /data/misc/adb/adb\_keys file, and new authorized keys are appended to the same file as you accept them. Read-only 'vendor keys' are stored in the /adb\_keys file, but it doesn't seem to exist on current Nexus devices. The private key is in standard OpenSSL PEM format, while the public one consists of the Base 64 encoded key followed by a `user@host` user identifier, separated by space. The user identifier doesn't seem to be used at the moment and is only meaningful on Unix-based OS'es, on Windows it is always 'unknown@unknown'.

While the USB debugging confirmation dialog helpfully displays a key fingerprint to let you verify you are connected to the expected host, the adb client doesn't have a handy command to print the fingerprint of the host key. You might think that there is little room for confusion: after all there is only one cable plugged to a single machine, but if you are running a couple of VMs, thing can get a little fuzzy. Here's one of way of displaying the host key's fingerprint in the same format the confirmation dialog uses (run in $HOME/.android or specify the full path to the public key file):

awk '{print $1}' < adbkey.pub|openssl base64 -A -d -a \

|openssl md5 -c|awk '{print $2}'|tr '[:lower:]' '[:upper:]'

We've reviewed how secure ADB debugging is implemented and have shown why it is needed, but just to show that all of this solves a real problem, we'll finish off with a screenshot of what a failed ADB attack against an 4.2.2 device from another Android device looks like:

[](https://3.bp.blogspot.com/-1qhk4Ck5Nvs/USObO6T_69I/AAAAAAAAL3c/YIr50qWytpw/s1600/p2p-adb-offline.png)