

REVERSING SLIDELOCK 1.1

Disclaimers

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this document have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched, and cracked versions were available since a lot of time. ARTeam or the authors of the paper cannot be considered responsible for damages to the companies holding rights on those programs. The scope of this tutorial as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art. We are not releasing any cracked application.

Verification

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: <http://releases.accessroot.com>

Table of Contents

1. Introduction to SlideLock 1.1	2
1.1. What is it, and why do I care?	2
1.2. Target:	2
1.3. Tools used.....	2
1.4. References.....	2
2. Investigating Slide Lock	3
2.1. Finding the SlideLOCK Classes.....	3
2.2. Analyzing the Classes.....	3
2.2.1. RightsInfo	3
2.2.2. RightsManager.....	4
2.2.3. RightsActivity	5
2.2.4. RightsActivity\$3.....	6
2.2.5. Finding the Developers Class	7
3. Patching/Removing the DRM	8
3.1. Patching the Switch Statement	8
3.2. Changing the Startup Class	8
4. What about the Application KEY???	9
5. Conclusions	10
6. Greetings.....	10

1. Introduction to SlideLock 1.1

1.1. What is it, and why do I care?

SlideLOCK is a DRM system for AndroidOS programs that aims to prevent the sharing of purchased APKs amongst users. The protection lies in special classes that the programmer must implement into his/her own code that does server-side checking with device-specific information to ensure the user is authorized to access the application. Each application is assigned a 'key' that is unique to the application and no longer than 32 characters. It is important for this key to stay unknown to users, we'll find out why later.

While DRM protection is OK in some instances it doesn't seem good in this one. Let's say you bought the game AntiBody2 for \$2, and successfully installed it on your phone, and then your phone breaks/dies/gets upgraded, shouldn't you still be entitled to play the game without purchasing it again? I think so.

So with the motive above I dug deeper into SlideLock 1.1 and will detail my findings here, and the end of the tutorial you should be able to properly remove SlideLock 1.1 protection from any application.

It is important to note that since SlideLOCK 1.1 is NOT an out-of-the-box solution, no two scenarios will be exactly the same, so you must learn from this tutorial WHY I did what I did, and not just try to mimic the steps! Here We Go!

1.2. Target:

- AntiBody2 (Game)
 - Available for 1.99 USD on SlideME Market
 - Also included in this distribution (uncracked)

1.3. Tools used

- APK Manager
 - For Decompiling/Recompiling/Signing APKs
- ADB
 - For pulling and installing APKs
- Notepad
 - For reading the smali/xml files
- Brain

1.4. References







- http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

REVERSING SLIDELOCK 1.1

2. Investigating Slide Lock

2.1. Finding the SlideLOCK Classes

Upon decompiling the application with APKManager we can find the SlideLOCK specific classes in the /smali/org/slideme/android/drm folder, and looks like this:

 RightsActivity\$1.smali	11/29/2010 11:11 ...	Text Document	9 KB
 RightsActivity\$2.smali	11/29/2010 11:11 ...	Text Document	2 KB
 RightsActivity\$3.smali	11/29/2010 11:29 ...	Text Document	6 KB
 RightsActivity.smali	11/29/2010 11:16 ...	Text Document	10 KB
 RightsInfo.smali	11/29/2010 11:11 ...	Text Document	2 KB
 RightsManager.smali	11/29/2010 11:11 ...	Text Document	12 KB

While there appears to be 6 classes, there are really only 3

- RightsActivity
- RightsInfo
- RightsManager

2.2. Analyzing the Classes

2.2.1. RightsInfo

This is the smallest of the 3 classes in the DRM system, so it seems natural to look at this one first. Upon opening the file in notepad, the first thing we'll see is that it uses an explicit constructor (one taking arguments):

```
# direct methods
.method public constructor <init>(Ljava/lang/String;Ljava/lang/Class;)V
    .locals 2
    .parameter "key"
    .parameter "mainClass"

    .prologue|
    .line 9
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
```

We see that the constructor takes a String named 'key' and a Class named 'main' this information will be important later. For now we just need to know that any RightsInfo object requires the key and the mainClass of the application to be passed in upon creation.

2.2.2. RightsManager

This is the second largest class, and by name seems to be the one that manages whether or not we are allowed to access the application. Once open in notepad we can see that this Class uses the Default Constructor (no parameters) and requires no specialized setup. It also provides 2 functions:

```
.method private static checkForwardLock(Ljava/net/URL;Z)Z
    .locals 6
    .parameter "url"
    .parameter "hasFile"
.method isForwardLocked(Ljava/lang/String;Ljava/lang/String;ILandroid/telephony/TelephonyManager;Z)Z
    .locals 5
    .parameter "key"
    .parameter "version"
    .parameter "versionCode"
    .parameter "mng"
    .parameter "hasFile"
```

In Dalvik Annotation, Z represents a Boolean Value, so as we can see above, the 2nd routine **isForwardLocked** returns a Boolean value, and is also accessible from outside the class. We can see that the first one **checkForwardLock** is private which means chances are this is a helper routine that is called from inside **isForwardLocked**.

We should remember that this class is responsible for validating us. I have provided Java equivalents of these routines here in the distribution of this tutorial and also online here:

- `checkForwardLock` -> <http://pastebin.com/fyaxzdRq>
- `isForwardLocked` -> <http://pastebin.com/YmLaBiex>

After analyzing the converted sources, it becomes apparent that in the **isForwardLocked** routine they are grabbing a LOT of device-specific user information (most of which is NOT needed for their purposes) and sending it to their server. The information includes:

- Device ID -> Only one really needed
- Network Operator
- Network Operator Name
- Network Country ISO
- Device Software Version
- Phone Type
- Network Type
- Sim Country ISO
- Sim Operator
- Sim Serial Number -> Maybe this one too (For GSM users)
- Subscriber ID
- Network Roaming

While the information provided here in the RightsManager class isn't directly related to the removal of the SlideLOCK DRM, it is quite alarming to see the amount of data they are pulling for no apparent reason.

2.2.3. RightsActivity

This is the largest of the 3 classes and perhaps the most complex. For this reason I will not be providing a full conversion of the class into Java, but rather just the important pieces. First thing to notice when opening up the file in notepad is that this is an abstract class, which means that the person using this class (the developer) must create a class that inherits from this one and implements the missing functions.

Also we should note the declared constants in the file as they may be helpful later:

```
.field private static final LOCKED:I = 0x0  
.field private static final ERROR:I = 0x1  
.field private static final IO_ERROR:I = 0x2  
.field private static final NOT_LOCKED:I = 0x3
```

The functions contained in this class are as follows:

```
.method private startMainApplication()V  
.method public exitOnError()Z  
.method public abstract getRightsInfo()Lorg/slideme/android/drm/RightsInfo;  
.method public final onCreate(Landroid/os/Bundle;)V  
.method protected onCreateDialog(I)Landroid/app/Dialog;
```

Note that the `getRightsInfo()` routine is labeled abstract, and as such is not implemented here, but rather in the developer's class that inherits this one.

The routine **startMainApplication** does just what it says, it calls `getRightsInfo` to get the `RightsInfo` object associated with this program, which if you recall holds the `Applications KEY` and the true Main class, for all intents and purposes we can consider this Main Class as OEP :P

The `exitOnError()` routine just tells the DRM whether or not the app should exit if there is an error when trying to validate (i.e. no internet connection) so this one is unimportant for our needs. Might be worth noting that by default this routine returns 0 or FALSE but the developer may override this function to return TRUE. That is up to the implementation and developer's wishes.

The last 2 routines are pretty unimportant to us so we will not discuss them here.

2.2.4. RightsActivity\$3

I came across this class simply by just looking through them all; one thing you'll notice in this class is that it contains some bad boy messages:

- `const-string v2, "This application is locked to another device. Shutting down."`
- `const-string v2, "An error detecting lock on device. Is network active?"`
- `const-string v2, "There was a problem with the network."`

Now we need to learn some things about smali code. It is equivalent to ASM code for x86 reversing. With some practice it is as readable as java code, but logic statements such as loops and ifs and switches can be confusing. For those who have developed in java before we know that switches must act upon numerical values only, which is different than .NET languages where you can switch on a string.

The routine that contains these bad-boy strings is using a switch statement to determine which one to go to, we can see the setup for the switch statement here:

```
iget v1, p1, Landroid/os/Message;->what:I  
packed-switch v1, :pswitch_data_0
```

The code above says to read in the integer value of Message.what and store it in the v1 variable. The Message Object they are referring to is p1 (parameter 1) that was passed into the routine. After the value is stored, they then perform a switch on the value in v1, and use :pswitch_data_0 to define the range of values used in the switch. By doing a Ctrl+F on that string we find it here:

```
:pswitch_data_0  
.packed-switch 0x0  
:pswitch_0  
:pswitch_1  
:pswitch_2  
:pswitch_3  
.end packed-switch
```

This shows that the values for the switch start at 0 and that there are 4 test cases, each case being the next number higher which results in a switch on the values 0 – 3 inclusive. Recall the constants we found earlier :

```
.field private static final LOCKED:I = 0x0  
.field private static final ERROR:I = 0x1  
.field private static final IO_ERROR:I = 0x2  
.field private static final NOT_LOCKED:I = 0x3
```

Seems like this is where they come into play ☺.

The lines between .packed-switch and .end packed-switch show the labels that define the start of that case's code. So to find the code for the case that v1 == 0 simply do a Ctrl+F for “:pswitch_0”. A couple lines down from this we see the bad-boy message that the application is locked, this confirms our suspicion that it is here that the above constants are used. This also looks like a good place to patch <<hint>>

You can find the java equivalent sources of the above RightsActivity class (remember it's only a partial conversion) in this package and also here: <http://pastebin.com/9P8SE9a9>

2.2.5. Finding the Developers Class

We could stop here with what we know and successfully patch the application to bypass the lock, but we're here to reverse as much as we can. So the next thing we need to figure out is which class the developer created that extends the RightsActivity class. There is a few ways we could do this, but only one sure fire way.

We could check the AndroidManifest.xml file to see if the class is set to the startup class, but this is not guaranteed. The only sure fire way is to navigate to the smali folder in a command prompt and execute this line of code to rename all .smali files to .txt:

```
FOR /R %x IN (*.smali) DO ren "%x" *.txt
```

Once all the files are renamed to .txt, we need to search all file contents for this string:

```
.super Lorg/slideme/android/drm/RightsActivity;
```

The results of this search will show that the file SlideLock.smali located in /smali/creafire/com/antibody2 extends the RightsActivity Class. Remember, whichever class extends the RightsActivity one MUST implement the getRightsInfo() routine. So let's take a look at the developer's implementation ☺.

```
.method public getRightsInfo()Lorg/slideme/android/drm/RightsInfo;
    .locals 3

    .prologue
    .line 9
    new-instance v0, Lorg/slideme/android/drm/RightsInfo;

    const-string v1, "dnsfgkgi5h44k5uFTJshgui45hgdsfnb"

    const-class v2, Lcreafire/com/antibody2/Main;

    invoke-direct {v0, v1, v2}, Lorg/slideme/android/drm/RightsInfo; -> <init>

    return-object v0
.end method
```

We see here that they are creating a new instance of the RightsInfo Object, which if we remember takes a Key and a Class as constructor arguments. The object is set to the variable v0, then v1 is assigned a 32 character string (the key), and v2 is assigned to class Main located in the /smali/creafire/com/antibody2 folder. They then call the RightsInfo object's constructor and pass these 2 variables in. Seems we've found the elusive application key and the "OEP". Now let's put everything we have found together :P

3. Patching/Removing the DRM

With all the information we've gathered, we can easily come up with a 2 ways to patch or remove the DRM from this application, I will cover both options for the sake of completeness. 1 option given is clearly the most desirable.

3.1. Patching the Switch Statement

Remember the switch statement that accepts the values 0 – 3? Well we could insert a single line of smali code right before the switch statement begins to alter the value it uses. We could force the value to always equal 3 (good value) prior to entering the switch statement.

The line of code we could use is this:

```
const/4 v1, 0x3
```

This line tells the application to assign the constant 4-bit value 0x3 to the variable v1.

After our patch the code would look like this:

```
iget v1, p1, Landroid/os/Message;->what:I  
const/4 v1, 0x3  
packed-switch v1, :pswitch_data_0
```

The above code will ensure that regardless of the status code really sent it will ALWAYS equal 3 prior to the switch statement and thus ALWAYS think we passed. After adding this line of code we simply need to recompile/resign/reinstall the application. And tada! The Server check has been defeated!.

NOTE: Because we altered the file and signed it with a different key than the original application we MUST uninstall the application first prior to installing our modified one.

3.2. Changing the Startup Class

Recall that during our investigation, we discovered what the true startup class is, effectively the OEP of the application. We could simply try to change the applications startup class to Main and bypass SlideLOCK all together :P. Let's take a look.

Every Android application contains an AndroidManifest.xml file that tells the operating system certain things about the application. Things like what permissions the application requires to run properly, and whether the screen should be portrait or landscape. Also included in this file is a list of all Activities (or screens) the application contains, and specific things about them as well.

Let's take a look at the AndroidManifest file for this application.

REVERSING SLIDELOCK 1.1

```
<activity android:label="@string/app_name" android:name=".SlideLock">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:label="@string/app_name" android:name=".Main" android:screenOrientation="landscape"
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

We can see here that there are 2 activities listed, SlideLock being 1 and Main being the other. Notice how BOTH have the <category> of LAUNCHER and BOTH have the <action> of MAIN. This was a design error on the developer's part. The only activity that should be listed here as MAIN and LAUNCHER is the SlideLock. I'm sure you noticed upon installing the original APK there were 2 entries for AntiBody2. This is the reason. The first entry that shows up is the SlideLock'd Activity that uses the DRM to verify you are allowed to run it. The second however is a completely unprotected activity!

All we have to do here, is remove the <activity> entry that lists SlideLock as the activity name. This will result in only one entry in the App Drawer, and that entry will automatically launch the Main class (which remember has no DRM in it).

If however you come across an application that has this part done correctly all you need to do is replace the Activities name with that of the real Main Class that we found in part 2.2.5 and recompile. The resulting APK will bypass all the DRM code leaving you with an unbound application :D

4. What about the Application KEY???

While the key itself is useless to us for removing the application's DRM, it could be useful in other scenarios. Let's say you purchase an application for \$0.50, that's pretty cheap, let's also say you can find out the application's key. Once you've done that all you would need to do to use a different (perhaps more expensive) application is swap the expensive app's key with the key of an application you legitimately bought. After you swapped it so both applications have the key of the \$0.50 application, in theory they should both pass validation. You will need to change other things too like Version and VersionCode, but that will not be covered here because here at ARTeam we do not agree with piracy.

5. Conclusions

Through our adventure today into SlideLOCK 1.1 we have learned quite a bit about the DRM and the company itself. We have learned how to bypass the Server Check by inserting smali code to affect the switch statement, we also learned how to completely bypass the DRM mechanisms by altering the AndroidManifest.xml file, and we learned that they are using quite a bit of information (or so it seems) to verify that you are authorized to run the application.

It would seem that removing the DRM completely would be the safest bet. By removing it we prevent SlideLOCK from sending our information to the servers. According to their site they are only grabbing your Device ID to verify, but we learned they grab a LOT more than that, so what are they doing with all the additional data? I will leave that up to you to decide!

6. Greetings

I would like to thank the following people (in no particular order)

- Nilrem for getting me involved in RCE
- ARTeam for becoming my 'home'
- SSIEvIN for the Template
- Ghandi for all the long talks on reversing and the endless hours of teaching :P
- JesusFreke for Baksmali/Smali
- Daneshm90 for APK Manager
- The reader for um... ya reading this :P

-- Nieylana