Smart Phones Security

How (Not) To Summon The Devil

Tomáš Rosa
http://crypto.hyperlink.cz

Smart Cards & Devices Forum 2012, May 17th, Prague

Abstract

- We present several real-life vulnerabilities that the author has found when experimenting with mobile applications.
  - It is not hard to guess the area of those application… (*hint: try author's personal web*).
- These are often based on innocent-looking constructions.
  - Furthermore, once known, the observations seem really trivial.
  - When unknown, however, they remain silently hidden waiting for the day of their exploit.
  - The final impact can be then really dramatic…

Part ONE
Introduction

ATA Scenario

Definition. *Let the After-Theft Attack (ATA) be any attacking scenario that assumes the attacker has unlimited physical access to the user's smart phone.*

- Imagine somebody steals your mobile phone…
- Despite being really obvious threat, it is often totally neglected in contemporary applications.

- By a robbery, the attacker can even get access to unlocked screen, hence receiving another considerable favor!

Forensic Techniques Lessons

- Hackers conferences are not the only one place where to look for an inspiration.
- There are also forensic experts who publish very interesting results [4], [5], [15], [24].
  - Actually, they often take hacking techniques and refine them to another level of maturity.
  - The main purpose is to prosecute criminals, of course.
  - But it is like a pistol – it is a question of who is holding the gun…
  - Anyway, security experts shall definitely consider looking into forensic publications, at least time to time.

Cross-Platform Attacks

- Interestingly, forensics also shows how to exploit certain access to both the mobile phone and the "paired" computer.
  - Such situation is rarely studied at hackers conferences, yet.
  - This model, however, fits nicely cross-platform attacks that arise e.g. with banking applications.
  - Again, we shall really look at what those forensic experts can do…

2root || !(2root) ? Don't!

- Running highly sensitive applications on rooted or jailbroken devices shall be avoided.
  - Already rooted or jailbroken device definitely makes the attacker's job easier.
    - In the same way as it already helps in forensics [15], [24].
    - Furthermore, <u>the runtime protection is almost none.</u>
    - As you can also see in Cycript experiments in Part Three.
  - Sometimes, the attacker can even hope to get an access to memory dumps of sleeping processes.
    - Consider the unlocked screen and the ability to run anything as root with no sandbox…

2root || !(2root) ? Do!

- We shall admit, however, the device gets rooted or jailbroken without user's incentive.
  - In JailbreakMe tools, for instance, it was enough to point the Mobile Safari at innocent-looking page [28].
- Developers, therefore, shall test their applications on such devices!
  - Just to be able to see their applications from other perspective…
  - From the perspective of the enemy.

Experimental Setup

- Experiments noted in this presentation were exercised on:
  - (rooted) Google Nexus S I9023XXKF1 with Android version 2.3.6, build GRK39F,
  - (jailbroken) Apple iPhone 4S – 16 GB MD235B with iOS v. 5.0.1 (9A406).

Part TWO
Latent PIN Prints

Memento ATA

- We shall assume that:
  - Once having unlimited physical access to the mobile device,
  - the attacker can read any plaintext data stored in its memory.
  - This also applies to certain encryption keys! [2], [14], [15], [23], [24].
- Despite not being trivial, we shall further assume this also applies to the content of the volatile RAM.

PIN Prints

- This can be any direct or indirect function value that:
  - once known to the attacker,
  - can be used for a successful brute force attack on the PIN,
  - under the particular attack scenario.
- Principally, the same applies to general passwords, too.
  - However, we can mitigate the risk by enforcing strong password policy here.

Pitfall No. 1

- There was RSA private key encrypted by a derivative of a decimal PIN.
  - According to PKCS#1 [22], there is a huge redundancy based on the ASN.1 structure syntax [8].
  - Furthermore, there is a terrible amount of algebraic-based redundancy in the private key numbers themselves [18].
- So, the decimal PIN was in fact packed together with the encrypted key store.
  - …as a bonus gift to the attacker!

Pitfall No. 2

- If the PIN is used for OTP generation,
  - then any OTP itself is a valuable PIN print.
- This is true even if the OTP is also based on some symmetric key.
  - Or, we have to prove the key cannot be retrieved by respective techniques like [2], [14], [15], [23], [24].
- Therefore, we shall:
  - not store OTPs in permanent memory,
  - wipe OTPs out of the volatile memory as soon as possible.

Padding Issues

- Consider the HOTP according to RFC 4226.
  - This is a popular OTP generator based on HMAC-SHA-1.
  - Its reference Java implementation [16], however, contains a security flaw.
    - OK, it is a reference design in the sense of test vectors.
    - On the other hand, the RFC does not warn clearly that this code shall not be used for real implementations.
    - Especially on Android, it is probably tempting to simply copy-paste the code. Do not do that!

Padding by RFC 4226

```
result = Integer.toString(otp);
while (result.length() < digits) {
  result = "0" + result;
}
return result;
```

Behind Those “+” and “=”

- With each iteration, there are two new instances created:
  - (“+”) `java.lang.StringBuffer` or `StringBuilder` to perform the concatenation,
  - (“=”) `java.lang.String` to hold the result.
- However, the references to the previous iteration `result` and to the concatenation instance are lost.

Memory Footprint

- With each iteration, we have at least one copy of the precious OTP left unattended in the memory.
  - We do not have a reference to them.
  - So, we cannot wipe them securely!
- Furthermore, there is the unfortunate choice of using `String` to hold the `result`.
  - This is by standard immutable object, so we need to invest an extra effort to wipe such values properly.

Android Proof-Of-Concept

- We have compiled the original HOTP padding procedure for Gingerbread.
  - To exhibit the faulty behavior, we have deliberately shortened the input integer, so we were able to see the padding in action.
  - In particular, we set:
    - **`otp = 755224,`**
    - **`digits = 9.`**

Dalvík Code View by IDA Pro

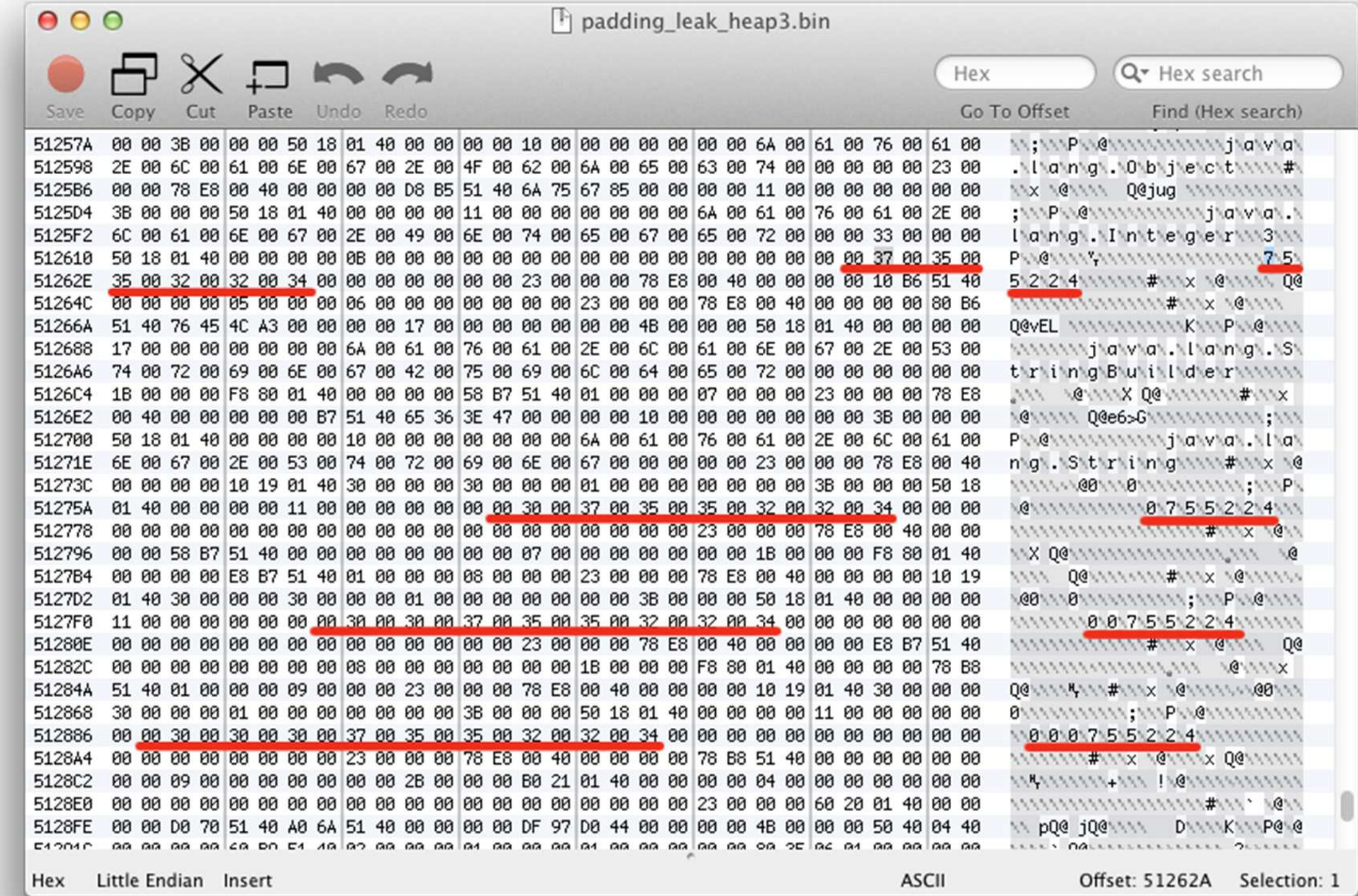
```
                invoke-static                  {p0}, <ref Integer.toString(int) imp. @ _def_Integer_toString@LI>
                move-result-object             v0

loc_4A0:                            # CODE XREF: PaddingLeak_doPad@LII+3C↓j
                invoke-virtual                 {v0}, <int String.length() imp. @ _def_String_length@I>
                move-result                    v1
                if-lt                          v1, p1, loc_4AE

locret:
                return-object                  v0
# ---------------------------------------------------------------------------

loc_4AE:                            # CODE XREF: PaddingLeak_doPad@LII+10↑j
                new-instance                   v1, <t: StringBuilder>
                const/16                       v2, 0x30
                invoke-static                  {v2}, <ref String.valueOf(char) imp. @ _def_String_valueOf@LC>
                move-result-object             v2
                invoke-direct                  {v1, v2}, <void StringBuilder.<init>(ref) imp. @ _def_StringBuilder__init_@V
                invoke-virtual                 {v1, v0}, <ref StringBuilder.append(ref) imp. @ _def_StringBuilder_append@LL
                move-result-object             v1
                invoke-virtual                 {v1}, <ref StringBuilder.toString() imp. @ _def_StringBuilder_toString@L>
                move-result-object             v0
                goto                           loc_4A0
```

Android Leakage Illustration



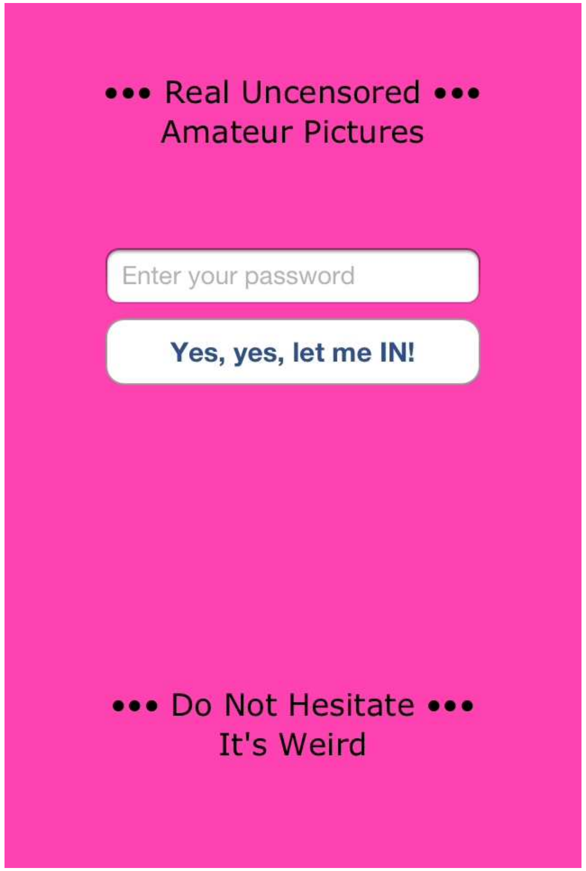
Part THREE
My name is C. Objective-C

Note on the Root Account

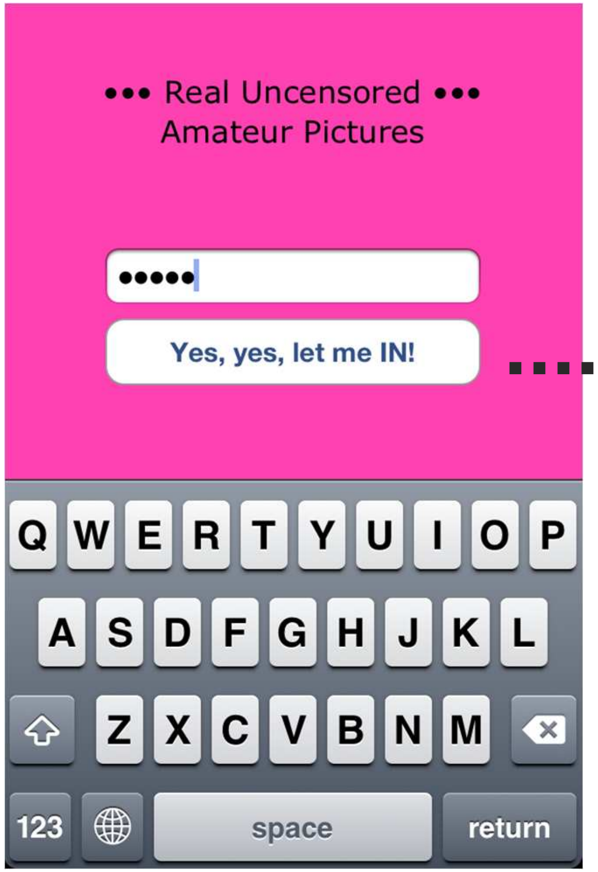
- The following experiments expose (ab)using the *root* account on a jailbroken iPhone.
  - It was, however, verified that everything shown here can be done under the *mobile* account as well.
  - Once a jailbreak environment is already set, the *root* is not such important for a malicious application.
    - Obviously, it is potentially dangerous to install any “underground” (Cydia, etc.) application side by side with e.g. sensitive banking application.
    - Recall, almost all runtime protections are gone!
    - We shall, on the other hand, constantly bear on mind that a kind of jailbreak can happen without user’s incentive.

Weird Pictures Demo

- Well, it would not be fair to use real-life applications here.
- We will use a modest iPhone joke that was written especially for this purpose to exhibit all those weaknesses we want to talk about.



Password: “kubrt”



It’s just the front camera in action…

Cycript

- Delicate combination of JavaScript and Objective-C interpreter running on iOS [31], [32].
  - Provides REPL (Read-Eval-Print Loop) interface.
- It can attach to an already running process and start commanding its Objective-C runtime.
  - It uses MobileSubstrate framework to do that [32].
  - This requires a jailbreak, but remember what we said before – it can happen without user's incentive.
- Its original purpose probably was not application hacking (in security sense).
  - Anyway, it is an excellent tool for vulnerability research and demonstration [24].

Cycript Taste

- As an illustration, we show a Cocoa Touch style `alert()` function in Cycript.

```
function cocoAlert(msg) {
   var alertView = [[UIAlertView alloc]
      initWithTitle:"Alert"
      message:(msg!==undefined) ? msg : ""
      delegate:null
      cancelButtonTitle:"OK"
      otherButtonTitles:null];
   [alertView show];
   [alertView release];
}
```

Back to Weird Pictures

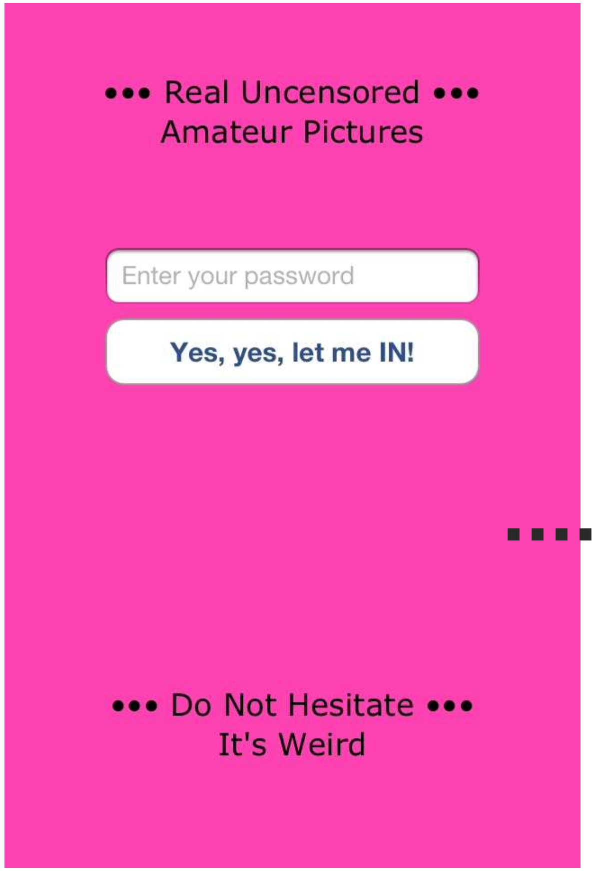
- How is the login view managed?
  - What if it is just a modal view controller presented by the root view controller of the application?
  - We mean having something like this in e.g. the method `applicationWillResignActive:` [33]:

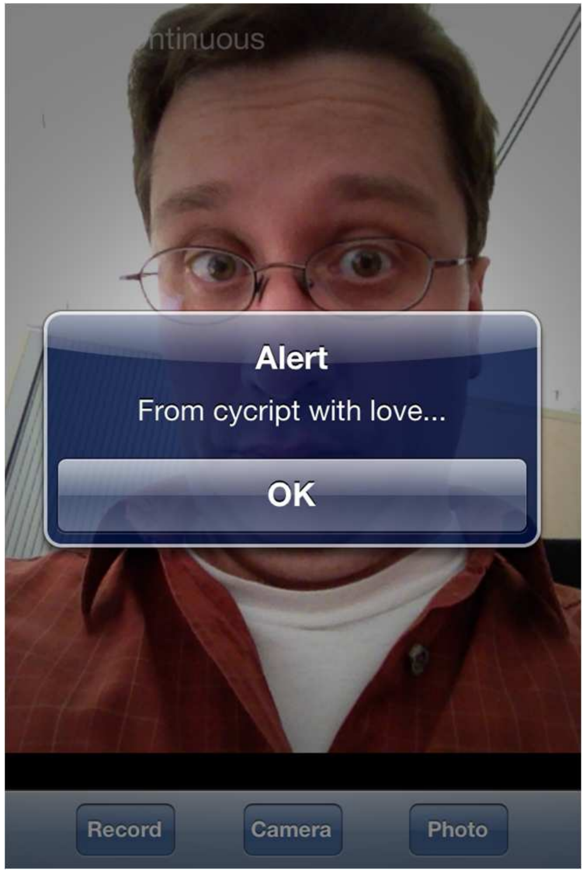
```
[self.viewController presentViewController:
     [WPLoginViewController getDefault]
 animated:NO
 completion:^{NSLog(@"modal login");}
];
```

Consider This (hack1.cy)

```
function AppVC() {
    var window = [UIApp keyWindow];
    this.viewController = [window
    rootViewController];
}
AppVC.prototype.unlock =
    function(animated/*opt*/) {
    [this.viewController
    dismissModalViewControllerAnimated:animated];
    cocoAlert("From cycript with love...");
}
var ac = new AppVC();
ac.unlock();
```

cycript -p WeirdPictures hack1.cy





Lesson Learned

- Do not assume that plain GUI provides any reasonable data protection.
- We shall assume the attacker can get access to all local plaintext data.
  - Especially important to consider under the After-Theft Attack assumption.
  - If we need to control data access, we shall encrypt this data [24].
    - But pay really high attention not to create any useful PIN prints this way!

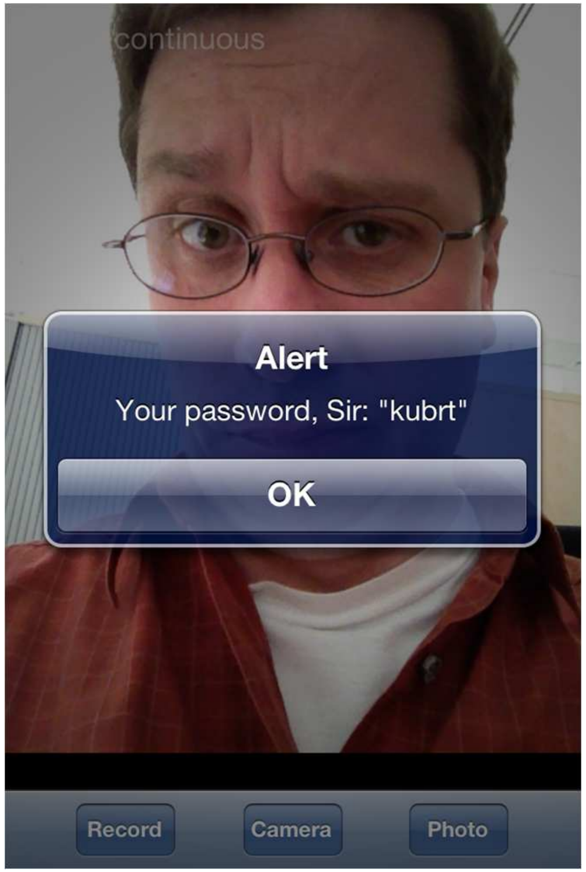
Consider Yet This (hack2.cy)

```
function LoginVC() {
    this.viewController = [WPLoginViewController
    getDefault];
}
LoginVC.prototype.showPwd = function() {
    var pwd = [[this.viewController passwordField] text];
    if (pwd == null)
        cocoAlert("Sorry Sir.");
    else
        cocoAlert("Your password, Sir: \"" +
    pwd.toString() + "\"");
}
var lc = new LoginVC();
lc.showPwd();
```

cycript -p WeirdPictures hack2.cy

- We shall consider using one-way derivatives, if we *really* need to keep user secrets in memory for some purpose.
  - Furthermore, it is wise not to expose anything like

  -(id)passwordField !

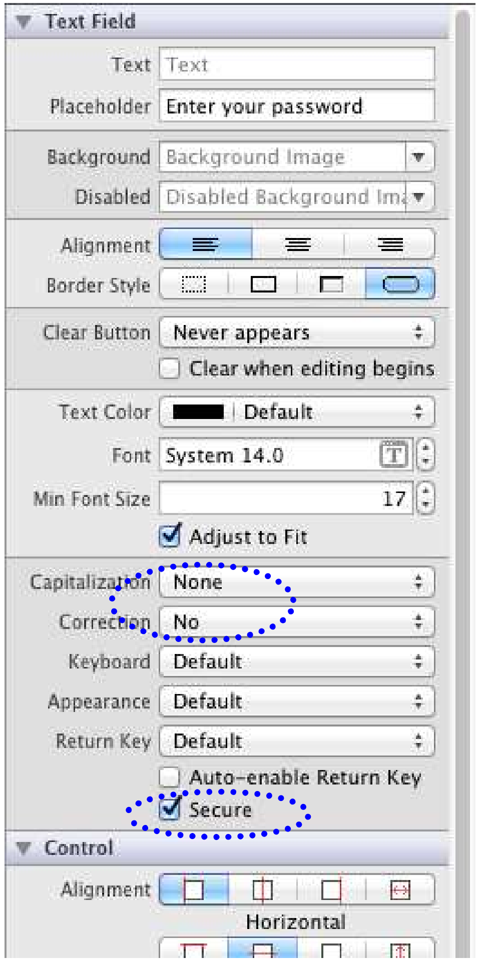


Cocoa Shaken, Not Stirred

- So far, we had the code under our control.
  - When we understand what is wrong, we can fix it.
- What if the problem is out of reach of our hands?
  - For instance, in Cocoa Touch.
  - Right around the `UITextField` control.

UITextField in Weird Pictures

- We use this control view to let users to type their password.
- Of course, we have marked it “Secure”.
  - But, is it enough?



Consider This Gdb Script

```
set variable $sel = (void*)sel_getUid("text")
set variable $cla = (void*)objc_getClass("UITextField")
set variable $addr = (void*)(((unsigned
    long)class_getMethodImplementation($cla, $sel)) & 0xFFFFFFFE)

break *($addr+118)
  commands
    printf "from: 0x%lx\n", $lr
    if ($lr != 0x0)
      x/i $lr
    end
    printf "return: 0x%lx\n", $r0
    if ($r0 != 0x0)
      x/a $r0
      call (unsigned char*)CFStringGetCStringPtr($r0, (unsigned
    long)CFStringGetSystemEncoding())
    end
    c
  end
```

saved as /var/root/tfexp.gdb

Notes on the Gdb Script

- Loaded by the gdb `source` command.
  - We use the original Xcode gdb running right on the iOS device [17].
  - We attach to the existing process of WeirdPictures.
- Well, there may be ASLR [25].
  - So, we abuse the wonderful Objective-C runtime to query for the `-[UITextFiled text]` implementation.
  - We then setup a breakpoint at the end of this method.
    - *This offset can change, we have verified it for iOS v. 5.0.1 (9A406) and v. 5.1 (9B176).*
  - This way, we can monitor who is querying our precious `passwordField` and what is the result.

Loading into Gdb

```
(gdb) source /var/mobile/tfexp.gdb
Breakpoint 1 at 0x324d508a

(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x324d508a <-[UITextField text]+118>
        printf "from: 0x%lx\n", $lr
        if ($lr != 0x0)
          x/i $lr
        end
        printf "return: 0x%lx\n", $r0
        x/a $r0
        if ($r0 != 0x0)
          x/a $r0
          call (unsigned char*)CFStringGetCStringPtr($r0,
                   (unsigned long)CFStringGetSystemEncoding())
        end
        c
(gdb) c
Continuing.
```

What a Surprise…

- As the user starts typing on the virtual keyboard, we can see:

```
…
Breakpoint 1, 0x324d508a in -[UITextField text] ()
from: 0x3242bb91
0x3242bb91 <-[UITextField _updateAutosizeStyleIfNeeded]+69>…
return: 0x14d750
0x14d750: 0x3f4712c8 <OBJC_CLASS_$___NSCFString>
$2 = (unsigned char *) 0x0

Breakpoint 1, 0x324d508a in -[UITextField text] ()
from: 0x3242bb91
0x3242bb91 <-[UITextField _updateAutosizeStyleIfNeeded]+69>…
return: 0x12f860
0x12f860: 0x3f4712c8 <OBJC_CLASS_$___NSCFString>
$3 = (unsigned char *) 0x35c2c1 "k"
```

...And It Continues...

```
Breakpoint 1, 0x324d508a in -[UITextField text] ()
from: 0x3242bb91
0x3242bb91 <-[UITextField _updateAutosizeStyleIfNeeded]+69>:       movw          r6, #5276     ; 0x149c
return: 0x1483f0
0x1483f0:     0x3f4712c8 <OBJC_CLASS_$___NSCFString>
$4 = (unsigned char *) 0x159ae1 "ku"

Breakpoint 1, 0x324d508a in -[UITextField text] ()
from: 0x3242bb91
0x3242bb91 <-[UITextField _updateAutosizeStyleIfNeeded]+69>:       movw          r6, #5276     ; 0x149c
return: 0x3179f0
0x3179f0:     0x3f4712c8 <OBJC_CLASS_$___NSCFString>
$5 = (unsigned char *) 0x35eed1 "kub"

Breakpoint 1, 0x324d508a in -[UITextField text] ()
from: 0x3242bb91
0x3242bb91 <-[UITextField _updateAutosizeStyleIfNeeded]+69>:       movw          r6, #5276     ; 0x149c
return: 0x15a3d0
0x15a3d0:     0x3f4712c8 <OBJC_CLASS_$___NSCFString>
$6 = (unsigned char *) 0x13dca1 "kubr"

Breakpoint 1, 0x324d508a in -[UITextField text] ()
from: 0x3242bb91
0x3242bb91 <-[UITextField _updateAutosizeStyleIfNeeded]+69>:       movw          r6, #5276     ; 0x149c
return: 0x113e40
0x113e40:     0x3f4712c8 <OBJC_CLASS_$___NSCFString>
$7 = (unsigned char *) 0x15a3d1 "kubrt"
```

…Then Comes Our Query

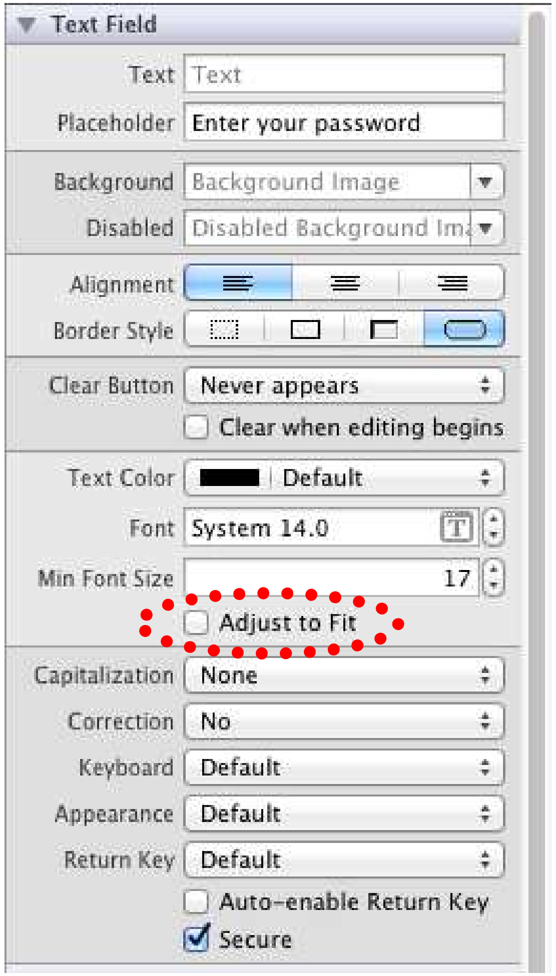
```
Breakpoint 1, 0x324d508a in -[UITextField text] ()
from: 0x7e47
0x7e47 <-[WPLoginViewController login:]+75>…
return: 0x1325b0
0x1325b0:     0x3f4712c8 <OBJC_CLASS_$___NSCFString>
$8 = (unsigned char *) 0x1544e1 "kubrt"
```

What The Hell...?!

- Apparently, we are not the only one who is interested in the `passwordField` value.
  - For some reason, UIKit framework (of Cocoa Touch) continuously monitors this value, too.
  - Furthermore, it was observed that this activity leads to a considerable memory footprint.

Then, We Start Getting the Idea

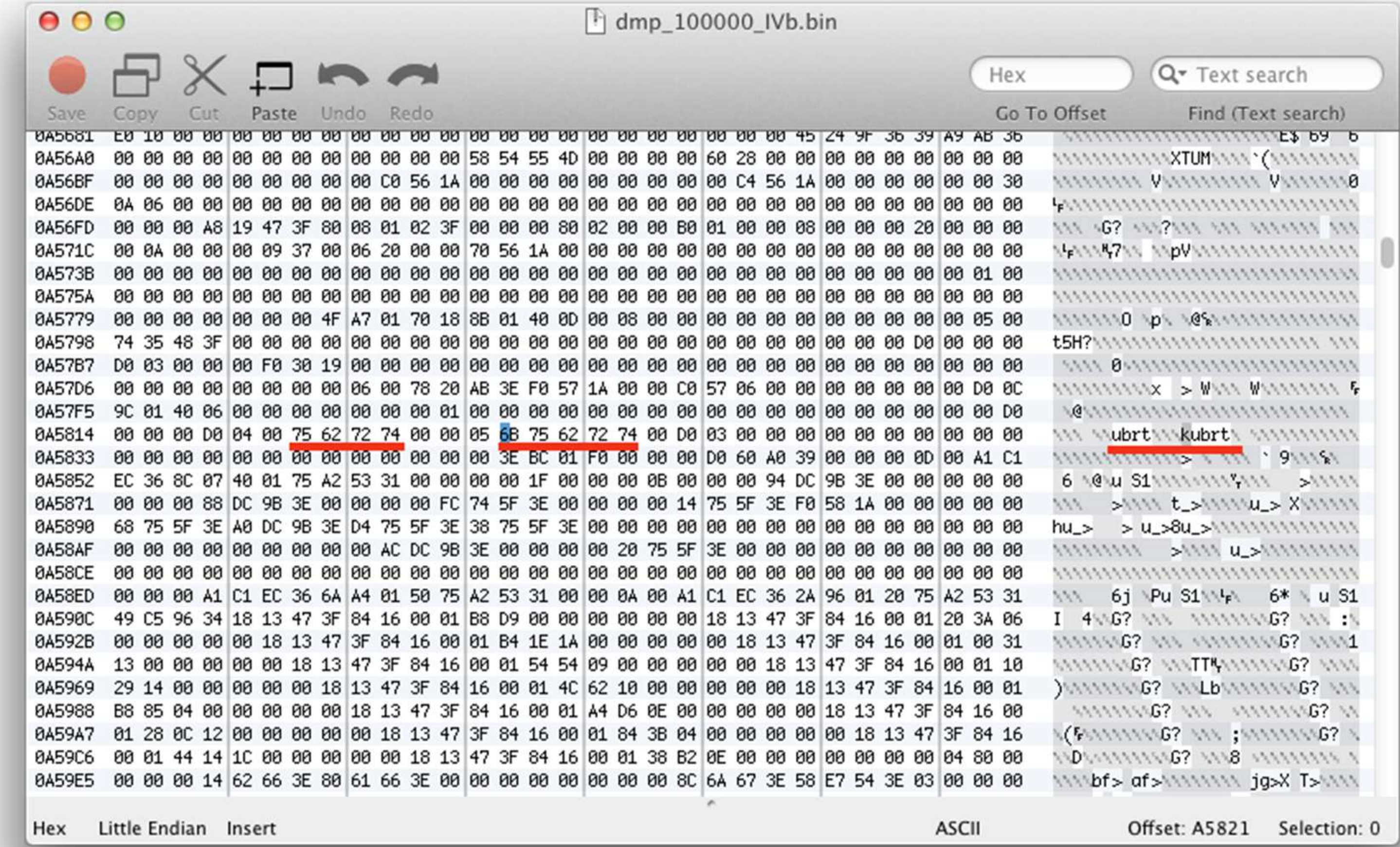
- We shall also turn off the automatic font adjusting.
  - This rule would remain silently hidden if we did not experiment with the gdb and jailbreak!
- However, one question still remains.
  - Is this enough, or could there be a similar issue somewhere else???
  - Or, we may already need the “Adjust to Fit” flag set…



Spraying The Secret

- Various parts of our secret string were identified in dumps of the process memory.
  - Of course, we have eliminated other potential sources for the experiment.
  - Anyway, the values were found not only at the addresses noted in the previous gdb listing.
  - Probably there is some further processing that finally "sprays" these values around the memory heap.
    - Again - did we already stop the whole leakage or just sealed up one particular hole?
    - Actually, we have already seen further automatic gathering made for the `-[UITextField _text]`…

Illustration of Heap Pollution



Memento ATA Again

- Regarding the After-Theft Attack, this can be really dangerous.
- According to the official documentation:
  - *"…[iOS] keeps suspended apps in memory for as long as possible, removing them only when the amount of free memory gets low…"* [33]
  - From the user perspective, however, the application is simply done.

Risk Assessment

- What if an attacker steals a device with such a suspended process?
  - It is a question of being able to dump RAM without cycling the power.
  - We cannot claim that there is always a chance to get these data.
    - However, we either cannot claim it will not happen.
  - Clearly, end users shall not jailbreak their devices with sensitive applications.
    - As this can help the attacker considerably.
  - Developers, on the other hand, shall test their own application with a jailbreak!
    - As this helps them to see things in a different light…

What Shall We Do With Drunken Framework?

- Obviously, it is not wise to try to improve the UIKit framework itself.
  - We cannot be sure we have patched all holes.
  - Furthermore, there can be serious compatibility issues.

- Simple workaround is to avoid using `UITextField` at all and devise our own control view instead.
  - Sometimes, however, we want to use `UITextField` for compatible look-and-feel, etc.
  - Then, the cryptography is here to help…

Encrypted Keyboard Idea

- Devise custom keyboard that for each character typed generates its cryptogram.
  - The `UITextField` does no longer operate with plaintext.
  - It is being fed by “crypto-chars” instead.

- When finished, we retrieve the crypto-char text, decrypt it, and wipe out the ephemeral key used.
  - The heap can still be polluted.
  - But this is just a gibberish text, since the key is already gone.

Clear Idea, But…

- The implementation presents some interesting problems:
  - We are talking about some kind of a stream cipher [18].
  - So, how to solve the keystream synchronization?
  - How to cope with potential keystream reuse?
  - How to generate the keystream fast enough?
- OK, this deserves a separate lecture.
  - Please see:
    - Dvořák, P. and Rosa, T.: *How the Brave Permutation Rescued a Naughty Keyboard*,
    - at Mobile DevCamp 2012, http://www.mdevcamp.cz/

Part FOUR
Cross-Platform Attacks

Overview

- We first show the Screen Lock Bypass (SLB) application at work.
  - This is an interesting forensics/hacking technique in itself.
- We then conclude by noting a possible way of an effective malware cross-infection.
  - The observation is trivial. Its impact, however, can really be dramatic.
  - Especially in the area of two-factor authentication applications.

Version Alert

- The following part of this presentation was researched in November 2011.
  - It was the time of Android Market and the Gingerbread was quite recent version.
  - It is the era of Google Play and Ice Cream Sandwich, now.
  - The ideas and concepts presented here, however, still apply.

Screen Lock Bypass (SLB)

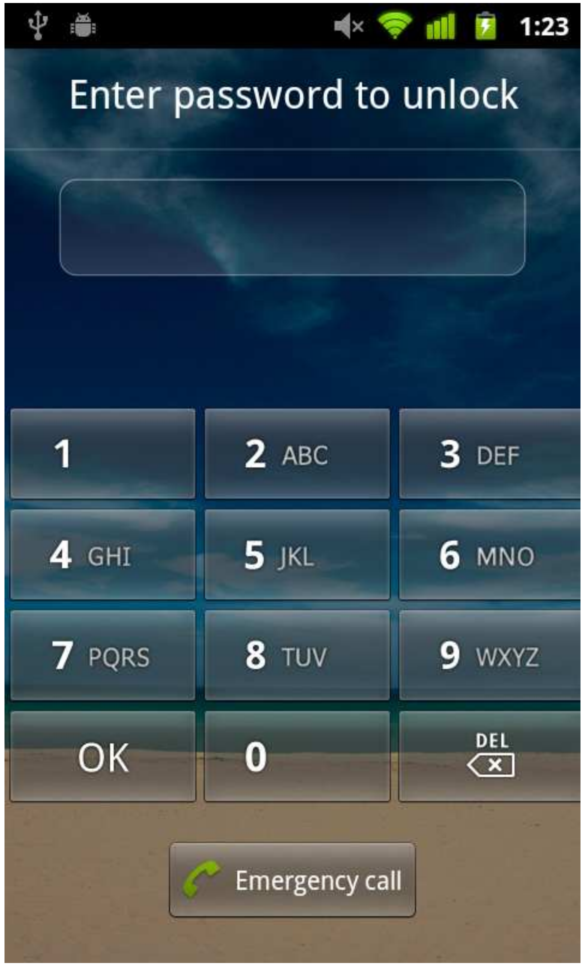
- Developed by Thomas Cannon [29], popularized by Andrew Hoog [15], and freely available on the Android Market (now Google Play).
- Its official purpose is to help users who accidentally forgot their screen lock gesture or PIN.
  - Anybody who knows the login name/password for the Gmail account associated with the particular Android device can use this application to try to unlock the screen.
  - The success ratio may not be 100 %, but it is quite high anyway.
  - In particular, we did not encounter any problem during several trials we have made for this presentation.

The Dark Side

- As was already noted in [15], this application may be used *not only* by the legitimate device owner.
  - Just anybody, who knows the respective Gmail credentials can give it a try.
  - Obviously, the Gmail credentials seems to be quite “magic”.
    - And that is just the beginning…

The Screen (Un)Lock At Work

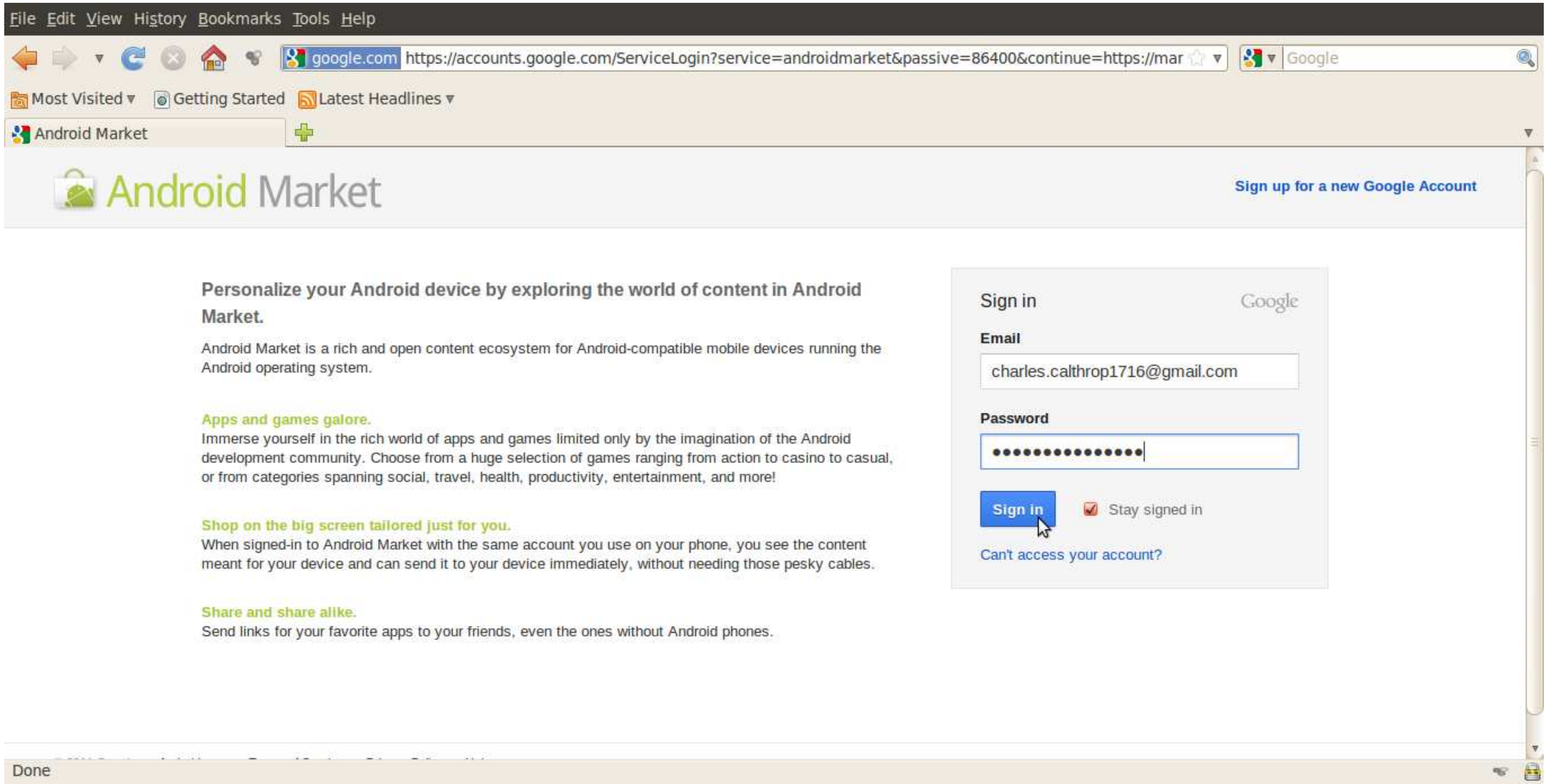
- Let us assume that the device display is locked by a PIN that we somehow cannot recall…



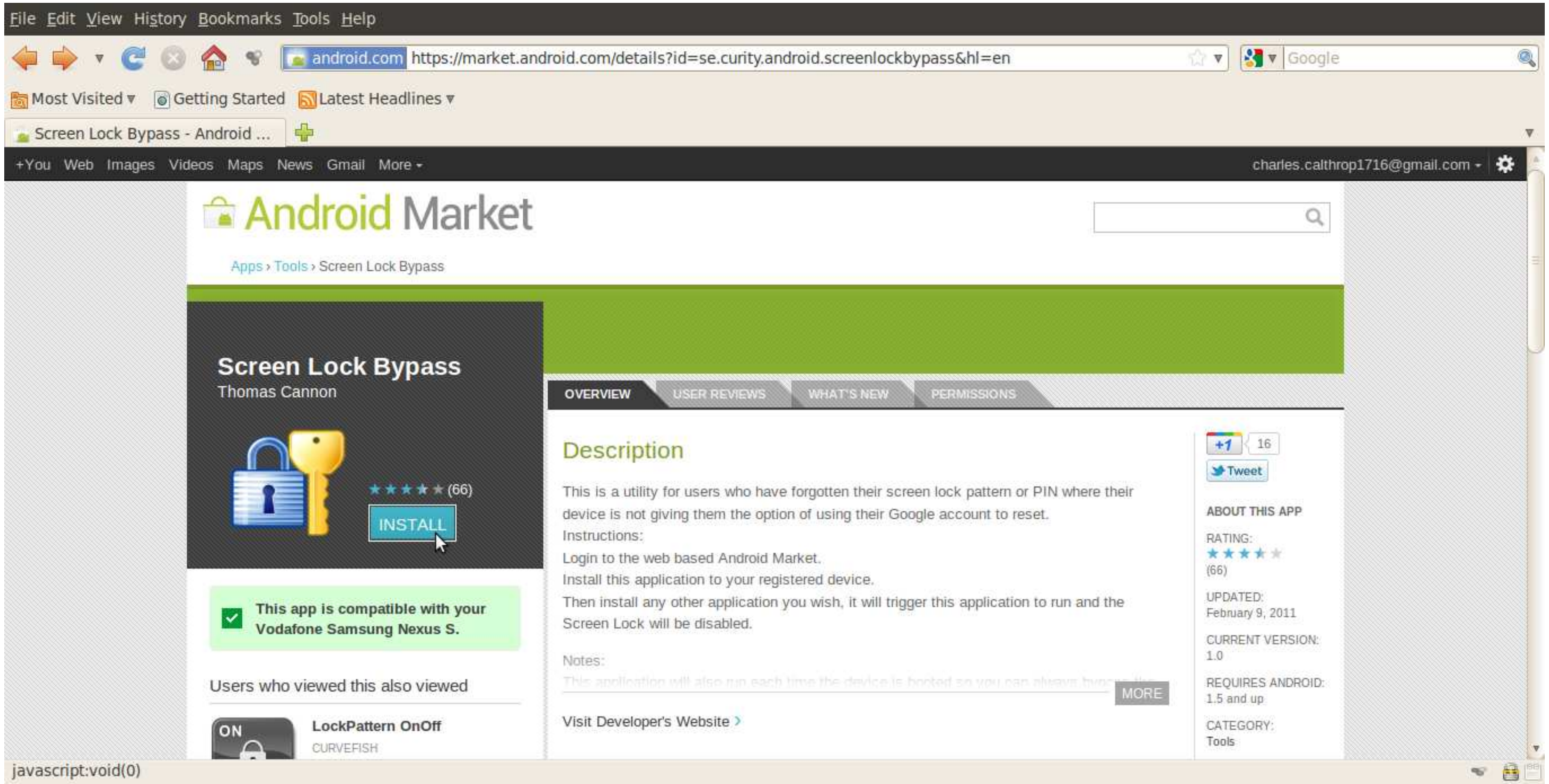
Gmail Account Sidekick

- Let us assume we somehow *can* recall the associated Gmail account login name/password…
- So, we do the following (from any PC/Mac)
  1. go to http://market.android.com
  2. use the name/pwd to log in – note the same credentials apply here as for that Gmail account
  3. find the "Screen Lock Bypass" application and let it install to the associated Android device

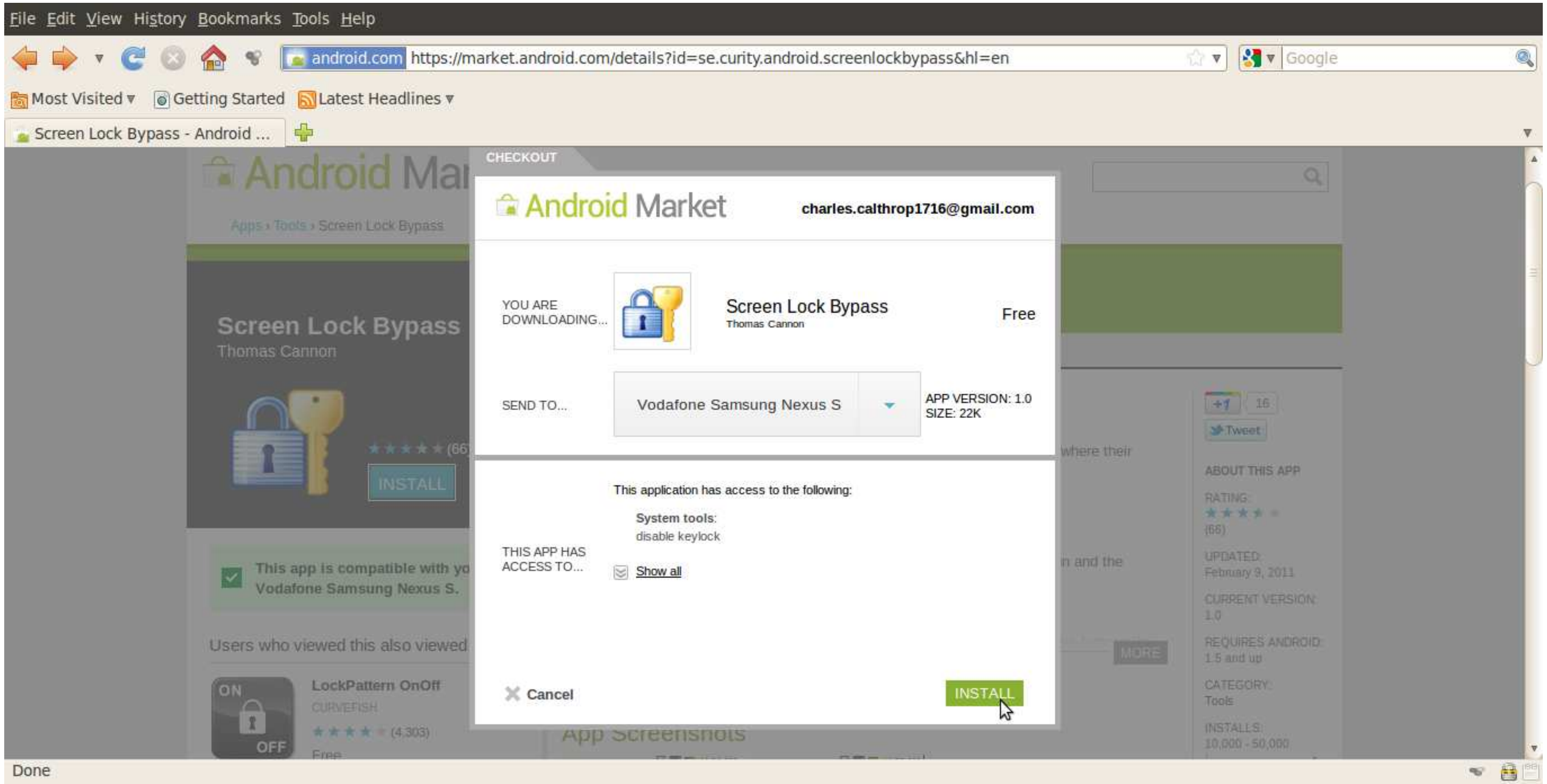
Android Market Login



Finding SLB Application



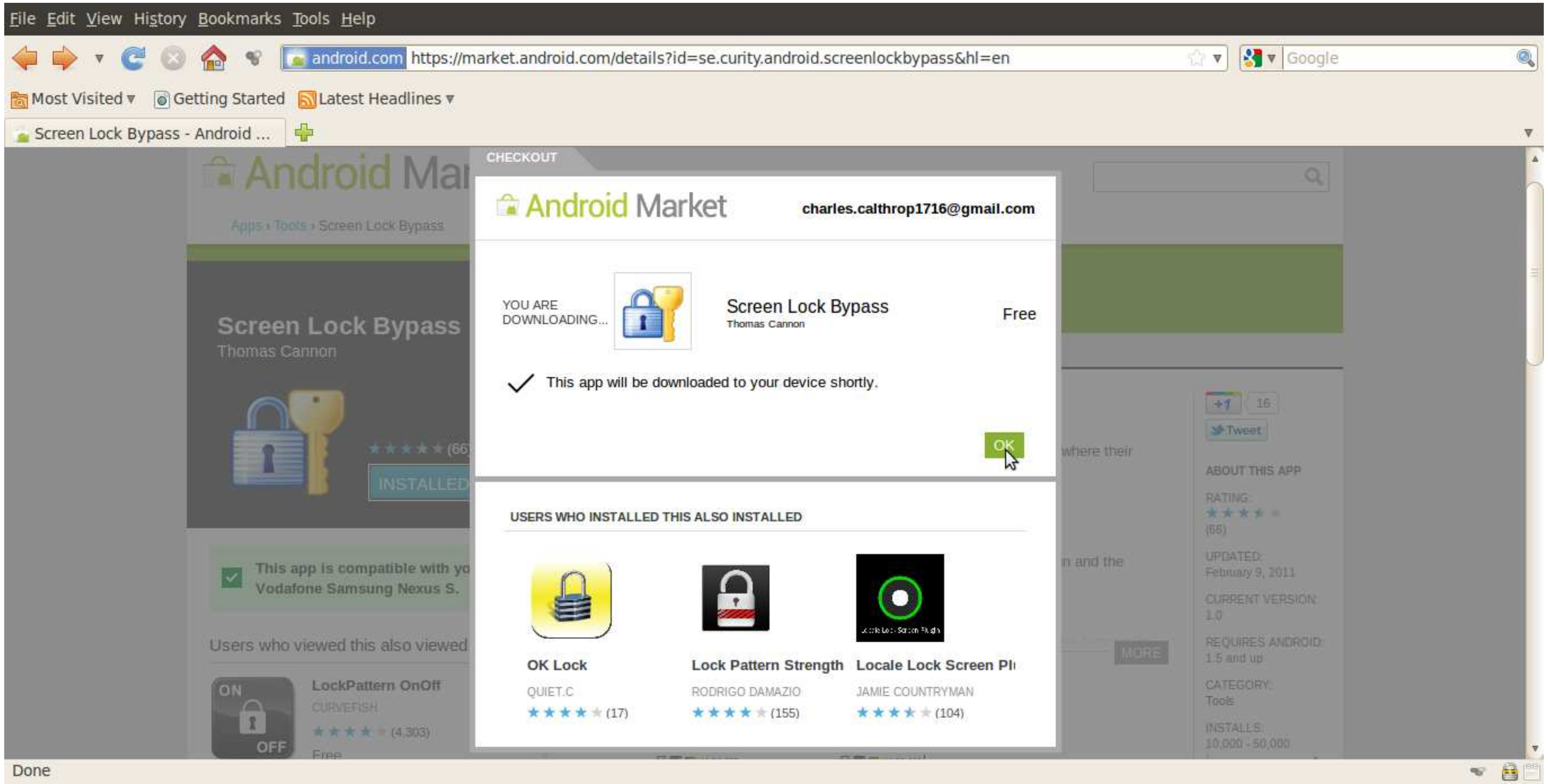
Starting SLB Installation



Telephone Number – Who Cares?

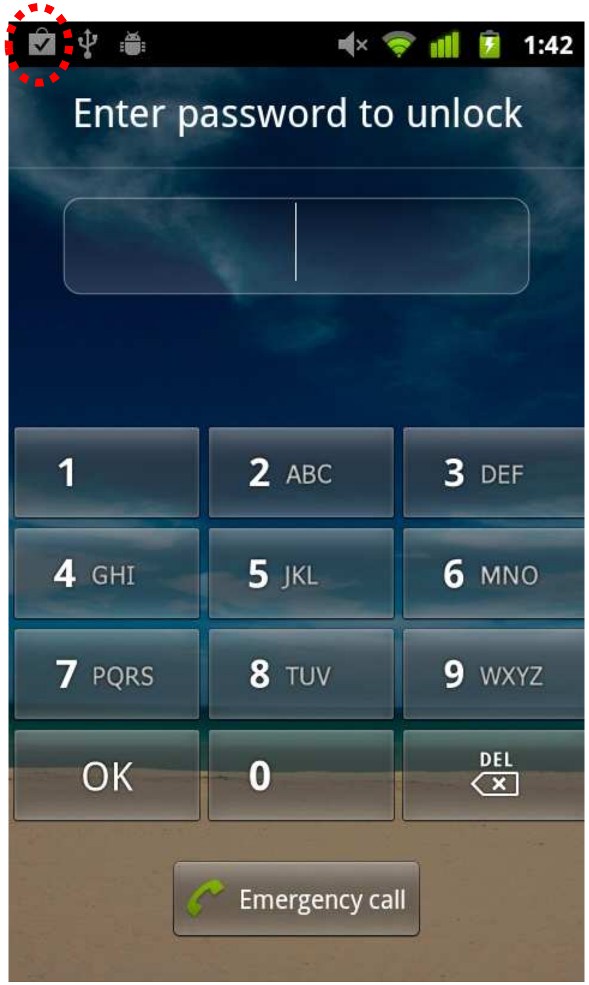
- We should emphasize it is unnecessary to know the telephone number of the target Android device.
- We either do not need to know any other a-priori identification of the device.
- This is because of Android Market offering us the list of associated devices automatically.
  - All we have to do is to choose a device from the list.

Installation In Progress



Meanwhile On the Device

- While the application is being installed, there is no user interaction required at the mobile device side at all.
- The name of the application flashes briefly in the status bar, leaving on just a tiny symbol of a successful installation.



Recall, OTA = Over The Air

- Note the SLB application was installed through a service channel that Google uses to silently manage Android devices worldwide.
  - This permanent data path is kept automatically by each Android device linked to the Android Market portal.
  - That means, we do not need to tweak the mobile phone in any way to start downloading.
  - It may be resting on a table as well as in somebody's pocket – just in any place with GSM/UMTS service coverage.
  - The display does not have to be turned on before the installation starts.
  - Well, this all really is a silent service…

Hands-Off Application Startup

- So, we have downloaded the (pirate) application on the Android device.
- The question is, however, <u>how to make this code run?</u>
  - Obviously, we cannot do that manually, since the screen is locked.
  - Unfortunately, the Android OS provides several reliable ways on how to do that.

Android Broadcast Receiver

- This is an application component [26] responsible for inter-process communication based on broadcast `Intent` mechanism.
  - Usually, developers use a `BrodcastReceiver` derivatives to hook up for asynchronous system events like:
    - `android.provider.Telephony.SMS_RECEIVED`
    - `android.net.conn.CONNECTIVITY_CHANGE`
    - `android.intent.action.PHONE_STATE`
    - etc.

Broadcast Receiver Setup

- To register a `BroadcastReceiver` component, it suffices to list it in the respective `AndroidManifest.xml`.
  - This xml file is stored in the application package and it gets processed automatically during the application installation [26].
  - Therefore, no single code instruction of our application needs to be run to hook up for a particular broadcast `Intent`.

Registration Example

- Remember – it is all done in a package configuration file.
  - We do not need to run our code to register for a broadcast `Intent`.

```
…
 <receiver android:name=".SniffReceiver">
   <intent-filter android:priority="256">
     <action
    android:name="android.provider.Telephony.SMS_RECEIVED"/>
   </intent-filter>
 </receiver>
…
```

Once Upon A Broadcast…

- When the particular broadcast is fired, the Android operating system invokes those registered receivers.
- This way our `onReceive()` method gets called and – yes, we have got it – our application code is up and running!
  - Actually, it is a bit complicated when it comes to the order of calling these receivers and possible event cancellation, but this is not important for us here.

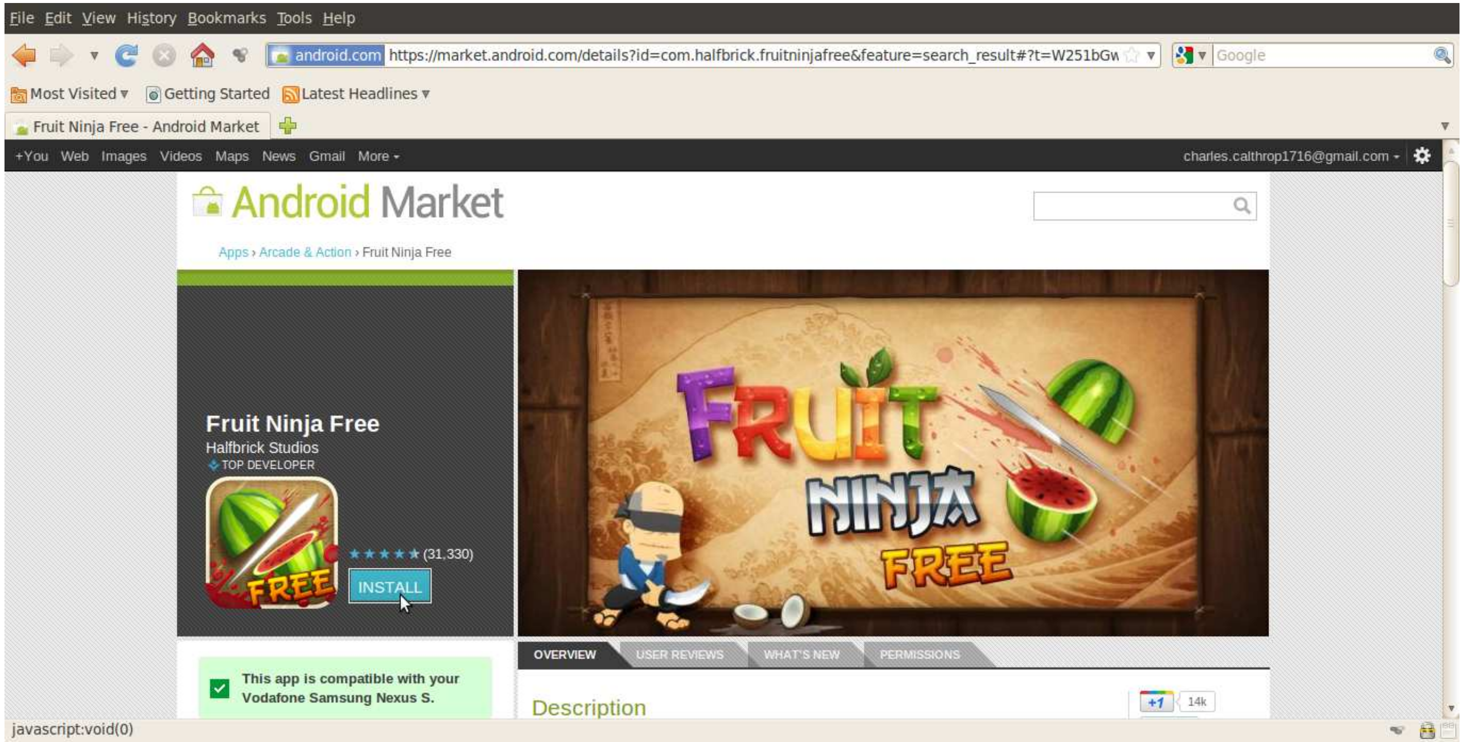
Back To SLB

- The Screen Lock Bypass, in particular, registers to the following broadcasts:
  - `android.intent.action.PACKAGE_ADDED`
    - Triggers when a new package is installed.
  - `android.intent.action.BOOT_COMPLETED`
    - Triggers after finishing OS boot and startup procedures.

Two Ways to Unlock

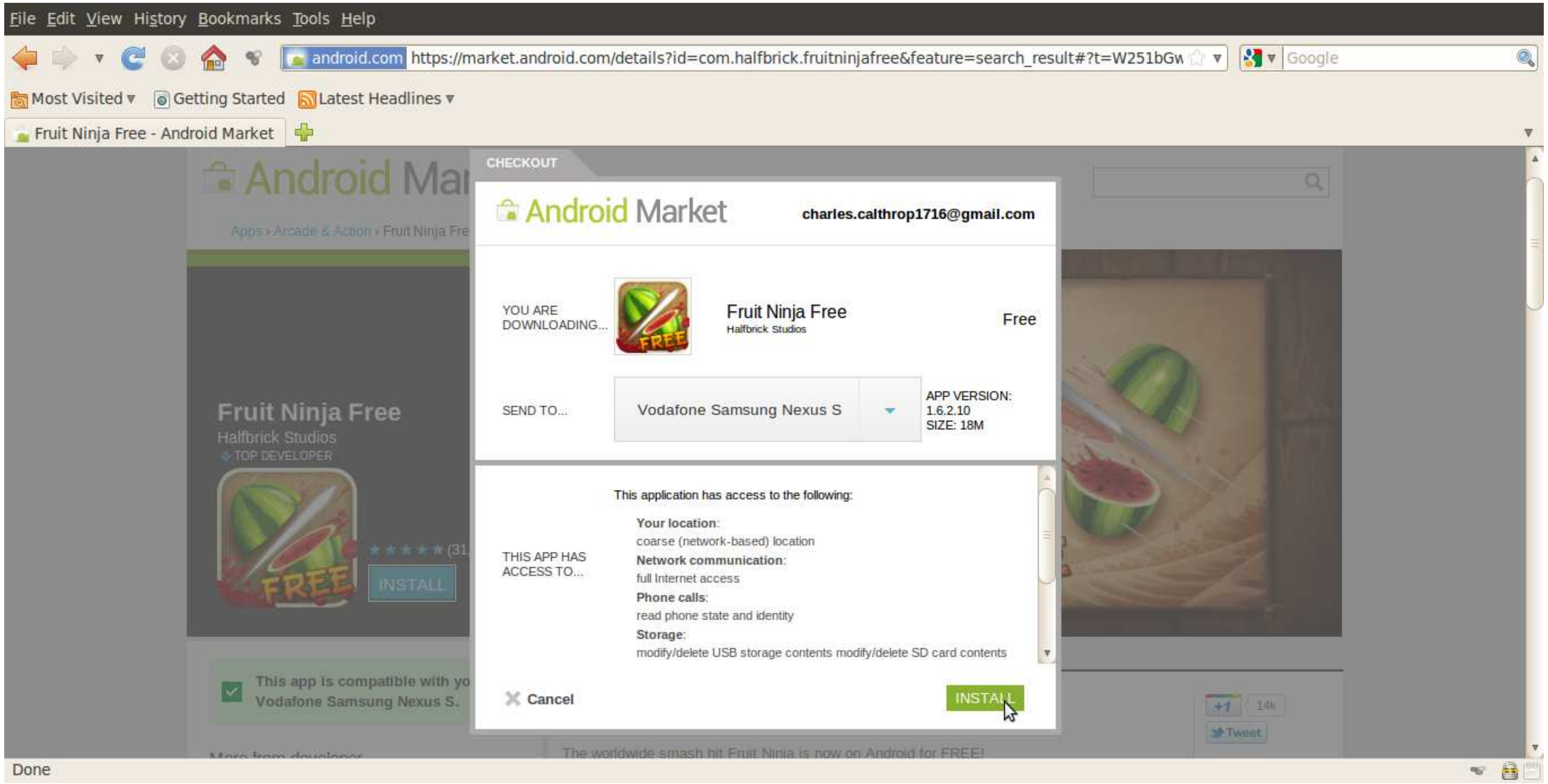
- According to the aforementioned events, there are basically two ways on how to trigger SLB activity.
  1. To install just another application package from the Android Market in the same way as we did for SLB itself.
  2. To switch off/on the device, hence triggering the `BOOT_COMPLETED`.
- We have verified both ways worked well in our experimental setup.

Going the First Way

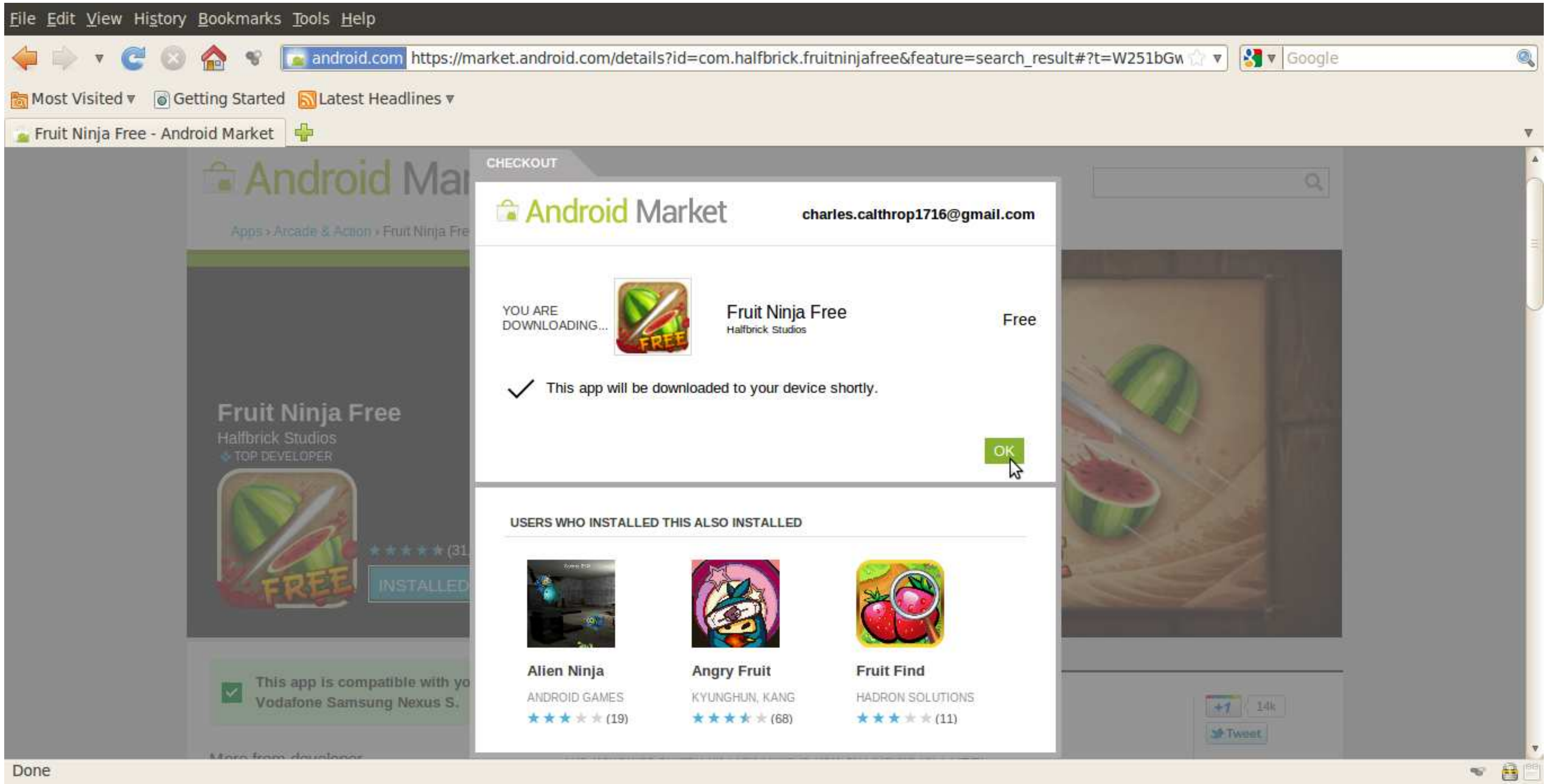


- It really does not matter what application we choose.
- Important is just the final event that triggers our `onReceive()`.

Installing Dummy Application

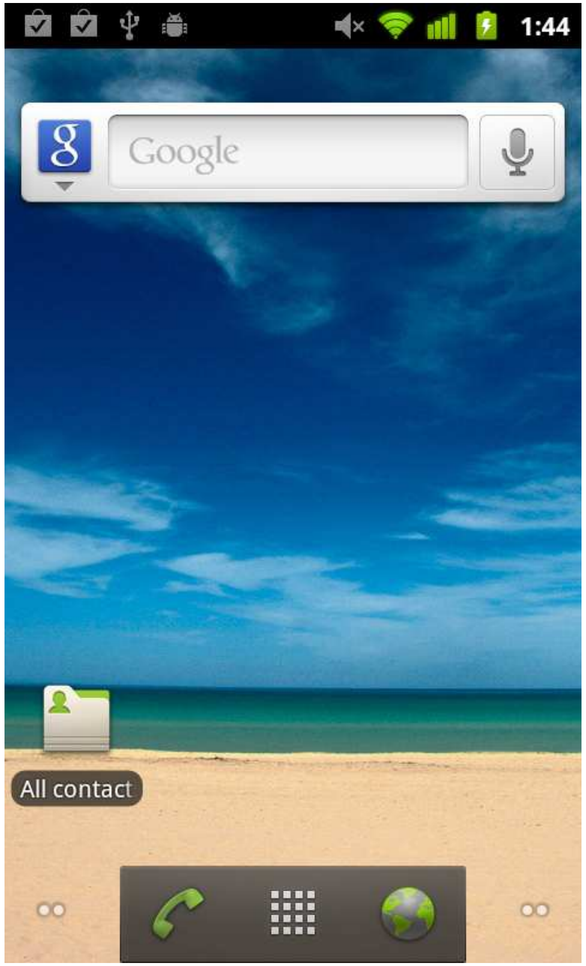


Installation In Progress



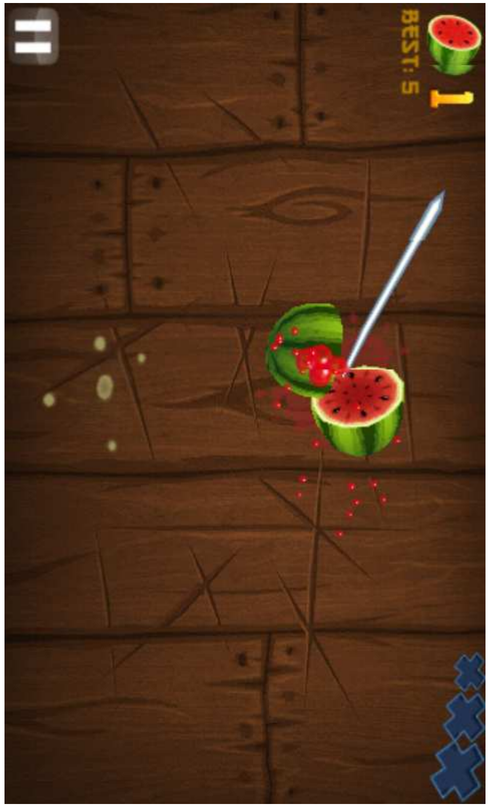
Having Triggered SLB

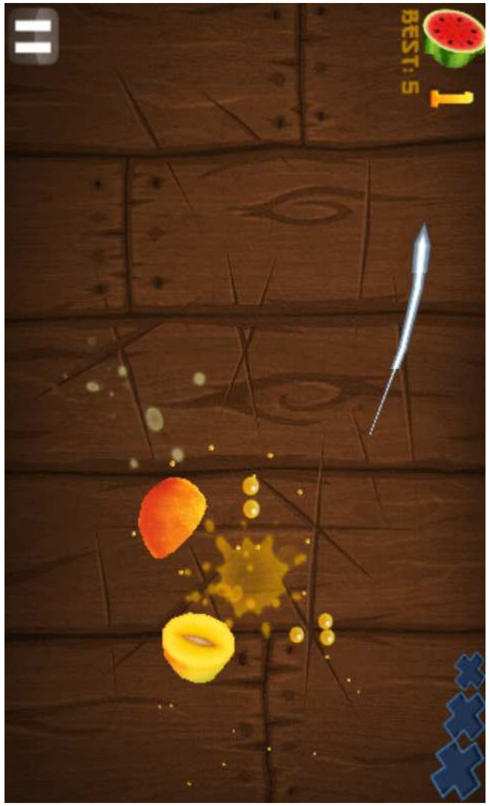
- Secondary installation triggered `PACKAGE_ADDED`.
- This in turn starts the SLB trap.
- Suddenly, the screen lock disappears…



Possibly

- Well, we can also enjoy playing Fruit Ninja.
- But we do not have to.
- Just for fun…



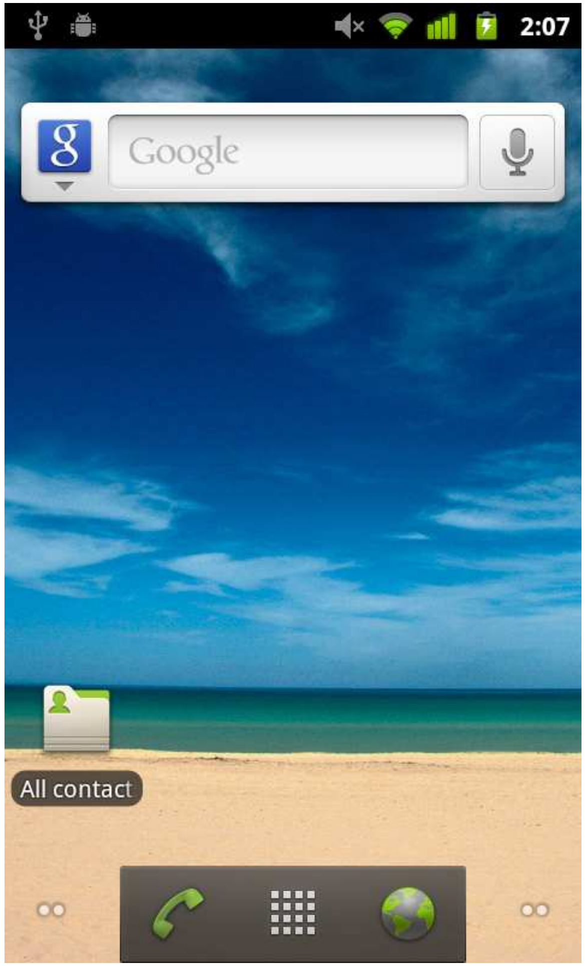


Remember… (regarding SLB)

- We have downloaded an application package on the Android device.
- We have granted any user permissions we needed to that package.
- We have run a code of that package.
- We did not need to directly operate with the mobile device in any way.
  - Furthermore, we even did not need to know the telephone number.
- The only thing we needed was an internet access and a valid login name/password for the associated Gmail account!

Working The Other Way

- By simply switching off/on the device, we can trigger `BOOT_COMPLETED`.
- This again runs a SLB code.
- Again, the screen lock disappears happily…



Recall Again

- The only thing we needed was an internet access and a valid login name/password for the associated Gmail account!
  - Well, this time we used the power off/on switch.
  - The attacker, however:
    1. Can use the former approach using a dummy package installation.
    2. Can just wait until users “recycle” their devices by themselves.

Access Rights Revisited

- The Android operating system relies mainly on user-granted permissions [26].
- During the application installation, the user is asked whether to allow or deny permissions required by the particular `AndroidManifest.xml` [26].
  - Well, this model itself is quite questionable as users may not be fully aware of the possible impact.
  - Furthermore, it is especially non-trivial to discover the risk of various permission synergy effects.
  - Anyway, this is not the topic we want to address here.

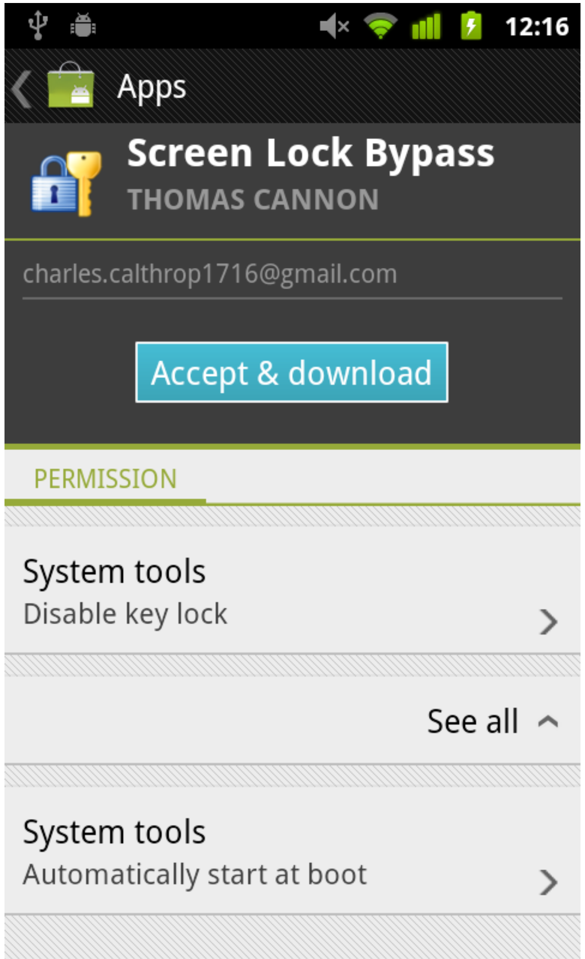
User-Granted Permissions Limits

- We should note that there are some privileges that cannot be granted even by explicit user confirmation.
  - For instance, it is not possible to directly grant root access to the underlying Embedded Linux core.
  - With user-granted privileges, we can, however, run a possible root exploit…
- On the other hand, the power of user-granted permissions is still considerable.
  - For instance, permissions needed by an SMS sniffer can be fully granted this way.

Let Us Experiment

- To see permission granting process at work, we can try installing SLB directly from the Android Market application running on the particular Android device.
  - Well, this does not make a sense, but we do this for another purpose.
  - We want to demonstrate how the user-granted permission mechanism works.

Illustrative Screenshots



As Bad As It Looks

- Well, but when we installed SLB through the web interface, we did not need to grant these permissions. Or did we?
  - We did, but that time it was granted through the web interface instead (cf. the former screenshots).
- Does it really mean…?!
  - Unfortunately, yes.
  - Provided we have respective Gmail credentials, we can choose any application from the Market, give it any user-granted permission, send it to the victim's device, and run it!

Cross-Infection Highway

- Time to time, users log to their e-mail accounts from “ordinary” computers, too.
  - What about if that PC/Mac is infected by a malware that steals Gmail login credentials?
  - The conclusion is immediate – such a malware can instantly spread to the associated Android device.
    - Compromised Gmail account implies compromised associated Android device.
  - There is no need for any further user cooperation!
  - This all in fact effectively breaks those popular SMS-based two-factor authentication schemes…

How About iOS

- We have seen one particular way of possible cross-infection on one particular platform.
  - There will hardly be only one such example.
- Consider, for instance, an infected computer that is synced via USB with an iOS device.
  - Furthermore, consider those exploits behind jailbreaking applications [28] and their forensic payloads [24].
    - Yet, we are only talking about those public ones…
  - Apparently, it is hard to believe that such iOS device can always withstand refined attempts for malware spreading.

Conclusion

- Was not this all happening to PCs in 90's?
  - Did not we lose the game?
  - PCs are considered insecure environment, now.
- However, this is an unavoidable evolution.
  - There is a yearning for mobile applications that we can hardly resist.
  - If we only could wait some time…
  - But we cannot.
  - The war has already begun.

Thank You For Attention

Tomáš Rosa, Ph.D.

http://crypto.hyperlink.cz

References I

1. Bachman, J.: *iOS Applications Reverse Engineering*, Swiss Cyber Storm, 2011
2. Bédrune, J.-B. and Sigwald, J.: *iPhone Data Protection in Depth*, HITB Amsterdam, 2011
3. Blazakis, D.: *The Apple Sandbox*, Black Hat DC, 2011
4. Breeuwsma, M.-F., de Jongh, M., Klaver, C., van der Knijff, R., and Roeloffs, M.: *Forensic Data Recovery from Flash Memory*, Small Scale Digital Device Forensics Journal, Vol. 1, No. 1, June 2007
5. Breeuwsma, M.-F.: *Forensic Imaging of Embedded Systems Using JTAG (boundary-scan)*, Digital Investigation 3, pp. 32 - 42, 2006
6. Chin, E., Felt, A.-P., Greenwood, K., and Wagner, D.: *Analyzing Inter-Application Communication in Android*, MobiSys'11, 2011
7. Dhanjani, N.: *New Age Application Attacks Against Apple's iOS (and Countermeasures)*, Black Hat Barcelona, 2011
8. Dubuisson, O.: *ASN.1 - Communication Between Heterogeneous Systems*, Morgan Kaufmann Academic Press, 2001
9. Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S.: *A Study of Android Application Security*, Proc. of the 20th USENIX Security Symposium, 2011
10. Fairbanks, K.-D., Lee, C.-P., and Owen III, H.-L.: *Forensics Implications of Ext4*, Proc. of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research, ACM, 2010

References II

11. Felt, A.-P., Finifter, M., Chin, E., Hanna, S., and Wagner, D.: *A Survey of Mobile Malware in the Wild*, SPSM'11, 2011
12. Halbronn, C. and Sigwald, J.: *iPhone Security Model & Vulnerabilities*, HITB KL, 2010
13. Hay, R. and Amit, Y.: *Android Browser Cross-Application Scripting*, CVE-2011-2357, IBM Rational Application Security Research Group, 2011
14. Heider, J. and Boll, M.: *Lost iPhone? Lost Passwords!*, Fraunhofer SIT Report, cf. also [23], 2011
15. Hoog, A.: *Android Forensics – Investigation, Analysis and Mobile Security for Google Android*, Elsevier, 2011
16. HOTP: *An HMAC-Based One-Time Password Algorithm*, RFC 4226, 2005
17. Jaden and Pod2G: *How To: Install GNU Debugger (GDB) On The iOS 5 Firmware Generation*, iJailbreak, February 24, 2012, http://www.ijailbreak.com/cydia/how-to-install-gnu-debugger-gdb-on-ios-5/
18. Menezes, A.-J., van Oorschot, P.-C., and Vanstone, S.-A.: *Handbook of Applied Cryptography*, CRC Press, 1996
19. Miller, C. and Iozzo, V.: *Fun and Games with Mac OS X and iPhone Payloads*, Black Hat Europe, 2009
20. Miller, C. and Zovi, D.-A.-D.: *The Mac Hacker's Handbook*, Wiley Publishing, Inc., 2009

References III

21. Oudot, L.: *Planting and Extracting Sensitive Data Form Your iPhone's Subconscious*, HITB Amsterdam, 2011
22. PKCS #1 v2.1: *RSA Cryptography Standard*, RSA Laboratories, June 14, 2002
23. Toomey, P.: *"Researchers Steal iPhone Passwords In 6 Minutes" - True, But Not the Whole Story*, Security Blog, http://labs.neohapsis.com/2011/02/28/researchers-steal-iphone-passwords-in-6-minutes-true-but-not-the-whole-story/ , 2011
24. Zdziarski, J.: *Hacking and Securing iOS Applications*, O'Reilly Media, 2012
25. Zovi, D.-A.-D.: *Apple iOS 4 Security Evaluation*, Black Hat USA, 2011

References IV

26. http://developer.android.com
27. http://developer.apple.com
28. http://theiphonewiki.com
29. http://thomascannon.net/blog/2011/02/android-lock-screen-bypass/
30. http://www.bbc.co.uk/news/technology-15635408
31. http://www.cycript.org
32. http://www.iphonedevwiki.net

33. *iOS App Programming Guide*, Apple Developer Guide, Apple Inc., 2011