Introduction to Advanced Security Analysis of iOS Applications with iNalyzer

By Chilik Tamir (chilik@appsec-labs.com)

Introduction

Performing security analysis of iOS applications is a tedious task; there is no source code and there is no true emulation available. Moreover, communication is usually signed or encrypted by the application, leaving the standard tampering and injection attacks worthless. Needless to say that time spent on testing such applications increases substantially due to the fact that not every automatic tool can be used on the captured signed-traffic, including the conventional scanners (such as Burp, Accunetix, Webinspect and AppScan).

In the following article I will present my latest research on a new approach to performing security assessments of iOS applications utilizing the iNalyzer, a free open-source framework for security assessment of iOS Applications.

But before we dive deep into iOS security analysis let us review some of the basic rules…

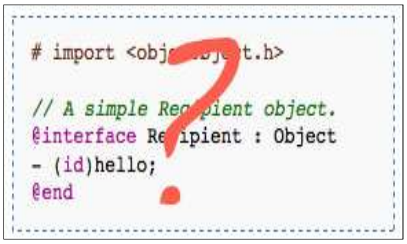
The Ten Commandments of iOS Security Analysis:

I. Thou shalt have… Root Access on iOS.

In order to convince the application that your data is legitimate you must own a greater permission set, namely root. Therefore you need to perform analysis on a jail-broken device, usually running analysis under root.

II. Thou shalt not … have any source of iOS App.

Unlike Android or other platforms, there are no true emulators for the iOS hardware; applications are built using Objective-C and are packed in Mach-O archives. Reversing these archives leaves you with assembly instructions which are a head-over for a simple security analysis test.



III. Thou shalt not … perform analysis on a device without backup.

During tests you will likely destroy most of your data, so backup the device on a regular basis.

IV. Remember the … device identification strings (UDID, IMEI etc.).

Hence, during analysis you can identify their usage in outgoing requests and inside database content.

V. Honor thy … application request encryption and signing scheme

You may notice that some requests are sent encrypted or signed to server side. Tampering with these requests usually ends up with broken responses from the server or with false positive findings. Using the normal proxy-tampering approach on such requests is impractical.

VI. Thou Shalt Not Kill … a process without saving a restoration point.

Often you will end up killing the application process during analysis, hence, keeping a clean note on the analysis tests and current stage is highly recommended. These notes will assist you to resume tests within seconds after you re-launch the process.

VII. Thou Shalt Not Commit Adultery… never install pirated software.

Pirated software tends to be hosted by peculiar sources, once installed they can easily modify the true nature of the application behavior. Never install pirated software on your testing device.

VIII. Thou Shalt Not Steal… never distribute pirated software.

Security analysis process includes disablement of the internal application encryption module which acts as DRM for all Apple applications. Distributing copyrighted applications is wrong and considered a crime in most parts of the world.

IX. Thou Shalt Not Bear … answer a call during testing.

Incoming calls tend to break testing logic and suspend the targeted application, hence, during testing place the device in airplane mode or remove the SIM card.

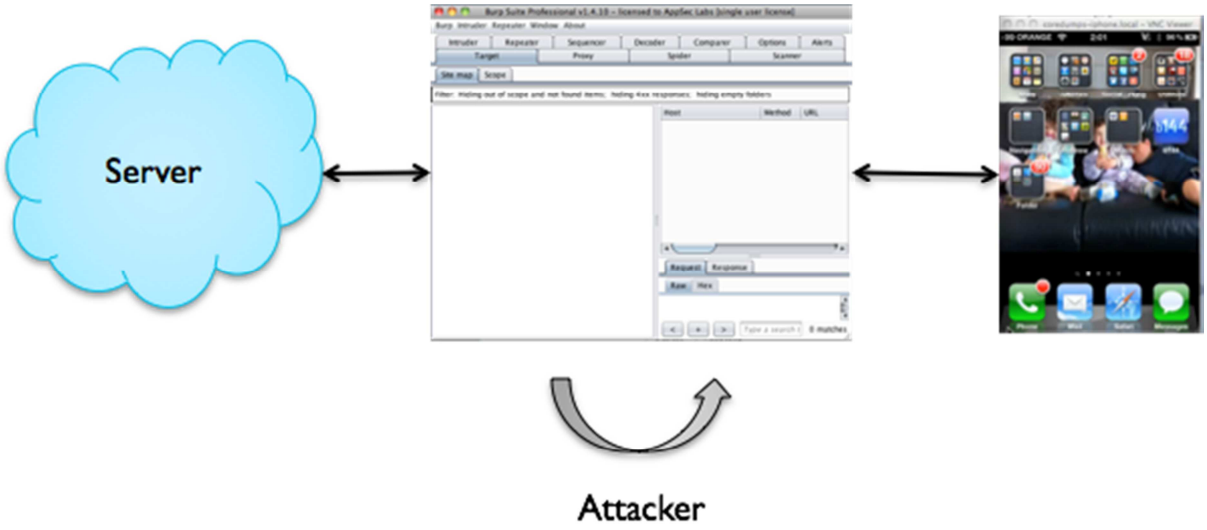
X. Thou Shalt Not Covet.

It is true that in order to conduct security analysis on iOS applications you will need an actual device, but that doesn't mean you can demand a diamond studded, platinum iPhone5 from your employer.

Now that we have clarified these points, let's review the work method we have been familiar with until now and see what it takes to adjust it for iOS testing.

This is what we have

Until now, in order to test a web application we would set up a testing environment in the following scheme:



This means an external server, proxy and a mobile device. This is good for simple applications: you interact with the proxy, connect a scanner and let it run automatically while you focus on the manual tests. But there are some crucial questions that need answering in our analysis before we can say that the test is complete:

What do I know about the system?

'Houston, we have a problem' – where does the application connect to? Do I know for a fact that I have covered all of the systems' access points by performing a manual test? What is my coverage rate? What have I missed…?

→ *We do not have the ability to find out what are the peers of the system in the outer world!*

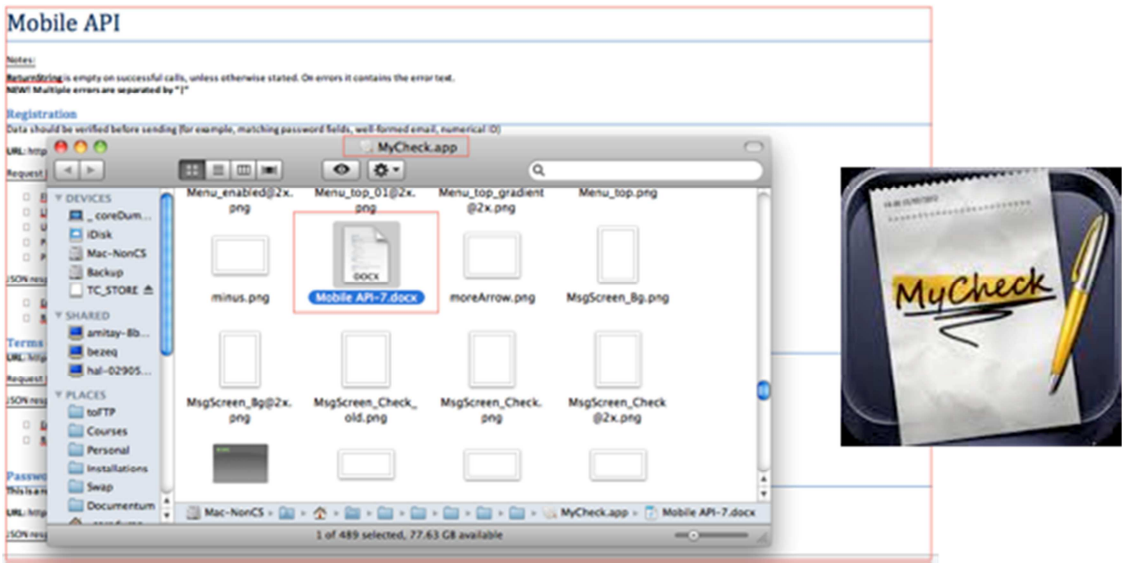
'Please sign here' – what happens if we want to test an application when the client sends sealed requests, or when it sends requests via 3G and we don't have the ability to intercept and play with them? We have a high chance of missing coverage and problems, since all of our requests will fail validation or server side compatibility.

→ *We do not have the ability to play with signature, so if the client seals request it's game over!*

'Easter Eggs' – which functionalities are hiding in my client application that I don't know about? "I didn't see it during testing", "I didn't know it existed…", "How do I know with complete certainty that I covered the entire system and don't have any Easter Eggs in my product?"

→ *We do not have the ability to uncover all of the hidden functionalities within the system.*

'Where are the keys' - are there any sensitive keys in the code? Are there other sensitive parameters hidden in the application? Sometimes the developers do silly things like upload the application together with a complete document describing the entire API



→ *We lack information on the application files and its' true content!*

The problem with all of these questions is that we cannot answer them from the proxy, we don't know neither if we covered all of the potential requests nor if all of the relevant files were loaded on the device. In many cases the content is encrypted so even regular tampering tests are problematic and we will get a lot of false-positives.

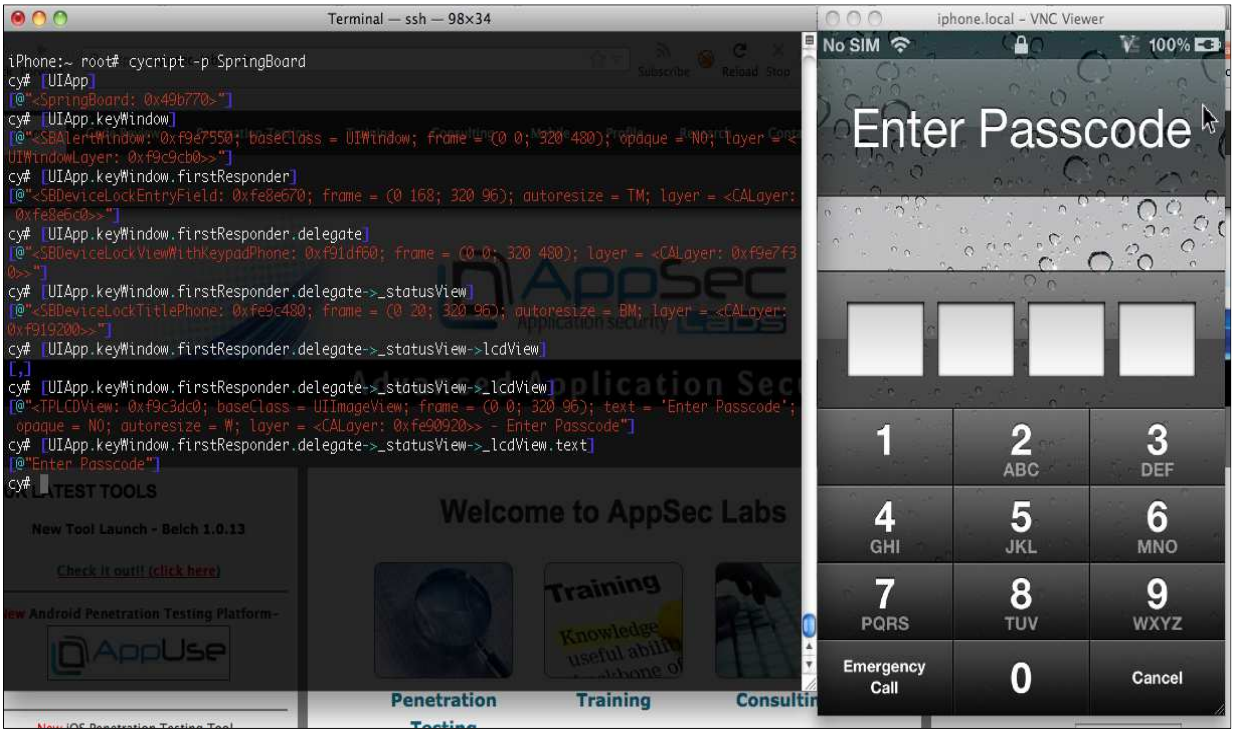
Cycript – changing approach

We understand the application itself possesses the know-how of performing signatures and of passing all stages until the request is sent to the server, so, we would like a convenient way of feeding fake or tampered values into the application. Moreover, if we had the ability to debug the application we could change memory addresses and values on-the-fly and let the application send the tampered messages to the server. But as we said for objC, the compilation is to a Mach-O file in machine language with no mediation language, so our general ability to debug the application is to work with GDB.

Fortunately for us, there is a man named Jay Freeman (AKA Saurik) who really loves challenging the Apple guys and keeps coming out with wonderful tools (such as Cydia) for public use. One of these tools is Cycript.

Cycript is an interpreter that combines ObjC syntax with JavaScript and allows us to connect to an existing process and play with it directly.

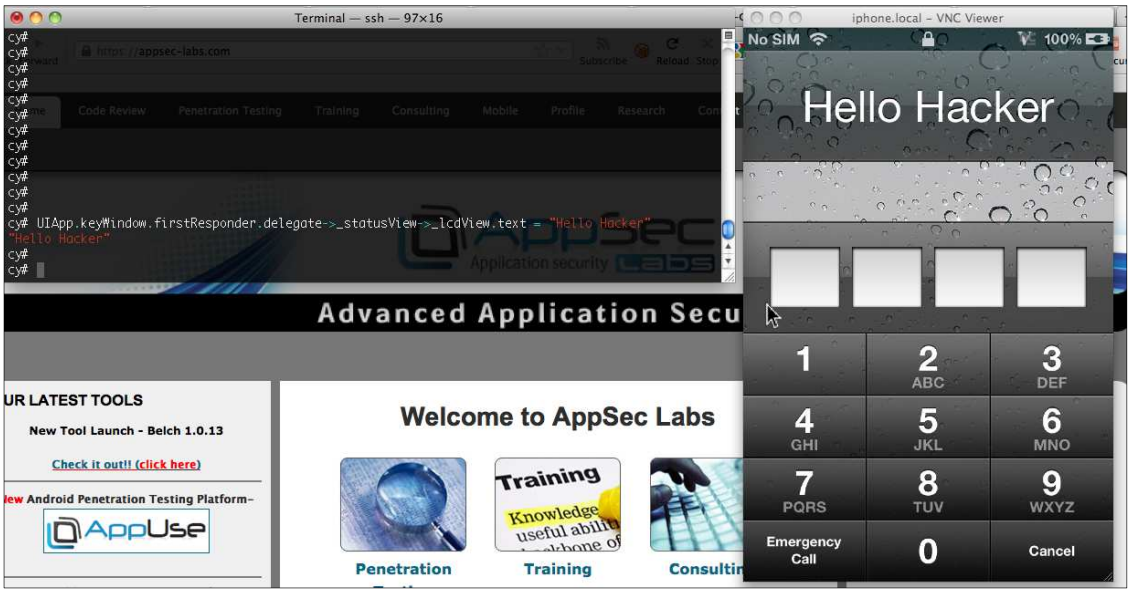
Let's look at an example in which we connect with the SpringBoard process and start playing with it using Cyript:



When I write:

```
UIApp.delegate.keyWindow.firstResponder.delegate->_statusView->_lcdView.text="Hello Hacker"
```

We will immediately see the system and the display change:



With Cycript we can easily deal with the runtime of JavaScript and ObjC, the only problem with using Cycript is the need to know the different objects in the system and the selectors and methods it can accept.

Like we said – we are performing a Black Box test, so where do we get the information from?

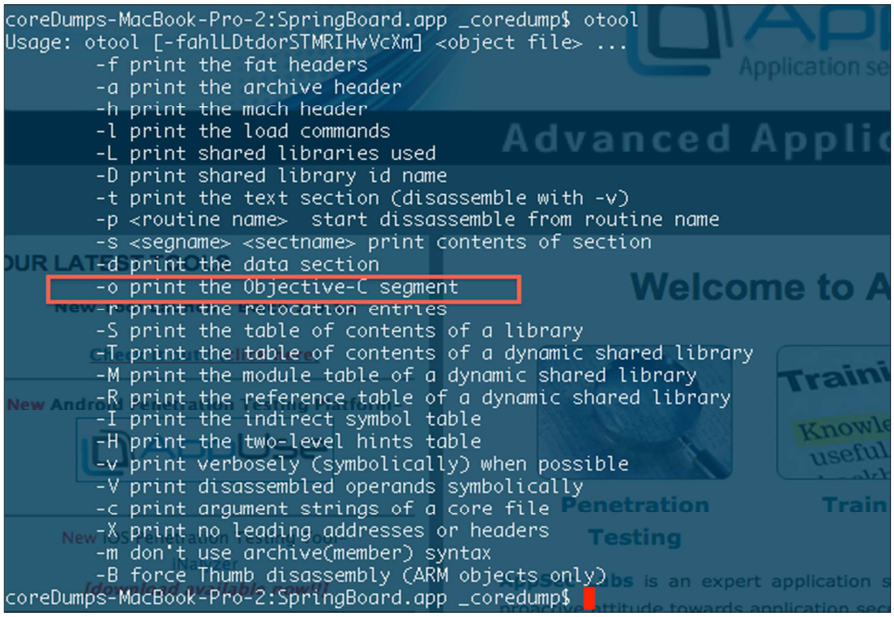
The answer is: from the Mach-O !

Don't Mach-O about!

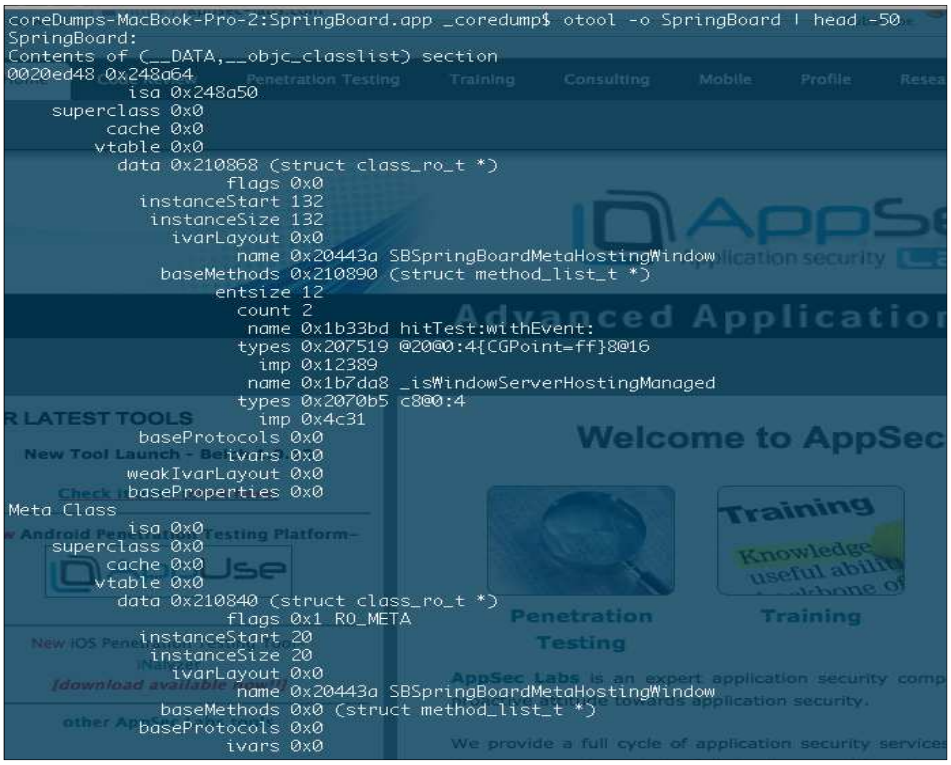
Yes, we are going to take advantage of the unique structure of the Mach-O file for the sake of extracting information about our system.

The structure of this type of file encompasses all the templates of the objects it needs in order to run. In other words, I can ask our system file which objects and methods it requires in order to run.

Let's experiment: we will use Apple's otool in order to question a Mach-O file, we are mainly interested in what data is stored wrote into its ObjC segment:

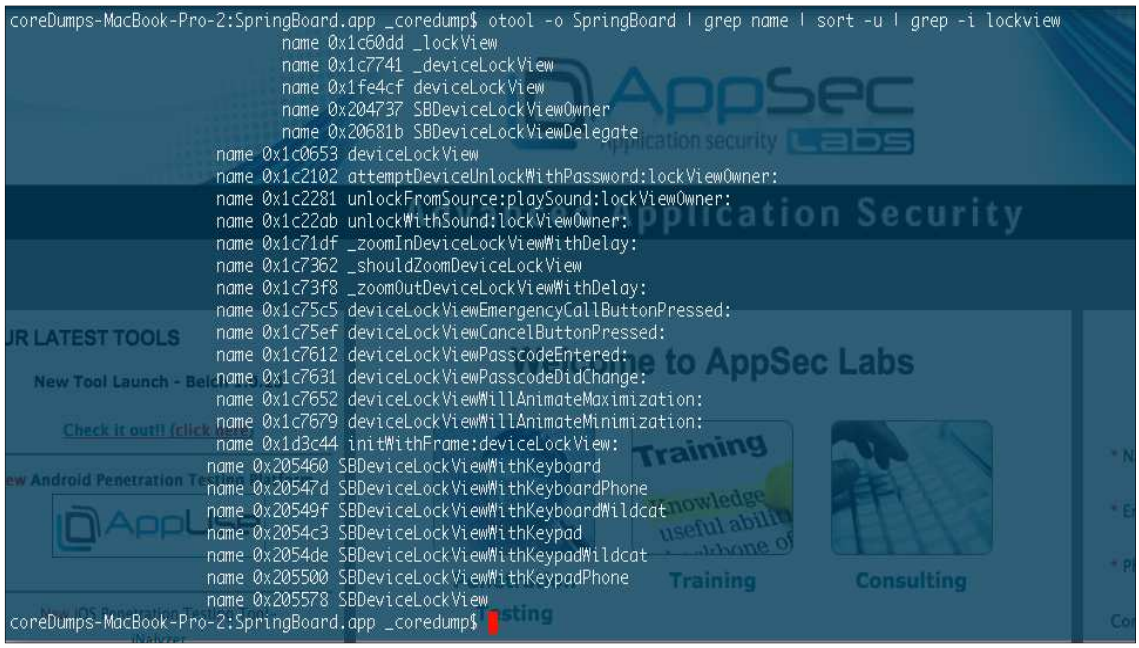


We will then run the otool on our system file (in this case the iOS 5.0.1 SpringBoard) and request all of the listings under the ObjC segment of the Mach-O fie (we trimmed the output to show first 50 lines):



As you can see, what we got is the structure of the object, its name and which parameters it receives and sends back.

So let's see what objects in the system file are related to the LockView:



We check our system file like we did last time and with the use of some command-line-nunjitsu we get a list of every appearance of the words LockView in the system file. (Note, for example, the @deviceLockViewPasscodeEntered method).

So, at this point we can question a Mach-O file and extract details from it about the different objects.

But it is one huge job to start cross referencing otools output to a meaningful objects and methods map, one like that will enable us to use in the tests… And this is where class-dump-z comes into the picture!

Classy class-dump-z

[Now that *IS* classy]

As mentioned, all of the information already appears in the Mach-O file and all we have left is to put this puzzle of objects together, that is what class-dump-z does for us: it compiles all of the data to the form of an original header!

And here is an example:

In the above image we requested class-dump-z to assemble every interface/object that implements the call to the method/selector “deviceLockViewPasscodeEntered”, the software collected and cut and pasted the raw information we saw in the ObjC segment of the system file, and created an original header file for us which can be worked with in Cycript.

We can see the selector we chose was realised in four objects, including SBSlidingAlertDisplay, SBDeviceLockViewDelegate etc.

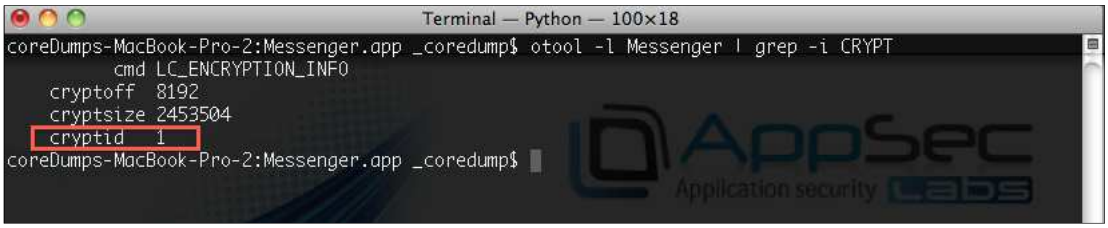
In this case, we can use Cycript to play with the application with a deeper understanding of its methods and objects. This means that we will be dealing with the application instead of the communication – and will trust that the output will come out the way we want it to.

All this refers to cases in which the application is not encrypted, but the files downloaded from the AppStore are encrypted – so it is important that we discuss the process when referring to an iOS encrypted application.

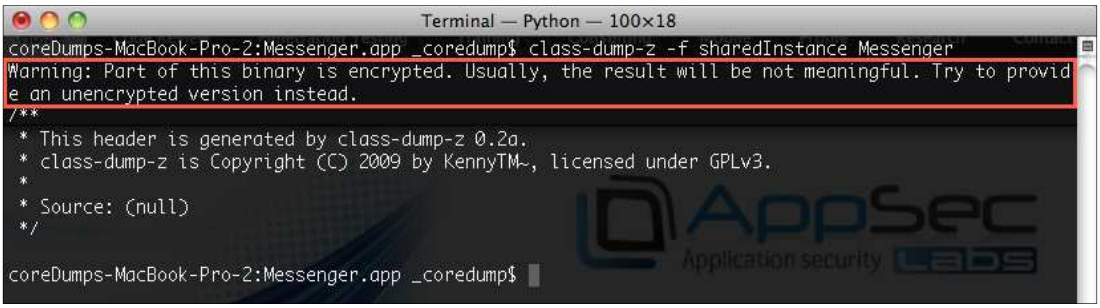
Stop! Password!

A short review of the encryption process: the process of downloading applications from the AppStore includes an encryption of the application's Mach-O file by the Apple server. The encryption is performed using private keys on the device, which theoretically prevents the user from running the application on a different device. The applications won't work since the keys don't match the encryption keys belonging to the device that downloaded the application.

One can easily identify whether the Mach-O file is encrypted by asking otool for the segments loading (-l) commands:



The above example shows that the cryptid flag is on, so we know this Mach-O file is encrypted, if we ask the class-dump-z to show the different objects we will receive a warning such as this, along with a missing output:



Class-dump-z notifies us that the file is encrypted and so we cannot extract information from it, it recommends we use an unencrypted file.

In order to get over this obstacle we need some understanding of the decoding process: during activation of the application, the system loads the keys and decodes the encrypted segment within the Mach-O file downloaded from the AppStore. Once the decoding is complete the application starts to run.

This means that the application is stored in the memory in decoded state one second before it starts to run, if we could connect with the application using GDB we will be able to set a DB on a start address and dump from there to the memory of that segment. Then we will be able to edit the Mach-O file so it contains the unencrypted version dumped from the memory, and then we will be able to use class-dump-z and let it work its magic.

Sounds like a lot of work:

1. First, decode application
2. Then analyze the objects
3. Then connect all the objects into a meaningful hierarchy map
4. Then use Cycript and the information collected to analyze the running process

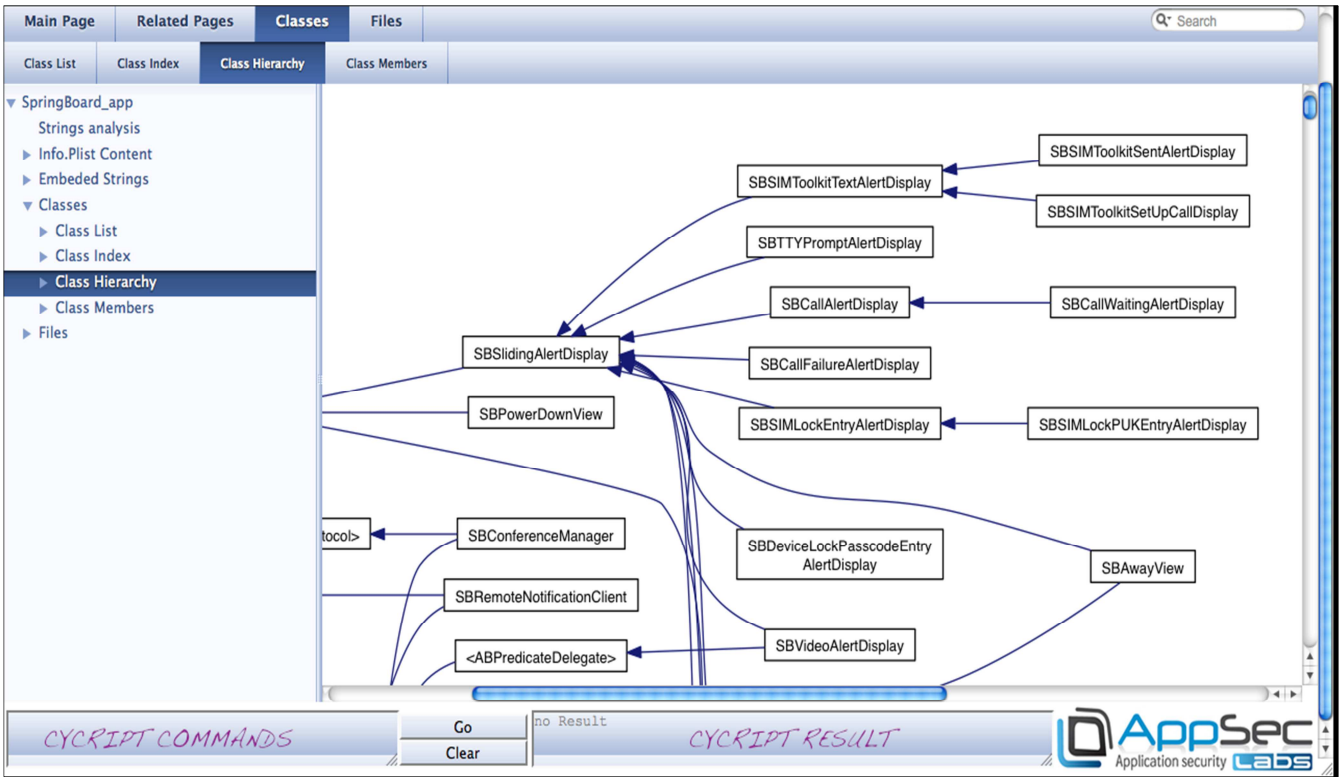
Isn't there a tool that can perform all of this tedious work for us?

Well there wasn't one, so I wrote the iNalyzer…

The iNalyzer !

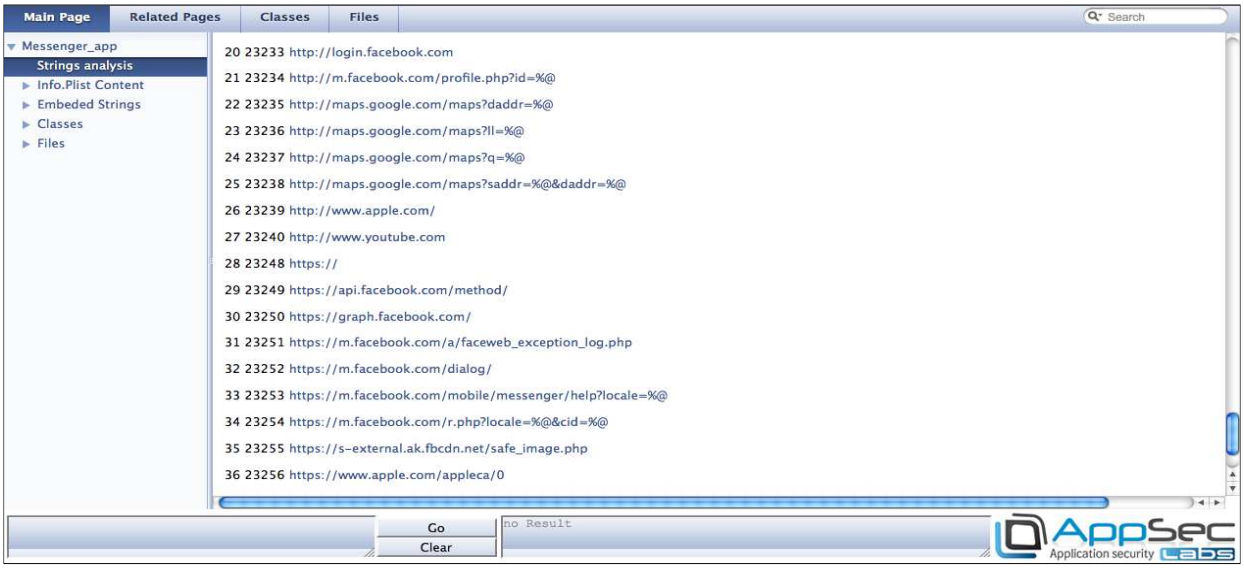
The iNalyzer is a special environment for testing iOS based systems. It collects all of the data from the system file and from class-dump-z and then generates a Doxygen-based Command&Control interface for Cycript, which gives us a full testing environment.

Some advantages of the iNalyzer: automatic activation of class-dump-z, collecting all of the information we need for the tests, it is a single tool that will perform all of the dirty work and let us deal only with the system and Cycript with no signatures, no games; an automatic way of decoding dump:

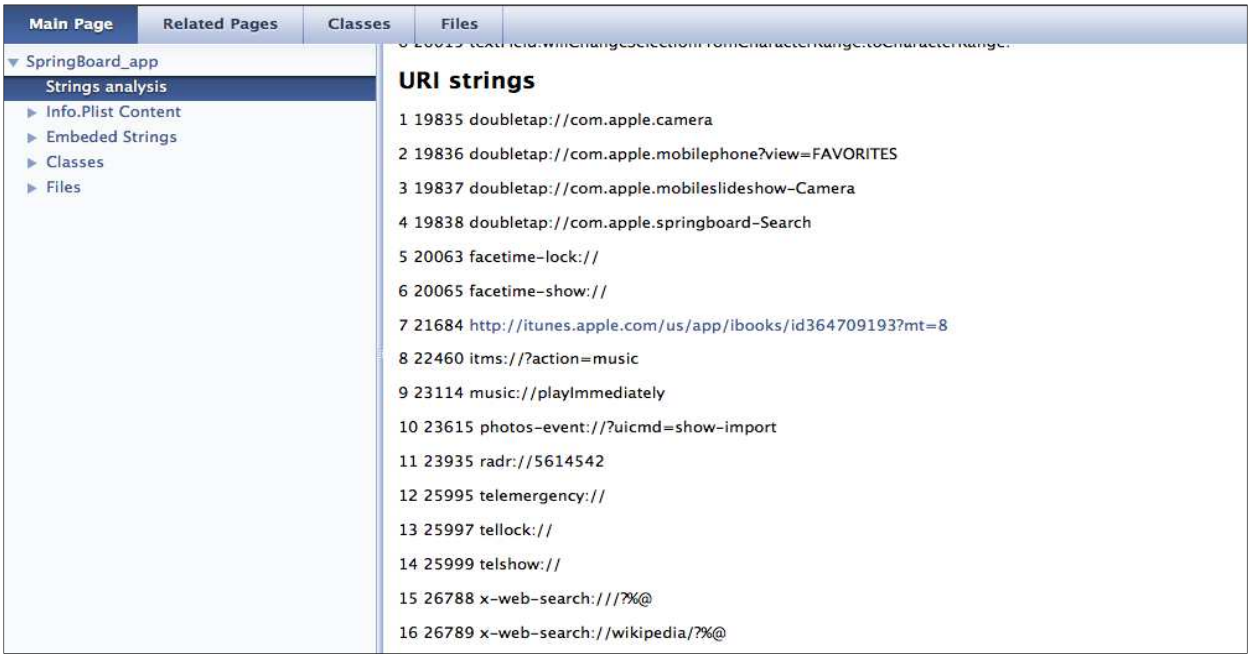


Here are a number of examples of what iNalyzer provides:

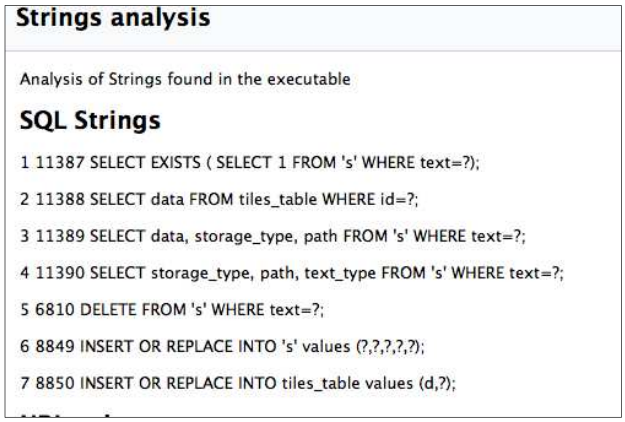
Presentation of external links, for the sake of mapping interface points with external servers:



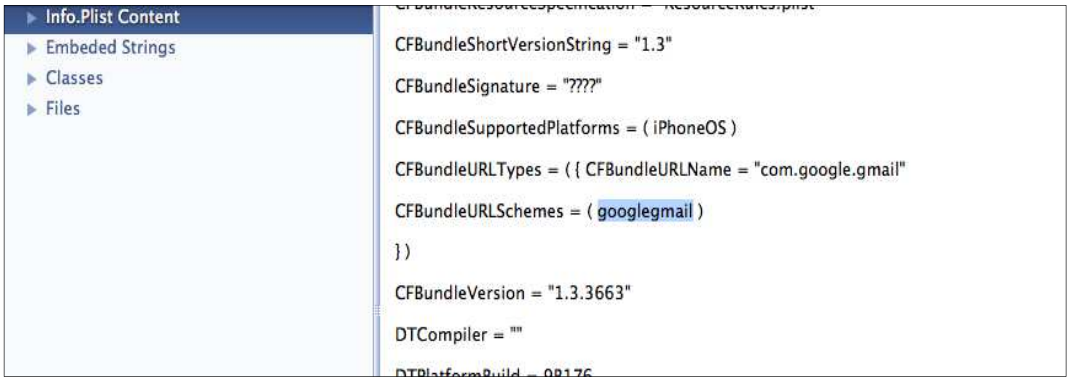
Showing used URI interface, for the sake of mapping out external activations of other applications and injections:



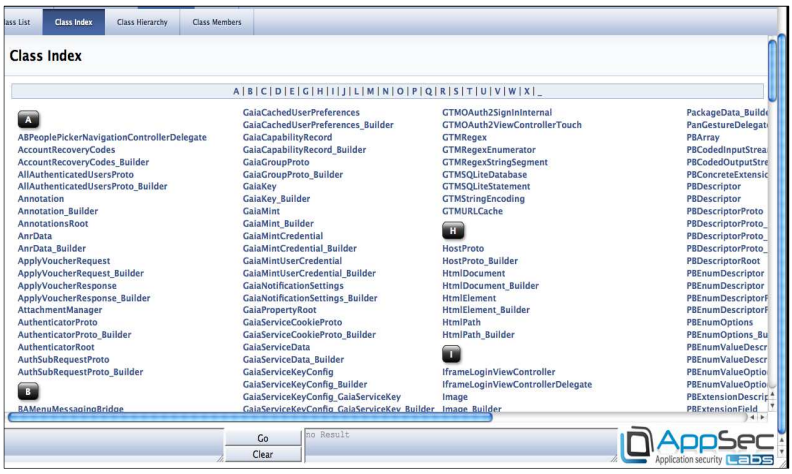
Showing SQL strings in use, for the sake of vulnerability analysis of local and remote injections:



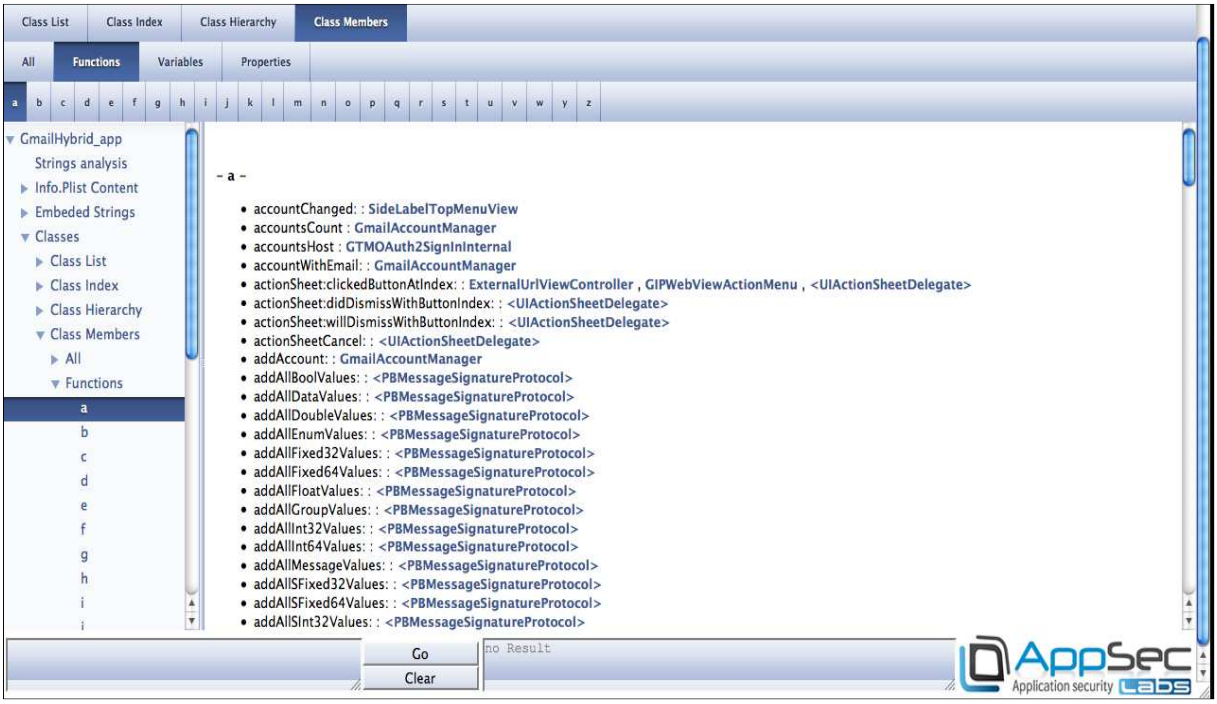
CFURL interface which is registered with the system and act as additional activation point, and a convenient attack vector:



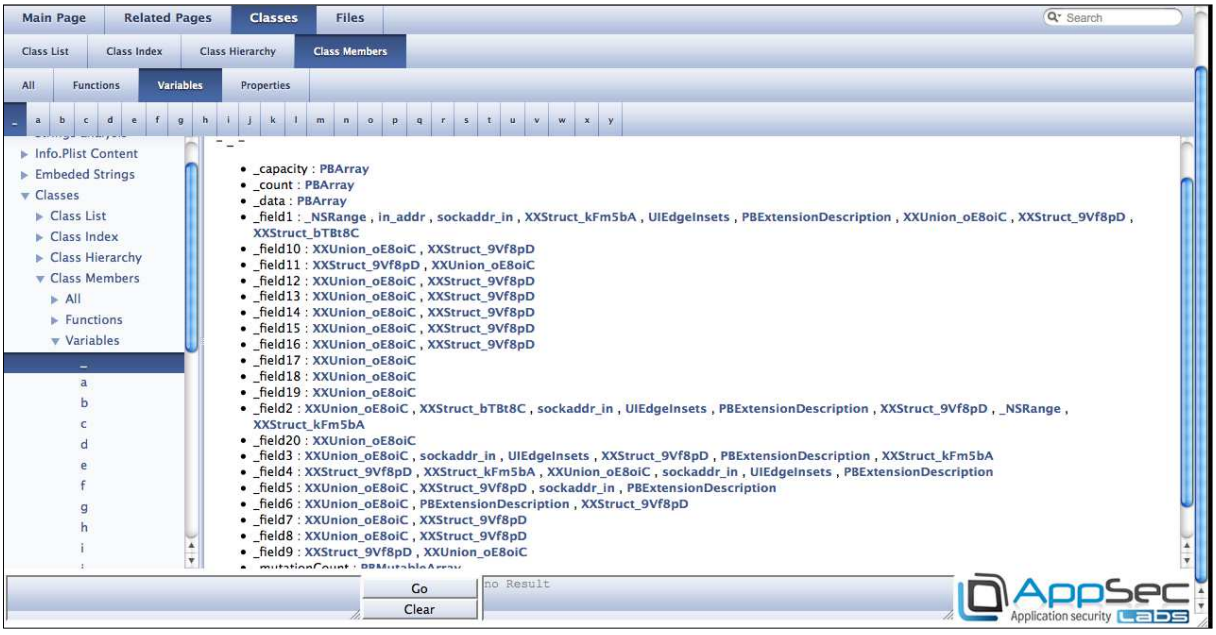
Presentation of all the system objects, for the sake of exposing problematic or redundant functionalities:



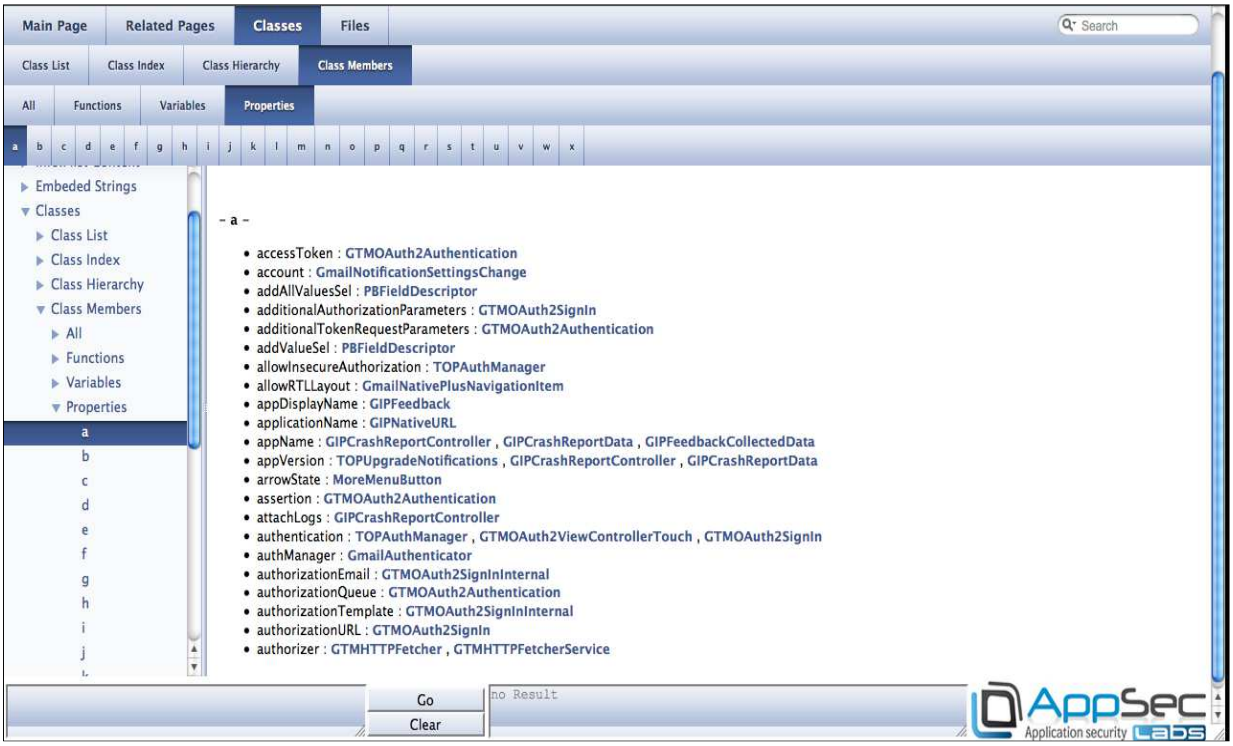
Presenting all of the system methods, for the sake of direct activation by Cycript:



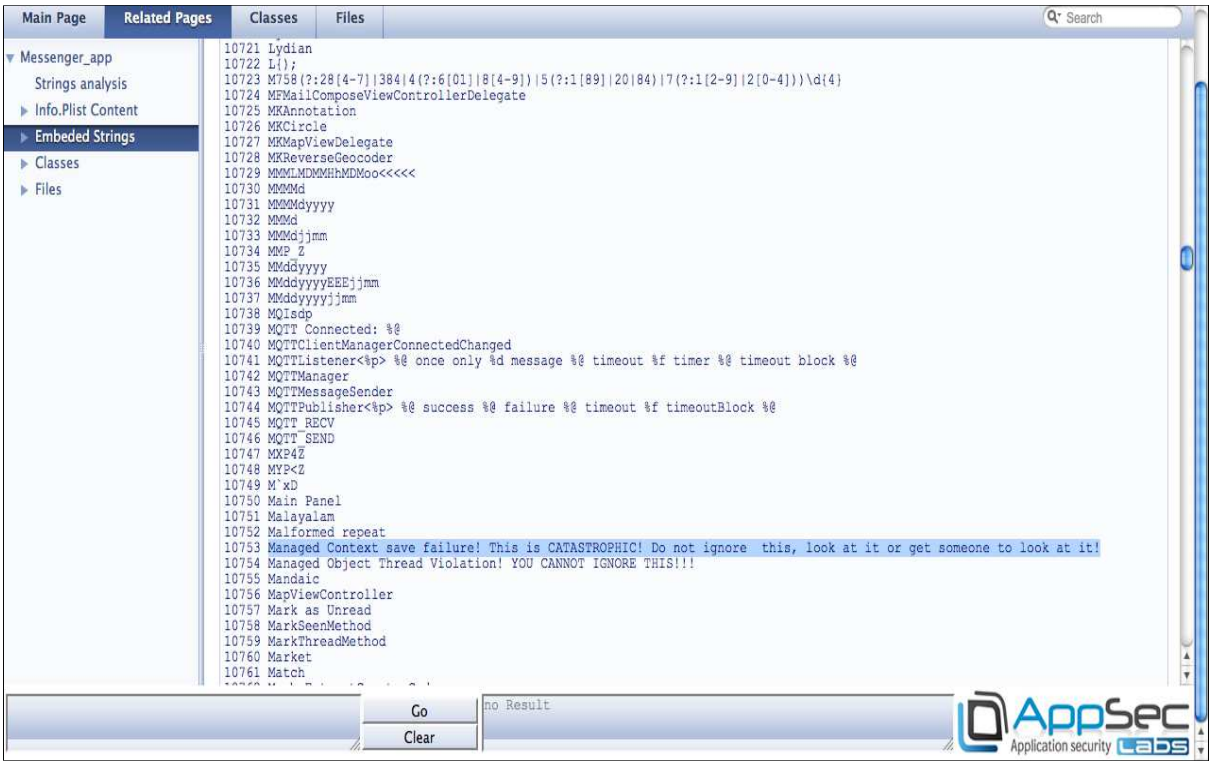
Presenting all of the system variables, for the sake of tampering attacks:



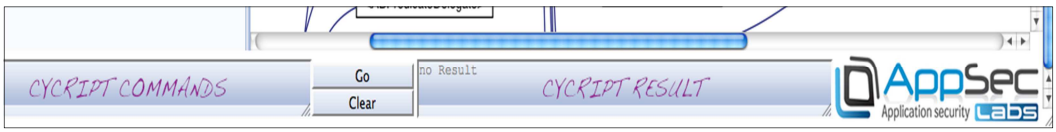
Presenting all of the system characteristics, for the sake of injection / tampering attacks:



Presenting all of the embedded strings from the Mach-O file, for the sake of sensitive information leakage damage analysis:



In addition to all this, the interface includes a Cycript activation environment directly to the device, so there is no need to open SSH and work from a terminal:



iNalyzer enables us to stop referring to information security tests on iOS systems as Black Box; it allows me to receive all of the available information in a convenient way into a user-friendly testing interface.

Now instead of using proxy to perform attacks like in regular web-based systems, I turn the application into the spearhead in the testing process, so our testing environment looks like the following diagram:

Summary

In this article, I tried to squeeze in as many subjects related with the iOS application testing environment, I did not cover most of the subjects in depth, but I am a great believer in ones' ability to learn and advance at his own pace. The main idea was to introduce you to the key players in this process, as well as presenting some of the methodology it encompasses. In addition, I presented the iNalyzer to you as an advanced system for this type of testing and I demonstrated some of its advantages.

I will be very happy if you find the iNalyzer to be helpful in your analysis process, and will be even happier if you decide to customize it according to your needs. This is why it is distributed as a free, open source tool. You can download it from our website and find other updates and movies. The entire article is given for free, for the good of the community and I am hopeful you find it interesting and helpful, if you would like you are welcome to contact me for questions and feedback.

About the author

Chilik Tamir is an information security expert with over two decades of experience in research, development, testing, consulting and training in the field of applicative information security for clients in the fields of finance, security, government offices and corporations. Among his previous publications you will find AppUse – a testing environment for Android applications developed together with Erez Metula; Belch – an automatic tool for analysis and testing of binary protocols such as Flex and Java-Serialization; as well as his lectures in conferences in Israel such as OWASP IL 2011 and OWASP IL 2012.

He is the Chief Scientist at AppSec Labs, where he acts as head of R&D and innovation.