# Chapter 4 iOS toolkit

In chapter 3, we’ve introduced the OSX toolkit for iOS reverse engineering. To get our work done, we still need to install and configure several tools on iOS to combine both platforms. All operations in this chapter are finished on iPhone 5, iOS 8.1, if you encounter any problems, please talk to us on http://bbs.iosre.com.

## 4.1 CydiaSubstrate



Figure 4- 1 Logo of CydiaSubstrate

CydiaSubstrate (as shown in figure 4-1) is the infrastructure of most tweaks. It consists of MobileHooker, MobileLoader and Safe mode.

### 4.1.1 MobileHooker

MobileHooker is used to replace system calls, or namely, hook. There are two major functions:

void MSHookMessageEx(Class class, SEL selector, IMP replacement, IMP \*result);

void MSHookFunction(void\* function, void\* replacement, void\*\* p\_original);

MSHookMessageEx works on Objective-C methods. It calls method\_setImplementation to replace the original implementation of [class selector] with “replacement”. What exactly does this mean? For example, if we send the message hasSuffix: to an NSString object (i.e, call [NSString hasSuffix:]), in normal situation, this method’s implementation is to indicate whether an NSString object has a certain suffix. But if we change this implementation with the implementation of hasPrefix:, then after an NSString object receives hasSuffix: message, it actually verifies whether an NSString object has a certain prefix. Isn’t it easy to understand?

Logos syntax, which we’ve introduced in chapter 3, is actually an encapsulation of MSHookMessageEx. Although Logos is clean and elegant, while making it easy to write Objective-C hooks, it’s still based on MSHookMessageEx. For Objective-C hooks, we recommend using Logos instead of MSHookMessageEx. If you are interested in the use of MSHookMessageEx, you can take a look at its official document, or Google “cydiasubstrate fuchsiaexample”, the link starting with “<http://www.cydiasubstrate.com>” is what you are looking for.

MSHookFunction is used for C/C++ hooks, and works in assembly level. Conceptually, when the process is about to call “function”, MSHookFunction makes it execute “replacement” instead, and allocate some memory to store the original “function” and its return address, making it possible for the process to execute “function” optionally, and guarantees the process can run as usual after executing “replacement”.

Maybe it’s hard to understand the above paragraph, so here comes an example. Let’s take a look at figure 4-2.

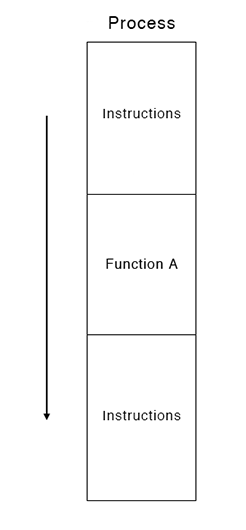


Figure 4- 2 Normal execution flow of a process

As shown in figure 4-2, a process executes some instructions, then calls function A, and afterward executes the remaining instructions. If we hook function A and replace it with function B, then this process’ execution flow changes to figure 4-3.

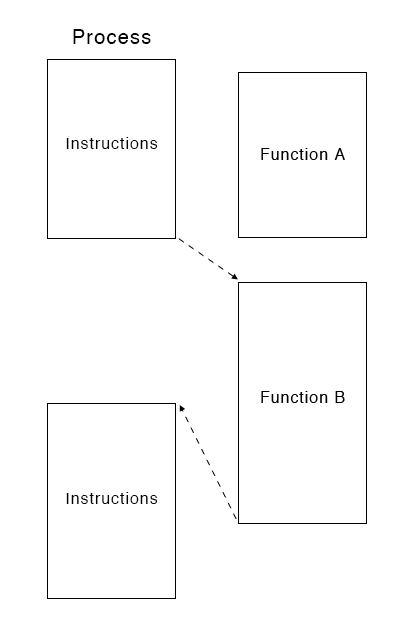


Figure 4- 3 Replace Function A with B

We can see in figure 4-3 that this process executes some instructions at first, but then calls function B at where it’s supposed to call function A, with function A stored elsewhere. Inside function B, it’s up to you whether and when to call function A. After function B finishes execution, the process will continue to execute the remaining instructions.

There’s one more thing to notice. MSHookFunction has a requirement on the length of the function it hooks, the total length of all its instructions must be bigger than 8 bytes (This number is not officially acknowledged). So here comes the question, how to hook these less-than-8-bytes short functions?

One workaround is hooking functions inside the short functions. The reason why a function is short is often because it calls other functions and they’re doing the actual job. Some of the other functions are long enough to be hooked, so we can choose these functions to be MSHookFunction’s targets, then do some logical judgements in “replacement” to tell if the short function is the caller. If we can make sure the short function is calling the “replacement”, then we can write our modification to the short function right inside “replacement”.

If you are still confused about MSHookFunction, here is a simple example. To be honest, this example contains too much low-level knowledge, hence is quite hard for beginners to understand. No worry if you happen to be a newbie, just skip to section 4.1.2. When you encounter a similar situation later in practice, review this section and you’ll know what we’re talking about. Anyway, welcome to <http://bbs.iosre.com> for further discussion.

Follow me:

1) Create iOSRETargetApp with Theos. The commands are as follows:

snakeninnys-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl

NIC 2.0 - New Instance Creator

------------------------------

[1.] iphone/application

[2.] iphone/library

[3.] iphone/preference\_bundle

[4.] iphone/tool

[5.] iphone/tweak

Choose a Template (required): 1

Project Name (required): iOSRETargetApp

Package Name [com.yourcompany.iosretargetapp]: com.iosre.iosretargetapp

Author/Maintainer Name [snakeninny]: snakeninny

Instantiating iphone/application in iosretargetapp/...

Done.

2) Modify RootViewController.mm as follows:

#import "RootViewController.h"

class CPPClass

{

public:

void CPPFunction(const char \*);

};

void CPPClass::CPPFunction(const char \*arg0)

{

for (int i = 0; i < 66; i++) // This for loop makes this function long enough to validate MSHookFunction

{

u\_int32\_t randomNumber;

if (i % 3 == 0) randomNumber = arc4random\_uniform(i);

NSProcessInfo \*processInfo = [NSProcessInfo processInfo];

NSString \*hostName = processInfo.hostName;

int pid = processInfo.processIdentifier;

NSString \*globallyUniqueString = processInfo.globallyUniqueString;

NSString \*processName = processInfo.processName;

NSArray \*junks = @[hostName, globallyUniqueString, processName];

NSString \*junk = @"";

for (int j = 0; j < pid; j++)

{

if (pid % 6 == 0) junk = junks[j % 3];

}

if (i % 68 == 1) NSLog(@"Junk: %@", junk);

}

NSLog(@"iOSRE: CPPFunction: %s", arg0);

}

extern "C" void CFunction(const char \*arg0)

{

for (int i = 0; i < 66; i++) // This for loop makes this function long enough to validate MSHookFunction

{

u\_int32\_t randomNumber;

if (i % 3 == 0) randomNumber = arc4random\_uniform(i);

NSProcessInfo \*processInfo = [NSProcessInfo processInfo];

NSString \*hostName = processInfo.hostName;

int pid = processInfo.processIdentifier;

NSString \*globallyUniqueString = processInfo.globallyUniqueString;

NSString \*processName = processInfo.processName;

NSArray \*junks = @[hostName, globallyUniqueString, processName];

NSString \*junk = @"";

for (int j = 0; j < pid; j++)

{

if (pid % 6 == 0) junk = junks[j % 3];

}

if (i % 68 == 1) NSLog(@"Junk: %@", junk);

}

NSLog(@"iOSRE: CFunction: %s", arg0);

}

extern "C" void ShortCFunction(const char \*arg0) // ShortCFunction is too short to be hooked

{

CPPClass cppClass;

cppClass.CPPFunction(arg0);

}

@implementation RootViewController

- (void)loadView {

self.view = [[[UIView alloc] initWithFrame:[[UIScreen mainScreen] applicationFrame]] autorelease];

self.view.backgroundColor = [UIColor redColor];

}

- (void)viewDidLoad

{

[super viewDidLoad];

CPPClass cppClass;

cppClass.CPPFunction("This is a C++ function!");

CFunction("This is a C function!");

ShortCFunction("This is a short C function!");

}

@end

We’ve written a CPPClass::CPPFunction, a CFunction, and a ShortCFunction as our hooking targets. Here, we’ve intentionally added some useless code in the CPPClass::CPPFunction and CFuntion for the purpose of increasing the length of these two functions, which validates MSHookFunction. However, MSHookFunction will fail on ShortCFunction because of its short length, and we have a plan B for this situation.

3) Modify Makefile and install the tweak:

export THEOS\_DEVICE\_IP = iOSIP

export ARCHS = armv7 arm64

export TARGET = iphone:clang:latest:8.0

include theos/makefiles/common.mk

APPLICATION\_NAME = iOSRETargetApp

iOSRETargetApp\_FILES = main.m iOSRETargetAppApplication.mm RootViewController.mm

iOSRETargetApp\_FRAMEWORKS = UIKit CoreGraphics

include $(THEOS\_MAKE\_PATH)/application.mk

after-install::

install.exec "su mobile -c uicache"

In the above code, "su mobile - C uicache” is used to refresh the UI cache of SpringBoard so that iOSRETargetApp’s icon can be shown on SpringBoard. Run "make package install" in Terminal to install this tweak on the device. Launch iOSRETargetApp, ssh into iOS after the red background shows, and see whether it outputs as expected:

FunMaker-5:~ root# grep iOSRE: /var/log/syslog

Nov 18 11:13:34 FunMaker-5 iOSRETargetApp[5072]: iOSRE: CPPFunction: This is a C++ function!

Nov 18 11:13:34 FunMaker-5 iOSRETargetApp[5072]: iOSRE: CFunction: This is a C function!

Nov 18 11:13:35 FunMaker-5 iOSRETargetApp[5072]: iOSRE: CPPFunction: This is a short C function!

4) Create iOSREHookerTweak with Theos, the commands are as follows:

snakeninnys-MacBook:Code snakeninny$ /opt/theos/bin/nic.pl

NIC 2.0 - New Instance Creator

------------------------------

[1.] iphone/application

[2.] iphone/library

[3.] iphone/preference\_bundle

[4.] iphone/tool

[5.] iphone/tweak

Choose a Template (required): 5

Project Name (required): iOSREHookerTweak

Package Name [com.yourcompany.iosrehookertweak]: com.iosre.iosrehookertweak

Author/Maintainer Name [snakeninny]: snakeninny

[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]: com.iosre.iosretargetapp

[iphone/tweak] List of applications to terminate upon installation (space-separated, '-' for none) [SpringBoard]: iOSRETargetApp

Instantiating iphone/tweak in iosrehookertweak/...

Done.

5) Modify Tweak.xm as follows:

#import <substrate.h>

void (\*old\_\_ZN8CPPClass11CPPFunctionEPKc)(void \*, const char \*);

void new\_\_ZN8CPPClass11CPPFunctionEPKc(void \*hiddenThis, const char \*arg0)

{

if (strcmp(arg0, "This is a short C function!") == 0) old\_\_ZN8CPPClass11CPPFunctionEPKc(hiddenThis, "This is a hijacked short C function from new\_\_ZN8CPPClass11CPPFunctionEPKc!");

else old\_\_ZN8CPPClass11CPPFunctionEPKc(hiddenThis, "This is a hijacked C++ function!");

}

void (\*old\_CFunction)(const char \*);

void new\_CFunction(const char \*arg0)

{

old\_CFunction("This is a hijacked C function!"); // Call the original CFunction

}

void (\*old\_ShortCFunction)(const char \*);

void new\_ShortCFunction(const char \*arg0)

{

old\_CFunction("This is a hijacked short C function from new\_ShortCFunction!"); // Call the original ShortCFunction

}

%ctor

{

@autoreleasepool

{

MSImageRef image = MSGetImageByName("/Applications/iOSRETargetApp.app/iOSRETargetApp");

void \*\_\_ZN8CPPClass11CPPFunctionEPKc = MSFindSymbol(image, "\_\_ZN8CPPClass11CPPFunctionEPKc");

if (\_\_ZN8CPPClass11CPPFunctionEPKc) NSLog(@"iOSRE: Found CPPFunction!");

MSHookFunction((void \*)\_\_ZN8CPPClass11CPPFunctionEPKc, (void \*)&new\_\_ZN8CPPClass11CPPFunctionEPKc, (void \*\*)&old\_\_ZN8CPPClass11CPPFunctionEPKc);

void \*\_CFunction = MSFindSymbol(image, "\_CFunction");

if (\_CFunction) NSLog(@"iOSRE: Found CFunction!");

MSHookFunction((void \*)\_CFunction, (void \*)&new\_CFunction, (void \*\*)&old\_CFunction);

void \*\_ShortCFunction = MSFindSymbol(image, "\_ShortCFunction");

if (\_ShortCFunction) NSLog(@"iOSRE: Found ShortCFunction!");

MSHookFunction((void \*)\_ShortCFunction, (void \*)&new\_ShortCFunction, (void \*\*)&old\_ShortCFunction); // This MSHookFuntion will fail because ShortCFunction is too short to be hooked

}

}

In the above code, we should pay extra attention to some points:

* The use of MSFindSymbol

Simply put, the role of MSFindSymbol is to search the symbol to be hooked. Well, what’s a symbol?

In computer, the instructions of a function are stored in memory. When the process is going to call the function, it needs to know where to locate the function in memory, and then executes its instructions at there. That is to say, the process needs to know the memory address of a function according to its name. The mapping of function names and addresses is stored in the "symbol table”. “symbol” is the name of the function, according to which the process locates the function’s address in memory and then jumps there to execute it.

Imagine such a scenario: Your App calls a lookup function in a dylib to query information on your server. If another App gets to know the symbol of “lookup”, then it can import the dylib, and call the function as it wishes, causing great consumption of your server resources.

To avoid this, symbols are divided into 2 types, i.e. public symbols and private symbols (Besides, there are stripped symbols, but they have little to do with this chapter. If you are interested in stripped symbols, please visit the following reference links or google by yourselves). Private symbols are not property of yours, you can not make use of them as you wish. That’s to say, MSHookFunction will fail on private symbols without further manipulation. So saurik provides the MSFindSymbol function to access private symbols. If the concept of symbol is still beyond comprehension, just keep the following code pattern in mind:

MSImageRef image = MSGetImageByName("/path/to/binary/who/contains/the/implementation/of/symbol");

void \*symbol = MSFindSymbol(image, "symbol");

The parameter of MSGetImageByName is “The full path of the binary which contains the implementation of the function". For example, the implementation of NSLog is in the Foundation framework, so the parameter should be "/System/Library/Frameworks/Foundation.framework/Foundation". Got it?

You can refer to the official document at http://www.cydiasubstrate.com/api/c/MSFindSymbol/ for a more detailed explanation of MSFindSymbol. As for the types and definition of symbols, please read http://msdn.microsoft.com/en-us/library/windows/hardware/ff553493(v=vs.85).Aspx and http://en.wikibooks.org/wiki/Reverse\_Engineering /Mac\_OS\_X#Symbols\_Types.

* The origin of symbol

You may have already noticed that, the functions we defined in RootViewController.mm were CPPClass:: CPPFunction, CFunction and ShortCFunction. How did they change into \_\_ZN8CPPClass11CPPFunctionEPKc, \_CFunction and \_ShortCFunction respectively in tweak.xm? In brief, that was because the compiler “mangled” (changed) the function name. It’s unnecessary here for us to know how every name is mangled, we are only concerned with the results. Where does these 3 underline prefixed symbols come from? In reverse engineering, normally we don’t have the right to access the source code of our targets, so these symbols are all extracted via IDA, as illustrated in this example.

Drag and drop iOSRETargetApp’s binary into IDA. The Functions window after initial analysis is shown in figure 4-4.

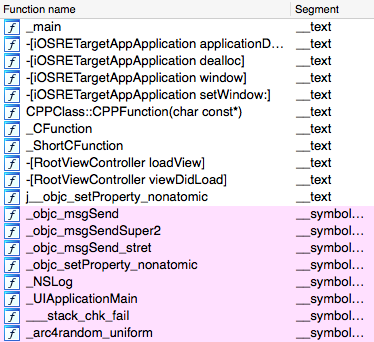


Figure 4- 4 Functions window

As we can see, CPPClass::CPPFunction(char const\*), \_CFunction and \_ShortCFunction are listed here. Double click “CPPClass::CPPFunction(char const\*)” to go to its implementation, as shown in figure 4-5.

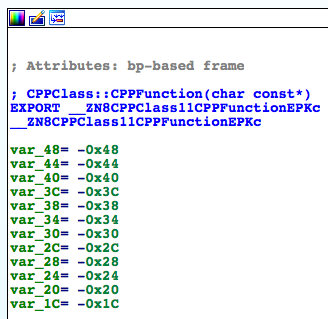


Figure 4- 5 CPPClass::CPPFunction(char const\*)

The underline prefixed string in line 4 is exactly the symbol we’re looking for. In the same way, where \_CFunction and \_ShortCFunction come from is obviously shown in figure 4-6 and figure 4-7.

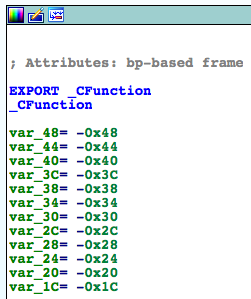


Figure 4- 6 CFunction

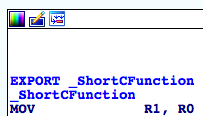


Figure 4- 7 ShortCFunction

This approach of symbol locating applies to all kinds of symbols. In the beginning stage, we suggest you keep in mind that a symbol and its corresponding function name are different, while ignore the hows and whys. During your whole process of studying reverse engineering, the concept of symbol will imperceptibly goes into your knowledge system, thus there is no need to push it for now.

* The writing pattern of MSHookFunction

The 3 parameters of MSHookFunction are: the original function to be hooked/replaced, the replacement function, and the original function saved by MobileHooker. Just like Sherlock Holmes needs Dr. Watson’s assistance, MSHookFunction doesn’t work alone, it only functions with a conventional writing pattern, shown as follows:

#import <substrate.h>

returnType (\*old\_symbol)(args);

returnType new\_symbol(args)

{

// Whatever

}

void InitializeMSHookFunction(void) // This function is often called in %ctor i.e. constructor

{

MSImageRef image = MSGetImageByName("/path/to/binary/who/contains/the/implementation/of/symbol");

void \*symbol = MSFindSymbol(image, "symbol");

if (symbol) MSHookFunction((void \*)symbol, (void \*)&new\_ symbol, (void \*\*)&old\_ symbol);

else NSLog(@"Symbol not found!");

}

You’ll recognize this pattern if you review Tweak.xm in iOSREHookerTweak. Again, we cannot get the source code of the function to hook, so we don’t know the prototype of the function: What is the returnType? How many args are there and what’re their types? At this moment, we need the help of more advanced reverse engineering skills to reconstruct the prototype of the function. Chapter 6 focuses on this knowledge, so don’t worry if you can’t catch up for now. I strongly suggest you review this section after finishing chapter 6, I bet you will get a better understanding at that time.

6) Modify Makefile and install the tweak:

export THEOS\_DEVICE\_IP = iOSIP

export ARCHS = armv7 arm64

export TARGET = iphone:clang:latest:8.0

include theos/makefiles/common.mk

TWEAK\_NAME = iOSREHookerTweak

iOSREHookerTweak\_FILES = Tweak.xm

include $(THEOS\_MAKE\_PATH)/tweak.mk

after-install::

install.exec "killall -9 iOSRETargetApp"

Now please relaunch iOSRETargetApp and see if the output matches our expectation:

FunMaker-5:~ root# grep iOSRE: /var/log/syslog

Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: Found CPPFunction!

Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: Found CFunction!

Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: Found ShortCFunction!

Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: CPPFunction: This is a hijacked C++ function!

Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: CFunction: This is a hijacked C function!

Nov 18 11:29:14 FunMaker-5 iOSRETargetApp[5327]: iOSRE: CPPFunction: This is a hijacked short C function from new\_\_ZN8CPPClass11CPPFunctionEPKc!

It is worth mentioning that, we failed hooking the short function (i.e. ShortCFunction), otherwise it would print "This is a hijacked short C function from new\_ShortCFunction!”. But we succeeded in hooking other functions (i.e. CPPClass::CPPFunction) inside the short function. We could tell if the caller was ShortCFuncation by judging the callee’s argument, thus indirectly hooked short function and met our needs. The introduction of MSHookFunction above covers almost every situation a beginner may encounter. Since Theos only provides encapsulation for MSHookMessageEx, thorough understanding of the use of MSHookFunction is particularly important. If MSHookFunction still confuses you, get to us on http://bbs.iosre.com.

### 4.1.2 MobileLoader

The role of MobileLoader is to load third-party dylibs. When iOS launches, launchd will load MobileLoader into memory, then MobileLoader will call dlopen according to tweaks’ plist filters to load dylibs under /Library/MobileSubstrate/DynamicLibraries/ into different processes. The format of the plist filter here has been explained in details in the previous Theos section, which saves my words here. For most rookie iOS reverse engineers, MobileLoader works transparently, knowing the existence of it is enough.

### 4.1.3 Safe mode

iOS crashes when tweak sucks. A tweak is essentially a dylib residing in another process, once something goes wrong in it, the entire process crashes. If it unfortunately happens to be SpringBoard or other system processes, tweak crash leads to a system paralysis. So CydiaSubstrate introduces Safe mode: It captures SIGTRAP, SIGABRT, SIGILL, SIGBUS, SIGSEGV and SIGSYS signals, then enter safe mode, as shown in figure 4-8.

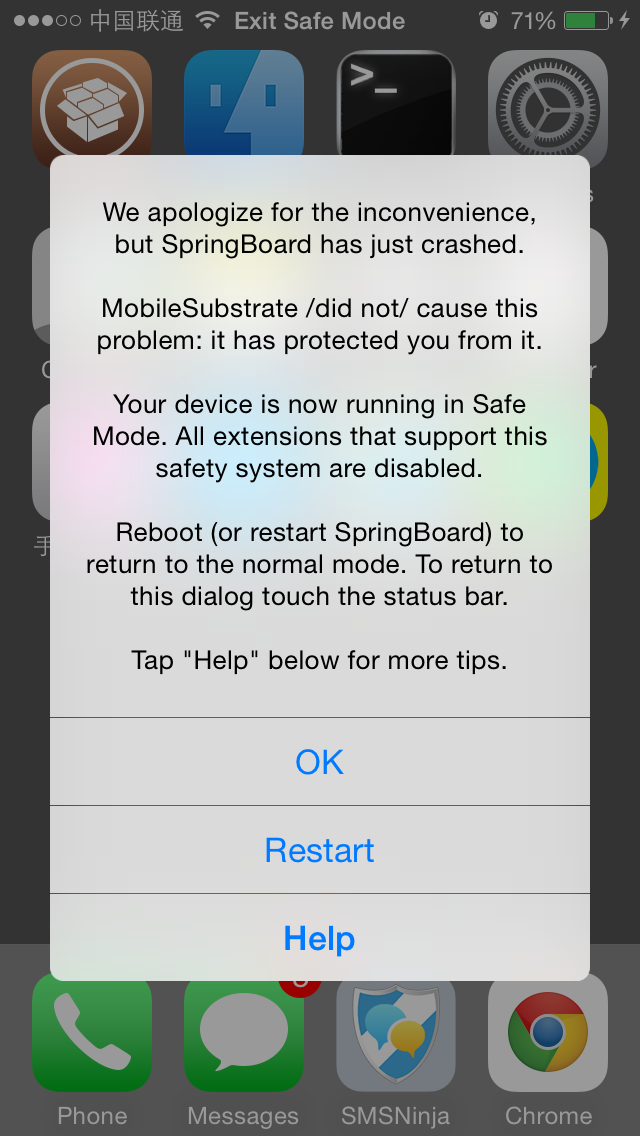


Figure 4- 8 Safe mode

In safe mode, all third-party tweaks that base on CydiaSubstrate will be disabled for troubleshooting. But safe mode can’t guarantee absolute safety, in many cases, devices fail to boot because of problematic third-party dylibs. In this situation, you can perform a hard reboot by pressing and holding the home and lock buttons, then completely disable CydiaSubstrate by holding the volume "+" button. After iOS successfully reboots, you can start error checking. When the problems are fixed, reboot iOS again to enable CydiaSubstrate.

## 4.2 Cycript

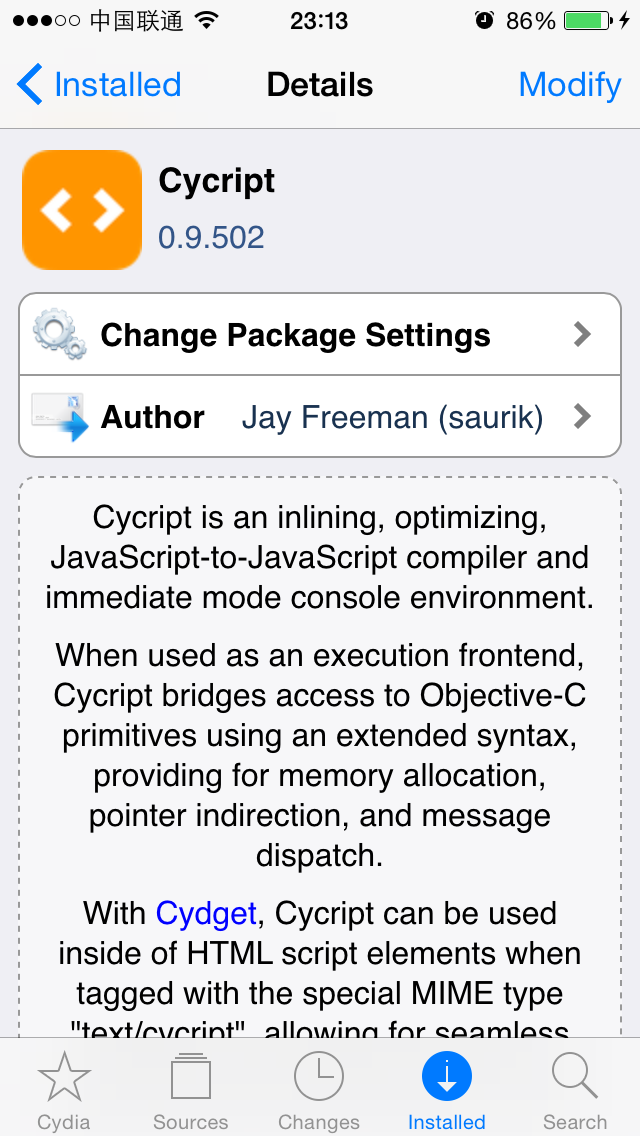


Figure 4- 9 Cycript

Cycript (As shown in figure 4-9) is a scripting language developed by saurik. You can view Cycript as Objective-JavaScript. A lot of you may not be familiar with JavaScript, then subconsciously think Cycript is very obscure. In fact, I, as a lazy learner, do not know JavaScript either, so in a long time, I’ve ignored Cycript deliberately. It wasn’t until not long ago when I was playing with MTerminal during my company's boring meeting and tested a few Objective-C methods in Cycript, then I had a new awareness of this simple yet powerful language. In fact, for Objective-C programmers, scripting languages are not difficult to use, as long as we overcome our fear of difficulty, we will be able to handle them very quickly, and Cycript is no exception. Cycript has the convenience of scripting language, you can even write App in Cycript, but saurik himself said, "This isn't quite 'ready for primetime’". In my humble opinion, the most practical usage of Cycript is testing private methods in an easy manner, possessing both safety and efficiency. Therefore, this book will only use Cycript to test methods. For its complete manual, please visit the official website <http://www.cycript.org>.

We can launch Cycript either in MTerminal or via ssh. Input “cycript” and it outputs “cy#”, which indicates Cycript’s successful launch.

FunMaker-5:~ root# cycript

cy#

After that, you can start coding. Instead of writing Apps, we mainly use Cycript to test methods, so we need to inject and run code in an existing process. Let’s exit Cycript by pressing "control + D" for now. Generally speaking, which process to inject depends on what methods we’re testing: Suppose the methods to be tested are from class A, and class A exists in process B, then you should inject into process B to test the methods. Stop beating around the bush, let’s see an example to make everything more straightforward.

If now we want to test the class method +sharedNumberFormatter in class PhoneApplication to reconstruct its prototype, we have to inject into the process MobilePhone because PhoneApplication only exists in MobilePhone; Similarly, for the instance method [SBUIController lockFromSource:], we have to inject into SpringBoard; Naturally, for [NSString length], we can inject into any process who imports Foundation.framework. Because most of the methods we test are private, so the general rules are that if the methods you’re testing are from a process, inject right into that process; If they’re from a lib, inject into the processes that import this lib.

Testing methods via process injection is rather simple. Take SpringBoard for an example, first we need to find out its process name or process ID (PID):

FunMaker-5:~ root# ps -e | grep SpringBoard

4567 ?? 0:27.45 /System/Library/CoreServices/SpringBoard.app/SpringBoard

4634 ttys000 0:00.01 grep SpringBoard

As we can see, SpringBoard’s PID is 4634. Input “cycript -p 4634” or “cycript -p SpringBoard” to inject Cycript into SpringBoard. Now Cycript has been injected into SpringBoard and we can start method testing.

UIAlertView is a most frequently used UI class on iOS. Only 3 lines of code in Objective-C are needed for a popup:

UIAlertView \*alertView = [[UIAlertView alloc] initWithTitle:@"iOSRE" message:@"snakeninny" delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];

[alertView show];

[alertView release];

It’s easy to convert the above Objective-C code into Cycript code:

FunMaker-5:~ root# cycript -p SpringBoard

cy# alertView = [[UIAlertView alloc] initWithTitle:@"iOSRE" message:@"snakeninny" delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil]

#"<UIAlertView: 0x1700e580; frame = (0 0; 0 0); layer = <CALayer: 0x164146c0>>"

cy# [alertView show]

cy# [alertView release]

No need to declare the type of an object, no need to add a semicolon at the end of each line, that’s Cycript. If a function has a return value, Cycript will print its memory address and description in real time, which is very intuitive. After Cycript executes the above code, a popup shows on SpringBoard, as shown in figure 4-10.

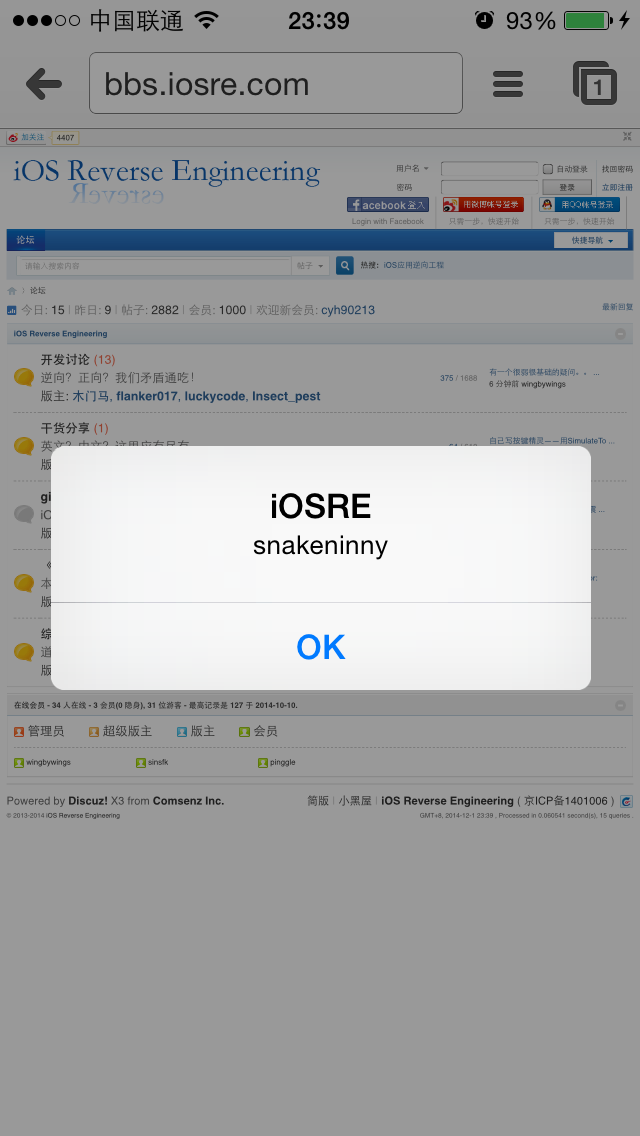


Figure 4- 10 Code execution in Cycript

If we already know the memory address of an object, we can use “#” operator to access the object like this:

cy# [[UIAlertView alloc] initWithTitle:@"iOSRE" message:@"snakeninny" delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil]

#"<UIAlertView: 0x166b4fb0; frame = (0 0; 0 0); layer = <CALayer: 0x16615890>>"

cy# [#0x166b4fb0 show]

cy# [#0x166b4fb0 release]

If we know an object of a certain class exists in the current process but don’t know its memory address, we cannot obtain the object with “#”. Under such circumstance, we can try “choose” out:

cy# choose(SBScreenShotter)

[#"<SBScreenShotter: 0x166e0e20>"]

cy# choose(SBUIController)

[#"<SBUIController: 0x16184bf0>"]

“choose” a class, Cycript returns its objects. This command is so convenient that it doesn’t succeed all the time. When it fails to get you a usable object, you’re on your own. We’ll talk about how to get our target objects manually in chapter 6, please stay tuned.

When it comes to testing private methods, a combination of the above Cycript commands would be enough. Let’s summarize the use of Cycript with an example of logging in to iMessage with my Apple ID. First we need to get an instance of iMessage login controller:

FunMaker-5:~ root# cycript -p SpringBoard

cy# controller = [CNFRegController controllerForServiceType:1]

#"<CNFRegController: 0x166401e0>"

Then login with my Apple ID:

cy# [controller beginAccountSetupWithLogin:@"snakeninny@gmail.com" password:@"bbs.iosre.com" foundExisting:NO]

#"IMAccount: 0x166e7b30 [ID: 5A8E19BE-1BC9-476F-AD3B-729997FAA3BC Service: IMService[iMessage] Login: E:snakeninny@gmail.com Active: YES LoginStatus: Connected]"

This is an equivalent of logging in iMessage as shown in figure 4-11.

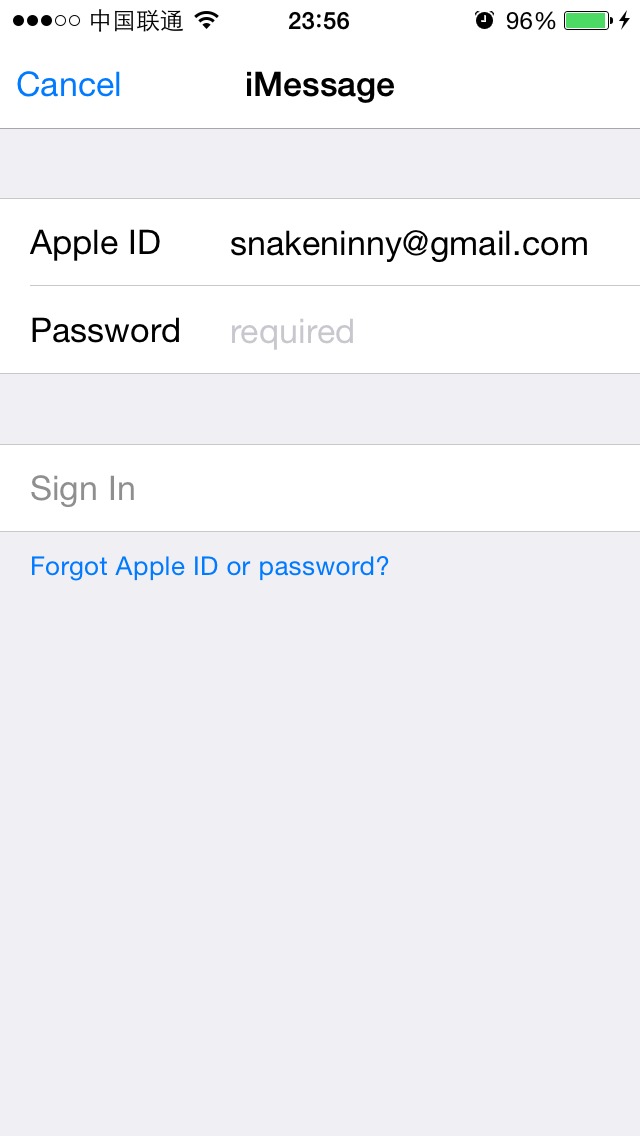


Figure 4- 11 Log in iMessage

This method returns a logged in IMAccount, i.e my iMessage account. Then select the addresses for sending and receiving iMessages:

cy# [controller setAliases:@[@"snakeninny@gmail.com"] onAccount:#0x166e7b30]

1

This is an equivalent of selecting iMessage addresses as shown in figure 4-12.

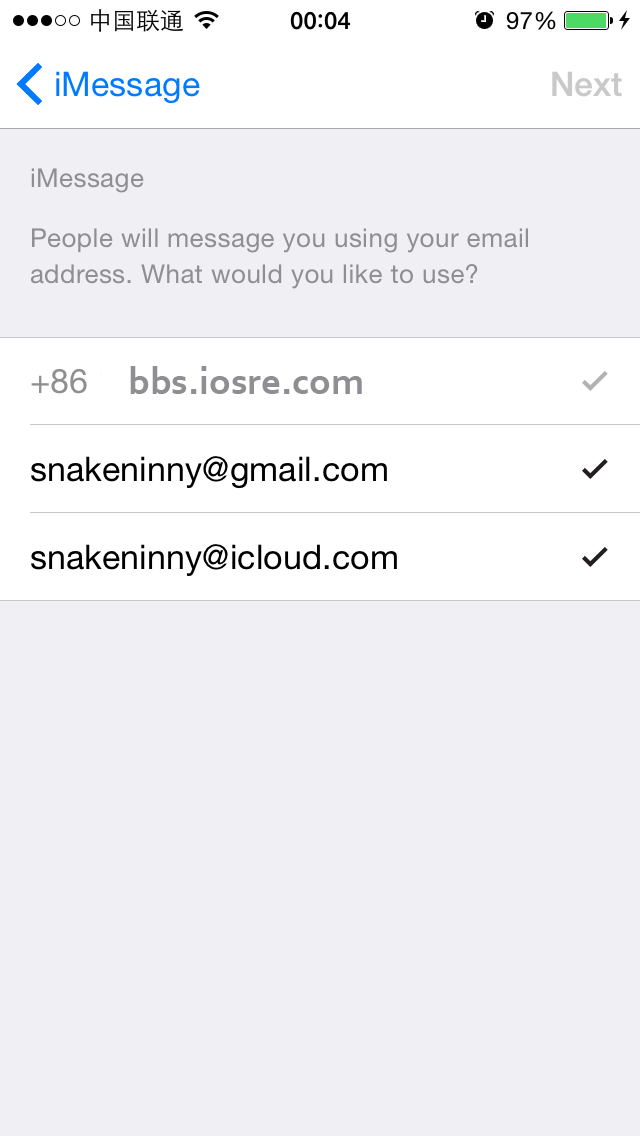


Figure 4- 12 Select iMessage addresses

The return value indicates our correctness by far. Finally, let’s check if my iMessage account is ready to rock!

cy# [#0x166e7b30 CNFRegSignInComplete]

1

1 in number is YES in BOOL. We can start iMessaging others right now.

Simple and clear, right? No further explanation needed. As the exercise of this section, now it’s your turn to convert the above Cycript code into Objective-C code, and write a tweak to verify your conversion as well get familiar with Cycript. One last note, remember to change my Apple ID to yours.

## 4.3 LLDB and debugserver

### 4.3.1 Introduction to LLDB

If IDA is caliburn, then LLDB is excalibur, they are at roughly the same position in iOS reverse engineering. LLDB, a production of Apple, stands for "Low Level Debugger". It’s the Xcode built-in dynamic debugger supporting C, C++ and Objective-C, working on OSX, iOS and the iOS simulator.

LLDB’s functionality sums up in 4 points:

❏ Launch the program under the conditions you specify;

❏ Stop the program under the conditions you specify;

❏ Inspect the internal status of a program when it stops;

* Modify the program when it stops, and observe the modification of its execution flow.

LLDB is a command line tool, it does not have a graphical interface. Its mass output in Terminal scares off beginners easily, but once you master the basic commands of LLDB, you’ll be surprised by its formidable combination with IDA. LLDB runs in OSX, so to debug iOS, we need another tool’s assistance on iOS, which is debugserver.

### 4.3.2 Introduction to debugserver

debugserver runs on iOS. As its name suggests, it plays the role of a server and executes the commands from LLDB (as a client), then returns the results to LLDB to show to the user. This working mode is called “remote debugging”. By default, debugserver is not installed on iOS. We need to connect the device to Xcode, configure it to enable debugging in menu Window→Devices, then debugserver will be installed to "/Developer/usr/bin/" on iOS.

However, because of the lack of task\_for\_pid permission, the raw debugserver installed by Xcode can only debug our own Apps. Debugging our own Apps is no mystery in App development, and since we’re reverse engineering, we have our own Apps’ source code, there is no need to reverse them. It’d only be cool if we can debug other Apps. No worry, here comes the solution. With a little hacking, debugserver and LLDB can be used to debug other Apps, maximizing their power.

### 4.3.3 Configure debugserver

#### 1. Help debugserver lose some weight

Find the corresponding ARM type of your device according to table 4-1.

|  |  |
| --- | --- |
| Name | ARM |
| iPhone 4s | armv7 |
| iPhone 5 | armv7s |
| iPhone 5c | armv7s |
| iPhone 5s | arm64 |
| iPhone 6 Plus | arm64 |
| iPhone 6 | arm64 |
| iPad 2 | armv7 |
| iPad mini | armv7 |
| The New iPad | armv7 |
| iPad with Retina display | armv7s |
| iPad Air | arm64 |
| iPad Air 2 | arm64 |
| iPad mini with Retina display | arm64 |
| iPad mini 3 | arm64 |
| iPod touch 5 | armv7 |

Table 4-1 iOS 8 Compatible devices

My device is iPhone 5, its matching ARM type is armv7s. Copy the raw debugserver from iOS to “/Users/snakeninny/” on OSX.

snakeninnysiMac:~ snakeninny$ scp root@iOSIP:/Developer/usr/bin/debugserver ~/debugserver

Then help it lose some weight:

snakeninnysiMac:~ snakeninny$ lipo -thin armv7s ~/debugserver -output ~/debugserver

Note that you need to change “armv7s” here to the corresponding ARM type of your device.

#### 2. Grant task\_for\_pid permission to debugserver

Download <http://iosre.com/ent.xml> to “/Users/snakeninny/” on OSX, then run the following command:

snakeninnysiMac:~ snakeninny$ /opt/theos/bin/ldid -Sent.xml debugserver

Note, there is no space between “-S” and “ent.xml”.

If everything goes fine, ldid will take less than 5 seconds to finish its job. But if ldid gets stuck and times out, just try another workaround: Download <http://iosre.com/ent.plist> to “/Users/snakeninny/”, then run the following command:

snakeninnysiMac:~ snakeninny$ codesign -s - --entitlements ent.plist -f debugserver

#### 3. Copy the modified debugserver back to iOS

Copy the modified debugserver to iOS and grant it execute permission with the following commands:

snakeninnysiMac:~ snakeninny$ scp ~/debugserver root@iOSIP:/usr/bin/debugserver

snakeninnysiMac:~ snakeninny$ ssh root@iOSIP

FunMaker-5:~ root# chmod +x /usr/bin/debugserver

One thing to clarify, the reason we put the modified debugserver under “/usr/bin/" instead of overriding the original one is because, first, the original debugserver is not writable, we just cannot override it; Second, we don’t need to input full paths to execute commands under “/usr/bin/”, just run “debugserver” wherever you want, and debugserver is ready to roll out.

### 4.3.4 Process launching and attaching using debugserver

2 most commonly used scenarios of debugserver are process launching and attaching. Both possess very simple commands:

debugserver -x backboard IP:port /path/to/executable

debugserver will launch the specific executable and open the specific port, then wait for LLDB’s connection from IP.

debugserver IP:port -a "ProcessName"

debugserver will attach to process with the name “ProcessName” and open the specific port, then wait for LLDB’s connection from IP.

For example:

FunMaker-5:~ root# debugserver -x backboard \*:1234 /Applications/MobileSMS.app/MobileSMS

debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89

for armv7.

Listening to port 1234 for a connection from \*...

The above command will launch MobileSMS and open port 1234, then wait for LLDB’s connection from any IP. And for the following command:

FunMaker-5:~ root# debugserver 192.168.1.6:1234 -a "MobileSMS"

debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89

for armv7.

Attaching to process MobileNotes...

Listening to port 1234 for a connection from 192.168.1.6...

debugserver will attach to MobileSMS and open port 1234, then wait for LLDB’s connection from 192.168.1.6.

If something goes wrong when executing the above commands, such as:

FunMaker-5:~ root# debugserver \*:1234 -a "MobileSMS"

dyld: Library not loaded: /Developer/Library/PrivateFrameworks/ARMDisassembler.framework/ARMDisassembler

  Referenced from: /usr/bin/debugserver

  Reason: image not found

Trace/BPT trap: 5

It means necessary debugging data under “/Developer/” is missing. This is generally because we did not enable development mode on this device in Xcode’s Window→Devices menu. You can fix the issue by re-enabling development mode on this device.

When you exit debugserver, the process being debugged also exits. The configuration of debugserver is over for now, the following operation are performed on LLDB.

### 4.3.5 Use LLDB

Before introducing LLDB, we need to know a big bug in the latest LLDB: LLDB (version 320.x.xx) in Xcode 6 sometimes messes up ARM with THUMB instructions on armv7 and armv7s devices, making it impossible to debug. Before the publishing of this book, the bug has not been fixed yet. A temporary solution is to download and install Xcode 5.0.x from https://developer.apple.com/downloads/index.action, their built-in LLDB (version 300.x.xx) works fine on armv7 and armv7s devices. When you’re installing the old version of Xcode, make sure you install it in a different path from the current Xcode, say “/Applications/OldXcode.app”, thus it won't affect the current Xcode. To launch the old LLDB, you need to specify the full path:

snakeninnysiMac:~ snakeninny$ /Applications/OldXcode.app/Contents/Developer/usr/bin/lldb

Then the old LLDB will launch and you can connect it to the waiting debugserver:

(lldb) process connect connect://iOSIP:1234

Process 790987 stopped

\* thread #1: tid = 0xc11cb, 0x3995b4f0 libsystem\_kernel.dylib`mach\_msg\_trap + 20, queue = 'com.apple.main-thread, stop reason = signal SIGSTOP

frame #0: 0x3995b4f0 libsystem\_kernel.dylib`mach\_msg\_trap + 20

libsystem\_kernel.dylib`mach\_msg\_trap + 20:

-> 0x3995b4f0: pop {r4, r5, r6, r8}

0x3995b4f4: bx lr

libsystem\_kernel.dylib`mach\_msg\_overwrite\_trap:

0x3995b4f8: mov r12, sp

0x3995b4fc: push {r4, r5, r6, r8}

Note, the execution of "process connect connect://iOSIP:1234” will take a rather long time (approximately more than 3 minutes in a WiFi environment) to connect to debugserver, please be patient. In section 4.6, there will be an introduction to connecting to debugserver through USB, which will save a lot of time. When the process is stopped by debugserver, we can start debugging. Let’s have a look at the commonly used commands in LLDB.

#### 1. image list

“image list” is similar to "info shared” in GDB, which is used to list the main executable and all dependent libraries (hereinafter referred to as images) in the debugged process. Because of ASLR (Address Space Layout Randomization, see http://theiphonewiki.com/wiki/ASLR), every time the process launches, a random offset will be added to the starting address of all images in that process, making their virtual memory addresses hard to predict.

For example, suppose there is an image B in process A, and image B is 100 bytes in size. When process A launches for the 1st time, image B may be loaded into virtual memory at 0x00 to 0x64; For the 2nd time, image B may be loaded into 0x10 to 0x74, and 0x60 to 0xC4 for the 3rd time. That is to say, although image B’s size stays 100 bytes, every launch changes the starting address, which happens to be a key value in our following operations. Then comes the question, how do we get this key value?

The answer is"image list -o -f". After LLDB has connected to debugserver, run "image list -o -f" to view its output:

(lldb) image list -o -f

[ 0] 0x000cf000 /private/var/db/stash/\_.29LMeZ/Applications/SMSNinja.app/SMSNinja(0x00000000000d3000)

[ 1] 0x0021a000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x000000000021a000)

[ 2] 0x01645000 /usr/lib/libobjc.A.dylib(0x00000000307b5000)

[ 3] 0x01645000 /System/Library/Frameworks/Foundation.framework/Foundation(0x0000000023c4f000)

[ 4] 0x01645000 /System/Library/Frameworks/CoreFoundation.framework/CoreFoundation(0x0000000022f0b000)

[ 5] 0x01645000 /System/Library/Frameworks/UIKit.framework/UIKit(0x00000000264c1000)

[ 6] 0x01645000 /System/Library/Frameworks/CoreGraphics.framework/CoreGraphics(0x0000000023238000)

……

[235] 0x01645000 /System/Library/Frameworks/CoreGraphics.framework/Resources/libCGXType.A.dylib(0x00000000233a2000)

[236] 0x0008a000 /usr/lib/dyld(0x000000001fe8a000)

In the above output, the 1st column, [X], is the sequence number of the image; the 2nd column is the image’s random offset generated by ASLR (hereinafter referred to as the ASLR offset); the 3rd column is the full path of this image, the content in brackets is the original starting address plus the ASLR offset. Do all these offsets and addresses confuse you? Take it easy, hopefully you’ll sort it through after an example.

Suppose the virtual memory is a shooting range with 1000 target positions. You can regard the images in a process as targets and now there are 600 of them. All these targets are uniformly arranged in a row with target 1 in position 1, target 2 in position 2, target 600 in position 600, etc. And positions 601 to 1000 are all empty. You can see the layout in figure 4-13 (The number at the top is the target position number, and the target number is at the bottom).

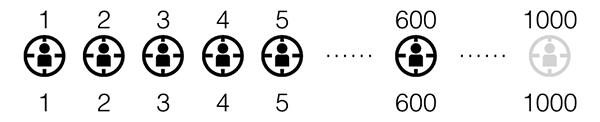


Figure 4- 13 Shooting range (1)

The images’ starting addresses in virtual memory are like the target positions of the 600 targets, which are named image base addresses in terminology. Now the owner of this shooting range thinks the previous targets are arranged rashly, shooters will hit all bulls’-eyes as soon he gets used to the arrangement. So the owner relocates all these targets randomly. After relocation, target 1 is placed in position 5, target 2 is placed in position 6, target 3 is placed in position 8, target 4 is placed in position 13, target 5 is placed in position 15...... Target 600 is placed in position 886, as shown in figure 4-14.

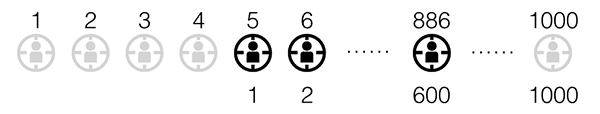


Figure 4- 14 Shooting range (2)

That’s to say, the offsets for target 1, 2, 3, 4, 5 and 600 are 4, 4, 5, 9, 10 and 286 respectively. This random offset (ASLR) greatly increases the shooting difficulty. For target 1, it used to be at position 1, and it is at position 5 for now, so the offset is 4, i.e.

image base address with offset = image base address without offset + ASLR offset

Back to the reverse engineering scene, let’s take the 4th image (i.e. Foundation) in the output of "image list -o -f" as an example, its ASLR offset is 0x1645000, its image base address with offset is 0x23c4f000, so according to the above formula, its image base address without offset is 0x23c4f000 - 0x1645000 = 0x2260A000.

You may wonder, where does 0x2260A000 come from? Drag and drop Foundation’s binary into IDA, after the initial analysis, IDA looks like figure 4-15.

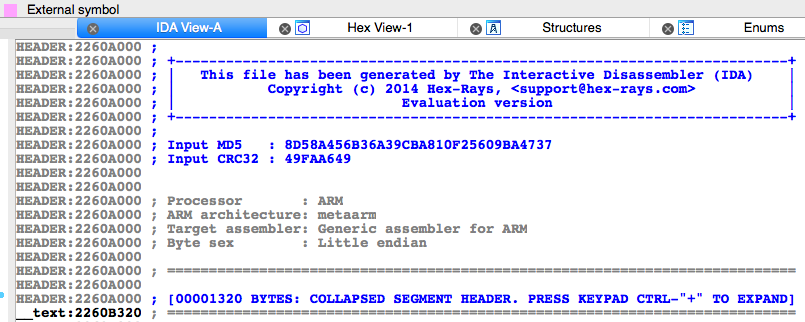


Figure 4- 15 Analyze Foundation in IDA

Scroll to the top of IDA View-A, do you see “HEADER:2260A000” in the first line? This is the origin of 0x2260A000.

Now that we’ve known "base address" means "starting address", let’s talk about another concept which is similar to “image base address”, i.e. “symbol base address”. Return to IDA and search for "NSLog" in the Functions window, and then jump to its implementation, as shown in figure 4-16.

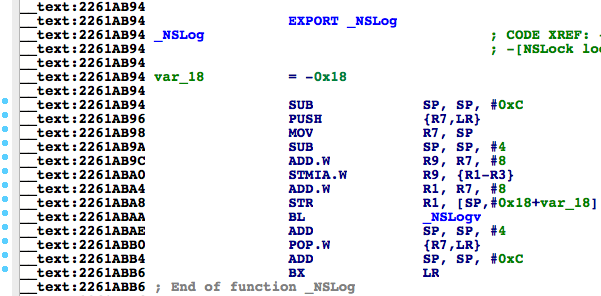


Figure 4- 16 NSLog

Because the base address of Foundation is a known number, and NSLog is in a fixed position inside Foundation, we can get the base address of NSLog according to the following formula:

base address of NSLog = relative address of NSLog in Foundation + base address of Foundation

What does “relative address of NSLog in Foundation” mean? Let’s go back to figure 4-16 and find the first instruction of NSLog, i.e. “SUB SP, SP, #0xC”. On the left, do you see the number 0x2261AB94? This the “address of NSLog in Foundation”. Subtract Foundation’s image base address without offset, i.e. 0x2260A000 from it, we get the “relative address of NSLog in Foundation”, i.e. 0x10B94.

Hence, the base address of NSLog is 0x10B94 + 0x23c4f000 = 0x23C5FB94. I guess some of you have already noticed that the formula

image base address with offset = image base address without offset + ASLR offset

With tiny modifications, is a new formula for symbols:

symbol base address with offset = symbol base address without offset + ASLR offset of the image containing the symbol

Let’s verify this formula.

NSLog’s symbol base address without offset is 0x2261AB94, ASLR offset of Foundation is 0x1645000, add these two numbers and we get 0x23C5FB94.

By analogy, we can also get the formula for instructions:

instruction base address with offset = instruction base address without offset + ASLR offset of the image containing the instruction

Naturally, symbol base address is the base address of the first instruction of the symbol’s corresponding function.

In the following content, base addresses with offset will be frequently used. Make sure you understand all concepts in this section then keep in mind: Base address without offset can be viewed in IDA, ASLR offset can be viewed in LLDB, add them together we get base address with offset. As for where in IDA and LLDB to search for the values, I bet you’ll get it after thoroughly reading this section.

#### 2. breakpoint

“breakpoint” is similar to “break” in GDB, it’s used to set breakpoints. In reverse engineering, we usually set breakpoints like these:

b function

Or

br s –a address

Or

br s –a ‘ASLROffset+address’

The former command is to set a breakpoint at the beginning of a function, for instance:

(lldb) b NSLog

Breakpoint 2: where = Foundation`NSLog, address = 0x23c5fb94

The latter two commands are to set a breakpoint at a specific address, for instance:

(lldb) br s -a 0xCCCCC

Breakpoint 5: where = SpringBoard`\_\_\_lldb\_unnamed\_function303$$SpringBoard, address = 0x000ccccc

(lldb) br s -a '0x6+0x9'

Breakpoint 6: address = 0x0000000f

Note that the “X” in the output “Breakpoint X:” is an integer id of that breakpoint, and we will use this number soon. When the process stops at a breakpoint, the line of code holding the breakpoint hasn’t been executed yet.

In reverse engineering, we'll be debugging assembly code, so in most cases we’ll be setting breakpoint on a specific assembly instruction instead of a function. To set a breakpoint on an assembly instruction, we have to know its base address with offset, which we have already explained in details. Now let’s take -[SpringBoard \_menuButtonDown:] for an example and set a breakpoint on the first instruction as a demonstration.

1. Find the base address without offset in IDA

Open SpringBoard’s binary in IDA, switch to Text view after the initial analysis and locate “- [SpringBoard \_menuButtonDown:]”, as shown in figure 4-17.

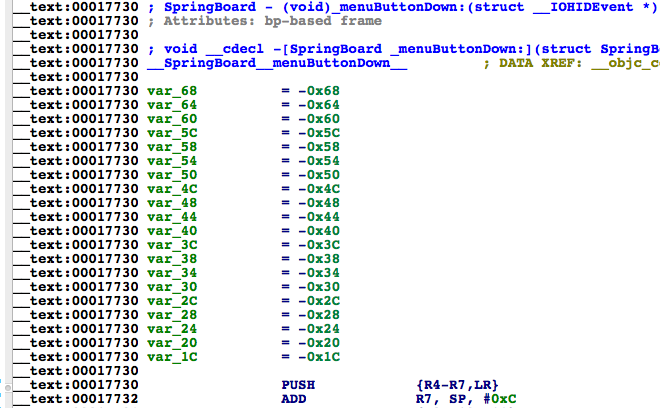


Figure 4- 17 [SpringBoard \_menuButtonDown:]

As we can see, the base address without offset of the first instruction “PUSH {R4-R7, LR}” is 0x17730.

1. Find the ASLR offset in LLDB

ssh into iOS to run debugserver with the following commands:

snakeninnysiMac:~ snakeninny$ ssh root@iOSIP

FunMaker-5:~ root# debugserver \*:1234 -a "SpringBoard"

debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89

for armv7.

Attaching to process SpringBoard...

Listening to port 1234 for a connection from \*...

Then connect to debugserver with LLDB on OSX, and find the ASLR offset:

snakeninnysiMac:~ snakeninny$ /Applications/OldXcode.app/Contents/Developer/usr/bin/lldb

(lldb) process connect connect://iOSIP:1234

Process 93770 stopped

\* thread #1: tid = 0x16e4a, 0x30dee4f0 libsystem\_kernel.dylib`mach\_msg\_trap + 20, queue = 'com.apple.main-thread, stop reason = signal SIGSTOP

frame #0: 0x30dee4f0 libsystem\_kernel.dylib`mach\_msg\_trap + 20

libsystem\_kernel.dylib`mach\_msg\_trap + 20:

-> 0x30dee4f0: pop {r4, r5, r6, r8}

0x30dee4f4: bx lr

libsystem\_kernel.dylib`mach\_msg\_overwrite\_trap:

0x30dee4f8: mov r12, sp

0x30dee4fc: push {r4, r5, r6, r8}

(lldb) image list -o -f

[ 0] 0x000b5000 /System/Library/CoreServices/SpringBoard.app/SpringBoard(0x00000000000b9000)

[ 1] 0x006ea000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000006ea000)

[ 2] 0x01645000 /System/Library/PrivateFrameworks/StoreServices.framework/StoreServices(0x000000002ca70000)

[ 3] 0x01645000 /System/Library/PrivateFrameworks/AirTraffic.framework/AirTraffic(0x0000000027783000)

……

[419] 0x00041000 /usr/lib/dyld(0x000000001fe41000)

(lldb) c

Process 93770 resuming

The ASLR offset of SpringBoard is 0xb5000.

1. Set and trigger the breakpoint

So the base address with offset of the first instruction is 0x17730 + 0xb5000 = 0xCC730. Input "br s -a 0xCC730" in LLDB to set a breakpoint on the first instruction:

(lldb) br s -a 0xCC730

Breakpoint 1: where = SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard, address = 0x000cc730

Then press the home button to trigger the breakpoint:

(lldb) br s -a 0xCC730

Breakpoint 1: where = SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard, address = 0x000cc730

Process 93770 stopped

\* thread #1: tid = 0x16e4a, 0x000cc730 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x000cc730 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard:

-> 0xcc730: push {r4, r5, r6, r7, lr}

0xcc732: add r7, sp, #12

0xcc734: push.w {r8, r10, r11}

0xcc738: sub sp, #80

(lldb) p (char \*)$r1

(char \*) $0 = 0x0042f774 "\_menuButtonDown:"

When the process stops, you can use "c" command to “continue” (running) the process. Compared to GDB, a significant improvement in LLDB is that you can enter commands while the process is running. But be careful, some processes (such as SpringBoard) will automatically relaunch because of timeout after stopping for a period of time. For this kind of processes, you should try to keep it running to avoid unexpected automatic relaunching.

You can also use commands like “br dis”, “br en” and “br del” to disable, enable and delete breakpoints. The command to disable all breakpoints is as follows:

(lldb) br dis

All breakpoints disabled. (2 breakpoints)

The command to disable a specific breakpoint is as follows:

(lldb) br dis 6

1 breakpoints disabled.

The command to enable all breakpoints is as follows:

(lldb) br en

All breakpoints enabled. (2 breakpoints)

The command to enable a specific breakpoint is as follows:

(lldb) br en 6

1 breakpoints enabled.

The command to delete all breakpoints is as follows:

(lldb) br del

About to delete all breakpoints, do you want to do that?: [Y/n] Y

The command to delete a specific breakpoint is as follows:

(lldb) br del 8

1 breakpoints deleted; 0 breakpoint locations disabled.

Another useful command is that we can set a series of commands on a breakpoint to be automatically executed when we hit the breakpoint. Suppose breakpoint 1 is set on a specific objc\_msgSend function, the commands to set a series of commands on breakpoint 1 are as follows:

(lldb) br com add 1

After executing the above command, LLDB will ask for a series of commands, ending with “DONE”.

Enter your debugger command(s). Type 'DONE' to end.

> po [$r0 class]

> p (char \*)$r1

> c

> DONE

Here we’ve input 3 commands, once breakpoint 1 is hit, LLDB will execute them one by one:

(lldb) c

Process 97048 resuming

\_\_NSArrayM

(char \*) $11 = 0x26c6bbc3 "count"

Process 97048 resuming

Command #3 'c' continued the target.

“br com add” is often used to automatically obverse the changes in the context of a breakpoint when it is hit, which often implies valuable reverse engineering clues. We’ll see how to use this command in the latter half of this book.

#### 3. print

Thanks to “print” command, "inspecting the internal status of a program when it stops” is possible. As its name implies, this command can print the value of a register, variable, expression, etc. Again, let’s illustrate the use of “print” with "-[SpringBoard \_menuButtonDown:]", as shown in figure 4-18.

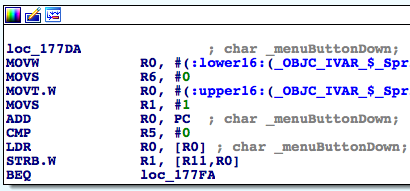


Figure 4- 18 [SpringBoard \_menuButtonDown:]

The base address with offset of “MOVS R6, #0” is known to be 0xE37DE, let’s set a breakpoint on it and print R6’s value when we hit the breakpoint:

(lldb) br s -a 0xE37DE

Breakpoint 2: where = SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 174, address = 0x000e37de

Process 99787 stopped

\* thread #1: tid = 0x185cb, 0x000e37de SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 174, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x000e37de SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 174

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 174:

-> 0xe37de: movs r6, #0

0xe37e0: movt r0, #75

0xe37e4: movs r1, #1

0xe37e6: add r0, pc

(lldb) p $r6

(unsigned int) $1 = 364526080

After this instruction is executed, R6 should be set to 0. Input "ni" to execute this instruction and reprint the value of R6:

(lldb) ni

Process 99787 stopped

\* thread #1: tid = 0x185cb, 0x000e37e0 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 176, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x000e37e0 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 176

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 176:

-> 0xe37e0: movt r0, #75

0xe37e4: movs r1, #1

0xe37e6: add r0, pc

0xe37e8: cmp r5, #0

(lldb) p $r6

(unsigned int) $2 = 0

(lldb) c

Process 99787 resuming

As we can see, command “p” has printed the value of R6 correctly.

In Objective-C, the implementation of [someObject someMethod] is actually objc\_msgSend(someObject, someMethod), among which the first argument is an Objective-C object, and the latter can be casted to a string (we will explain this in detail in chapter 6). As shown in figure 4-19, "BLX \_objc\_msgSend" executes [SBTelephonyManager sharedTelephonyManager].

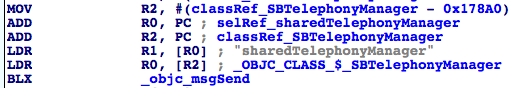


Figure 4- 19 objc\_msgSend([SBTelephonyManager class], @selector(sharedTelephonyManager))

The address with offset of “BLX \_objc\_msgSend” is known to be 0xCC8A2. Set a breakpoint on it and print the arguments of “objc\_msgSend” when we hit this breakpoint:

(lldb) br s -a 0xCC8A2

Breakpoint 1: where = SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 370, address = 0x000cc8a2

Process 103706 stopped

\* thread #1: tid = 0x1951a, 0x000cc8a2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 370, queue = 'com.apple.main-thread, stop reason = breakpoint 1.1

frame #0: 0x000cc8a2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 370

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 370:

-> 0xcc8a2: blx 0x3e3798 ; symbol stub for: objc\_msgSend

0xcc8a6: mov r6, r0

0xcc8a8: movw r0, #31088

0xcc8ac: movt r0, #74

(lldb) po [$r0 class]

SBTelephonyManager

(lldb) po $r0

SBTelephonyManager

(lldb) p (char \*)$r1

(char \*) $2 = 0x0042eee6 "sharedTelephonyManager"

(lldb) c

Process 103706 resuming

As you can see, we’ve used "po" command to print the Objective-C object, and "p (char \*)" to print the C object by casting. Quite simple, right? It’s worth mentioning that when the process stops on a "BL" instruction, LLDB will automatically parse this instruction and display the corresponding symbol:

-> 0xcc8a2: blx 0x3e3798 ; symbol stub for: objc\_msgSend

However, sometimes LLDB's parsing is wrong, mistaking the symbol. In this case, please refer to IDA’s static analysis of that symbol.

Finally, we can use “x” command to print the value stored in a specific address:

(lldb) p/x $sp

(unsigned int) $4 = 0x006e838c

(lldb) x/10 $sp

0x006e838c: 0x00000000 0x22f2c975 0x00000000 0x00000000

0x006e839c: 0x26c6bf8c 0x0000000c 0x17a753c0 0x17a753c8

0x006e83ac: 0x000001c8 0x17a75200

(lldb) x/10 0x006e838c

0x006e838c: 0x00000000 0x22f2c975 0x00000000 0x00000000

0x006e839c: 0x26c6bf8c 0x0000000c 0x17a753c0 0x17a753c8

0x006e83ac: 0x000001c8 0x17a75200

We’ve printed SP in hexadecimal with “p/x” command. SP is a pointer, whose value is 0x6e838c. And the “x/10” command has printed the 10 continuous words SP points to.

#### 4. nexti and stepi

Both of "nexti" and "stepi" are used to execute the next instruction, but the biggest difference between them is that the former does not go/step inside a function but the latter does. They are two of the most used commands, and can be abbreviated as "ni" and “si" respectively. You may wonder, what does “go inside a function or not” mean? Let’s still take "-[SpringBoard \_menuButtonDown:]" for example, as shown in figure 4-20.

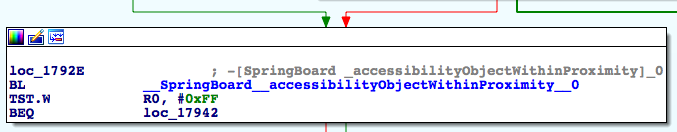


Figure 4- 20 [SpringBoard \_menuButtonDown:]

The base address with offset of “BL \_\_SpringBoard\_\_accessibilityObjectWithinProximity\_\_0” is 0xEE92E, this instruction calls \_SpringBoard\_\_accessibilityObjectWithinProximity\_\_0. Set a breakpoint on it and execute the “ni” command:

(lldb) br s -a 0xEE92E

Breakpoint 2: where = SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 510, address = 0x000ee92e

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee92e SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 510, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x000ee92e SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 510

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 510:

-> 0xee92e: bl 0x2fd654 ; \_\_\_lldb\_unnamed\_function16405$$SpringBoard

0xee932: tst.w r0, #255

0xee936: beq 0xee942 ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 530

0xee938: blx 0x403f08 ; symbol stub for: BKSHIDServicesResetProximityCalibration

(lldb) ni

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee932 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 514, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x000ee932 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 514

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 514:

-> 0xee932: tst.w r0, #255

0xee936: beq 0xee942 ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 530

0xee938: blx 0x403f08 ; symbol stub for: BKSHIDServicesResetProximityCalibration

0xee93c: movs r0, #0

(lldb) c

Process 731 resuming

As we can see, we haven’t gone inside \_SpringBoard\_\_accessibilityObjectWithinProximity\_\_0 by “ni”. Now, let’s try again with “si”:

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee92e SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 510, queue = 'com.apple.main-thread, stop reason = breakpoint 2.1

frame #0: 0x000ee92e SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 510

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 510:

-> 0xee92e: bl 0x2fd654 ; \_\_\_lldb\_unnamed\_function16405$$SpringBoard

0xee932: tst.w r0, #255

0xee936: beq 0xee942 ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 530

0xee938: blx 0x403f08 ; symbol stub for: BKSHIDServicesResetProximityCalibration

(lldb) si

Process 731 stopped

\* thread #1: tid = 0x02db, 0x002fd654 SpringBoard`\_\_\_lldb\_unnamed\_function16405$$SpringBoard, queue = 'com.apple.main-thread, stop reason = instruction step into

frame #0: 0x002fd654 SpringBoard`\_\_\_lldb\_unnamed\_function16405$$SpringBoard

SpringBoard`\_\_\_lldb\_unnamed\_function16405$$SpringBoard:

-> 0x2fd654: movw r0, #33920

0x2fd658: movt r0, #43

0x2fd65c: add r0, pc

0x2fd65e: ldrsb.w r0, [r0]

(lldb) c

Process 731 resuming

The base address without offset of “movw r0, #33920” is 0x226654, as shown in figure 4-21.

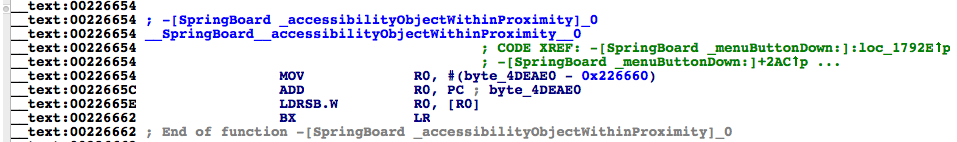


Figure 4- 21 SpringBoard\_\_accessibilityObjectWithinProximity\_\_0

This instruction is inside the \_SpringBoard\_\_accessibilityObjectWithinProximity\_\_0 function. That’s to say, the “si” command has gone inside the function, which is the meaning of “go inside a function or not”.

#### 5. register write

“register write” is used to write a specific value to a specific register, hence “modify the program when it stops, and observe the modification of its execution flow". According to the code in figure 4-22, the base address with offset of "TST.W R0, offset #0xFF" is known to be 0xEE7A2, if R0’s value is 0, the process will branch to the left, or to the right if R0 is not 0.

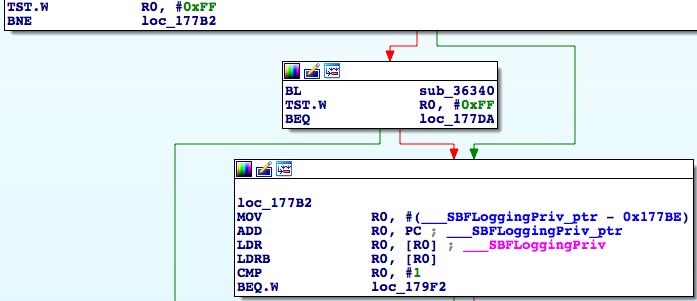


Figure 4- 22 Branches

Set a breakpoint here to see the value of R0 as follows:

(lldb) br s -a 0xEE7A2

Breakpoint 3: where = SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 114, address = 0x000ee7a2

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee7a2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 114, queue = 'com.apple.main-thread, stop reason = breakpoint 3.1

frame #0: 0x000ee7a2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 114

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 114:

-> 0xee7a2: tst.w r0, #255

0xee7a6: bne 0xee7b2 ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 130

0xee7a8: bl 0x10d340 ; \_\_\_lldb\_unnamed\_function1110$$SpringBoard

0xee7ac: tst.w r0, #255

(lldb) p $r0

(unsigned int) $0 = 0

Because the value of R0 is 0, so BNE makes the process branch to the left:

(lldb) ni

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee7a6 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 118, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x000ee7a6 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 118

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 118:

-> 0xee7a6: bne 0xee7b2 ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 130

0xee7a8: bl 0x10d340 ; \_\_\_lldb\_unnamed\_function1110$$SpringBoard

0xee7ac: tst.w r0, #255

0xee7b0: beq 0xee7da ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 170

(lldb) ni

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee7a8 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 120, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x000ee7a8 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 120

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 120:

-> 0xee7a8: bl 0x10d340 ; \_\_\_lldb\_unnamed\_function1110$$SpringBoard

0xee7ac: tst.w r0, #255

0xee7b0: beq 0xee7da ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 170

0xee7b2: movw r0, #2174

Trigger that breakpoint again, change R0’s value to 1 by “register write”, and see if the branch changes:

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee7a2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 114, queue = 'com.apple.main-thread, stop reason = breakpoint 3.1

frame #0: 0x000ee7a2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 114

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 114:

-> 0xee7a2: tst.w r0, #255

0xee7a6: bne 0xee7b2 ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 130

0xee7a8: bl 0x10d340 ; \_\_\_lldb\_unnamed\_function1110$$SpringBoard

0xee7ac: tst.w r0, #255

(lldb) p $r0

(unsigned int) $5 = 0

(lldb) register write r0 1

(lldb) p $r0

(unsigned int) $6 = 1

(lldb) ni

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee7a6 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 118, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x000ee7a6 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 118

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 118:

-> 0xee7a6: bne 0xee7b2 ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 130

0xee7a8: bl 0x10d340 ; \_\_\_lldb\_unnamed\_function1110$$SpringBoard

0xee7ac: tst.w r0, #255

0xee7b0: beq 0xee7da ; \_\_\_lldb\_unnamed\_function299$$SpringBoard + 170

(lldb)

Process 731 stopped

\* thread #1: tid = 0x02db, 0x000ee7b2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 130, queue = 'com.apple.main-thread, stop reason = instruction step over

frame #0: 0x000ee7b2 SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 130

SpringBoard`\_\_\_lldb\_unnamed\_function299$$SpringBoard + 130:

-> 0xee7b2: movw r0, #2174

0xee7b6: movt r0, #63

0xee7ba: add r0, pc

0xee7bc: ldr r0, [r0]

At this time, the program branches to the right as we expected.

There’re much more LLDB commands that worth attention, but we’re only covering 5 of the most frequently used ones in the beginning period of iOS reverse engineering, hope you can peep one spot and see the whole picture, as well feel the power of LLDB. LLDB is still under development, other than a few official websites, there is no satisfying tutorial; LLDB derives from GDB, although they have different commands, the thinking mode is almost the same. To learn LLDB in a more systematic way, I recommend you "Peter 's GDB tutorial" and "RMS's gdb Debugger Tutorial". IDA is good at static analysis, while LLDB is good at dynamic analysis. Mastery of these two tools removes all obstacles on your road to a master of reverse engineering.

### 4.3.6 Miscellaneous LLDB

#### 1. Binaries to be debugged must be right from iOS on device

If only our static and dynamic analysis target is exactly the same that the base address without offset, ASLR offset and the base address with offset are correspondent. For binaries to be analyzed in IDA, we can use dyld\_decache in chapter 3 to extract them from the shared cache on device. Binaries from SDK or iOS simulator usually don’t meet the condition.

#### 2. Shortcuts in LLDB

If you want to repeat the last command in LLDB, you can simply press “enter”. If you want to review all history commands, just press up and down on your keyboard.

LLDB commands are simple, but it’s not easy to solve complicated problems with these simples commands. In chapter 6, we will introduce more common scenarios of using LLDB, and before that, please be sure to understand the knowledge of this section.

## 4.4 dumpdecrypted

When introducing class-dump, we’ve mentioned that Apple encrypts all Apps from AppStore, protecting them from being class-dumped. If we want to class-dump StoreApps, we have to decrypt their executables at first. A handy tool, dumpdecrypted, by Stefan Esser (@i0n1c) is commonly used in iOS reverse engineering.

dumpdecrypted is open sourced on GitHub, you have to compile it by yourselves. Now let’s start from scratch to class-dump a virtual target, i.e. TargetApp.app to show you the steps of decrypting an App, please follow me.

1. Download dumpdecrypted’s source code from GitHub as follows:

snakeninnysiMac:~ snakeninny$ cd /Users/snakeninny/Code/

snakeninnysiMac:Code snakeninny$ git clone git://github.com/stefanesser/dumpdecrypted/

Cloning into 'dumpdecrypted'...

remote: Counting objects: 31, done.

remote: Total 31 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (31/31), 6.50 KiB | 0 bytes/s, done.

Resolving deltas: 100% (15/15), done.

Checking connectivity... done

1. Compile the source code and get dumpdecrypted.dylib:

snakeninnysiMac:~ snakeninny$ cd /Users/snakeninny/Code/dumpdecrypted/

snakeninnysiMac:dumpdecrypted snakeninny$ make

`xcrun --sdk iphoneos --find gcc` -Os -Wimplicit -isysroot `xcrun --sdk iphoneos --show-sdk-path` -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/Frameworks -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/PrivateFrameworks -arch armv7 -arch armv7s -arch arm64 -c -o dumpdecrypted.o dumpdecrypted.c

`xcrun --sdk iphoneos --find gcc` -Os -Wimplicit -isysroot `xcrun --sdk iphoneos --show-sdk-path` -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/Frameworks -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/PrivateFrameworks -arch armv7 -arch armv7s -arch arm64 -dynamiclib -o dumpdecrypted.dylib dumpdecrypted.o

After “make”, a dumpdecrypted.dylib will be generated under the current directory. This dylib can be reused, there’s no need to recompile.

1. Locate the executable to be decrypted with “ps” command

On iOS 8, all StoreApps are under /var/mobile/Containers/, and TargetApp.app’s executable is under /var/mobile/Containers/Bundle/Application/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX/TargetApp.app/. Since X is unknown, it’d be a great amount of work to locate the executable manually. But a simple trick will save our days: First close all StoreApps on iOS, then launch TargetApp and ssh into iOS to print all processes:

snakeninnysiMac:~ snakeninny$ ssh root@iOSIP

FunMaker-5:~ root# ps -e

PID TTY TIME CMD

1 ?? 3:28.32 /sbin/launchd

……

5717 ?? 0:00.21 /System/Library/PrivateFrameworks/MediaServices.framework/Support/mediaartworkd

5905 ?? 0:00.20 sshd: root@ttys000

5909 ?? 0:01.86 /var/mobile/Containers/Bundle/Application/03B61840-2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp

5911 ?? 0:00.07 /System/Library/Frameworks/UIKit.framework/Support/pasteboardd

5907 ttys000 0:00.03 -sh

5913 ttys000 0:00.01 ps –e

Because now there is only one running StoreApp, the only path that contains “/var/mobile/Containers/Bundle/Application/” is the full path of TargetApp’s executable.

1. Find out TargetApp’s Documents directory via Cycript

All StoreApps’ Documents directories are under /var/mobile/Containers/Data/Application/ YYYYYYYY-YYYY-YYYY-YYYY–YYYYYYYYYYYY/. Note that these Ys are different from those previous Xs, and they are not obtainable via “ps”. So this time we need to mak use of Cycript to reveal the Documents directory of TargetApp. The commands we use are as follows:

FunMaker-5:~ root# cycript -p TargetApp

cy# [[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory inDomains:NSUserDomainMask][0]

#"file:///var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents/"

1. Copy dumpdecrypted.dylib to TargetApp’s Documents directory:

snakeninnysiMac:~ snakeninny$ scp /Users/snakeninny/Code/dumpdecrypted/dumpdecrypted.dylib root@iOSIP:/var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents/

dumpdecrypted.dylib 100% 193KB 192.9KB/s 00:00

Here we’re using scp instead of iFunBox, anyway tools don’t matter.

1. Start decrypting

The usage of dumpdecrypted.dylib is as follows:

DYLD\_INSERT\_LIBRARIES=/path/to/dumpdecrypted.dylib /path/to/executable

For instance:

FunMaker-5:~ root# cd /var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents/

FunMaker-5:/var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents root# DYLD\_INSERT\_LIBRARIES=dumpdecrypted.dylib /var/mobile/Containers/Bundle/Application/03B61840-2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp

mach-o decryption dumper

DISCLAIMER: This tool is only meant for security research purposes, not for application crackers.

[+] detected 32bit ARM binary in memory.

[+] offset to cryptid found: @0x81a78(from 0x81000) = a78

[+] Found encrypted data at address 00004000 of length 6569984 bytes - type 1.

[+] Opening /private/var/mobile/Containers/Bundle/Application/03B61840-2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp for reading.

[+] Reading header

[+] Detecting header type

[+] Executable is a plain MACH-O image

[+] Opening TargetApp.decrypted for writing.

[+] Copying the not encrypted start of the file

[+] Dumping the decrypted data into the file

[+] Copying the not encrypted remainder of the file

[+] Setting the LC\_ENCRYPTION\_INFO->cryptid to 0 at offset a78

[+] Closing original file

[+] Closing dump file

A decrypted executable named TargetApp.decrypted will be created in the current directory:

FunMaker-5:/var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents root# ls

TargetApp.decrypted dumpdecrypted.dylib OtherFiles

Copy TargetApp.decrypted to OSX ASAP. class-dump and IDA have been waiting for ages!

I think these 6 steps are clear enough, but some of you may still wonder, why to copy dumpdecrypted.dylib to Documents directory?

Good question. We all know that StoreApps don’t have write permission to most of the directories outside the sandbox. Since dumpdecrypted.dylib needs to write a decrypted file while residing in a StoreApp and they have the same permission, so the destination of its write operation should be somewhere writable. StoreApp can write to its Documents directory, so dumpdecrypted.dylib should be able to work under this directory.

Let’s see what happens if dumpdecrypted.lib is not working under Documents directory:

FunMaker-5: /var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents root# mv dumpdecrypted.dylib /var/tmp/

FunMaker-5: /var/mobile/Containers/Data/Application/D41C4343-63AA-4BFF-904B-2146128611EE/Documents root# cd /var/tmp

FunMaker-5:/var/tmp root# DYLD\_INSERT\_LIBRARIES=dumpdecrypted.dylib /private/var/mobile/Containers/Bundle/Application/03B61840-2349-4559-B28E-0E2C6541F879/TargetApp.app/TargetApp

dyld: could not load inserted library 'dumpdecrypted.dylib' because no suitable image found. Did find:

dumpdecrypted.dylib: stat() failed with errno=1

Trace/BPT trap: 5

errno=1 means "Operation not permitted", dumpdecrypted.dylib failed to work as expected. If you encounter any problem or have any experience using dumpdecrypted, you are welcome to share with us at http://bbs.iosre.com.

## 4.5 OpenSSH

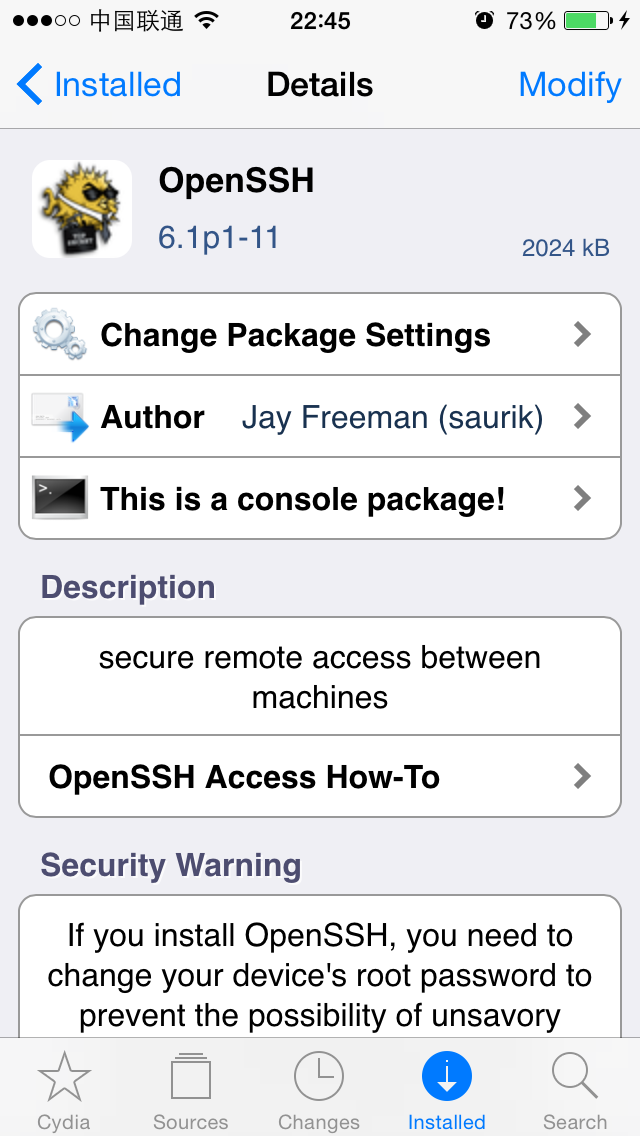


Figure 4- 23 OpenSSH

OpenSSH will install SSH service on iOS (as shown in figure 4-23). Only 2 commands are the most commonly used: ssh is used for remote logging, scp is used for remote file transfer. The usage of ssh is as follows:

ssh user@iOSIP

For instance:

snakeninnysiMac:~ snakeninny$ ssh mobile@192.168.1.6

The usage of scp is as follows:

1. Copy a local file to iOS:

scp /path/to/localFile user@iOSIP:/path/to/remoteFile

For instance:

snakeninnysiMac:~ snakeninny$ scp ~/1.png [root@192.168.1.6:/var/tmp/](mailto:root@192.168.1.6:/var/tmp/)

1. Copy a file from iOS to the local system:

scp user@iOSIP:/path/to/remoteFile /path/to/localFile

For instance:

snakeninnysiMac:~ snakeninny$ scp [root@192.168.1.6:/var/log/syslog](mailto:root@192.168.1.6:/var/log/syslog) ~/iOSlog

These two commands are relatively simple and intuitive. After installing OpenSSH, make sure to change the default login password “alpine”. There’re 2 users on iOS, i.e. root and mobile, we need to change both passwords like this:

FunMaker-5:~ root# passwd root

Changing password for root.

New password:

Retype new password:

FunMaker-5:~ root# passwd mobile

Changing password for mobile.

New password:

Retype new password:

If we forget to change the default password, there’re chances that viruses like Ikee login as root via ssh. This leads to very serious security disasters: all data on iOS including SMS, contacts, AppleID passwords and so on is at the risk of leaking, the intruder can take control over your device and do whatever he wants. Therefore, promise me you’ll change the default password after installing OpenSSH, OK?

## 4.6 usbmuxd

Most of you ssh into iOS via WiFi, which leads to slow responses in remote debugging or file copying. This is because of the instability of wireless network and the limitation of transmission speed. The well-known hacker, Nikias Bassen (@pimskeks) has written a tool named usbmuxd to forward local OSX/Windows port to remote iOS port. With this tool, we can ssh into iOS via USB, greatly increasing the speed of SSH connection. usbmuxd is easy to use:

1. Download and configure usbmuxd

Download usbmuxd from <http://cgit.sukimashita.com/usbmuxd.git/snapshot/usbmuxd-1.0.8.tar.gz> and decompress it. The files we are going to use are tcprelay.py and usbmux.py. Copy them to the same directory such as:

/Users/snakeninny/Code/USBSSH/

1. Forward local port to remote port with usbmuxd

Input the following command in Terminal:

/Users/snakeninny/Code/USBSSH/tcprelay.py -t Remote port on iOS:Local port on OSX/Windows

Now usbmuxd is forwarding local port on OSX/Windows to remote port on iOS.

Here comes an example of usage scenario: ssh into iOS via USB without WiFi, then debug SpringBoard with LLDB.

1. Forward local port 2222 on OSX to remote port 22 on iOS:

snakeninnysiMac:~ snakeninny$ /Users/snakeninny/Code/USBSSH/tcprelay.py -t 22:2222

Forwarding local port 2222 to remote port 22

1. ssh into iOS and attach debugserver to SpringBoard:

snakeninnysiMac:~ snakeninny$ ssh root@localhost -p 2222

FunMaker-5:~ root# debugserver \*:1234 -a “SpringBoard"

1. Forward local port 1234 on OSX to remote port 1234 on iOS:

snakeninnysiMac:~ snakeninny$ /Users/snakeninny/Code/USBSSH/tcprelay.py -t 1234:1234

Forwarding local port 1234 to remote port 1234

1. Start debugging in LLDB:

snakeninnysiMac:~ snakeninny$ /Applications/OldXcode.app/Contents/Developer/usr/bin/lldb

(lldb) process connect connect://localhost:1234

usbmuxd speeds up ssh connection to less than 15 seconds in general, and should be your first choice.

## 4.7 iFile

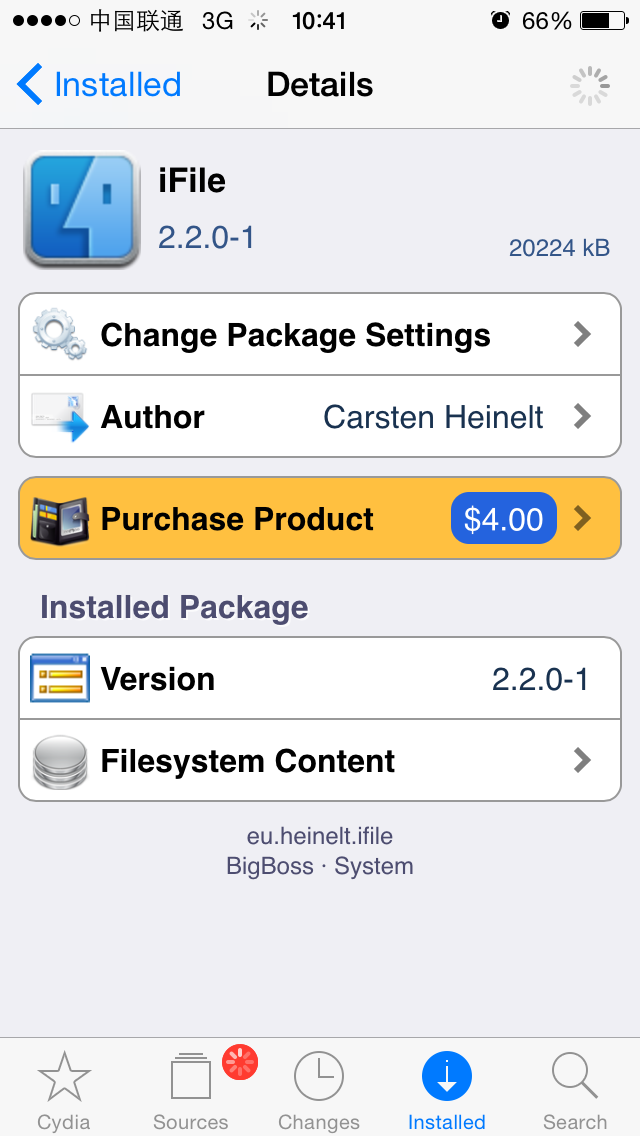


Figure 4- 24 iFile

iFile is a very powerful file management App, you can view it as Finder’s parallel on iOS. iFile is capable of all kinds of file operation including browsing, editing, cutting, copying and deb installing, possessing great convenience.

iFile is rather user-friendly. Before installing a deb, remember to close Cydia at first, then tap the deb file to be installed and choose “Installer” in the action sheet, as shown in figure 4-25.

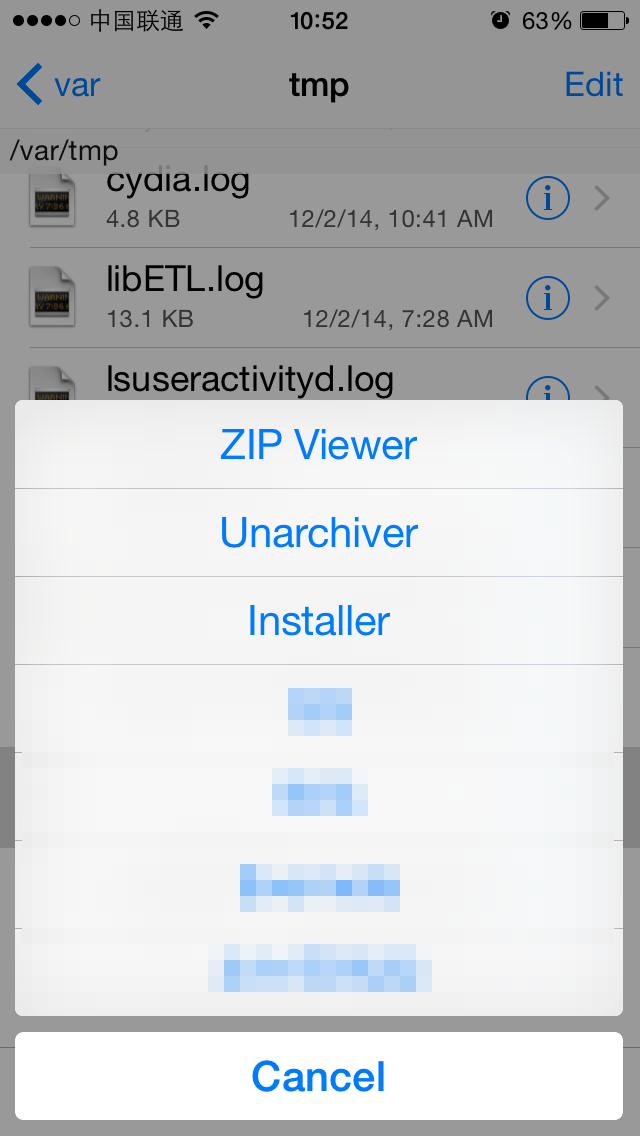


Figure 4- 25 Install deb file

## 4.8 MTerminal

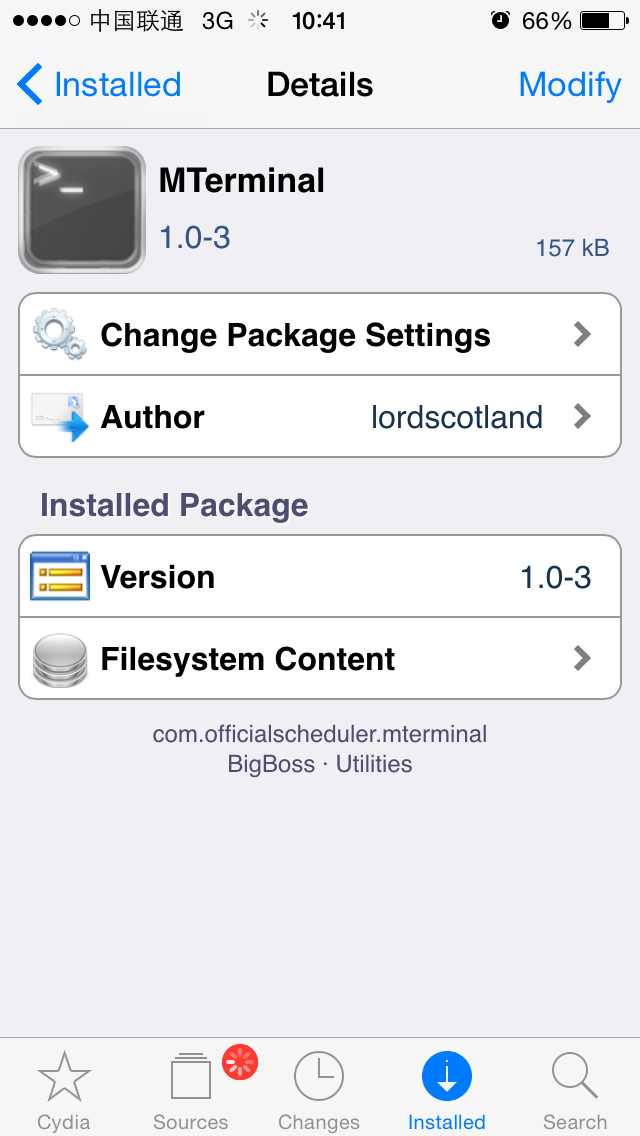


Figure 4- 26 MTerminal

MTerminal is an open sourced Terminal on iOS with all basic functions available. The usage of MTerminal is no much difference to Terminal, if we put the screen and keyboard size aside. I think the most practical scene of MTerminal is to test private methods in Cycript when we’re blanking out on the subway or something.

## 4.9 syslogd to /var/log/syslog

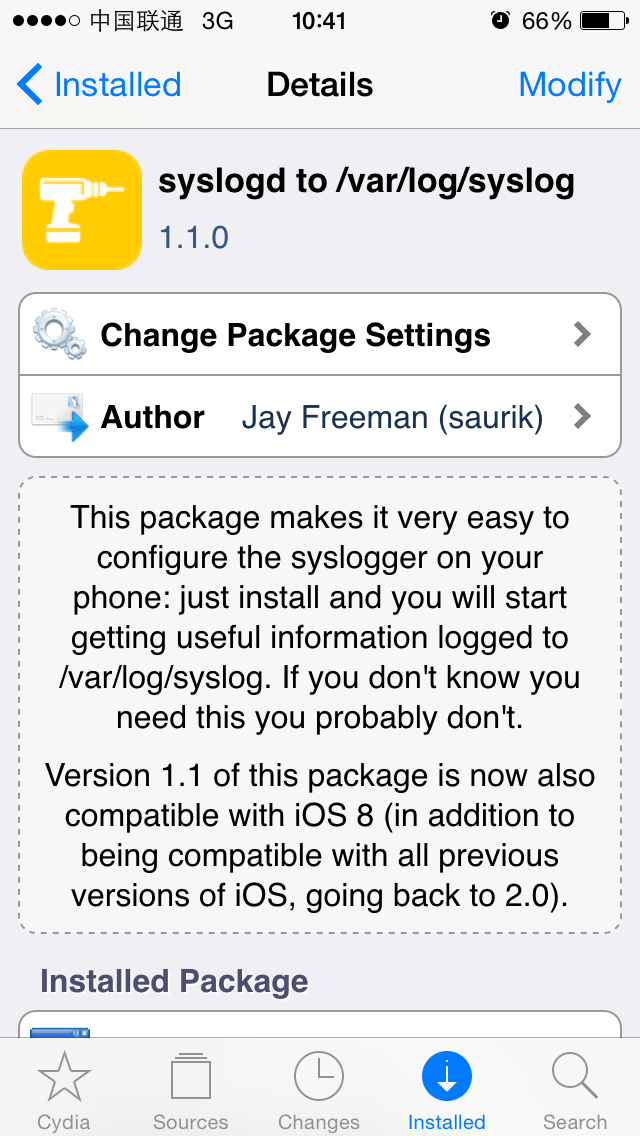


Figure 4- 27 syslogd to /var/log/syslog

syslogd is a daemon to record system logs on iOS, and "syslogd to /var/log/syslog" is used to write the logs to a file at "/var/log/syslog". You need to reboot iOS after you install this tweak to automatically create the file "/var/log/syslog". This file gets larger as time goes by, you can zero clear it with the following command:

FunMaker-5:~ root# cat /dev/null > /var/log/syslog

## 4.10 Conclusion

We’ve introduced 9 tools in this chapter, among which CydiaSubstrate, LLDB and Cycript are the top priorities. It is because of the existence of these iOS tools, along with the OSX toolkit in chapter 3, that we get a complete iOS reverse engineering environment. There's a famous Chinese saying that we should know how as well as know why. Now that we’ve already known how by finishing part 2 of this book, it’s time for us to know why in the next part. Stay tuned!