# Chapter 8 Practice 2: Mark user specific emails as read automatically

## 8.1 Mail

Email is one of the most popular communication channels in the era of Internet. Many people send and receive emails every day. Although there are lots of good email Apps on AppStore, such as Sparrow, Inbox, etc, they are not as highly integrated as the stock Mail App (hereafter referred to as Mail). Therefore, Mail is still the top choice during my daily life.

Among all emails we receive every day, most of them are valueless subscription emails like notifications and advertisements, which comes from our inadvertently clicks of subscriptions on various websites, as shown in figure 8-1.



Figure 8- 1 Mail

These emails always make me entangled. If we are kind enough to not think of them as spam messages, they are actually distracting our attention. However, if we mark them as spam messages, we may miss some useful information. So how to deal with these messages can be a real headache. I have an idea that we can add a whitelist feature to Mail, which saves our frequent contacts. Other emails outside whitelist will be marked as read automatically. With this solution, we can highlight the most valuable messages while not missing any useful information, as shown in figure 8-2.



Figure 8- 2 Mark messages outside whitelist as read

Our task for this chapter is to finish this tweak. We can divide the task into following 2 steps.

1. Add a button on the Mail UI and present an editable whitelist after pressing the button in order to add or delete entries in whitelist.
2. Every time the inbox get refreshed, mark all emails outside whitelist as read.

Simple and clear, let’s get started. All operations in this chapter are carried out on iPhone 5, iOS 8.1.1.

#### 8.2 Tweak prototyping

The initial view of Mail is shown in figure 8-3.

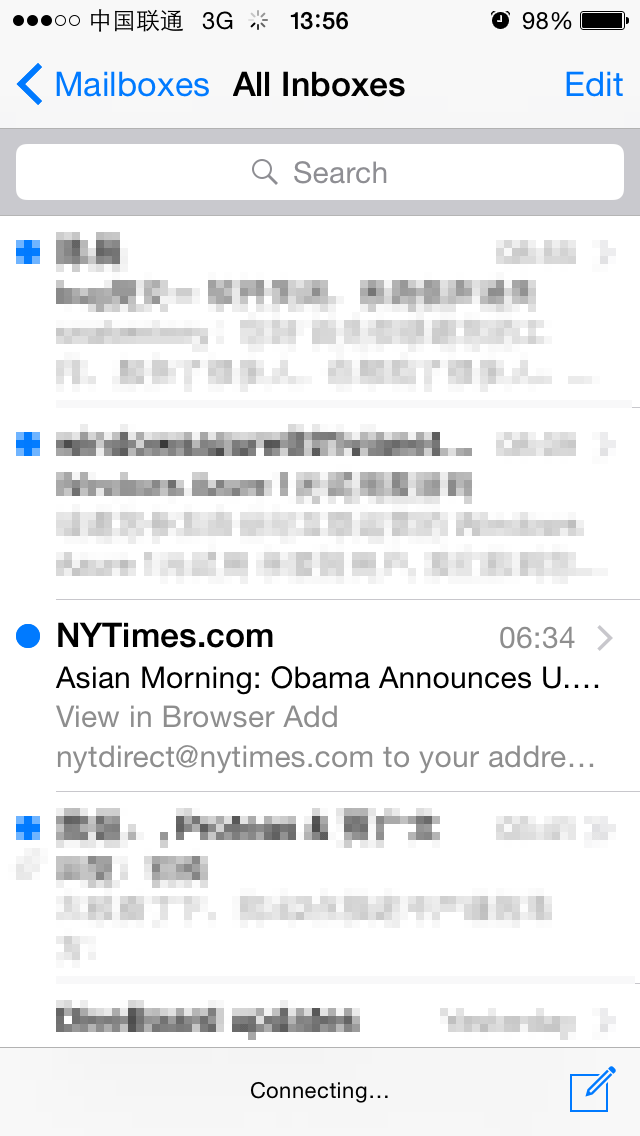


Figure 8- 3 Initial view of Mail

Where should we place the whitelist button for a better user-experience? In the “All Inboxes” view in figure 8-3, we can see that the left bottom corner is blank; maybe we can put the button here. Let’s try it out and the effect is shown in figure 8-4.

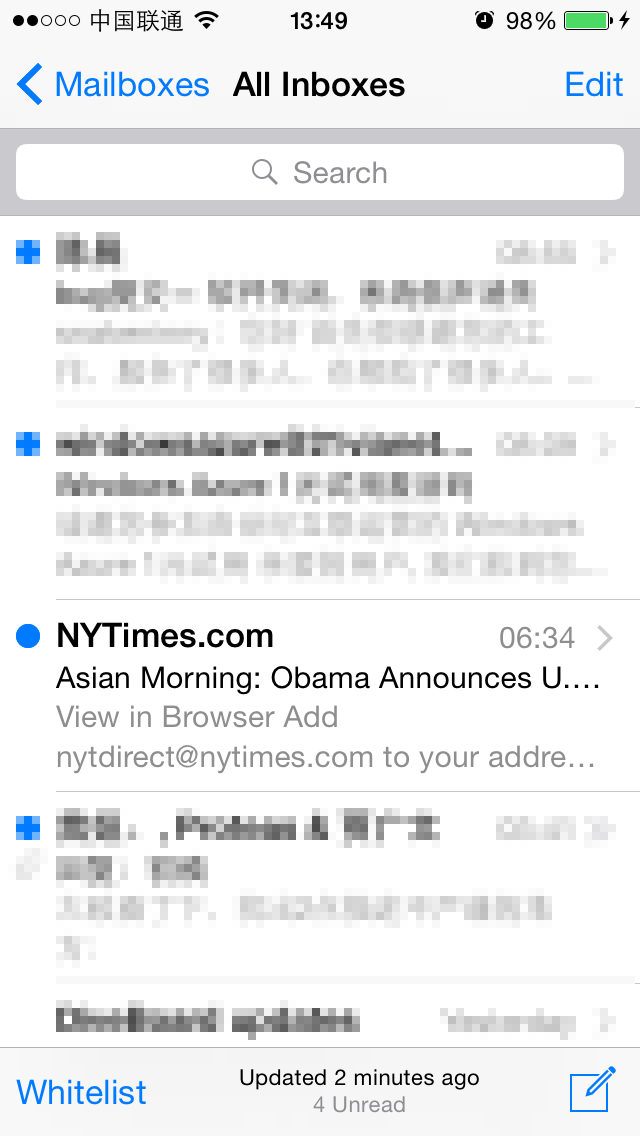


Figure 8- 4 Add whitelist button at the left bottom corner

Although the whitelist button is aligned with the compose button in right bottom corner, the former is text and the latter is an icon. They are in different forms and looks inharmonious. Therefore, we can see the left bottom corner is not suitable for text button. How about changing it to an icon? The problem is that there isn’t an accustomed icon to represent whitelist, while a random one may cause confusion. So in this view, no matter icon or text we use, we cannot get both understandability and harmony. Let’s click “Mailboxes” and go to the upper view, as shown in figure 8-5.

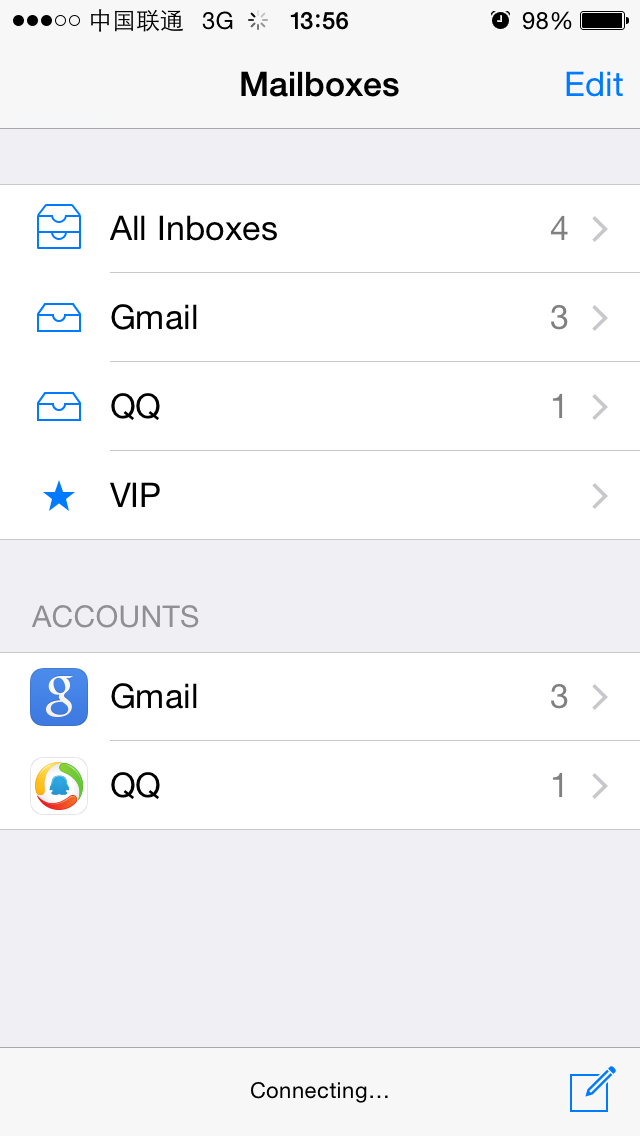


Figure 8- 5 Mailboxes

The top left and bottom left areas are both empty, as shown in figure 8-5. The bottom left is not suitable for the whitelist button as we’ve discussed just now. So let’s put the button on top left corner to see how it looks, as shown in figure 8-6.

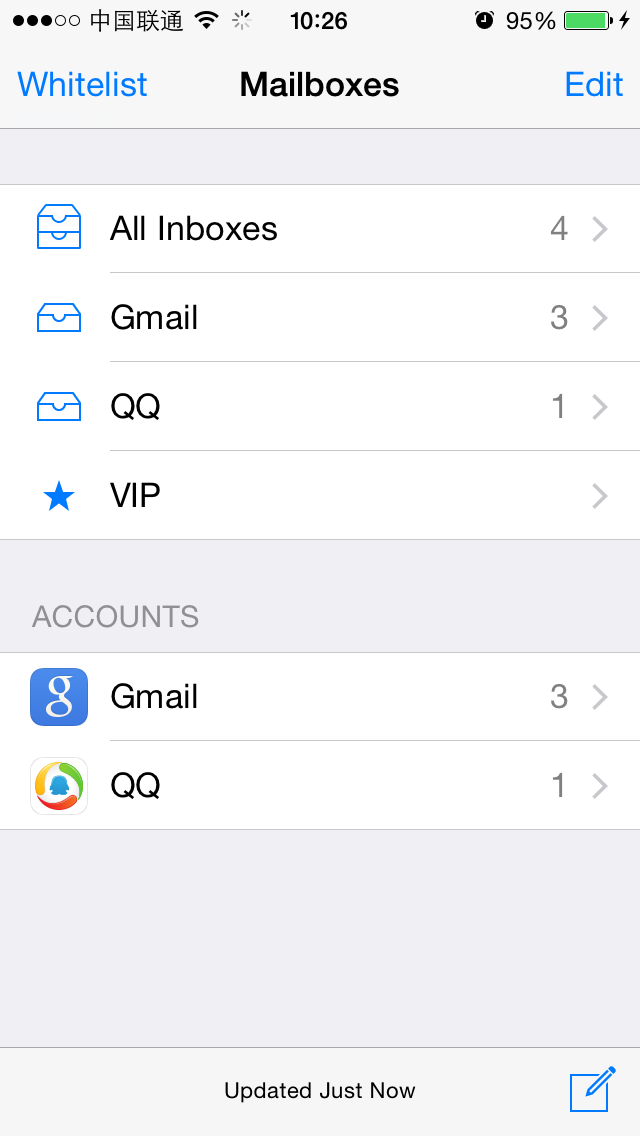


Figure 8- 6 Add whitelist button at top left corner

Not bad, this is it. To customize the view like figure 8-6, we just need to find the controller of “Mailboxes” view and then add the button by calling [controller.navigationItem setLeftBarButtonItem:]. We have repeated the process of finding C from V for many times previously and it has been proved as a feasible solution. After we know how to add the button, we can try to implement the function of whitelist. It can be divided into three steps.

1. Get all emails.
2. Extract their senders’ addresses.
3. Mark them as read according to whitelist.

Let’s analyze them step by step, hope you can still catch up.

* How can we get all emails? As we know, we can pull to refresh the inbox, as shown in figure 8-7.

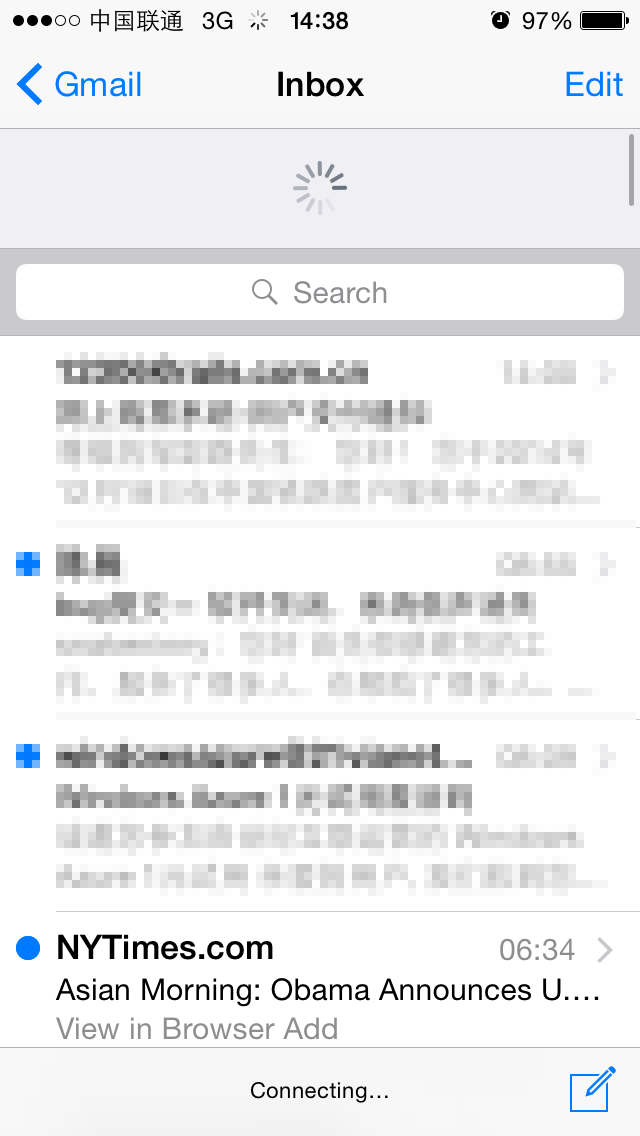


Figure 8-7 Pull to refresh

During refreshing, Mail will fetch all latest emails from mail servers. After refreshing, the UI will restore to the normal state as shown in figure 8-3, and at this moment, we’ve got all emails. As long as we can catch the refresh completion event and read the inbox after that, we can get all emails. Therefore, we can divide “getting all emails” into 2 steps: first, try to capture the refresh completion event; second, read the inbox. Normally, the refresh completion event handler would be a callback method in some protocols. So when analyzing the class-dump headers, we should pay attention to whether there are protocol methods with keywords like “didRefresh”, “didUpdate” or “didReload” in their names. By hooking such methods and read the inbox after their execution, we’ll be able to get all emails.

* An email is an object and it is generally abstracted as a class. From this class, we can extract information like the receiver, sender, title, content and whether it is read. If we can get this object, we can finish the second and third step together.

The overall ideas are not complicated, let’s realize them one by one.

### 8.2.1 Locate and class-dump Mail’s executable

We can easily locate the executable of Mail, “/Applications/MobileMail.app/MobileMail”, using “ps”. Since Mail is a stock App on iOS, it is not encrypted and we can class-dump it directly without decryption:

snakeninnys-MacBook:~ snakeninny$ class-dump -S -s -H /Users/snakeninny/Code/iOSSystemBinaries/8.1.1\_iPhone5/MobileMail.app/MobileMail -o /Users/snakeninny/Code/iOSPrivateHeaders/8.1.1/MobileMail

There’re 393 headers in total, as shown in figure 8-8.

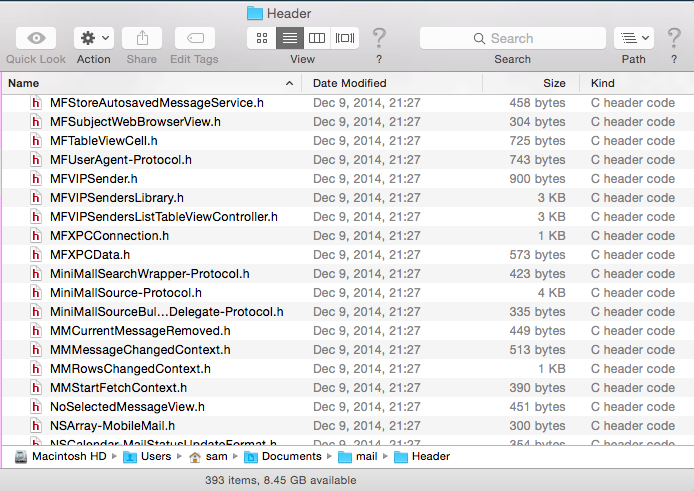


Figure 8- 8 class-dump headers

#### 8.2.2 Import headers into Xcode

The search and code highlighting features in Xcode are competent to present lots of headers, as shown in figure 8-9.

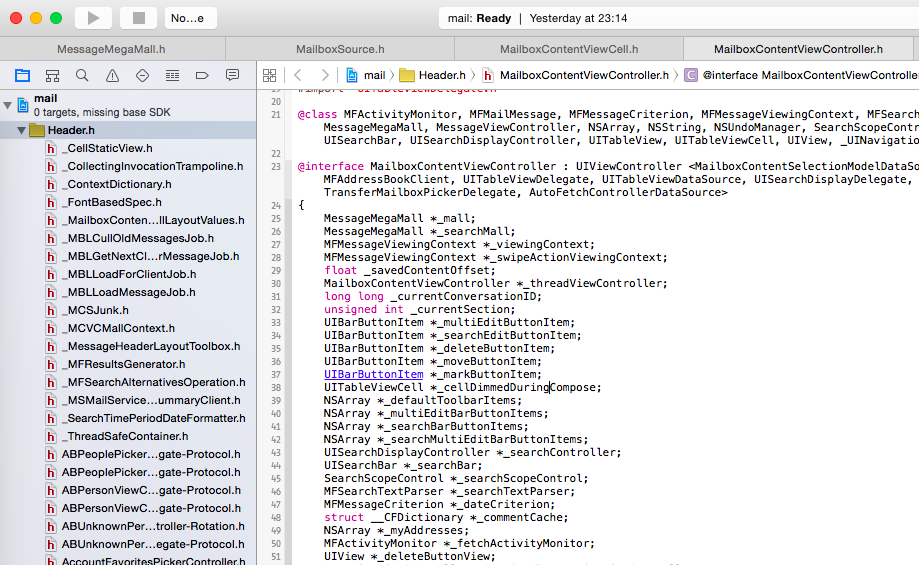


Figure 8- 9 Import headers into Xcode

Next, let’s start to find the point to cut into code from UI.

#### 8.2.3 Find the controller of “Mailboxes” view using Cycript

Firstly, use recursiveDescription to print out the view hierarchy of “Mailboxes” view, as shown below:

FunMaker-5:~ root# cycript -p MobileMail

cy# ?expand

expand == true

cy# [[UIApp keyWindow] recursiveDescription]

@"<UIWindow: 0x156bffe0; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x156bd390>; layer = <UIWindowLayer: 0x156c1be0>>

| <UIView: 0x15611490; frame = (0 0; 320 568); autoresize = W+H; gestureRecognizers = <NSArray: 0x15618e70>; layer = <CALayer: 0x15611420>>

| | <UIView: 0x15611210; frame = (0 0; 320 568); layer = <CALayer: 0x15611280>>

| | | <\_MFActorItemView: 0x15614660; frame = (0 0; 320 568); layer = <CALayer: 0x15614840>>

| | | | <UIView: 0x156150f0; frame = (-0.5 -0.5; 321 569); alpha = 0; layer = <CALayer: 0x15615160>>

| | | | <\_MFActorSnapshotView: 0x15614bb0; baseClass = UISnapshotView; frame = (0 0; 320 568); clipsToBounds = YES; hidden = YES; layer = <CALayer: 0x15614e00>>

| | | | | <UIView: 0x15614f40; frame = (-1 -1; 322 570); layer = <CALayer: 0x15614fb0>>

| | | | <UILayoutContainerView: 0x1572ec40; frame = (0 0; 320 568); clipsToBounds = YES; autoresize = LM+W+RM+TM+H+BM; layer = <CALayer: 0x1572ecc0>>

| | | | | <UIView: 0x1683d890; frame = (0 0; 320 0); layer = <CALayer: 0x16848140>>

| | | | | <UILayoutContainerView: 0x157246b0; frame = (0 0; 320 568); clipsToBounds = YES; gestureRecognizers = <NSArray: 0x156088e0>; layer = <CALayer: 0x15724890>>

……

| | | | | | | | | | <MailboxTableCell: 0x1572ad50; baseClass = UITableViewCell; frame = (0 28; 320 44.5); autoresize = W; layer = <CALayer: 0x168299f0>>

| | | | | | | | | | | <UITableViewCellContentView: 0x16829b70; frame = (0 0; 286 44); gestureRecognizers = <NSArray: 0x1682b060>; layer = <CALayer: 0x16829be0>>

| | | | | | | | | | | | <UILabel: 0x1682b0a0; frame = (55 12; 84.5 20.5); text = 'All Inboxes'; userInteractionEnabled = NO; layer = <\_UILabelLayer: 0x1682b160>>

……

The text of the UILabel at the bottom is “All Inboxes”, indicating its corresponding MailBoxTableCell is the top one in figure 8-5. Keep calling nextResponder until we get the controller:

cy# [#0x1572ad50 nextResponder]

#"<UITableViewWrapperView: 0x1572fe60; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x15730370>; layer = <CALayer: 0x157301a0>; contentOffset: {0, 0}; contentSize: {320, 568}>"

cy# [#0x1572fe60 nextResponder]

#"<UITableView: 0x1585a000; frame = (0 0; 320 568); clipsToBounds = YES; autoresize = W+H; gestureRecognizers = <NSArray: 0x1572fa20>; layer = <CALayer: 0x1572f540>; contentOffset: {0, -64}; contentSize: {320, 371}>"

cy# [#0x1585a000 nextResponder]

#"<MailboxPickerController: 0x156e9260>"

Aha. It’s very easy to get MailboxPickerController. Let’s try whether we can add a leftBarButtonItem:

cy# #0x156e9260.navigationItem.leftBarButtonItem = #0x156e9260.navigationItem.rightBarButtonItem

#"<UIBarButtonItem: 0x15729f00>"

The effect is shown in figure 8-10.

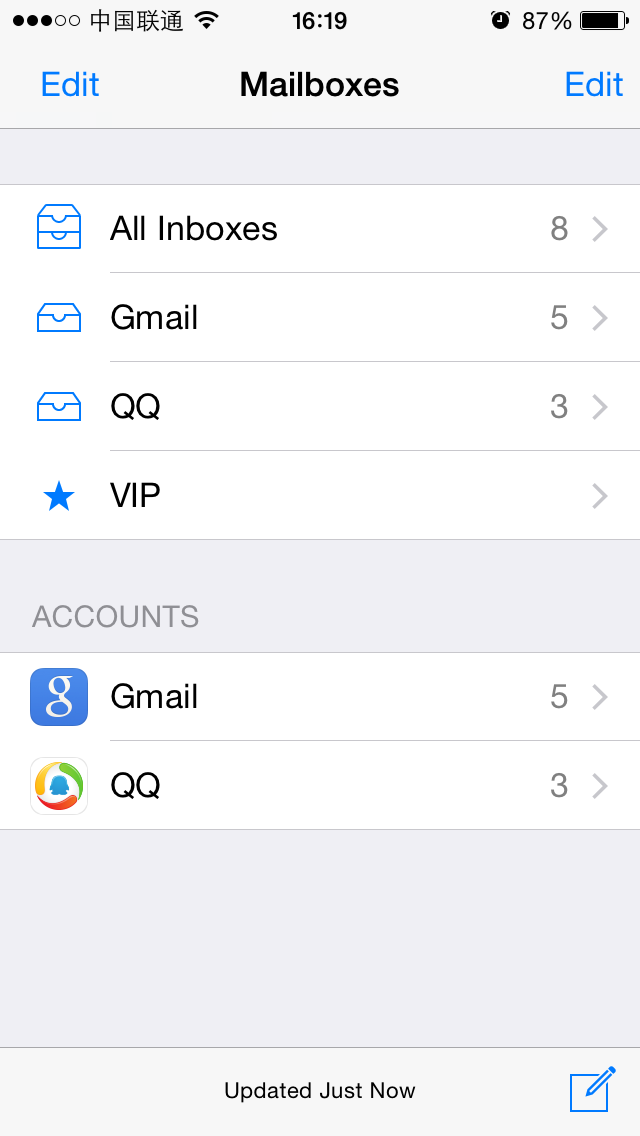


Figure 8- 10 After setLeftBarButtonItem:

No problem! We’ve successfully added the button. Therefore, we can confirm that MailboxPickerController is the controller of “Mailboxes” view.

#### 8.2.4 Find the delegate of “All Inboxes” view using Reveal and Cycript

After adding the whitelist button, we need to implement the function of it. First let’s take a look at how to capture the refresh completion event. Since the event is straightly showed on “All Inboxes” view, it is very likely that the callback method is defined in the delegate of this view. Now let’s turn to “All Inboxes” view in figure 8-3 and use Reveal rather than repeating what we’ve done with Cycript in section 8.2.3, to locate a cell of this view, and then turn back to Cycript to find its associated UITableView as well delegate.

With Reveal, we can easily locate the top cell, as shown in figure 8-11.

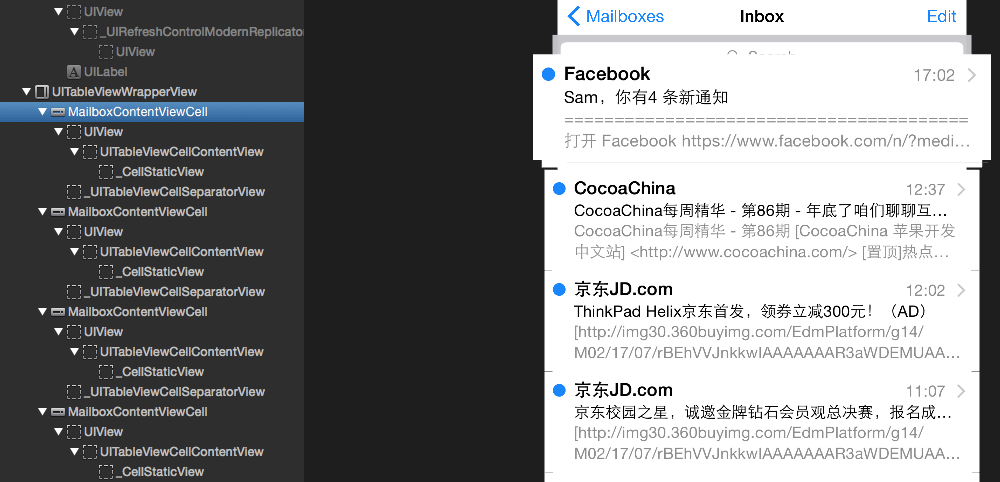


Figure 8- 11 See the view hierarchy using Reveal

MailboxContentViewCell is the cell class to show the sender, title and summary of an email. Next, we use Cycript to find its associated UITableView. Since we know there must be at least one MailboxContentViewCell object in current view, we can try to find these cells through command “choose” without using recursiveDescription.

FunMaker-5:~ root# cycript -p MobileMail

cy# choose(MailboxContentViewCell)

[#"<MailboxContentViewCell: 0x161f4000> cellContent",#"<MailboxContentViewCell: 0x1621c400> cellContent",#"<MailboxContentViewCell: 0x1621d000> cellContent",#"<MailboxContentViewCell: 0x16234800> cellContent",#"<MailboxContentViewCell: 0x1623ee00> cellContent",#"<MailboxContentViewCell: 0x1623f200> cellContent",#"<MailboxContentViewCell: 0x159c2c00> cellContent"]

“choose” has returned an NSArray of MailboxContentViewCell objects. Pick anyone and keep calling nextResponder.

cy# [choose(MailboxContentViewCell)[0] nextResponder]

#"<UITableViewWrapperView: 0x15660b80; frame = (0 0; 320 612); gestureRecognizers = <NSArray: 0x16855170>; layer = <CALayer: 0x16888f20>; contentOffset: {0, 0}; contentSize: {320, 612}>"

cy# [#0x15660b80 nextResponder]

#"<MFMailboxTableView: 0x16095000; baseClass = UITableView; frame = (0 0; 320 568); clipsToBounds = YES; autoresize = W+H; gestureRecognizers = <NSArray: 0x15607850>; layer = <CALayer: 0x16838210>; contentOffset: {0, -64}; contentSize: {320, 52364}>"

Its associated UITableView is an MFMailboxTableView object. Let’s take a look at its delegate.

cy# [#0x16095000 delegate]

#"<MailboxContentViewController: 0x16106000>"

Its delegate is MailboxContentViewController. Keep calling nextResonder and find what its controller is.

cy# [#0x16095000 nextResponder]

#"<MailboxContentViewController: 0x16106000>"

From the output, we can see that both the controller and delegate of MFMailboxTableView are MailboxContentViewController. Let’s validate the controller as below.

cy# [#0x16106000 setTitle:@"iOSRE"]

The effect is shown in 8-12.

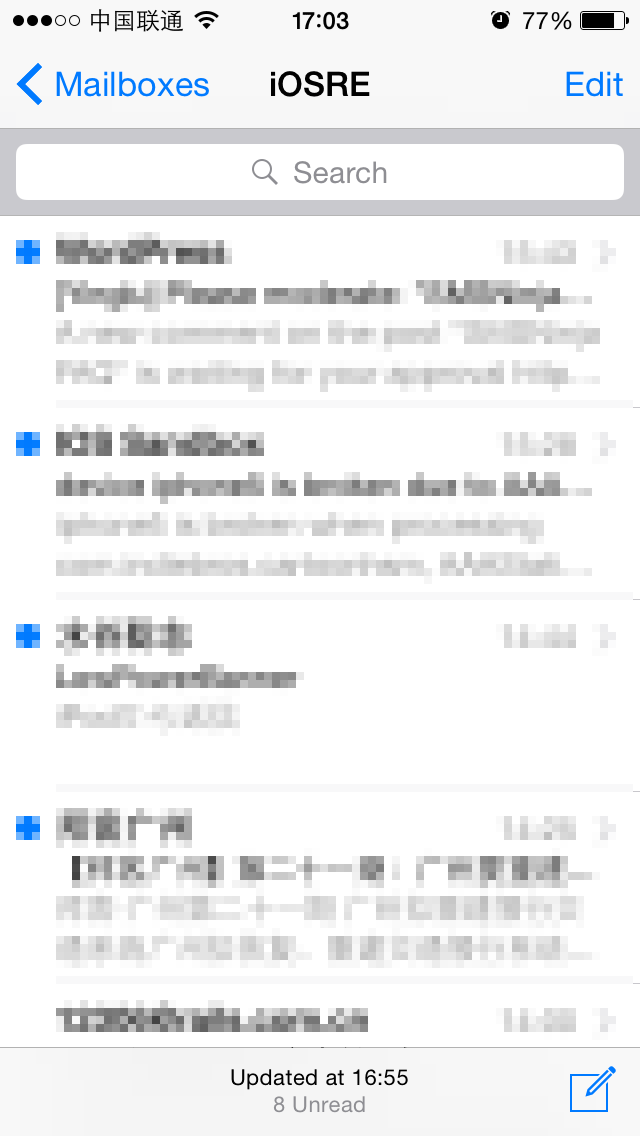


Figure 8- 12 After setTitle:

So far, we can confirm that our deduction is correct. Playing 2 important roles at the same time, it is very likely that we can find both the refresh completion event handler and inbox reading method in MailboxContentViewController. Let’s focus on this class from now on.

### 8.2.4 Locate the refresh completion callback method in MailboxContentViewController

Like what we did in Chapter 7, we should take a look at what protocol does MailboxContentViewController confirm to at first and then try to find our target method.

@interface MailboxContentViewController : UIViewController <MailboxContentSelectionModelDataSource, MFSearchTextParserDelegate, MessageMegaMallObserver, MFAddressBookClient, MFMailboxTableViewDelegate, UIPopoverPresentationControllerDelegate, UITableViewDelegate, UITableViewDataSource, UISearchDisplayDelegate, UISearchBarDelegate, TransferMailboxPickerDelegate, AutoFetchControllerDataSource>

We can exclude MFSearchTextParserDelegate, MFAddressBookClient, UIPopoverPresentationControllerDelegate, UITableViewDelegate, UITableViewDataSource, UISearchDisplayDelegate and UISearchBarDelegate just by name, because they seemingly have no relation with refresh completion. The rest protocols, MailboxContentSelectionModelDataSource, MessageMegaMallObserver, MFMailboxTableViewDelegate, TransferMailboxPickerDelegate and AutoFetchControllerDataSource are hard to determine by names. Let’s check them one by one from MailboxContentSelectionModelDataSource.

@protocol MailboxContentSelectionModelDataSource <NSObject>

- (BOOL)selectionModel:(id)arg1 deleteMovesToTrashForTableIndexPath:(id)arg2;

- (void)selectionModel:(id)arg1 getConversationStateAtTableIndexPath:(id)arg2 hasUnread:(char \*)arg3 hasUnflagged:(char \*)arg4;

- (void)selectionModel:(id)arg1 getSourceStateHasUnread:(char \*)arg2 hasUnflagged:(char \*)arg3;

- (id)selectionModel:(id)arg1 indexPathForMessageInfo:(id)arg2;

- (id)selectionModel:(id)arg1 messageInfosAtTableIndexPath:(id)arg2;

- (id)selectionModel:(id)arg1 messagesForMessageInfos:(id)arg2;

- (BOOL)selectionModel:(id)arg1 shouldArchiveByDefaultForTableIndexPath:(id)arg2;

- (id)selectionModel:(id)arg1 sourceForMessageInfo:(id)arg2;

- (BOOL)selectionModel:(id)arg1 supportsArchivingForTableIndexPath:(id)arg2;

- (id)sourcesForSelectionModel:(id)arg1;

@end

It looks like the function of this protocol is to read the data source rather than refresh it. Let’s move on to MessageMegaMallObserver, its contents are as below:

@protocol MessageMegaMallObserver <NSObject>

- (void)megaMallCurrentMessageRemoved:(id)arg1;

- (void)megaMallDidFinishSearch:(id)arg1;

- (void)megaMallDidLoadMessages:(id)arg1;

- (void)megaMallFinishedFetch:(id)arg1;

- (void)megaMallGrowingMailboxesChanged:(id)arg1;

- (void)megaMallMessageCountChanged:(id)arg1;

- (void)megaMallMessagesAtIndexesChanged:(id)arg1;

- (void)megaMallStartFetch:(id)arg1;

@end

There are many perfect tense verbs in the method names. Meanwhile, judging from the name like “LoadMessages”, “FinishedFetch” and “MessageCountChanged”, we guess that they may get called before or after refresh completion. So let’s set breakpoints at the beginning of these three methods using LLDB and pull to refresh the inbox to check if these methods are called. In the first place, attach LLDB to MobileMail and inspect its ASLR offset.

(lldb) image list -o -f

[ 0] 0x000b2000 /private/var/db/stash/\_.lnBgU8/Applications/MobileMail.app/MobileMail(0x00000000000b6000)

[ 1] 0x003b7000 /Library/MobileSubstrate/MobileSubstrate.dylib(0x00000000003b7000)

[ 2] 0x090d1000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1 (12B411)/Symbols/usr/lib/libarchive.2.dylib

[ 3] 0x090c3000 /Users/snakeninny/Library/Developer/Xcode/iOS DeviceSupport/8.1.1 (12B435)/Symbols/System/Library/Frameworks/CloudKit.framework/CloudKit

……

We can see the ASLR offset is 0x000b2000. Then drag and drop MobileMail into IDA and after the initial analysis has been finished, check the base addresses of [MailboxContentViewController megaMallDidLoadMessages:], [MailboxContentViewController megaMallFinishedFetch:] and [MailboxContentViewController megaMallMessageCountChanged:], as shown in figure 8-13, 8-14 and 8-15.

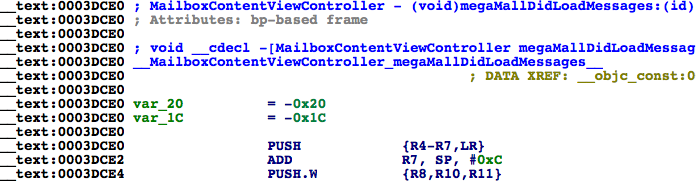


Figure 8- 13 [MailboxContentViewController megaMallDidLoadMessages:]

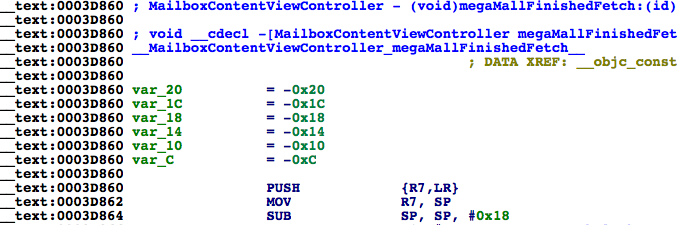


Figure 8- 14 [MailboxContentViewController megaMallFinishedFetch:]

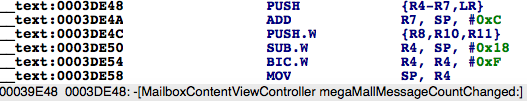


Figure 8- 15 [MailboxContentViewController megaMallMessageCountChanged:]

Their base addresses are 0x3dce0, 0x3d860 and 0x3de48 respectively. Set breakpoints on these addresses with LLDB and refresh the inbox to trigger the breakpoints:

(lldb) br s -a '0x000b2000+0x3dce0'

Breakpoint 1: where = MobileMail`\_\_\_lldb\_unnamed\_function992$$MobileMail, address = 0x000efce0

(lldb) br s -a '0x000b2000+0x3d860'

Breakpoint 2: where = MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail, address = 0x000ef860

(lldb) br s -a '0x000b2000+0x3de48'

Breakpoint 3: where = MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, address = 0x000efe48

Some of you may meet the same problem as me, which is none of three breakpoints get triggered. If you have experience in network development, you may have the idea that in order to reduce the burden of servers and save the network traffic of iOS, Mail may not fetch emails remotely on every refresh. If the time interval between two refreshes is very short, it will use cached content as data source of inbox; and as a result, methods in MessageMegaMallObserver will not get called. In order to validate our assumption, send an email to yourself and refresh to check whether breakpoints get triggered:

Process 73130 stopped

\* thread #44: tid = 0x14c10, 0x000ef860 MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail, stop reason = breakpoint 2.1

frame #0: 0x000ef860 MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail:

-> 0xef860: push {r7, lr}

0xef862: mov r7, sp

0xef864: sub sp, #24

0xef866: movw r1, #44962

(lldb) c

Process 73130 resuming

Process 73130 stopped

\* thread #44: tid = 0x14c10, 0x000ef860 MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail, stop reason = breakpoint 2.1

frame #0: 0x000ef860 MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail:

-> 0xef860: push {r7, lr}

0xef862: mov r7, sp

0xef864: sub sp, #24

0xef866: movw r1, #44962

(lldb) c

Process 73130 resuming

Process 73130 stopped

\* thread #1: tid = 0x11daa, 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90, stop reason = breakpoint 3.1

frame #0: 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail:

-> 0xefe48: push {r4, r5, r6, r7, lr}

0xefe4a: add r7, sp, #12

0xefe4c: push.w {r8, r10, r11}

0xefe50: sub.w r4, sp, #24

(lldb)

Process 73130 resuming

Process 73130 stopped

\* thread #1: tid = 0x11daa, 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90, stop reason = breakpoint 3.1

frame #0: 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail:

-> 0xefe48: push {r4, r5, r6, r7, lr}

0xefe4a: add r7, sp, #12

0xefe4c: push.w {r8, r10, r11}

0xefe50: sub.w r4, sp, #24

(lldb)

Process 73130 resuming

Process 73130 stopped

\* thread #44: tid = 0x14c10, 0x000ef860 MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail, stop reason = breakpoint 2.1

frame #0: 0x000ef860 MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function987$$MobileMail:

-> 0xef860: push {r7, lr}

0xef862: mov r7, sp

0xef864: sub sp, #24

0xef866: movw r1, #44962

(lldb) c

Process 73130 resuming

As expected, megaMallFinishedFetch: and megaMallMessageCountChanged: are called alternately. From their names we can see that an email is a message, megaMallFinishedFetch: will be called when iOS has fetched emails from servers successfully, and megaMallMessageCountChanged: will get called when email count changes, i.e. when we receive or delete emails. These two methods will definitely get called after refreshing; we can choose either one as the refresh completion callback method. We’ll take megaMallMessageCountChanged: in this chapter and our next task is to find the method for getting all emails.

#### 8.2.5 Get all emails from MessageMegaMall

Do you still remember the saying in chapter 7 that “The reason a protocol method gets called is generally that the corresponding event mentioned in the method name happened. And the thing that triggers the event is usually the method’s arguments”? So let’s delete the first two breakpoints and keep the last one on megaMallMessageCountChanged:, and take a look at its argument:

Process 73130 stopped

\* thread #1: tid = 0x11daa, 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90, stop reason = breakpoint 3.1

frame #0: 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail:

-> 0xefe48: push {r4, r5, r6, r7, lr}

0xefe4a: add r7, sp, #12

0xefe4c: push.w {r8, r10, r11}

0xefe50: sub.w r4, sp, #24

(lldb) po $r2

NSConcreteNotification 0x157e8af0 {name = MegaMallMessageCountChanged; object = <MessageMegaMall: 0x1576c320>; userInfo = {

"added-message-infos" = (

"<MFMessageInfo: 0x157c86d0> uid=1185, conversation=2777228998582613276"

);

destination = "{(\n)}";

inserted = "{(\n <NSIndexPath: 0x157e8ac0> {length = 2, path = 0 - 0}\n)}";

relocated = "{(\n)}";

updated = "{(\n)}";

}}

We can see that the argument is an NSConcreteNotification object. Checking its header file, we can learn that it inherits from NSNotification. Its name is MegaMallMessageCountChanged, object is MessageMegaMall and userInfo is its changelog. The thing that interests us is the name “MegaMall”, which seemingly has nothing to do with emails but is always next to “Message”, so I guess it’s a mega mall for emails instead of merchandises. Let’s see what’s in MessageMegaMall.h:

@interface MessageMegaMall : NSObject <MessageMiniMallObserver, MessageSelectionDataSource>

……

- (id)copyAllMessages;

@property (retain, nonatomic) MFMailMessage \*currentMessage;

- (void)loadOlderMessages;

- (unsigned int)localMessageCount;

- (unsigned int)messageCount;

- (void)markAllMessagesAsNotViewed;

- (void)markAllMessagesAsViewed;

- (void)markMessagesAsNotViewed:(id)arg1;

- (void)markMessagesAsViewed:(id)arg1;

……

@end

We’ve got some new clues: copyAllMessages, currentMessage, loadOlderMessages, localMessageCount, messageCount, markAllMessagesAsViewed, etc. From these methods and properties, we can confirm that MessageMegaMall is a model class in charge of all emails; a mega mall is a vivid analogy from Apple for its responsibility. So, can we get all emails with copyAllMessages? Let’s try it out in LLDB:

Process 73130 stopped

\* thread #1: tid = 0x11daa, 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90, stop reason = breakpoint 3.1

frame #0: 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail:

-> 0xefe48: push {r4, r5, r6, r7, lr}

0xefe4a: add r7, sp, #12

0xefe4c: push.w {r8, r10, r11}

0xefe50: sub.w r4, sp, #24

(lldb) po [[$r2 object] copyAllMessages]

{(

<MFLibraryMessage 0x15612030: library id 89, remote id 13020, 2014-11-25 20:32:16 +0000, 'Cydia/APT(A): LowPowerBanner (1.4.5)'>,

<MFLibraryMessage 0x1572ef10: library id 604, remote id 12718, 2014-10-01 21:34:28 +0000, 'Asian Morning: Told to End Protests, Organizers in Hong Kong Vow to Expand Them'>,

<MFLibraryMessage 0x168bd170: library id 906, remote id 13142, 2014-12-17 22:34:30 +0000, 'Asian Morning: Obama Announces U.S. and Cuba Will Resume Relations'>,

……

)}

(lldb) p (int)[[[$r2 object] copyAllMessages] count]

(int) $7 = 580

(lldb) p (int)[[$r2 object] localMessageCount]

(int) $8 = 580

(lldb) p (int)[[$r2 object] messageCount]

(int) $0 = 553

(lldb) po [[[$r2 object] copyAllMessages] class]

\_\_NSSetM

copyAllMessages has returned an NSSet with 580 MFLibraryMessage objects. There is an email summary in each MFLibraryMessage object and the count of this NSSet is the same to localMessageCount. Actually, 580 is far less than all email count, but this number is reasonable that to save network traffic and local storage, iOS doesn’t have to really fetch all emails and store them locally, several hundreds of emails would be enough. If users want to read more, iOS will fetch more with loadOlderMessages. Therefore, copyAllMessages can be considered the right method for getting all emails. Aha, we have achieved our 2nd goal. For the 3rd goal, we should pay attention to [MessageMegaMall markMessagesAsViewed:]. If nothing goes wrong, this is the method for marking emails as read and its argument seems to be an NSArray or NSSet with MFLibraryMessage objects. Is that so? We’ll see shortly.

### 8.2.6 Get sender address from MFLibraryMessage and mark email as read using MessageMegaMall

From the analysis in section 8.2.4, we can see that an email is an MFLibraryMessage object, whose description contains the summary of that email. However, you can’t find MFLibraryMessage.h in MobileMail headers. Why? Because MFLibraryMessage originates from an external dylib. Search “MFLibraryMessage” in iOS 8 class-dump headers, you will find it in Messages.framework, as shown in figure 8-16.

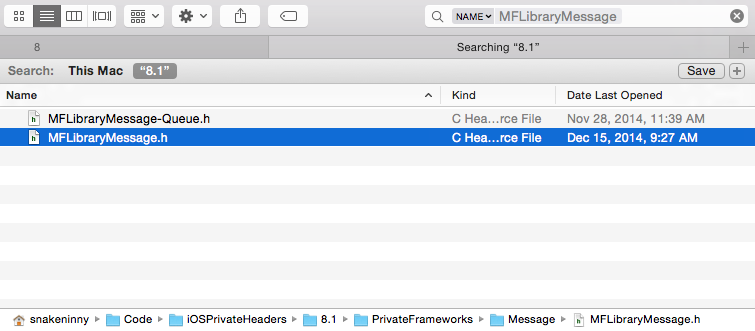


Figure 8- 16 Find MFLibraryMessage

Take a look at MFLibraryMessage.h:

@interface MFLibraryMessage : MFMailMessage

……

- (id)copyMessageInfo;

……

- (void)markAsNotViewed;

- (void)markAsViewed;

- (id)account;

……

- (unsigned long long)uniqueRemoteId;

- (unsigned long)uid;

- (unsigned int)hash;

- (id)remoteID;

- (void)\_updateUID;

- (unsigned int)messageSize;

- (id)originalMailboxURL;

- (unsigned int)originalMailboxID;

- (unsigned int)mailboxID;

- (unsigned int)libraryID;

- (id)persistentID;

- (id)messageID;

@end

In MFLibraryMessage.h, there are various IDs, but our target information seems to be missing. This doesn’t make sense: we have already found the email summary in the description of MFLibraryMessage, but haven’t found the corresponding methods to read the summary in MFLibraryMessage.h. Therefore, something must be ignored in our analysis. Recheck MFLibraryMessage.h, we notice that there is a method called copyMessageInfo. Let’s take a look at it.

Process 73130 stopped

\* thread #1: tid = 0x11daa, 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90, stop reason = breakpoint 3.1

frame #0: 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail:

-> 0xefe48: push {r4, r5, r6, r7, lr}

0xefe4a: add r7, sp, #12

0xefe4c: push.w {r8, r10, r11}

0xefe50: sub.w r4, sp, #24

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] copyMessageInfo]

<MFMessageInfo: 0x157c8040> uid=89, conversation=594030790676622907

We’ve got an object of MFMessageInfo, which has been mentioned in section 8.2.5. Is email summary in MFMessageInfo.h? Let’s try it.

@interface MFMessageInfo : NSObject

{

unsigned int \_flagged:1;

unsigned int \_read:1;

unsigned int \_deleted:1;

unsigned int \_uidIsLibraryID:1;

unsigned int \_hasAttachments:1;

unsigned int \_isVIP:1;

unsigned int \_uid;

unsigned int \_dateReceivedInterval;

unsigned int \_dateSentInterval;

unsigned int \_mailboxID;

long long \_conversationHash;

long long \_generationNumber;

}

+ (long long)newGenerationNumber;

@property(readonly, nonatomic) long long generationNumber; // @synthesize generationNumber=\_generationNumber;

@property(nonatomic) unsigned int mailboxID; // @synthesize mailboxID=\_mailboxID;

@property(nonatomic) long long conversationHash; // @synthesize conversationHash=\_conversationHash;

@property(nonatomic) unsigned int dateSentInterval; // @synthesize dateSentInterval=\_dateSentInterval;

@property(nonatomic) unsigned int dateReceivedInterval; // @synthesize dateReceivedInterval=\_dateReceivedInterval;

@property(nonatomic) unsigned int uid; // @synthesize uid=\_uid;

- (id)description;

- (unsigned int)hash;

- (BOOL)isEqual:(id)arg1;

- (int)generationCompare:(id)arg1;

- (id)initWithUid:(unsigned int)arg1 mailboxID:(unsigned int)arg2 dateReceivedInterval:(unsigned int)arg3 dateSentInterval:(unsigned int)arg4 conversationHash:(long long)arg5 read:(BOOL)arg6 knownToHaveAttachments:(BOOL)arg7 flagged:(BOOL)arg8 isVIP:(BOOL)arg9;

- (id)init;

@property(nonatomic) BOOL isVIP;

@property(nonatomic, getter=isKnownToHaveAttachments) BOOL knownToHaveAttachments;

@property(nonatomic) BOOL uidIsLibraryID;

@property(nonatomic) BOOL deleted;

@property(nonatomic) BOOL flagged;

@property(nonatomic) BOOL read;

@end

MFMessageInfo can tell if an email is read, but it still doesn’t contain the summary. Go back to MFLibraryMessage.h again, we see it inherits from MFMailMessage. Judging from its name, MailMessage is certainly more appropriate to represent an email than LibraryMessage. Take a look at MFMailMessage.h:

@interface MFMailMessage : MFMessage

……

- (BOOL)shouldSetSummary;

- (void)setSummary:(id)arg1;

- (void)setSubject:(id)arg1 to:(id)arg2 cc:(id)arg3 bcc:(id)arg4 sender:(id)arg5 dateReceived:(double)arg6 dateSent:(double)arg7 messageIDHash:(long long)arg8 conversationIDHash:(long long)arg9 summary:(id)arg10 withOptions:(unsigned int)arg11;

- (id)subject;

@end

summary, subject, sender, cc, bcc and some other frequently used phrases in emails come into our eyes. However, except subject, most of them are only setters, where are the getters? If you still remember how we’ve shifted our attention from MFLibraryMessage.h to MFMailMessage.h, you will notice that MFMailMessage inherits from MFMessage. Before inspecting MFMessage.h, let’s take a look at the return value of [MFMailMessage subject] through LLDB to verify the analysis by now.

Process 73130 stopped

\* thread #1: tid = 0x11daa, 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90, stop reason = breakpoint 3.1

frame #0: 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail:

-> 0xefe48: push {r4, r5, r6, r7, lr}

0xefe4a: add r7, sp, #12

0xefe4c: push.w {r8, r10, r11}

0xefe50: sub.w r4, sp, #24

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] subject]

Asian Morning: Told to End Protests, Organizers in Hong Kong Vow to Expand Them

We can see that the return value of [MFMailMessage subject] is exactly the email title. Take a look at MFMessage.h (Attention, MFMessage is a class in MIME.framework).

@interface MFMessage : NSObject <NSCopying>

……

- (id)headerData;

- (id)bodyData;

- (id)summary;

- (id)bccIfCached;

- (id)bcc;

- (id)ccIfCached;

- (id)cc;

- (id)toIfCached;

- (id)to;

- (id)firstSender;

- (id)sendersIfCached;

- (id)senders;

- (id)dateSent;

- (id)subject;

- (id)messageData;

- (id)messageBody;

- (id)headers;

……

@end

to, sender, subject, messageBody, getters for all email information are available now. It’s time to check their values with LLDB.

Process 73130 stopped

\* thread #1: tid = 0x11daa, 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail, queue = 'MessageMiniMall.0x157c2d90, stop reason = breakpoint 3.1

frame #0: 0x000efe48 MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function993$$MobileMail:

-> 0xefe48: push {r4, r5, r6, r7, lr}

0xefe4a: add r7, sp, #12

0xefe4c: push.w {r8, r10, r11}

0xefe50: sub.w r4, sp, #24

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] firstSender]

NYTimes.com <nytdirect@nytimes.com>

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] sendersIfCached]

<\_\_NSArrayI 0x16850850>(

NYTimes.com <nytdirect@nytimes.com>

)

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] senders]

<\_\_NSArrayI 0x16850850>(

NYTimes.com <nytdirect@nytimes.com>

)

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] to]

<\_\_NSArrayI 0x16850840>(

snakeninny@gmail.com

)

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] dateSent]

2014-10-01 21:30:32 +0000

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] subject]

Asian Morning: Told to End Protests, Organizers in Hong Kong Vow to Expand Them

(lldb) po [[[[$r2 object] copyAllMessages] anyObject] messageBody]

<MFMimeBody: 0x16852fc0>

Everything is too distinct to explain. firstSender returns a single sender, while sendersIfCached and senders both return an NSArray, which means on iOS, there could be multiple senders in an email. Although this situation is quite rare (at least for me, I have never seen multiple senders), to avoid missing any sender, I’ll still use “senders” to get all possible senders. The final task is to mark messages as read; do you still remember [MessageMegaMall markMessagesAsViewed:] in section 8.2.5? Is it the right method for marking messages as read? Let’s set a breakpoint on this method and check whether it will be called when we mark an email as read.

At first, we need to locate [MessageMegaMall markMessagesAsViewed:] in IDA and check its base address, as shown in figure 8-17.

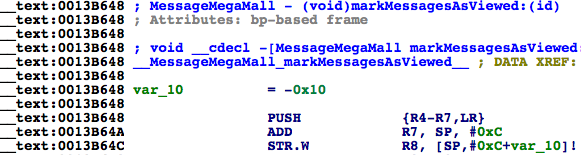


Figure 8- 17 [MessageMegaMall markMessagesAsViewed:]

Its base address is 0x13b648. Since the ASLR offset of MobileMail is 0xb2000, we can set a breakpoint like this:

(lldb) br s -a '0x000b2000+0x0013B648'

Breakpoint 4: where = MobileMail`\_\_\_lldb\_unnamed\_function7357$$MobileMail, address = 0x001ed648

Process 103910 stopped

\* thread #1: tid = 0x195e6, 0x001ed648 MobileMail`\_\_\_lldb\_unnamed\_function7357$$MobileMail, queue = 'com.apple.main-thread, stop reason = breakpoint 4.1

frame #0: 0x001df648 MobileMail`\_\_\_lldb\_unnamed\_function7357$$MobileMail

MobileMail`\_\_\_lldb\_unnamed\_function7357$$MobileMail:

-> 0x1ed648: push {r4, r5, r6, r7, lr}

0x1ed64a: add r7, sp, #12

0x1ed64c: str r8, [sp, #-4]!

0x1ed650: mov r8, r0

(lldb) po $r2

{(

<MFLibraryMessage 0x157b70b0: library id 906, remote id 13142, 2014-12-17 22:34:30 +0000, 'Asian Morning: Obama Announces U.S. and Cuba Will Resume Relations'>

)}

(lldb) po [$r2 class]

\_\_NSSetI

The output of LLDB validates our assumption. [MessageMegaMall markMessagesAsViewed:] is the right method for marking messages as read and its argument is an NSSet of MFLibraryMessage objects. Till now, we have successfully added the whitelist button, captured the refresh completion event, got all emails and their senders, as well marked them as read. Tweak prototyping comes to an end; let’s comb our thoughts before writing code.

## 8.3 Result interpretation

The practice in this chapter is highly modularized; every part in Mail has a clear division of work, which speeds up our tweak prototyping.

#### Find the place and method for adding whitelist button

Sticking to the pursuit of both understandability and harmony, we have tried several solutions and finally decided to put the whitelist button at the top left corner of “Mailboxes” view. We are all familiar with the pattern to get MailboxPickerController with Cycript, so there is no difficulty for us to add a button on its navigation bar.

#### Find the refresh completion callback methods in protocol

Again in this chapter, we’ve used the protocols in MailboxContentViewController.h as clues, walked through all corresponding headers and guessed the keywords, then finally found the refresh completion callback methods, just like what we’ve done in “find a method to monitor note text changes in real time”, chapter 7. After testing, megaMallMessageCountChanged: is called when email count changes, thus meet our requirements.

#### Get all emails from MessageMegaMall.

According to the experience that “The reason a protocol method gets called is generally that the corresponding event mentioned in the method name happened. And the thing that triggers the event is usually the method’s arguments”, we’ve found class MessageMegaMall from the argument of megaMallMessageCountChanged:. The name, MegaMall, is very obscure. With wild guesses and programmatic checks, we’ve discovered that it is the model for email managements. By calling [MessageMegaMall copyAllMessages], we can get all emails.

#### Get the sender’s address from MFLibraryMessage

[MessageMegaMall copyAllMessages] returns an array of MFLibraryMessage objects. By inspecting MFLibraryMessage.h and related headers, as well testing some suspicious properties and methods, we can easily get the sender’s addresses from this class.

#### Mark emails as read with MessageMegaMall

When we were studying MessageMegaMall.h, we have noticed the uncertain target method, markMessagesAsViewed:. We could even say for sure it was what we were looking for without any test. Of course, the result from LLDB proved our conclusion directly.

Notice: In order to simplify the tweak, the whitelist in section 8.4 consists of only one single email address, and it’s presented as a UIAlertView. As an exercise, it’s your turn to extend it with more addresses and use a UITableView to present it, make this tweak more useful.

## 8.4 Tweak writing

All difficulties have been overcome during the stage of prototyping. Now we just need to follow the conclusion we get in section 8.3 and write the tweak with elegant code.

### Create tweak project “iOSREMailMarker” using Theos

The Theos commands are as follows:

hangcom-mba:Documents sam$ /opt/theos/bin/nic.pl

NIC 2.0 - New Instance Creator

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

[1.] iphone/application

[2.] iphone/cydget

[3.] iphone/framework

[4.] iphone/library

[5.] iphone/notification\_center\_widget

[6.] iphone/preference\_bundle

[7.] iphone/sbsettingstoggle

[8.] iphone/tool

[9.] iphone/tweak

[10.] iphone/xpc\_service

Choose a Template (required): 9

Project Name (required): iOSREMailMarker

Package Name [com.yourcompany.iosremailmarker]: com.iosre.mailmarker

Author/Maintainer Name [sam]: sam

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

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

Instantiating iphone/tweak in iosremailmarker/...

Done.

### 2. Compose iOSREMailMarker.h

The finalized iOSREMailMarker.h looks like this:

@interface MailboxPickerController : UITableViewController

@end

@interface NSConcreteNotification : NSNotification

@end

@interface MessageMegaMall : NSObject

- (void)markMessagesAsViewed:(NSSet \*)arg1;

- (NSSet \*)copyAllMessages;

@end

@interface MFMessageInfo : NSObject

@property (nonatomic) BOOL read;

@end

@interface MFLibraryMessage : NSObject

- (NSArray \*)senders;

- (MFMessageInfo \*)copyMessageInfo;

@end

This header is composed by picking snippets from other class-dump headers. The existence of this header is simply for avoiding any warnings or errors when compiling the tweak.

### 8.4.3 Edit Tweak.xm

The finalized Tweak.xm looks like this:

#import "iOSREMailMarker.h"

%hook MailboxPickerController

%new

- (void)iOSREShowWhitelist

{

UIAlertController \*alertController = [UIAlertController alertControllerWithTitle:@"Whitelist" message:@"Please input an email address" preferredStyle:UIAlertControllerStyleAlert];

UIAlertAction \*okAction = [UIAlertAction actionWithTitle:@"OK" style:UIAlertActionStyleDefault handler:^(UIAlertAction \* action) {

UITextField \*whitelistField = alertController.textFields.firstObject;

if ([whitelistField.text length] != 0) [[NSUserDefaults standardUserDefaults] setObject:whitelistField.text forKey:@"whitelist"];

}];

UIAlertAction \*cancelAction = [UIAlertAction actionWithTitle:@"Cancel" style:UIAlertActionStyleCancel handler:nil];

[alertController addAction:okAction];

[alertController addAction:cancelAction];

[alertController addTextFieldWithConfigurationHandler:^(UITextField \*textField) {

textField.placeholder = @"snakeninny@gmail.com";

textField.text = [[NSUserDefaults standardUserDefaults] objectForKey:@"whitelist"];

}];

[self presentViewController:alertController animated:YES completion:nil];

}

- (void)viewWillAppear:(BOOL)arg1

{

self.navigationItem.leftBarButtonItem = [[[UIBarButtonItem alloc] initWithTitle:@"Whitelist" style:UIBarButtonItemStylePlain target:self action:@selector(iOSREShowWhitelist)] autorelease];

%orig;

}

%end

%hook MailboxContentViewController

- (void)megaMallMessageCountChanged:(NSConcreteNotification \*)arg1

{

%orig;

NSMutableSet \*targetMessages = [NSMutableSet setWithCapacity:600];

NSString \*whitelist = [[NSUserDefaults standardUserDefaults] objectForKey:@"whitelist"];

MessageMegaMall \*mall = [arg1 object];

NSSet \*messages = [mall copyAllMessages]; // Remember to release it later

for (MFLibraryMessage \*message in messages)

{

MFMessageInfo \*messageInfo = [message copyMessageInfo]; // Remember to release it later

for (NSString \*sender in [message senders]) if (!messageInfo.read && [sender rangeOfString:[NSString stringWithFormat:@"<%@>", whitelist]].location == NSNotFound) [targetMessages addObject:message];

[messageInfo release];

}

[messages release];

[mall markMessagesAsViewed:targetMessages];

}

%end

### 8.4.4. Edit Makefile and control files

The finalized Makefile looks like this:

THEOS\_DEVICE\_IP = iOSIP

ARCHS = armv7 arm64

TARGET = iphone:latest:8.0

include theos/makefiles/common.mk

TWEAK\_NAME = iOSREMailMarker

iOSREMailMarker\_FILES = Tweak.xm

iOSREMailMarker\_FRAMEWORKS = UIKit

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

after-install::

install.exec "killall -9 MobileMail"

The finalized control looks like this:

Package: com.iosre.mailmarker

Name: iOSREMailMarker

Depends: mobilesubstrate, firmware (>= 8.0)

Version: 1.0

Architecture: iphoneos-arm

Description: Mark non-whitelist emails as read!

Maintainer: sam

Author: sam

Section: Tweaks

Homepage: <http://bbs.iosre.com>

### 8.4.5 Test

Compile the tweak and install it on iOS. Open Mail but it seems nothing changed. That is because we haven’t configured iOSREMailMarker yet. As shown in figure 8-18, there are 44 unread messages currently.

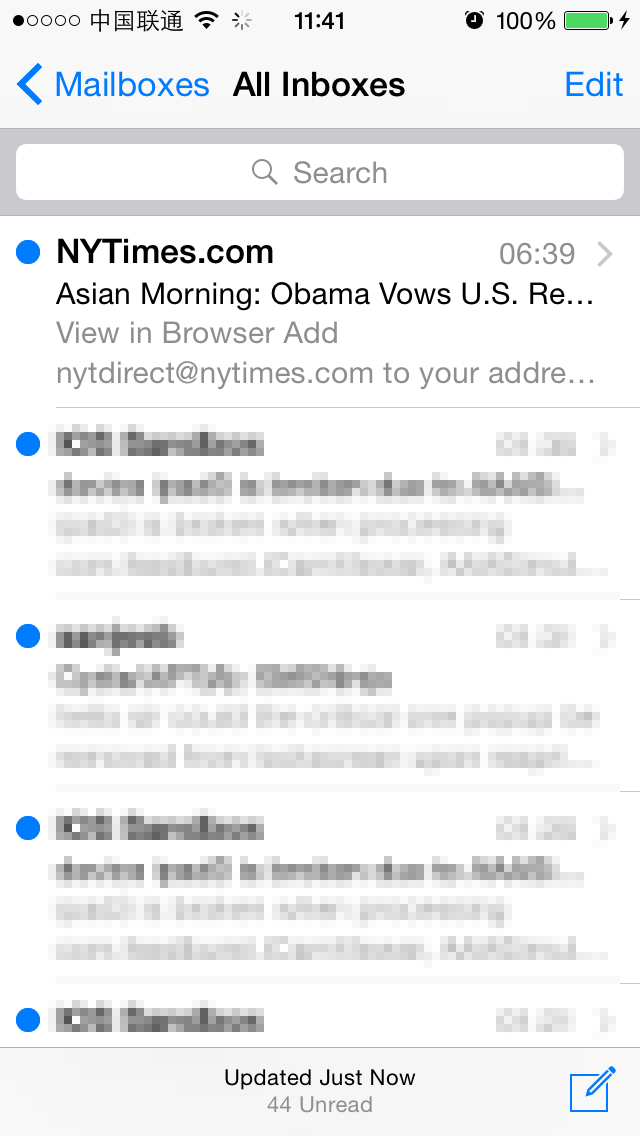


Figure 8- 18 44 unread emails

After entering the “Mailboxes” view, there is a new whitelist button on the left side of navigation bar. Press it and a new whitelist dialog will pop up, as shown in 8-19.

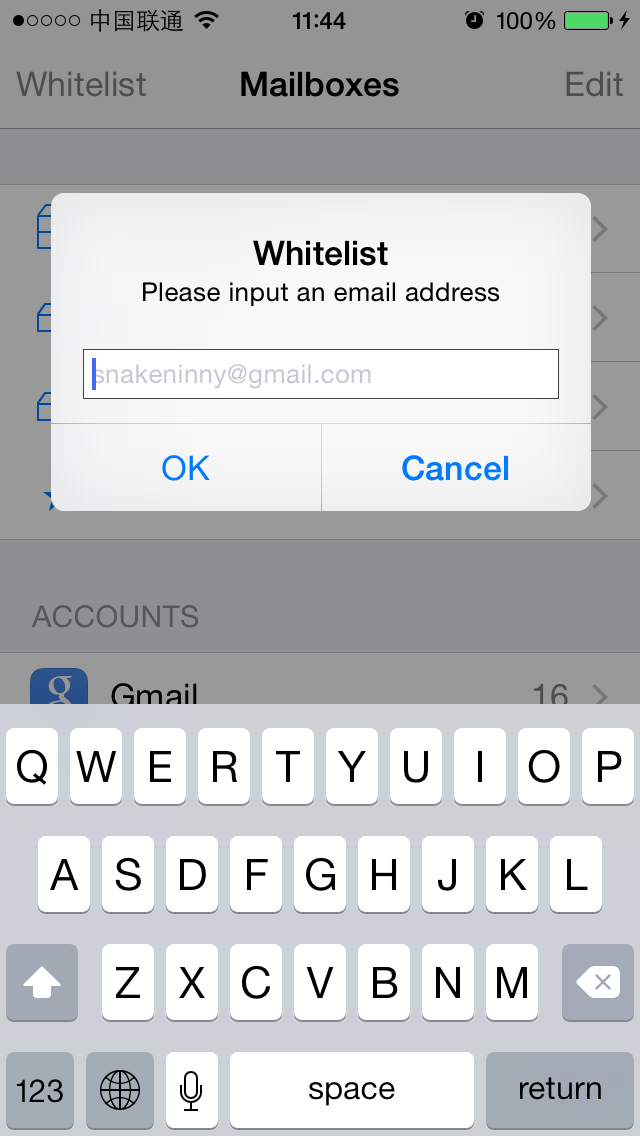


Figure 8- 19 Whitelist dialog

I’ve subscribed a copy of NYTimes and spend about 15 minutes on it every day. Let’s add NYTimes into whitelist, as shown in figure 8-20.

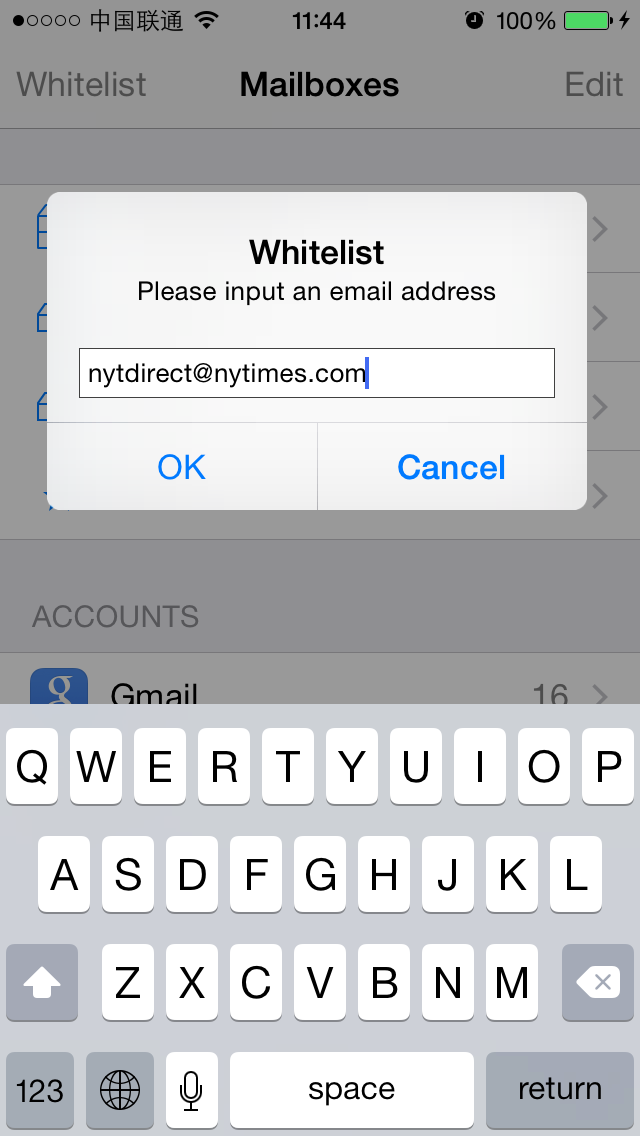


Figure 8- 20 Add NYTimes into whitelist

At last, send an email to myself to trigger megaMallMessageCountChanged:. After receiving the email, all emails except NYTimes are marked as read, as shown in 8-21.

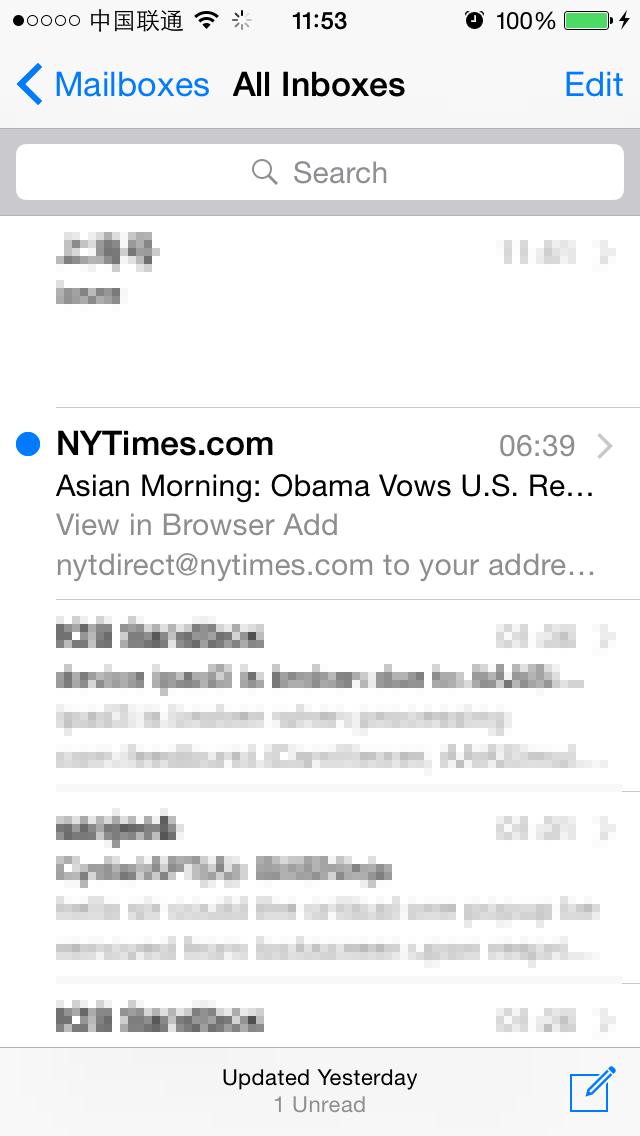


Figure 8-21 iOSREMailMarker marked all emails except NYTimes as read

So far, we have achieved all of our goals successfully.

## 8.5 Conclusion

In this chapter, we’ve taken Mail as an example and added a feature that can automatically mark emails outside whitelist as read, which helps us highlight the important emails. The filter condition of iOSREMailMarker is somewhat simple, and it may not be a good solution for everyone to simply mark emails as read. So I hope you can learn this chapter by analogy and intimidate the ideas to make your own unique tweaks. Of course, you are welcome to share your works on our website.

So far, we have gone through 2 practices. I hope everyone enjoyed them and had the feeling that our brains should keep one step ahead of our hands in iOS reverse engineering. Only when you get fully prepared during early stage analysis can you write an excellent tweak later. TiGa, a veteran reverse engineer, once said: “A reverser should know how/what is done before grabbing tools to complete the tasks automatically.” I believe that everyone will gradually realize the meaning of this sentence during continuously studying reverse engineering.