Damn Vulnerable IOS Application Solutions
http://damnvulnerableiosapp.com/

Application Patching – Login Method 1

In this challenge, we will be patching the binary on our system. This is just to demonstrate that you can learn application patching without the need of an IOS device. Make sure you have the latest source code of DVIA and run the application in Xcode. This will install the application on the IOS simulator and an application sandbox folder will be created at the following location as shown below.

/Users/$username/Library/Application Support/iPhone Simulator/$ios version of simulator/Applications/

In my case, the location is

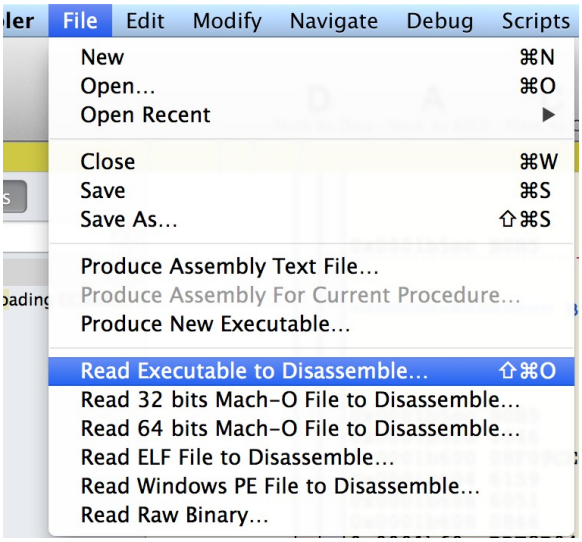
/Users/Prateek/Library/Application\ Support/iPhone\ Simulator/7.0.3/Applications/

Once you are in this directory, you have to find your application folder. Using the command **ls -al** will give you the last modified date of these folders. The latest one would be our application.

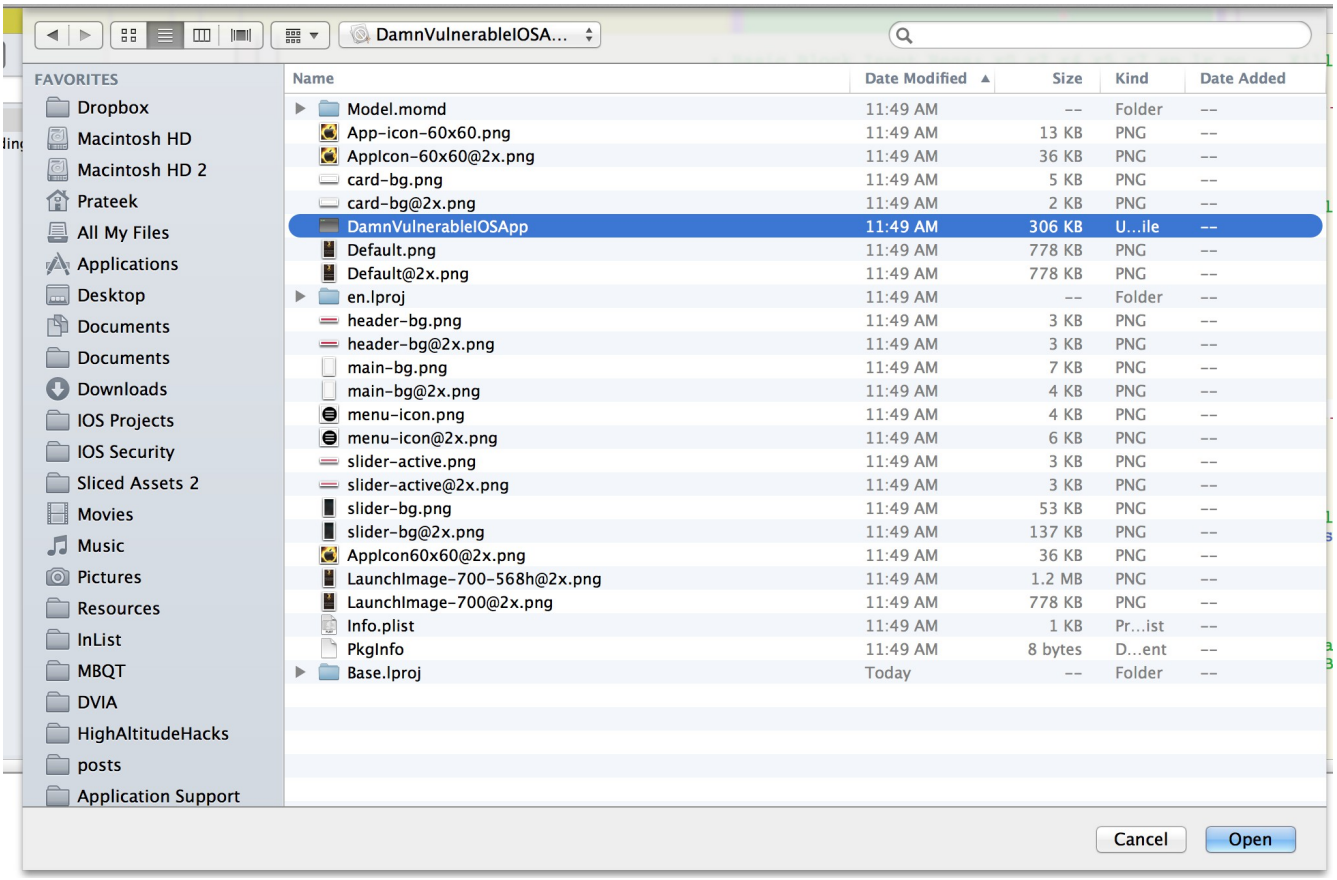
```
Prateeks-MacBook-Pro-2:254BFC17-66AC-4CDD-AC74-28E23BAAEB21 Prateek$ cd ~
Prateeks-MacBook-Pro-2:~ Prateek$ cd /Users/Prateek/Library/Application\ Support/iPhone\ Simulator/7.0.3/Applications/
Prateeks-MacBook-Pro-2:Applications Prateek$ ls -al
total 48
drwxr-xr-x  14 Prateek  staff    476 Jan 29 10:26 .
drwxr-xr-x  11 Prateek  staff    374 Jan 28 17:39 ..
-rw-r--r--@  1 Prateek  staff  21508 Jan 29 10:26 .DS_Store
drwxr-xr-x@  7 Prateek  staff    238 Jan 12 14:26 02233F84-8E14-41B8-81D2-A92DF6B2F6D2
drwxr-xr-x@  7 Prateek  staff    238 Jan 29 10:23 0B373879-8756-4C9E-A015-D06B808DC051
drwxr-xr-x@  7 Prateek  staff    238 Jan 29 10:26 254BFC17-66AC-4CDD-AC74-28E23BAAEB21
drwxr-xr-x@  7 Prateek  staff    238 Jan 14 21:55 2BF7B95D-F12F-4B61-AAB5-572A912531E3
drwxr-xr-x@  6 Prateek  staff    204 Nov 13 13:11 6AAF2D9E-4385-40E1-B77F-E7A28EF20AC3
drwxr-xr-x@  7 Prateek  staff    238 Jan 14 22:27 878F97C4-B756-43E1-8E61-1A2CD9E7818B
drwxr-xr-x@  6 Prateek  staff    204 Nov 13 13:11 96937D6F-2C4F-4748-9C11-A27E553191DF
drwxr-xr-x@  6 Prateek  staff    204 Nov 13 13:11 A1AB6F26-70B6-4719-98E4-ED6D81CC4300
drwxr-xr-x@  6 Prateek  staff    204 Nov 13 13:11 AF81AC4E-4821-43BE-9463-EC13B6A8ECFB
drwxr-xr-x@  6 Prateek  staff    204 Nov 13 13:11 CF8F4435-17C7-4446-81D1-56F636F82212
drwxr-xr-x@  7 Prateek  staff    238 Jan 13 12:13 EBF10E39-984C-4AD4-A320-38519DB7A9EC
Prateeks-MacBook-Pro-2:Applications Prateek$ cd 254BFC17-66AC-4CDD-AC74-28E23BAAEB21/
Prateeks-MacBook-Pro-2:254BFC17-66AC-4CDD-AC74-28E23BAAEB21 Prateek$ ls
DamnVulnerableIOSApp.app        Documents                       Library                         tmp
Prateeks-MacBook-Pro-2:254BFC17-66AC-4CDD-AC74-28E23BAAEB21 Prateek$
```

Now make sure you have Hopper installed on your computer. You can read an article on Hopper at http://highaltitudehacks.com/2014/01/17/ios-application-security-part-28-patching-ios-application-with-hopper . Hopper's licensed version costs just $60 but its a great tool for such a price. If you are serious about Reverse Engineering, I would recommend you to get a copy of it. You can also use IDA pro and Hex Fiend for the same purpose. To get a little bit idea on how to use it, you can read the article here http://highaltitudehacks.com/2013/12/17/ios-application-security-part-26-patching-ios-applications-using-ida-pro-and-hex-fiend

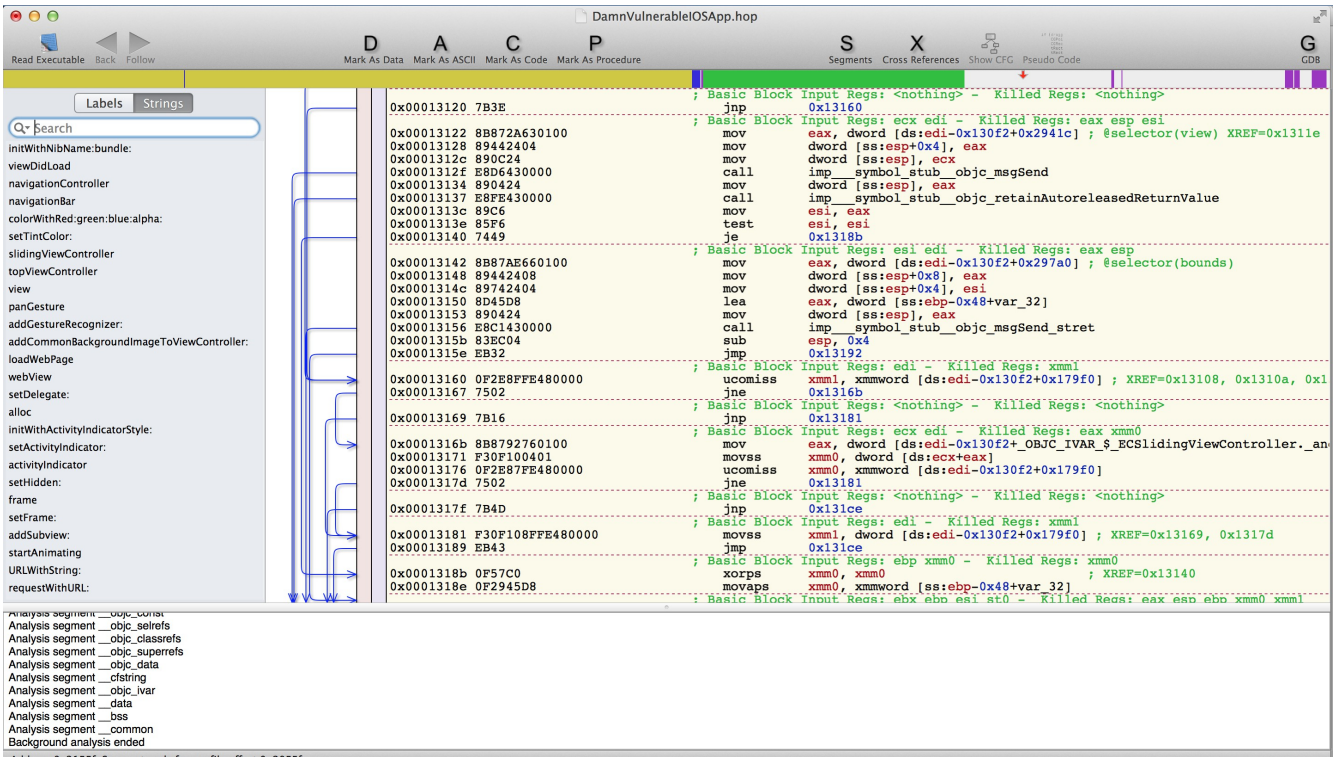
Now we need to give Hopper the binary for this application so we can modify it. Go to *Hopper → Read Executable To Disassemble*



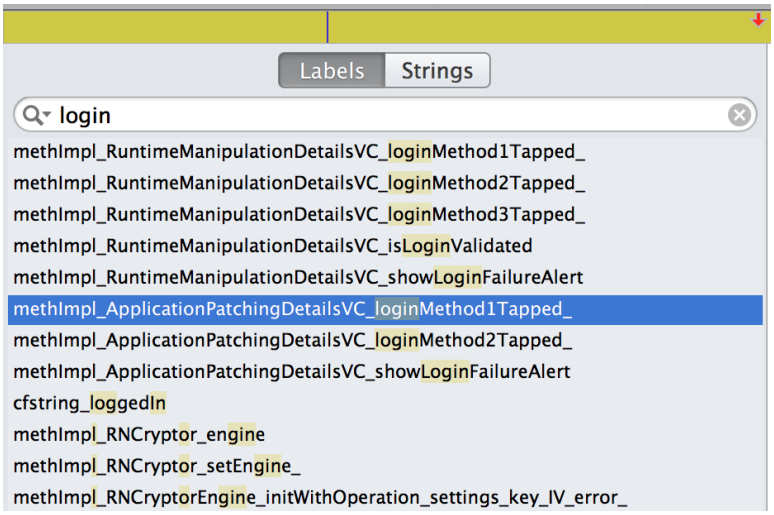
Now give the input as the binary for this app. **Make sure to save a copy of this binary somewhere as we will be needing it later.**



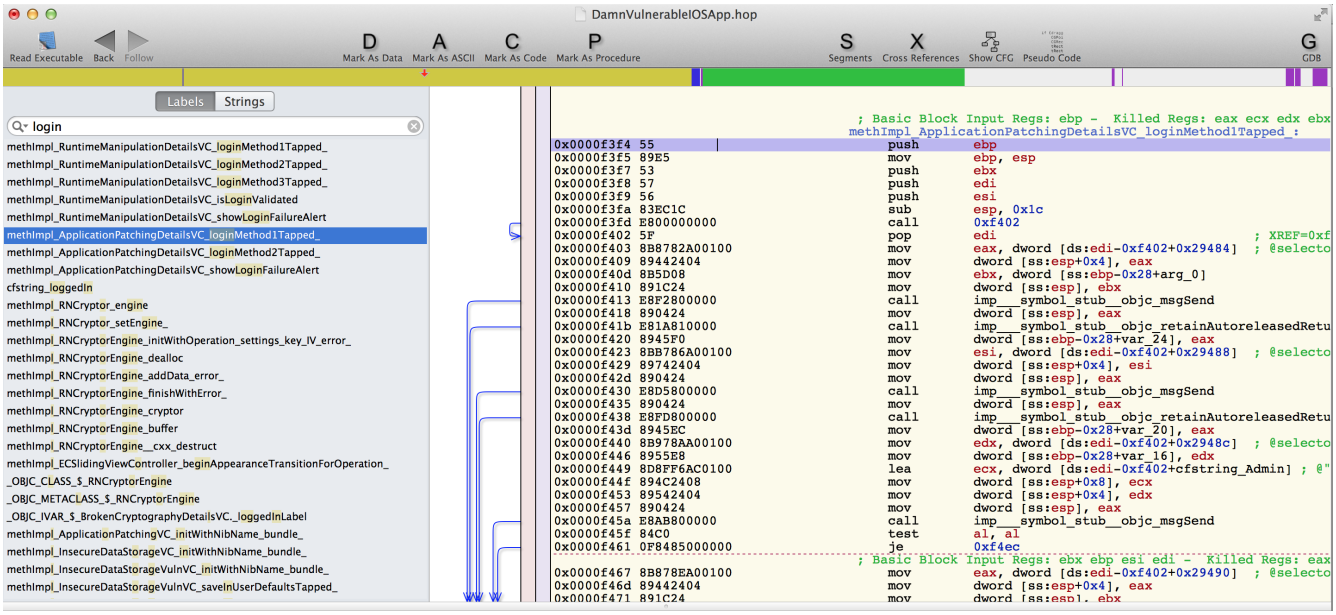
Hopper will produce the disassembly for this application.



Now go to the Labels section on the left and search for *login*. This will give you all method implementations that have the text *login.* Here, we can see the method that we are concerned with (highlighted).



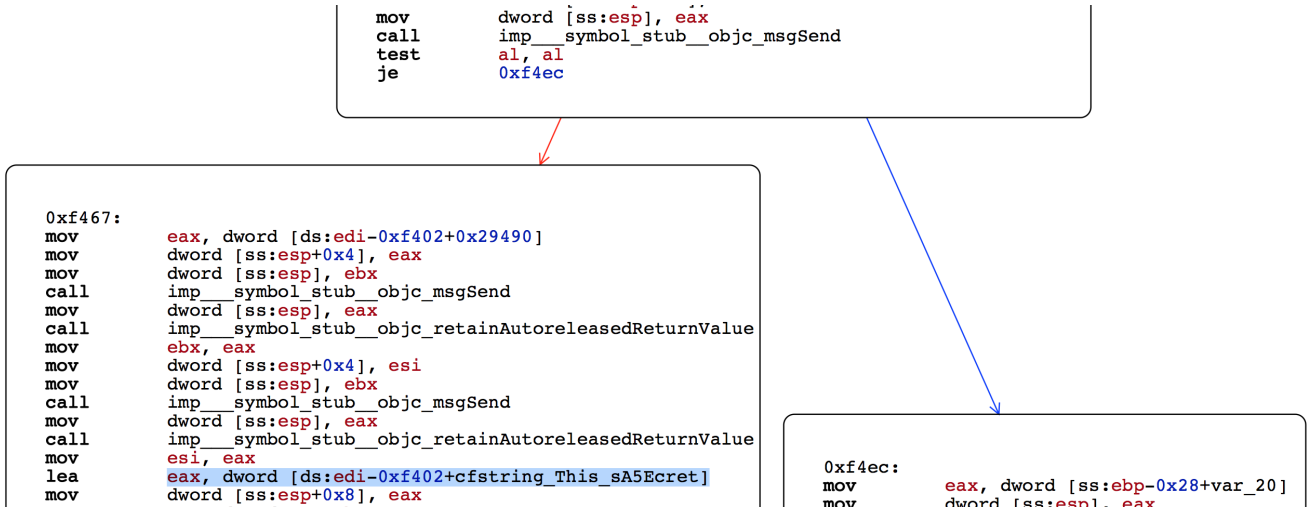
Click on it and we can see its disassembly to the right.



You can also see the CFG and Pseudo code for this method by clicking on *Show CFG* and *Pseudo Code* respectively. Both the CFG and Pseudo code for this method can be found in the same folder with the name cfg-login-1.pdf and pseudo-code-login-1.pdf respectively.

By looking at the CFG, we can see that there are different sections where the flow can go. We have to manipulate the binary to take the flow to the section we want.

If we look at the very first diversion...

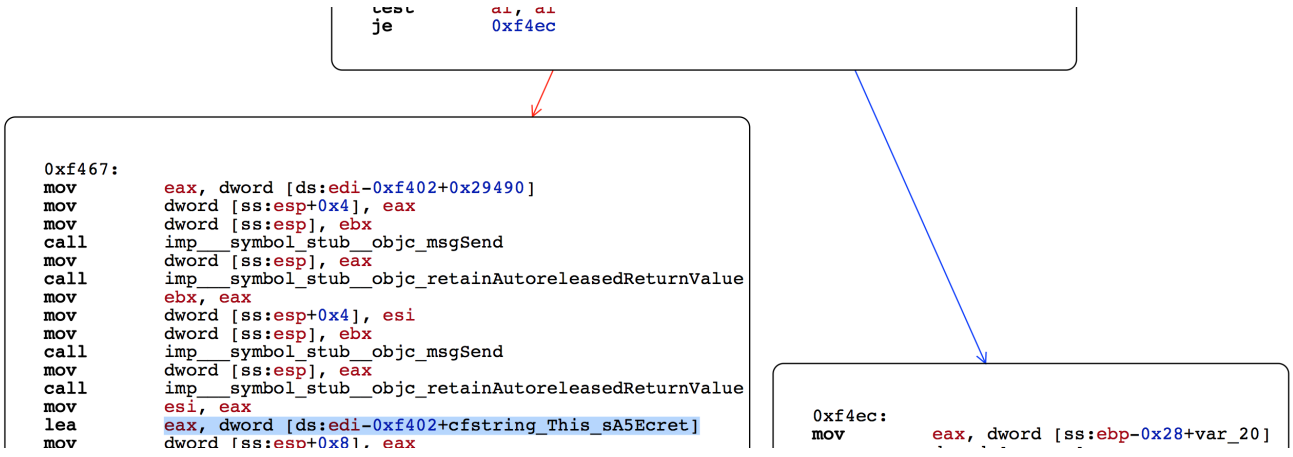


we can clearly assume that we want to take the flow to the left section. This is because we can see the assembly

```
eax, dword [ds:edi-0xf402+cfstring_This_sA5Ecret]
```

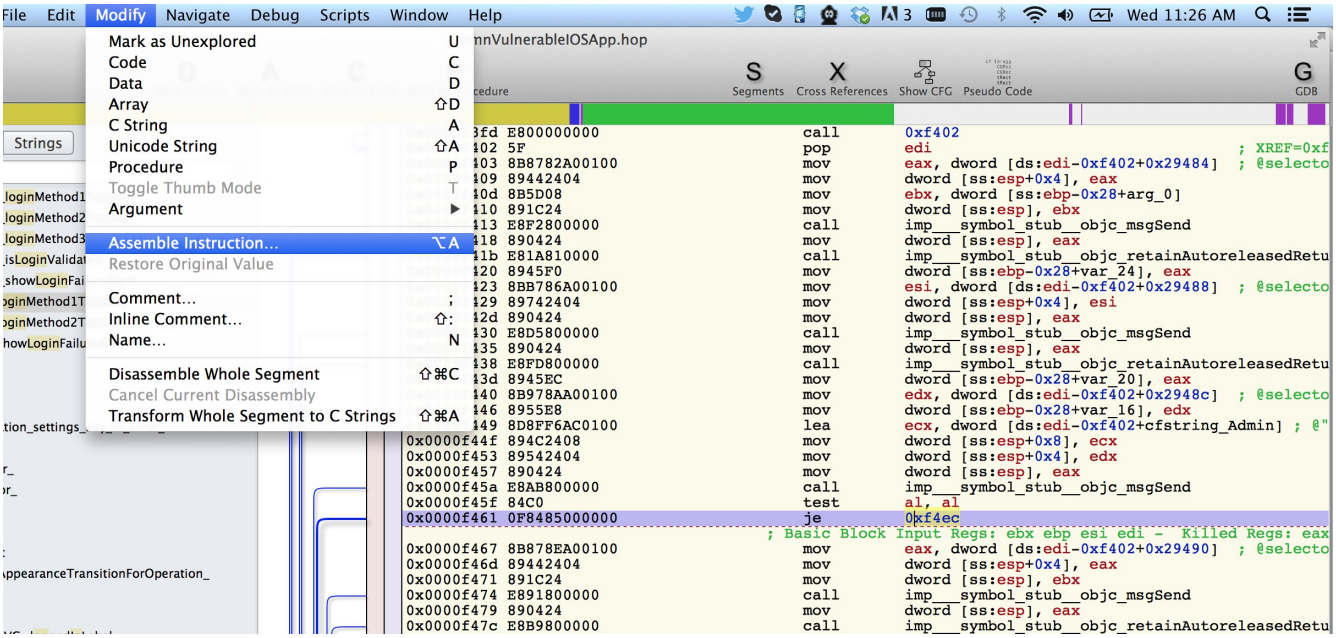
on the left side after diversion which denotes that some kind of extra validation is happening on the left side (maybe a check for the password ?), whereas the right side could just be the flow that denotes a failed validation. Optionally, you can also use hit and trial methods to check each flow. However, for now let's take the flow to the left section.

The last assembly instruction before the diversion is je 0xf4ec

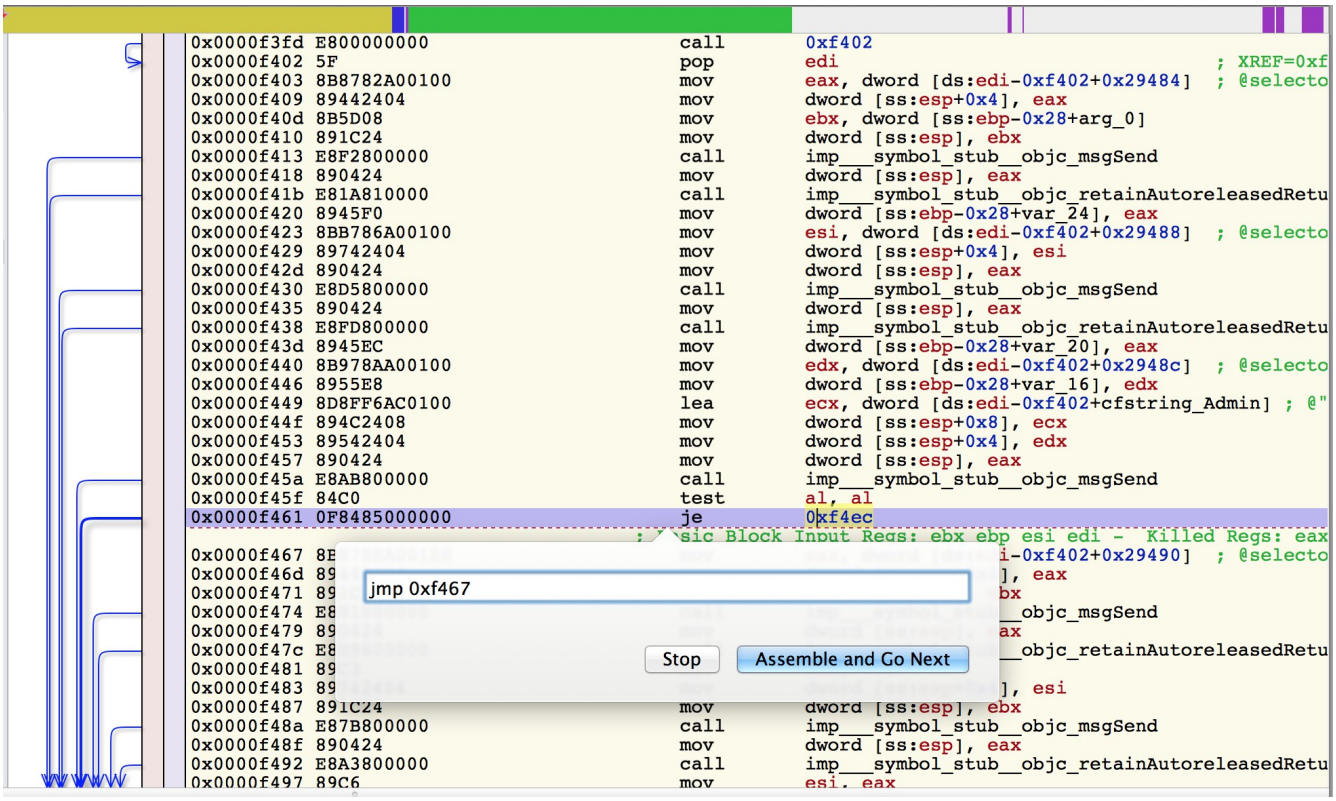


However, since the left section has the label *0xf467,* if we can change the last instruction to *jmp 0xf467* then the flow will always go the left section and this is what we want.

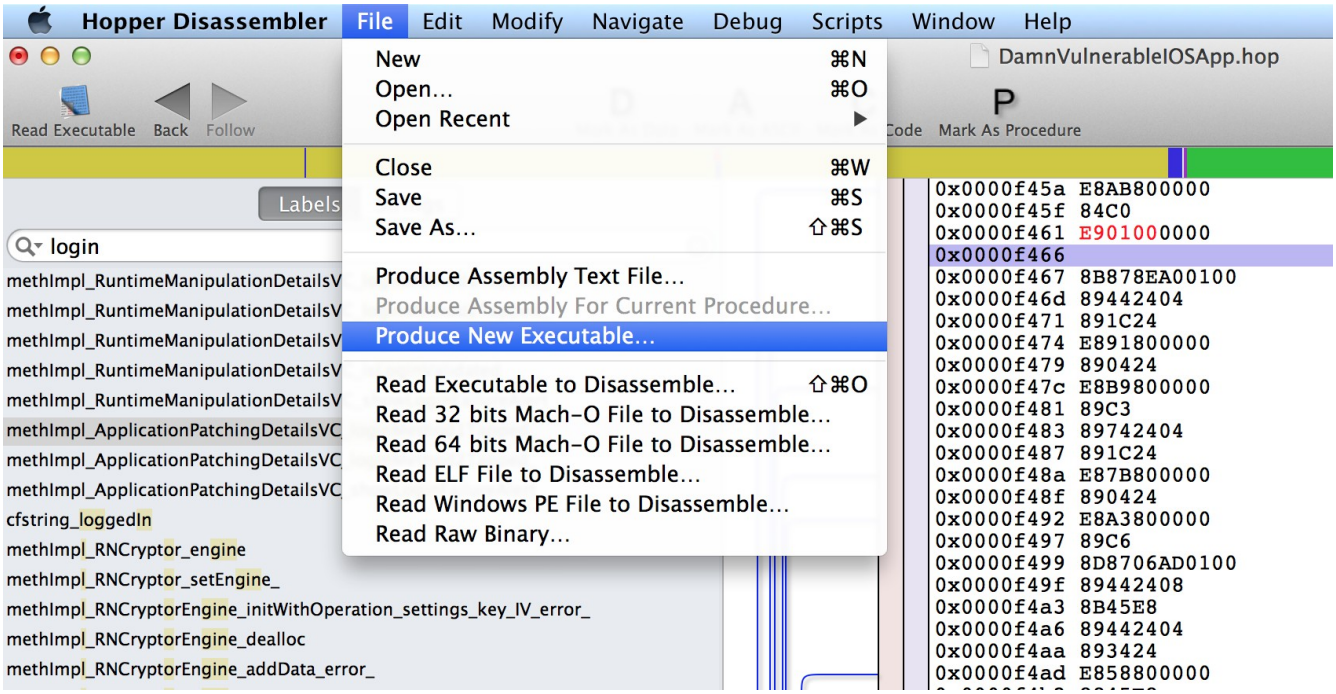
Head over to this instruction in the disassembly in Hopper. Then click on *Modify->Assemble Instruction*

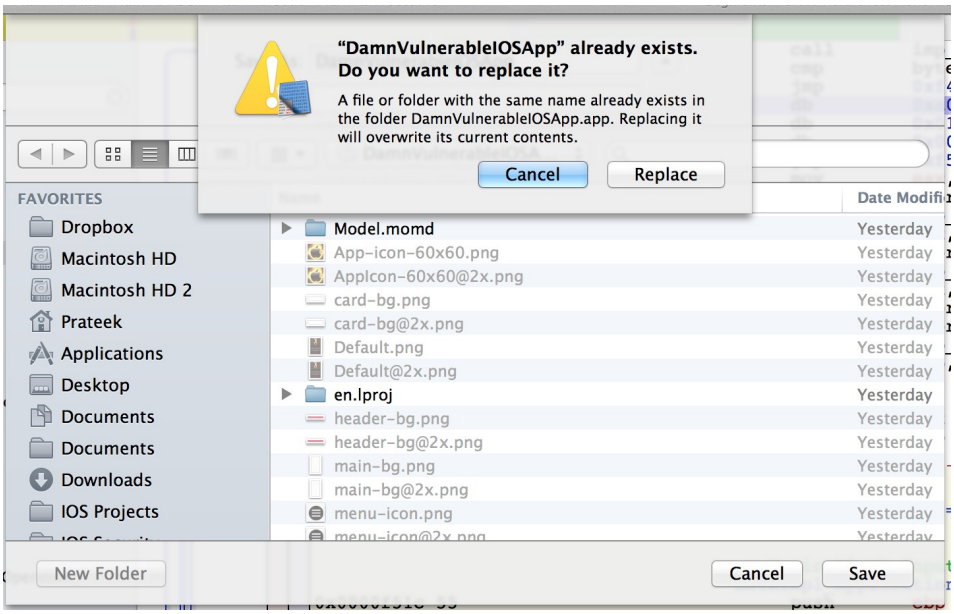


And write the instruction as *jmp 0xf467* and click on *Assemble and Go Next*

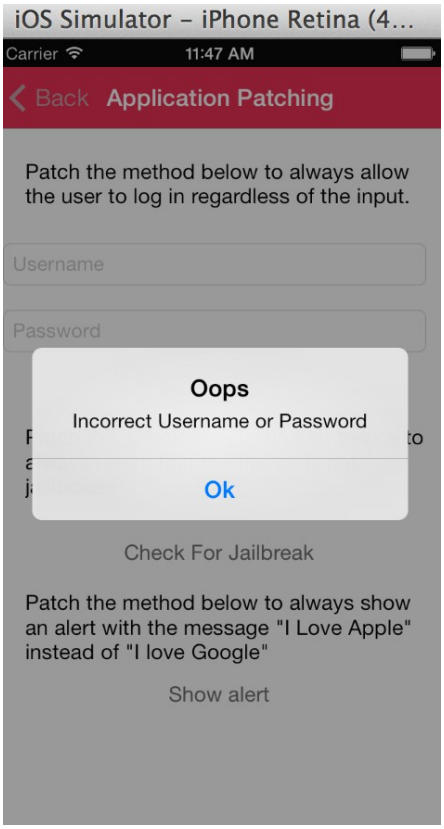


Now save the binary (Cmd +S) and click on *File->Produce New Executable* and overwrite the original application binary.

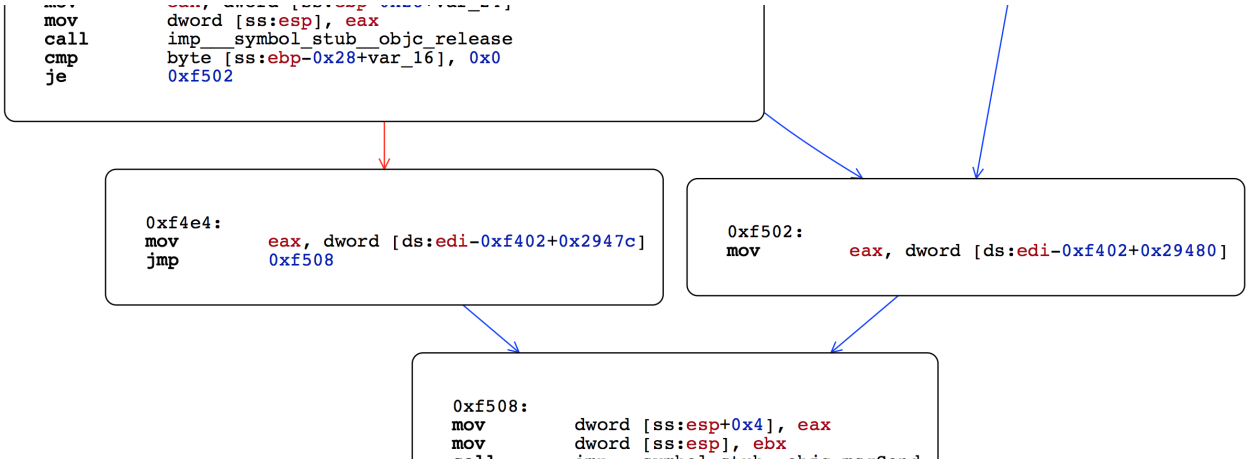




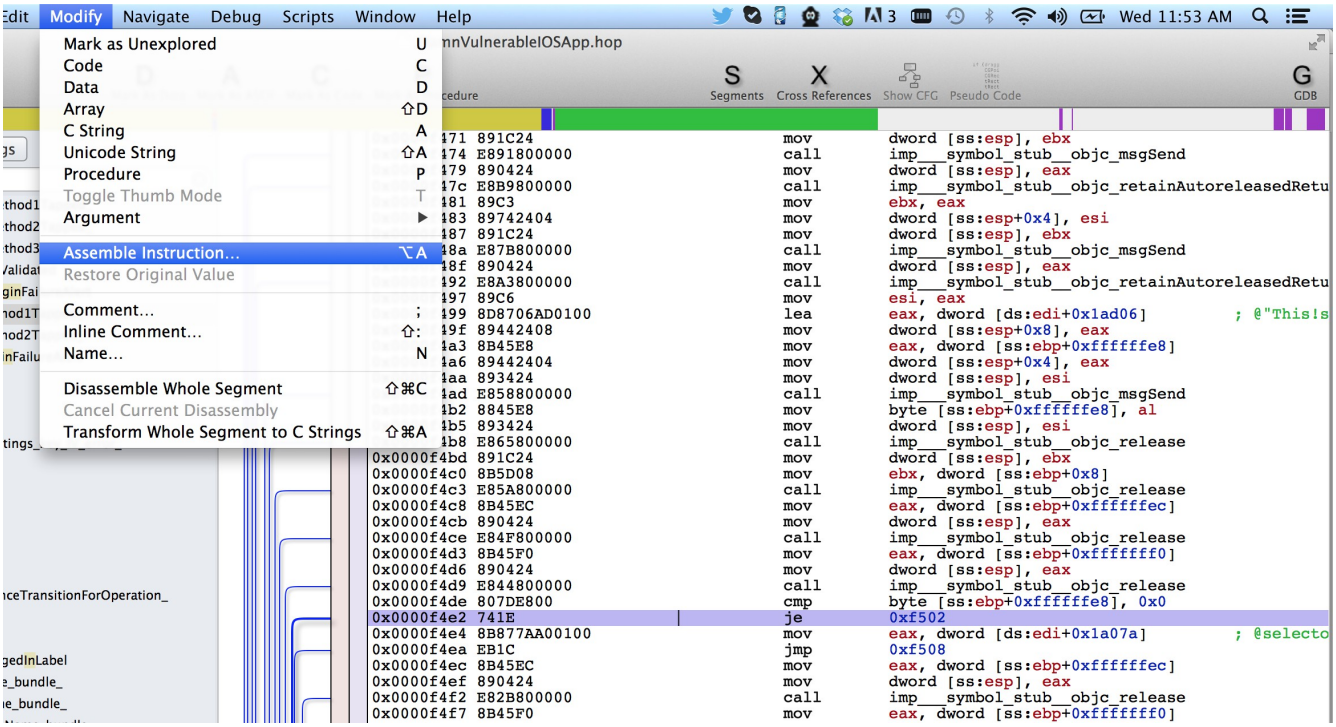
Now go to simulator, quit Xcode, quit the existing instance of the application by double tapping the home button (Cmd + Shift + H) and quitting the DVIA application. Now restart the application and tap on *Login Method 1* in the application patching section.

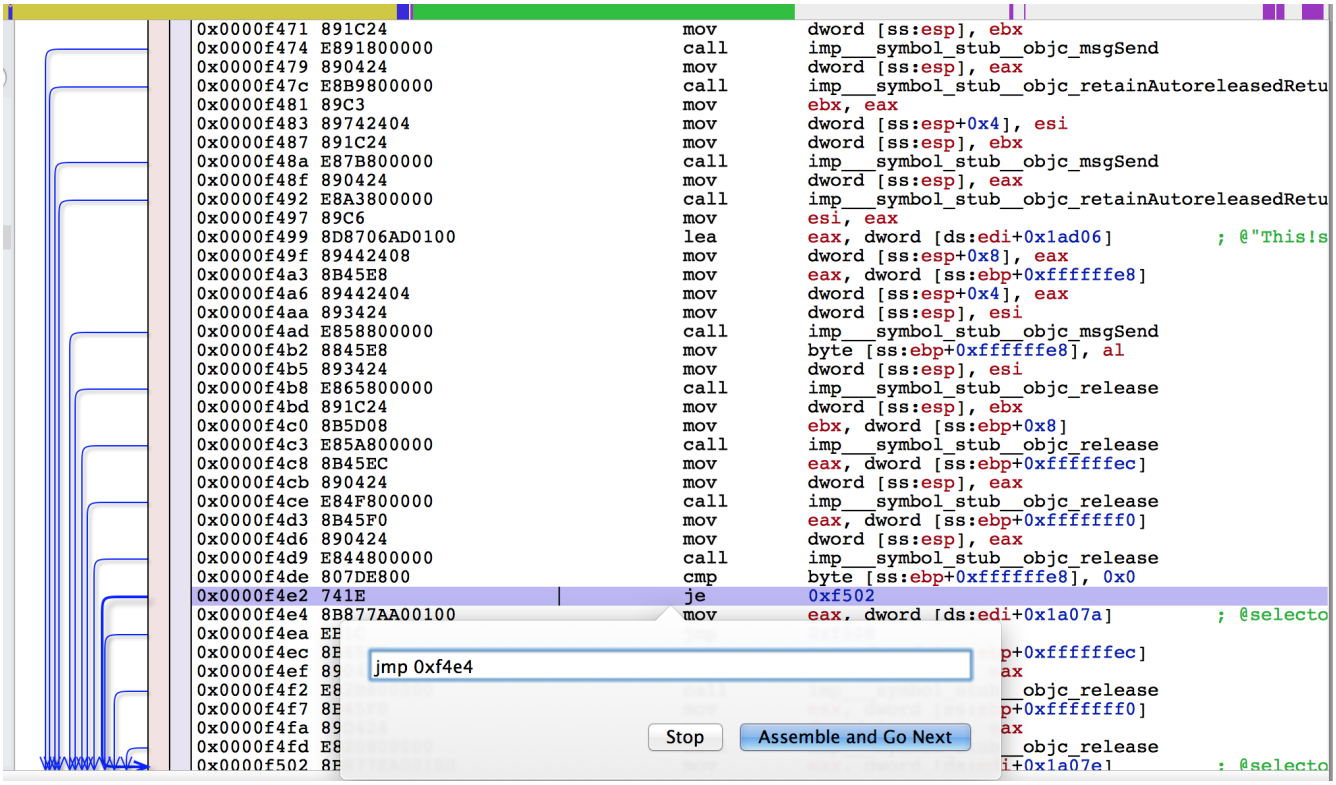


Well, still no Success ! Maybe our flow didn't reach the correct section. On looking at the CFG again, we can see that there is another diversion in the code as well.



For this diversion, let's try to make the flow go to the left section. For this change the last assembly instruction before the diversion which is *je 0xf502* to *jmp 0xf4e4* where 0xf4e4 is the label for the left section as seen in the image above.



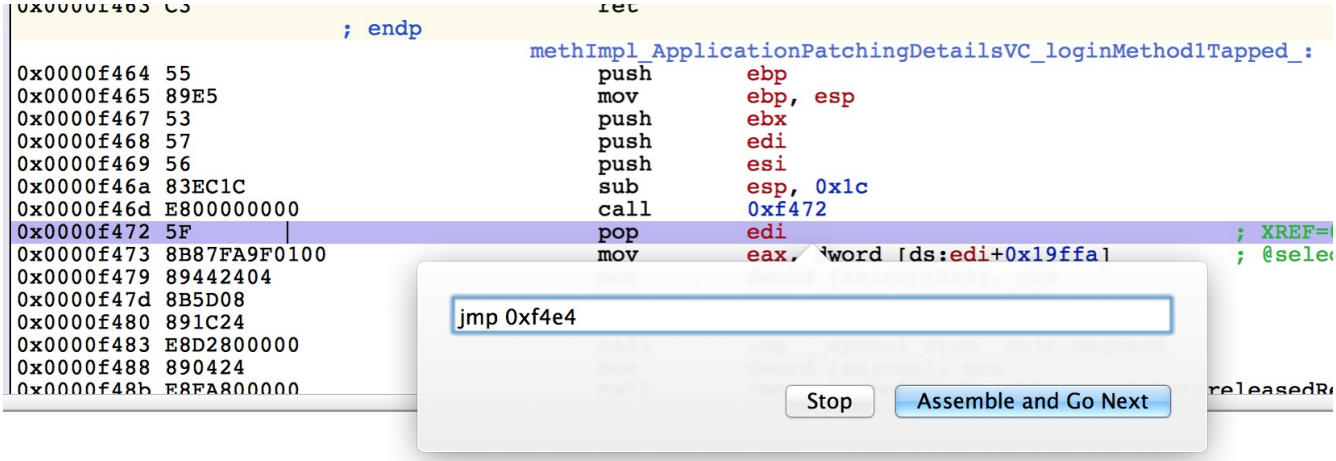


Again, let's save the binary, overwrite the existing one, quit the running instance of the application in the simulator and restart it. Then we head over to the application patching section and tap on *Login method 1*.

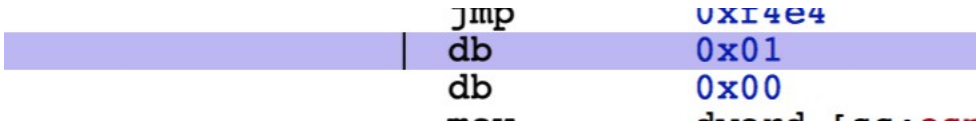
Oops, application crash :(

Well, there has been one thing we have been doing wrong. When we modified the 2nd instruction with our own instruction, we mistakenly overwrite some addresses with incorrect data. So when the execution went to that location, the application crashed. The basic thing here is to note that in order to head to the label *0xf4e4*, we don't need to modify instructions at 2 places. We can simply replace any instruction with the instruction *jmp 0xf4e4* and the flow should head over to that section.

Get the previous unmodified binary that you had saved somewhere as warned before. Open it in Hopper. Now just head over to any instruction in the assembly for the function we have been looking at and modify it with your instruction.



You will notice that some address locations are overwritten. But we know that these instructions will not get executed because the flow will jump to a different location just before it. To avoid crash anyways, you can replace these two db instructions with *nop* (No operation)



And now, if you quit the existing instance of the application, restart it and go to the Binary patching section and tap on *Login Method 1*, we will see that we have successfully bypassed the authentication check.