

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

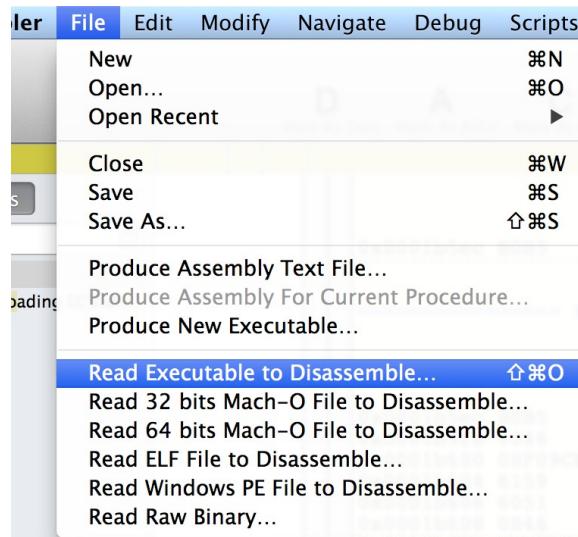
/Users/Prateek/Library/Application\ Support/iPhone\ Simulator/7.0.3/Applications/

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.

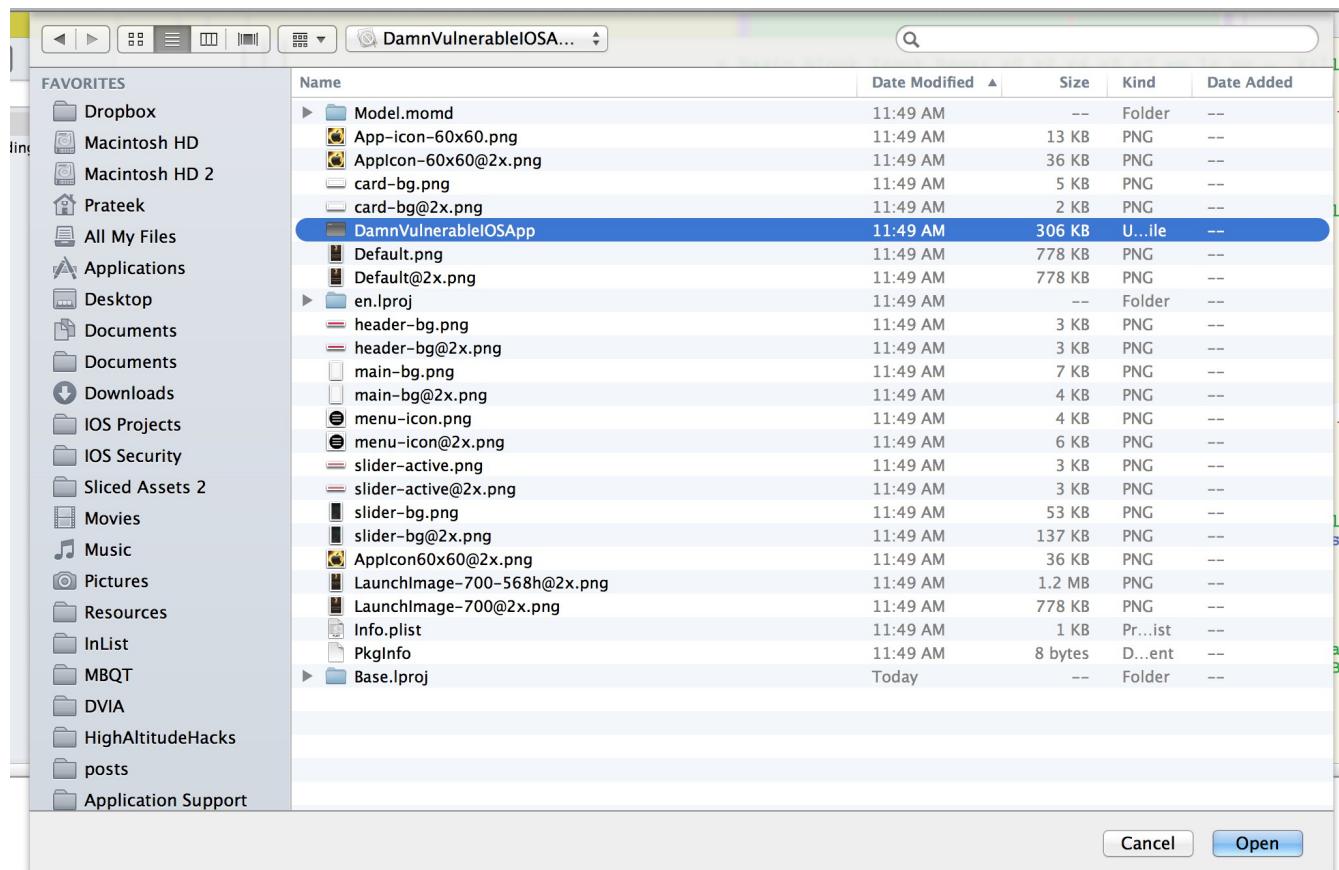
```
Prateeks-MacBook-Pro-2:254BFC17-66AC-4CDD-AC74-28E23BAAEB21 Prateek$ cd ~
Prateeks-MacBook-Pro-2:~ Prateek$ cd /Users/Prateek/Library/Application\ Support/iPhone\ Simulator/7.0.3/Applications/
Prateeks-MacBook-Pro-2:Applications Prateek$ ls -al
total 48
drwxr-xr-x 14 Prateek staff 476 Jan 29 10:26 .
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.

```

    ; Basic Block Input Regs: <nothing> - Killed Regs: <nothing>
    jnp 0x13160
    ; Basic Block Input Regs: ecx edi - Killed Regs: eax esp esi
    mov eax, dword [ds:edi-0x130f2+0x294ic] ; #selector(view) XREF=0x1311e
    mov dword [ss:esp+0x4], eax
    mov dword [ss:esp+0x4], edi
    call imp_symbol_stub_objc_msgSend
    mov dword [ss:esp], eax
    call imp_symbol_stub_objc_retainAutoreleasedReturnValue
    mov esi, eax
    test esi, esi
    je 0x1319b

    ; Basic Block Input Regs: esi edi - Killed Regs: eax esp
    mov eax, dword [ds:edi-0x130f2+0x297a0] ; #selector(bounds)
    mov dword [ss:esp+0x4], eax
    mov dword [ss:esp+0x4], esi
    lea eax, dword [ss:ebp-0x48+var_32]
    mov dword [ss:esp], eax
    sub esp, 0x4
    call imp_symbol_stub_objc_msgSend_stret
    jmp 0x13192

    ; Basic Block Input Regs: edi - Killed Regs: xmm1
    ucomiss xmm1, xmmword [ds:edi-0x130f2+0x179f0] ; XREF=0x13108, 0x1310a, 0x1311b
    jne 0x1316b
    ; Basic Block Input Regs: <nothing> - Killed Regs: <nothing>
    jnp 0x13181

    ; Basic Block Input Regs: ecx edi - Killed Regs: eax xmm0
    mov eax, dword [ds:edi-0x130f2+_OBJC_IVAR_$_ECSlidingViewController._an]
    movss xmm0, dword [ds:ecx+eax]
    ucomiss xmm0, xmmword [ds:edi-0x130f2+0x179f0]
    jne 0x13161

    ; Basic Block Input Regs: <nothing> - Killed Regs: <nothing>
    jnp 0x131ce

    ; Basic Block Input Regs: edi - Killed Regs: xmm1
    movss xmm1, dword [ds:edi-0x130f2+0x179f0] ; XREF=0x13169, 0x1317d
    jmp 0x131ce

    ; Basic Block Input Regs: ebp xmm0 - Killed Regs: xmm0
    xorps xmm0, xmm0
    movaps xmm0, xmmword [ss:ebp-0x48+var_32]
    ; Basic Block Input Regs: ebx ebp esi st0 - Killed Regs: eax esp ebx xmm0 xmm1
    : Basic Block Input Regs: ebx ebp esi st0 - Killed Regs: eax esp ebx xmm0 xmm1

Analysis segment __objc_const
Analysis segment __objc_selrefs
Analysis segment __objc_classrefs
Analysis segment __objc_superrefs
Analysis segment __objc_data
Analysis segment __cfstring
Analysis segment __objc_ivar
Analysis segment __data
Analysis segment __bss
Analysis segment __common
Background analysis ended

```

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).

Label
methImpl_RuntimeManipulationDetailsVC_loginMethod1Tapped_
methImpl_RuntimeManipulationDetailsVC_loginMethod2Tapped_
methImpl_RuntimeManipulationDetailsVC_loginMethod3Tapped_
methImpl_RuntimeManipulationDetailsVC_isLoginValidated
methImpl_RuntimeManipulationDetailsVC_showLoginFailureAlert
methImpl_ApplicationPatchingDetailsVC_loginMethod1Tapped_
methImpl_ApplicationPatchingDetailsVC_loginMethod2Tapped_
methImpl_ApplicationPatchingDetailsVC_showLoginFailureAlert
cfstring_loggedIn
methImpl_RNCryptor_engine
methImpl_RNCryptor_setEngine_
methImpl_RNCryptorEngine_initWithOperation_settings_key_IV_error_

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

Screenshot of the Immunity Debugger showing assembly code for the `methImpl_ApplicationPatchingDetailsVC_loginMethod1Tapped` function. The assembly code is annotated with comments and labels. The code involves multiple branches and calls to objc_msgSend and objc_release methods.

```

; Basic Block Input Regs: ebp - Killed Regs: eax ecx edx ebx
methImpl_ApplicationPatchingDetailsVC_loginMethod1Tapped:
    push    ebp
    mov     ebp, esp
    push    ebx
    push    edi
    push    esi
    sub    esp, 0x1c
    call    0x1402
    pop    ebx
    mov     eax, dword [ds:edi-0xf402+0x29484]; XREF=0xf
    mov     dword [ss:esp+0x4], eax
    mov     ebx, dword [ss:ebp-0x28+arg_0]
    mov     mov    dword [ss:esp+0x4], ebx
    mov     mov    dword [ss:esp+0x4], esi
    mov     mov    dword [ss:esp+0x4], ebx
    call    imp__symbol_stub_objc_msgSend
    mov     mov    dword [ss:esp], eax
    call    imp__symbol_stub_objc_retainAutoreleasedReturnValue
    mov     mov    eax, dword [ds:edi-0xf402+0x29484]; @selecto
    mov     mov    dword [ss:ebp-0x28+var_24], eax
    mov     mov    esi, dword [ds:edi-0xf402+0x29488]; @selecto
    mov     mov    dword [ss:esp+0x4], esi
    mov     mov    dword [ss:esp+0x4], ebx
    mov     mov    dword [ss:esp+0x4], edx
    mov     mov    dword [ss:esp+0x4], ecx
    mov     mov    dword [ss:esp+0x4], edx
    mov     mov    dword [ss:esp+0x4], eax
    call    imp__symbol_stub_objc_msgSend
    mov     mov    al, al
    test   test   al, al
    je     je     0x4ec

; Basic Block Input Regs: ebx ebp esi edi - Killed Regs: eax
0x0000f467: 8B878EA00100
0x0000f46d: 89442404
0x0000f471: 891C24

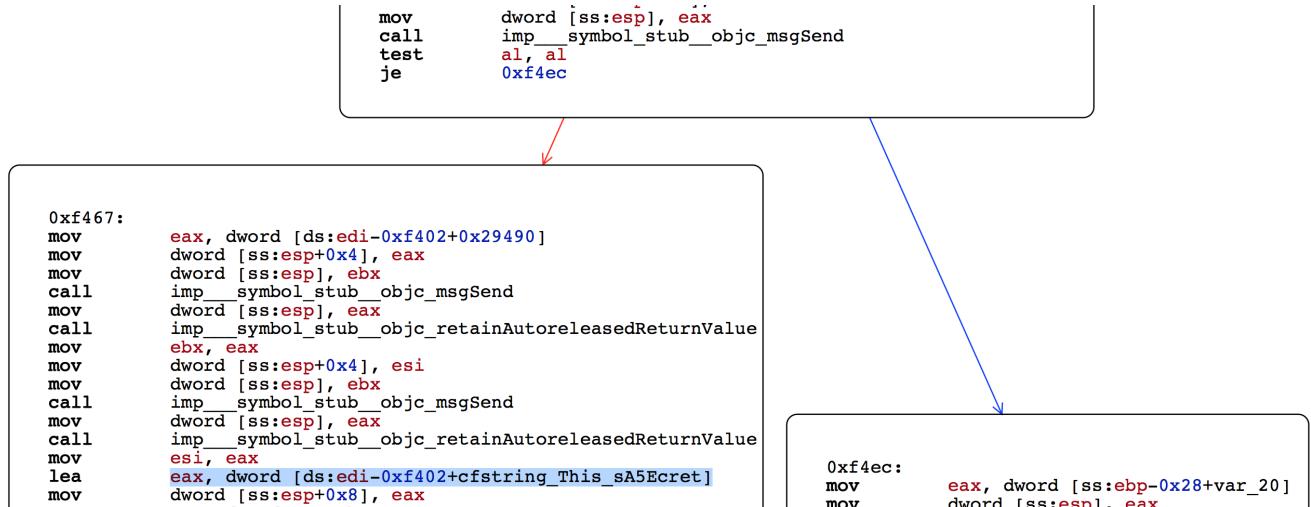
```

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...

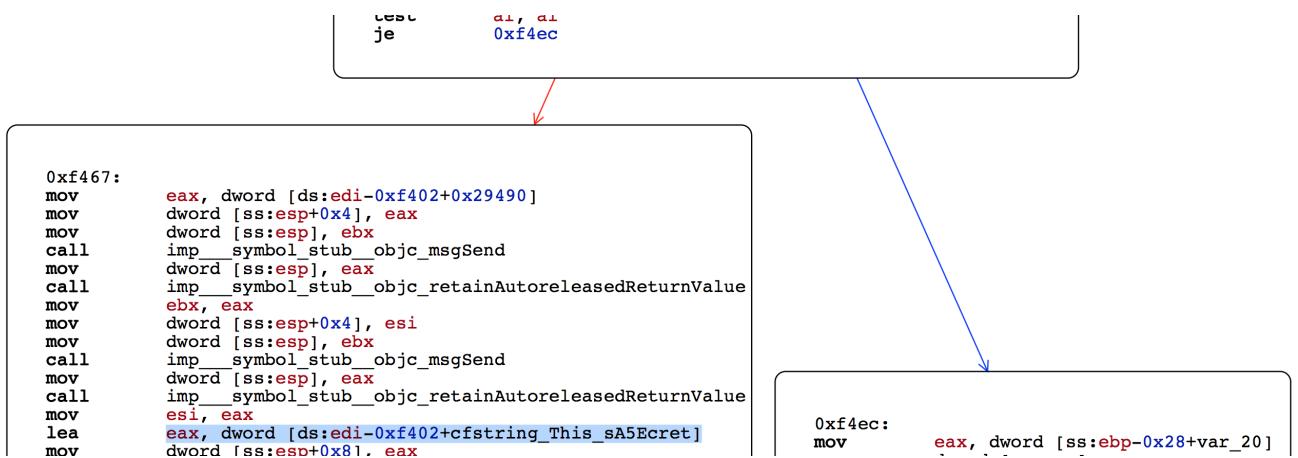


we can clearly assume that we want to take the flow to the left section. This is because we can see the assembly

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

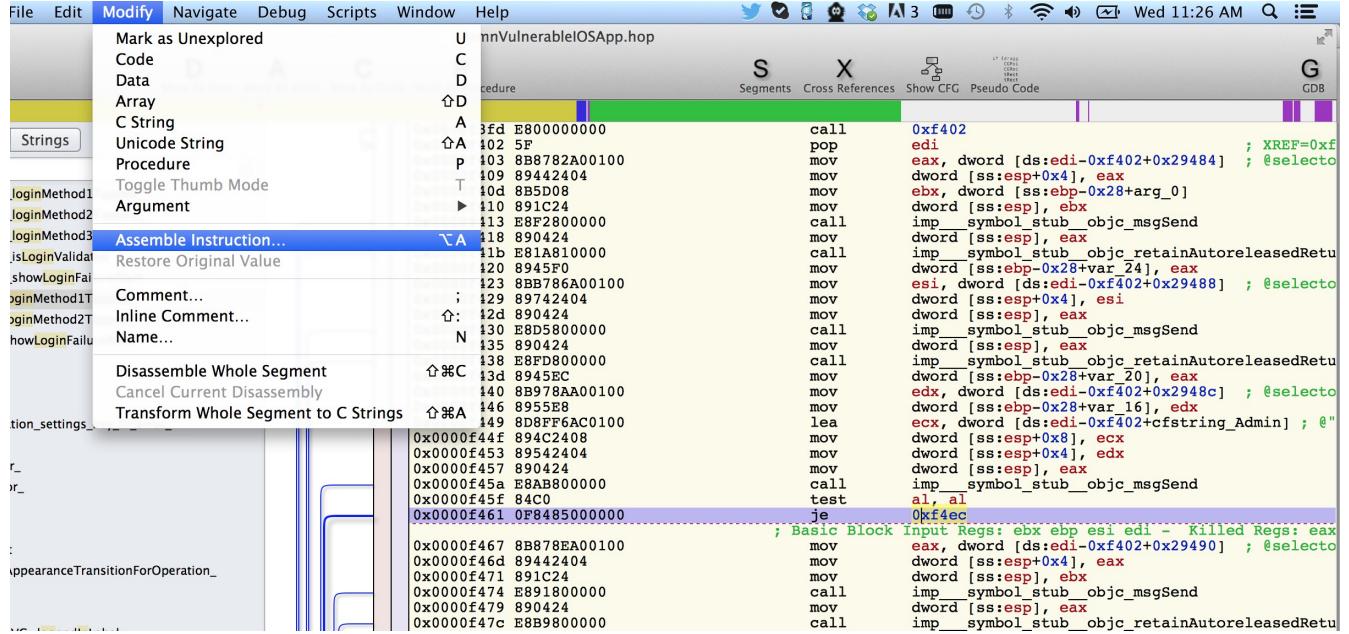
on the left side after diversion which denotes that some kind of extra validation is happening on the left side (maybe a check for the password ?), whereas the right side could just be the flow that denotes a failed validation. Optionally, you can also use hit and trial methods to check each flow. However,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

The last assembly instruction before the diversion is `je 0xf4ec`

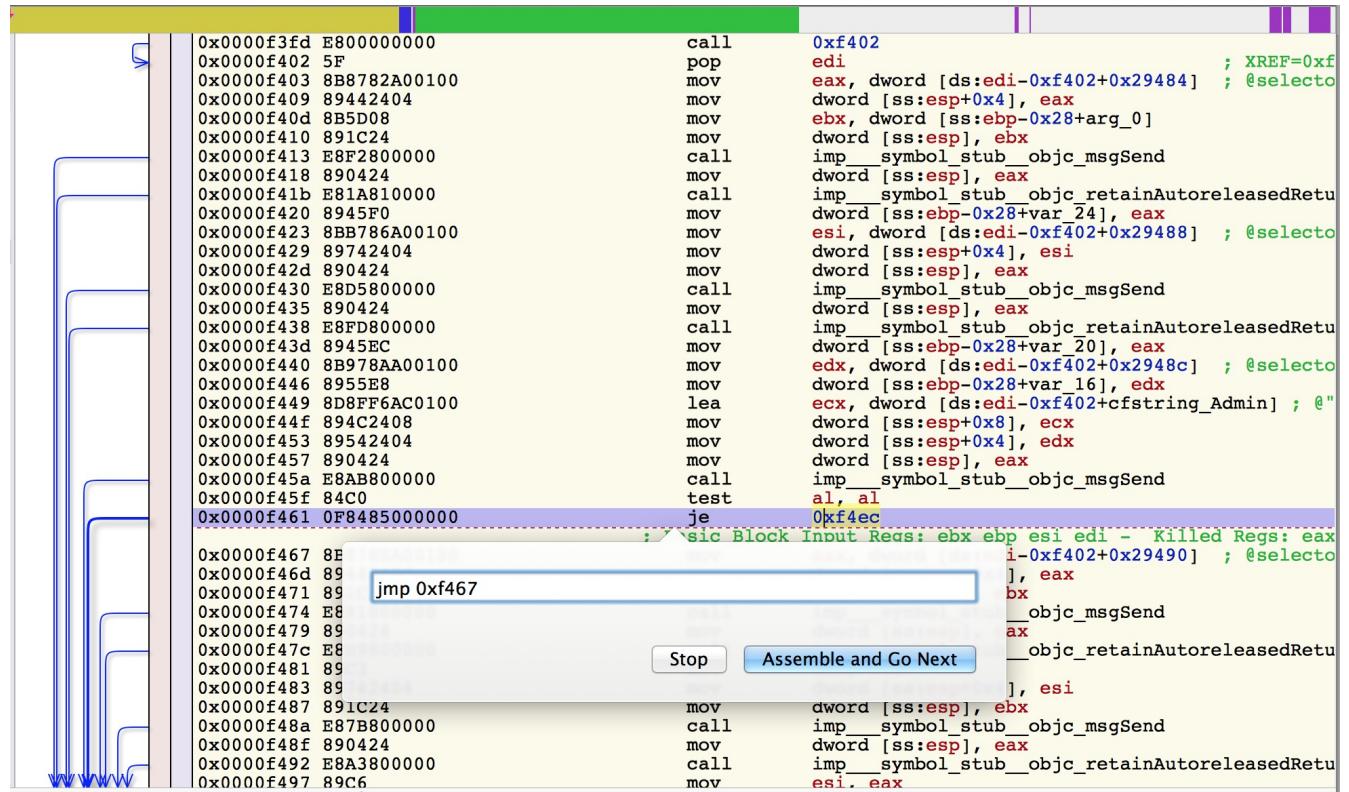


However, since the left section has the label `0xf467`, if we can change the last instruction to `jmp 0xf467` then the flow will always go the left section and this is what we want.

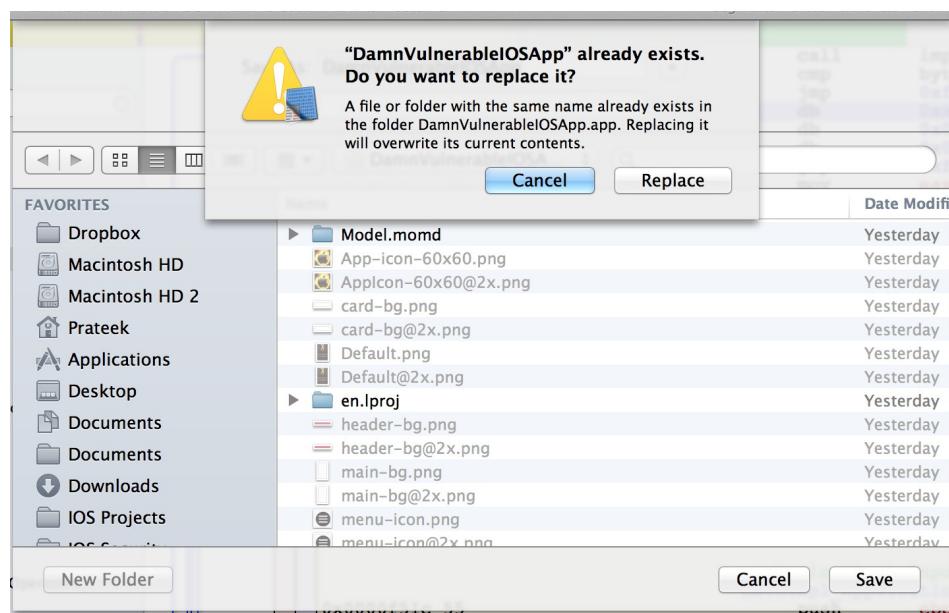
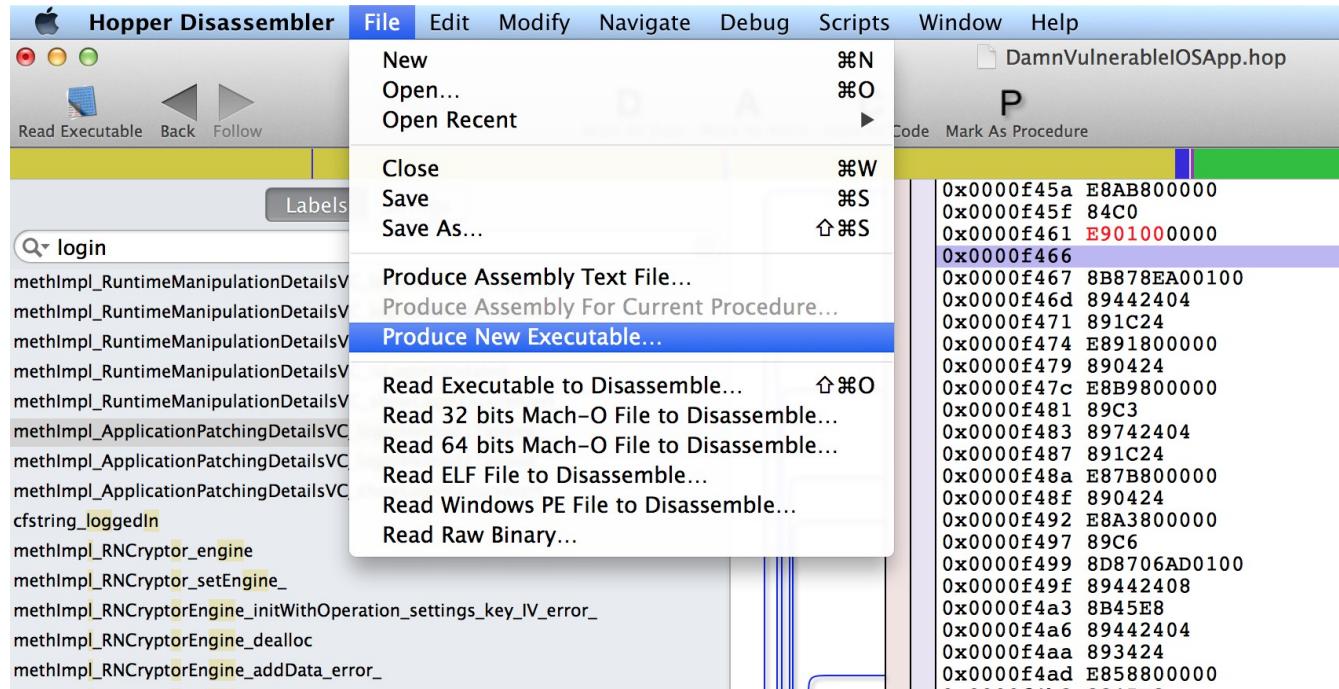
Head over to this instruction in the disassembly in Hopper. Then click on *Modify->Assemble Instruction*



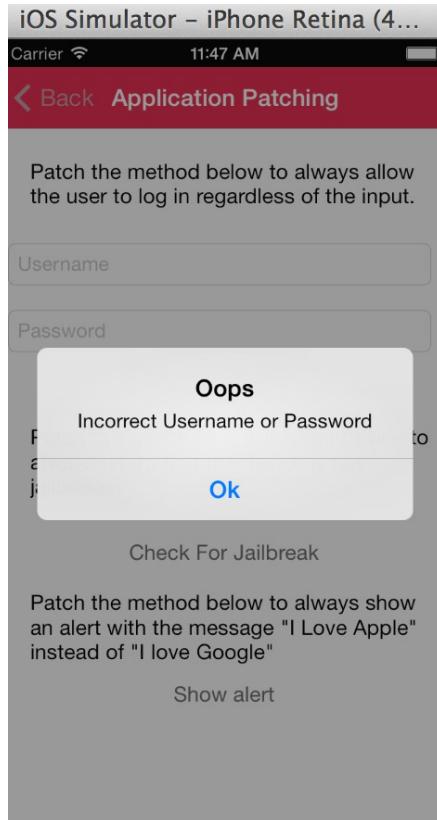
And write the instruction as *jmp 0xf467* and click on *Assemble and Go Next*



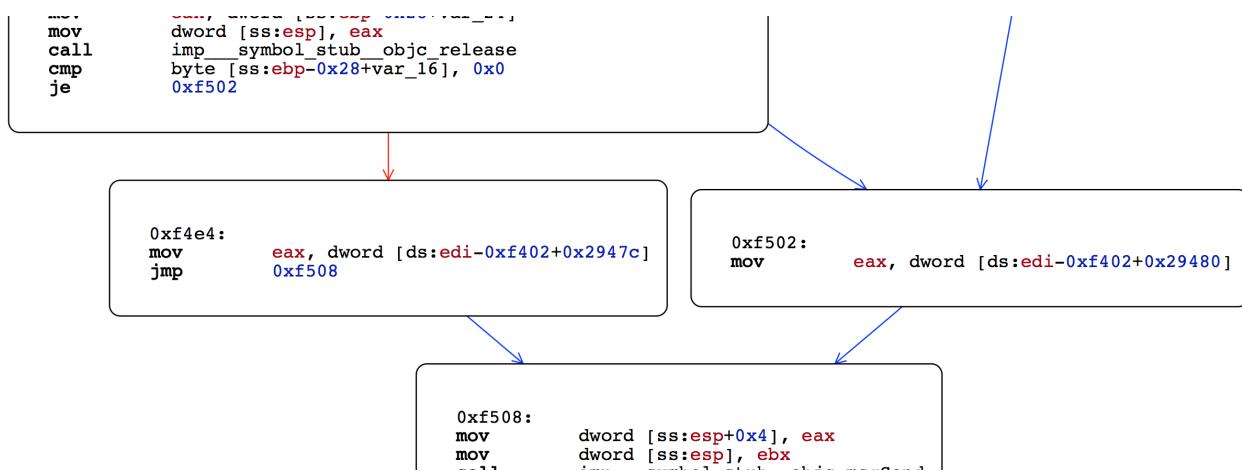
Now save the binary (Cmd + S) and click on *File->Produce New Executable* and overwrite the original application binary.



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.

Mark as Unexplored

Code

Data

Array

C String

Unicode String

Procedure

thod1 Toggle Thumb Mode

thod2 Argument

thod3 Assemble Instruction...

Validate Restore Original Value

ginFail

mod1T

mod2T

inFailu

Comment...

Inline Comment...

Name...

Disassemble Whole Segment

Cancel Current Disassembly

Transform Whole Segment to C Strings

ting...

anceTransitionForOperation_

gedInLabel

e_bundle_

ie_bundle_

nnVulnerableIOSApp.hop

```

U nnVulnerableIOSApp.hop
C
D
D procedure
A A 171 891C24 mov dword [ss:esp], ebx
A 174 E891800000 call imp__symbol_stub_objc_msgSend
P 179 890424 mov dword [ss:esp], eax
T 17c E8B9800000 call imp__symbol_stub_objc_retainAutoreleasedRetu
ebx, eax
A 181 89C3 mov dword [ss:esp+0x4], esi
A 183 89742404 call dword [ss:esp], ebx
A 187 891C24 mov imp__symbol_stub_objc_msgSend
call imp__symbol_stub_objc_retainAutoreleasedRetu
esi, eax
A 18a E87B800000 call dword [ss:esp+0x4], eax
A 18f 890424 mov imp__symbol_stub_objc_msgSend
call dword [ss:esp], eax
A 192 E8A3800000 mov imp__symbol_stub_objc_retainAutoreleasedRetu
esi, eax
A 197 89C6 lea eax, dword [ds:edi+0x1ad06] ; @"This!
A 19f 89442408 mov dword [ss:esp+0x8], eax
N 1a3 8B45E8 mov eax, dword [ss:ebp+0xffffffffe8]
A 1a6 89442404 mov dword [ss:esp+0x4], eax
A 1aa 893424 mov dword [ss:esp], esi
call imp__symbol_stub_objc_msgSend
mov byte [ss:ebp+0xffffffffe8], al
A 1b2 8845E8 mov dword [ss:esp], ebx
call imp__symbol_stub_objc_release
mov ebx, dword [ss:ebp+0x8]
A 1b5 893424 call imp__symbol_stub_objc_release
mov eax, dword [ss:ebp+0xfffffffffec]
A 1b8 E865800000 mov dword [ss:esp], eax
call imp__symbol_stub_objc_release
mov byte [ss:ebp+0xffffffffe8], 0x0
A 0x0000f4bd 891C24 cmp 0xf502
mov je 0xf502
A 0x0000f4e4 8B877AA00100 mov eax, dword [ds:edi+0x1a07a] ; @selecto
A 0x0000f4ea EE jmp 0xf508
A 0x0000f4ec 8E mov eax, dword [ss:ebp+0xfffffffffec]
A 0x0000f4ef 89 mov dword [ss:esp], eax
A 0x0000f4f2 E8 call imp__symbol_stub_objc_release
A 0x0000f4f7 8E mov byte [ss:ebp+0xffffffffe8], ax
A 0x0000f4fa 89 mov eax, dword [ss:ebp+0xfffffffff0]
A 0x0000f4fd E8 call imp__symbol_stub_objc_release
A 0x0000f502 8F mov eax, dword [ss:ebp+0xfffffffff0]

```

0x0000f471 891C24 mov dword [ss:esp], ebx
0x0000f474 E891800000 call imp__symbol_stub_objc_msgSend
0x0000f479 890424 mov dword [ss:esp], eax
0x0000f47c E8B9800000 call imp__symbol_stub_objc_retainAutoreleasedRetu
ebx, eax
0x0000f481 89C3 mov dword [ss:esp+0x4], esi
0x0000f483 89742404 call dword [ss:esp], ebx
0x0000f487 891C24 mov imp__symbol_stub_objc_msgSend
0x0000f48a E87B800000 call dword [ss:esp], eax
0x0000f48f 890424 mov imp__symbol_stub_objc_release
0x0000f492 E8A3800000 call imp__symbol_stub_objc_retainAutoreleasedRetu
esi, eax
0x0000f497 89C6 lea eax, dword [ds:edi+0x1ad06] ; @"This!
0x0000f499 8D8706AD0100 mov dword [ss:esp+0x8], eax
0x0000f49f 89442408 mov eax, dword [ss:ebp+0xfffffffffe8]
0x0000f4a3 8B45E8 mov dword [ss:esp+0x4], eax
0x0000f4a6 89442404 call imp__symbol_stub_objc_release
0x0000f4aa 893424 mov dword [ss:esp], esi
0x0000f4ad E858800000 call imp__symbol_stub_objc_msgSend
0x0000f4b2 8845E8 mov byte [ss:ebp+0xfffffffffe8], al
0x0000f4b5 893424 mov dword [ss:esp], esi
0x0000f4b8 E865800000 call imp__symbol_stub_objc_release
0x0000f4bd 891C24 mov dword [ss:esp], ebx
0x0000f4c0 8B5D08 call imp__symbol_stub_objc_release
0x0000f4c3 E85A800000 mov eax, dword [ss:ebp+0x8]
0x0000f4c8 8B45EC mov imp__symbol_stub_objc_release
0x0000f4cb 890424 mov dword [ss:esp], eax
0x0000f4ce E84F800000 call imp__symbol_stub_objc_release
0x0000f4d3 8B45F0 mov eax, dword [ss:ebp+0xfffffffff0]
0x0000f4d6 890424 mov dword [ss:esp], eax
0x0000f4d9 E844800000 call imp__symbol_stub_objc_release
0x0000f4de 807DE800 cmp byte [ss:ebp+0xfffffffffe8], 0x0
0x0000f4e2 741E je 0xf502
0x0000f4e4 8B877AA00100 mov eax, dword [ds:edi+0x1a07a] ; @selecto
0x0000f4ea EE p+0xfffffffffec]
0x0000f4ec 8E jmp 0xf508
0x0000f4ef 89 ax
0x0000f4f2 E8 _objc_release
0x0000f4f7 8E p+0xfffffffff0]
0x0000f4fa 89 ax
0x0000f4fd E8 _objc_release
0x0000f502 8F i+0x1a07e1 : @selecto

Stop Assemble and Go Next

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*.

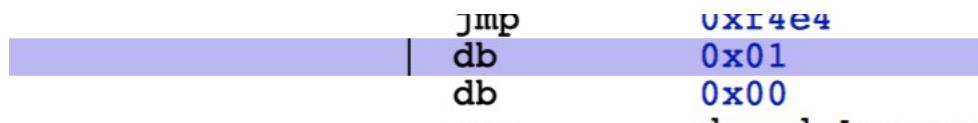
Oops, application crash :(

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.

```
0x000001403 C9      ; endp
                                ret
methImpl_ApplicationPatchingDetailsVC_loginMethod1Tapped_:
    push    ebp
    mov     ebp, esp
    push    ebx
    push    edi
    push    esi
    sub    esp, 0x1c
    call   0xf472
    pop    edi
    mov    eax, word [ds:edi+0x19ffa]
; XREF=1
; @selected
0x0000f472 5F          | jmp 0xf4e4
0x0000f473 8B87FA9F0100
0x0000f479 89442404
0x0000f47d 8B5D08
0x0000f480 891C24
0x0000f483 E8D2800000
0x0000f488 890424
0x0000f48b F8FA800000
```

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.

