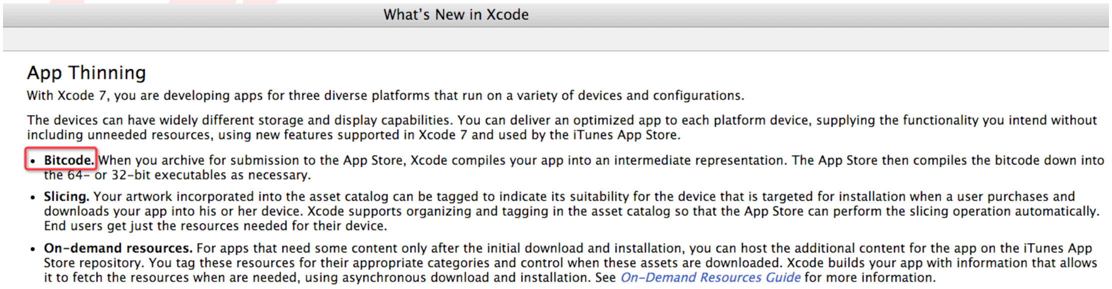
Workflow and Security Assessment of Bitcode in Xcode 7

Proteas of Qihoo 360 Nirvan Team

2015-12-10

Introduction

Along with the release of Xcode 7, Apple added a new feature of Bitcode[1] for Xcode:



What's New in Xcode

App Thinning

With Xcode 7, you are developing apps for three diverse platforms that run on a variety of devices and configurations.

The devices can have widely different storage and display capabilities. You can deliver an optimized app to each platform device, supplying the functionality you intend without including unneeded resources, using new features supported in Xcode 7 and used by the iTunes App Store.

- **Bitcode.** When you archive for submission to the App Store, Xcode compiles your app into an intermediate representation. The App Store then compiles the bitcode down into the 64- or 32-bit executables as necessary.
- **Slicing.** Your artwork incorporated into the asset catalog can be tagged to indicate its suitability for the device that is targeted for installation when a user purchases and downloads your app into his or her device. Xcode supports organizing and tagging in the asset catalog so that the App Store can perform the slicing operation automatically. End users get just the resources needed for their device.
- **On-demand resources.** For apps that need some content only after the initial download and installation, you can host the additional content for the app on the iTunes App Store repository. You tag these resources for their appropriate categories and control when these assets are downloaded. Xcode builds your app with information that allows it to fetch the resources when are needed, using asynchronous download and installation. See *On-Demand Resources Guide* for more information.

New features often mean potential new attack surfaces. This article firstly introduces what Bitcode is and its associated workflow. After getting familiar with the workflow of Bitcode, the next step is to evaluate attack surfaces associated with Bitcode. In the last part, test methods specific to each attack surface and current test results are included.

What is Bitcode

To put it simply, Bitcode is an on-disk binary representation of LLVM-IR. Please refer to [2] for more details on Bitcode. Here we will take an example to give you an intuitive idea of what Bitcode is.

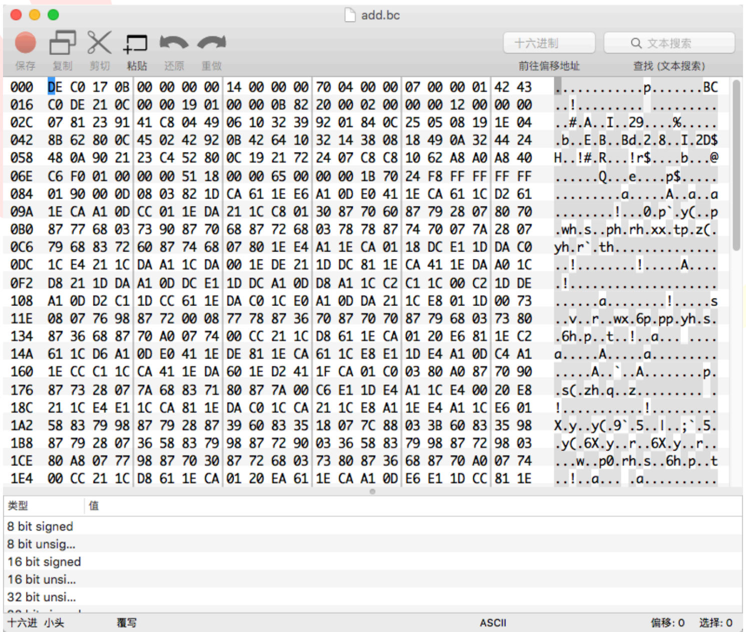
First let’ s write a simple C program to add two numbers. The code is as below:

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

If we save the above program as *add.c*, and then we compile the source code into bitcode:

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

By executing the above commands, we will have *add.bc*. If we use a binary editor to open this file, we can see its content:



Since Bitcode is the binary representation of LLVM-IR, the file is barely readable without knowing its encoding in advance, as in the above figure. Next, we will convert bitcode into text format:

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

If we open *add.ll* in a text editor, we can see the LLVM-IR of function *add* , as follows:

```
; 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()
; It can be noted that such representation requests a lot of variables,
; If you are interested, you may learn more about Static Single Assignment (SSA)
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
}
```

You may now have an intuitive understanding of LLVM-IR by comparing the source code with the annotated LLVM-IR of function *add()*. Next let' s see the workflow of Bitcode.

Workflow

Apple describes the workflow as such: “When you archive for submission to the

App Store, Xcode compiles your app into an intermediate representation. The App Store then compiles bitcode down into the 64- or 32-bit executables as necessary.”

The above workflow can be divided into two processes:

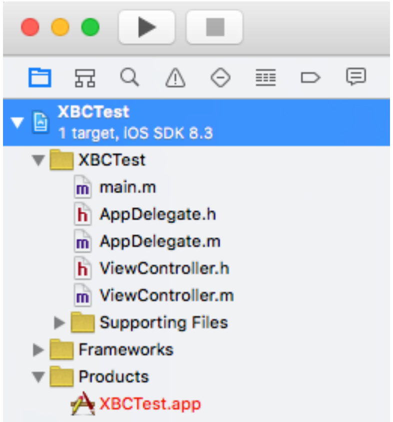
1. When you upload an application to the App Store, Xcode will also upload its Bitcode together.
2. The App Store will recompile Bitcode to executables for users to download.

Next we will split the complete workflow of Bitcode to the following questions and subflows, and explain each of them.

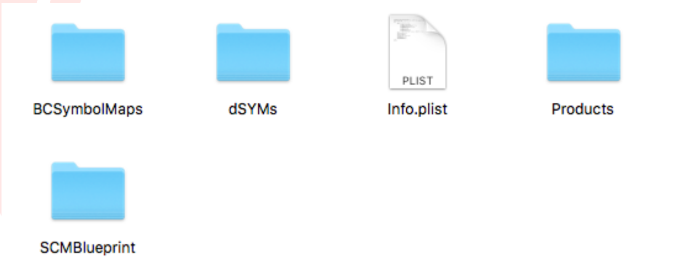
1. Where is Bitcode？
2. Methods to embed Bitcode
3. Ways to generate executables from Bitcode

Where is Bitcode？

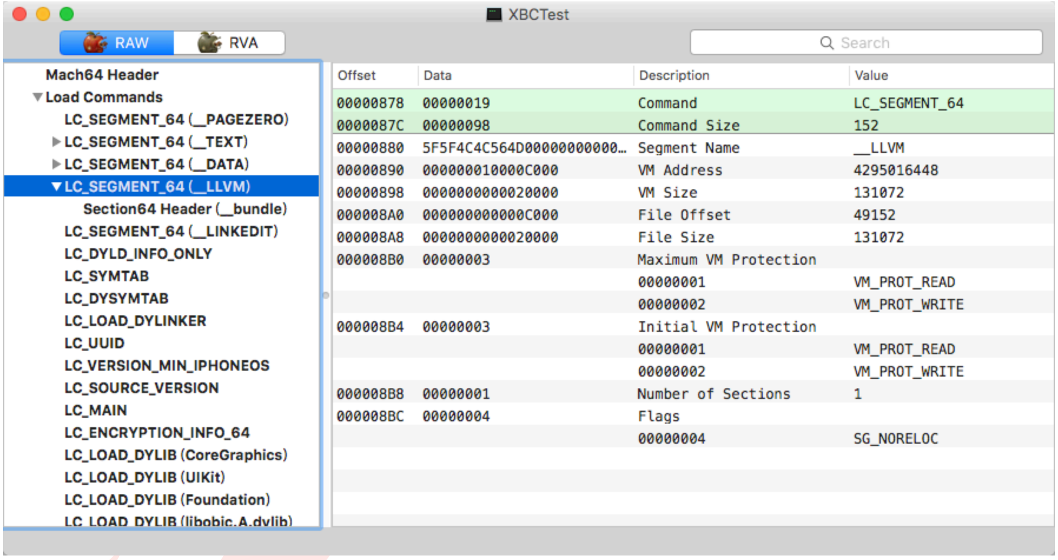
According to Apple’ s description, Bitcode is created only when an application is archived, so a test project is set up:



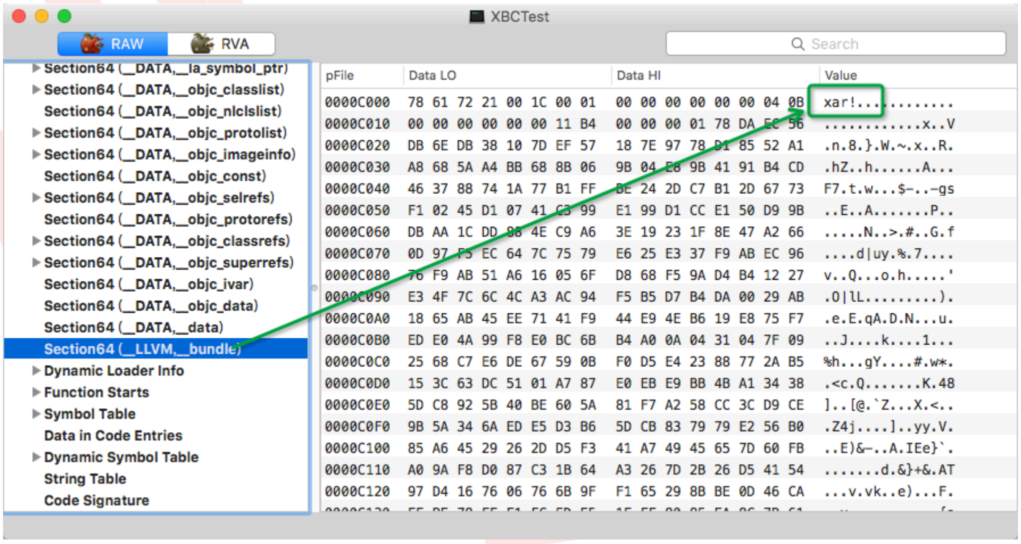
We executed the Archive command, and then checked the package structure generated:



After analyzing, we could not find Bitcode directly from the above directory, so the next step is to check the generated MachO. Opening the MachO file in MachOView, we saw the following results:



It can be seen from the above figure that the Segments and Sections related with LLVM have shown up in the final executable. Let' s go on to check the information on Sections:



As shown in the above figure, what is saved in the Section __bundle is a xar file.

We extracted the xar file, and then use the following command to unpack it:

```
Unpacking：xar -x -f XXX.xar
```

After unpacking, a Bitcode file can be seen.

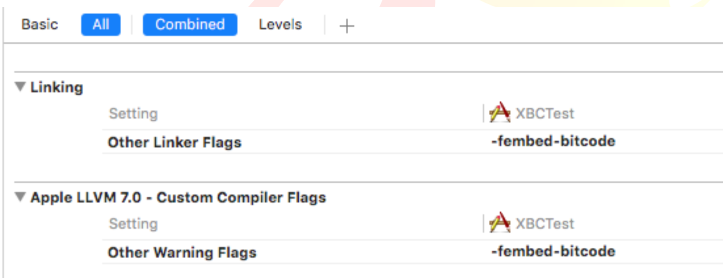
In summary: Bitcode for an application is packed into a xar file by Xcode, and embedded into MachO.

Next let’ s see how to embed Bitcode into MachO.

Methods to embed Bitcode

Method One

A comparison of the compilation parameters when archived or not archived finds that once such parameter as *-fembed-bitcode* is added in the space below, Xcode can also have Bitcode embedded into MachO even in regular compilations:



Method Two

Method One is very convenient, but too much has been done through IDE, which makes it not easy to understand the specific steps. Next we will compile an executable on our own. From source code to executable, there are two processes involved: compilation and linking. To control these two processes, we will cover configuration of the Makefile and arguments that can be used in the processes.

When a Makefile is used to compile an iOS program, there are some common configurations. Below are some common configurations for your reference:

```
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)
SDK_SETTINGS_iOS = $(SYS_ROOT) -I$(SDK_iOS)/usr/include -I$(SDK_iOS)/usr/local/include
MIN_VER_iOS = -miphoneos-version-min=8.0
ARCH_iOS = -arch arm64
```

To take *main.m* for example, we will demonstrate what arguments are needed in compilation:

```
CC_FLAGS_COMMON = -fblocks -std=gnu99 -fobjc-arc -g -fembed-bitcode
CC_FLAGS=-x objective-c $(ARCH_iOS) $(CC_FLAGS_COMMON)
COMPILE_iOS_OBJ=$(CC_iOS) $(MIN_VER_iOS) $(SDK_SETTINGS_iOS) $(CC_FLAGS)

$(COMPILE_iOS_OBJ) -c main.m -o main.o
```

Link *main.o*, *AppDelegate.o*, and *ViewController.o* as arguments for the executable program:

```
LDFLAGS=$(SYS_ROOT) \
        -dead_strip \
        -fembed-bitcode \
        -fobjc-arc -fobjc-link-runtime
LINK_iOS_BIN=$(LD_iOS) $(ARCH_iOS) $(MIN_VER_iOS) $(LDFLAGS)
LDFLAGS_CUSTOM=-framework Foundation -framework UIKit
```

```
$(LINK_iOS_BIN) $(LDFLAGS_CUSTOM) AppDelegate.o ViewController.o main.o -o XBCTest
```

If you make slight modifications on the above Makefile snippet, and include them in one Makefile, then you can embed Bitcode to an executable with Make commands.

Method Three

In this method, we will split the above steps further, specifically as:

source code→Bitcode→xar→executable

source code→Bitcode

In this process, we will compile source code of an iOS application into Bitcode to show the arguments that may apply in *main.m* as an example:

```
CC_FLAGS_COMMON_BC = $(CC_FLAGS_COMMON)
COMPILE_iOS_32_BC = $(CC_iOS) -cc1 -x objective-c $(CC_FLAGS_COMMON_BC) -triple thumbv7-apple-ios8.0.0 -disable-llvm-optzns -target-abi apcs-gnu -mfloat-abi soft $(SYS_ROOT)
COMPILE_iOS_64_BC = $(CC_iOS) -cc1 -x objective-c $(CC_FLAGS_COMMON_BC) -triple arm64-apple-ios8.0.0 -disable-llvm-optzns -target-abi darwinpcs $(SYS_ROOT)

$(COMPILE_iOS_64_BC) -emit-llvm-bc main.m -o main.bc
```

By completing this process, we can obtain three Bitcode files:

1. *main.bc*

2. *AppDelegate.bc*

3. *ViewController.bc*

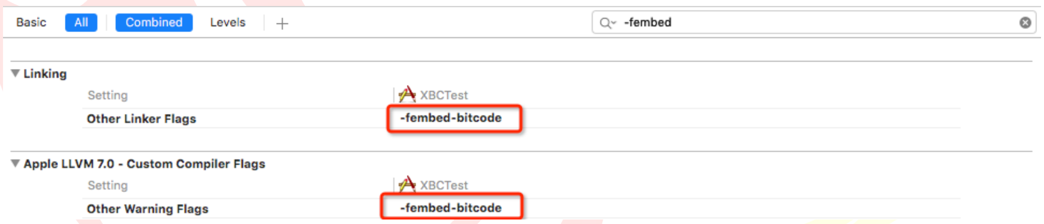
Bitcode→xar

In this step, we will archive the above three Bitcode files into one xar file. There is nothing special about archiving, but one point that needs attention is it should be compatible with the xar generated by Xcode. Specific arguments as below:

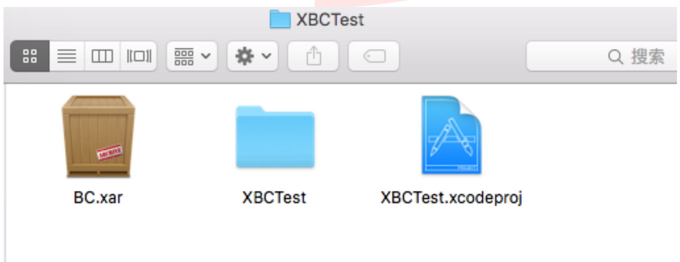
```
Generating : xar --toc-cksum none -c -f BC.xar main.bc AppDelegate.bc ViewController.bc
```

xar→executable

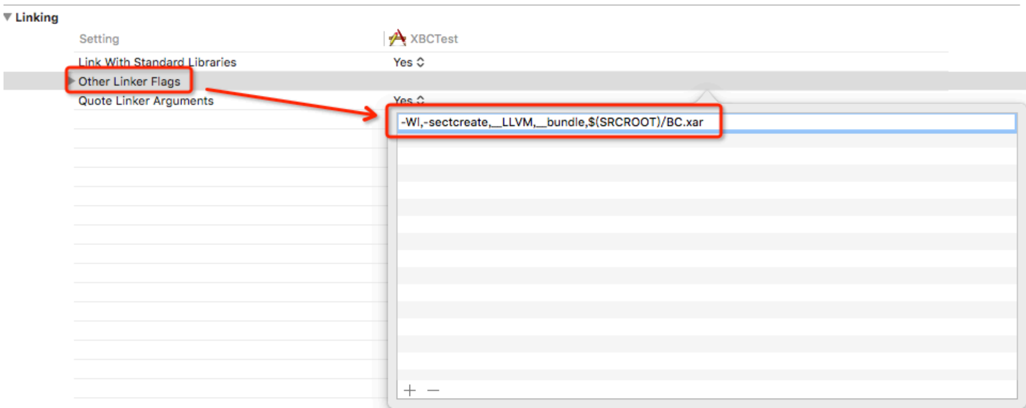
To simplify the process, here we will jump out of the Makefile. Instead, we use Xcode. First, clear all the following compilation arguments:



Copy the newly generated *BC.xar* to the root directory of the test project:



Edit the *Other Linker Flags* in the setting of the project, add: *-Wl,-sectcreate,__LLVM,__bundle,$(SRCROOT)/BC.xar*, as shown below:



View the generated MachO file in the compilation program, you will find that Bitcode has been added to MachO.

The method to embed Bitcode into MachO we introduced relates to the first process: "When you upload an application to the App Store, Xcode will also upload its Bitcode together." Next let's come to the second process.

Ways to generate executables from Bitcode

The second process is: "The App Store will recompile Bitcode to executables for users to download." The second process is conducted on Apple's server, so we are unable to get details. But it should be based on the same toolchain, and we can simulate such a process.

Extracting Bitcode from MachO

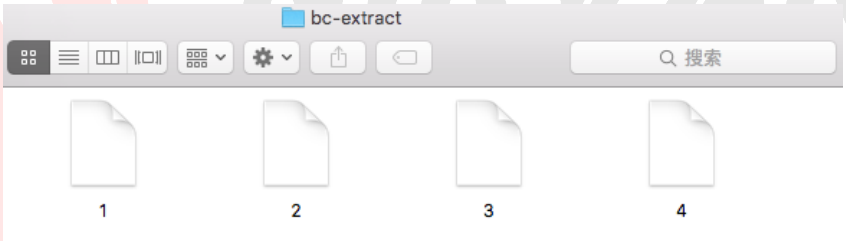
When IPA is uploaded to the App Store, a xar file including Bitcode will be extracted from the MachO file in the IPA. There is a tool named segedit in the toolchain of Xcode that can be utilized to extract Section from MachO. The specific xar arguments are as follows:

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

After extracting the xar, unpack it:

```
Unpacking：xar -x -f Embedded-BC.xar
```

The following Bitcode files are displayed:



We can also use the llvm-dis tool to process the above files into readable formats, so as to understand the content of each file.

Generating executable

With Bitcode, next we need to compile Bitcode into an executable. This can be done in two steps: compile Bitcode into an Object file; link the Object file to an executable.

Compiling Bitcode into an Object file

The Makefile snippet is as below:

```
COMPILE_iOS_BC_2_OBJ=$(CC_iOS) $(MIN_VER_iOS) $(SYS_ROOT) $(ARCH_iOS)

$(COMPILE_iOS_BC_2_OBJ) -c 1 -o 1.o
$(COMPILE_iOS_BC_2_OBJ) -c 2 -o 2.o
$(COMPILE_iOS_BC_2_OBJ) -c 3 -o 3.o
$(COMPILE_iOS_BC_2_OBJ) -c 4 -o 4.o
```

Linking the Object file to an executable

The Makefile snippet is as below:

```
LDFLAGS=$(SYS_ROOT) \
        -dead_strip \
        -fobjc-arc -fobjc-link-runtime
LINK_iOS_BIN=$(LD_iOS) $(ARCH_iOS) $(MIN_VER_iOS) $(LDFLAGS)
LDFLAGS_CUSTOM=-framework Foundation -framework UIKit
$(LINK_iOS_BIN) $(LDFLAGS_CUSTOM) 1.o 2.o 3.o 4.o -o XBCTest
```

As is shown above, we have regenerated an executable XBCTest from Bitcode.

Attack surfaces

Let' s first recall the local workflow of Bitcode: Xcode uploads a MachO embedded with Bitcode to the App Store. Through analysis, we can find there are two problems:

1. Consistency of MachO and Bitcode embedded. In other words: whether Bitcode of Program B can be embedded into Program A.

2. Whether the App Store trusted Xcode, without checking consistency, to allow a malformed MachO to be uploaded to the App Store.

After analyzing the potentially existing problems, we believe that if there are defects in the process and functions of Bitcode, then two targets might be threatened: regular users, Apple.

Regular users

Since Bitcode is transparent to regular users, no direct attacks can be made to them through their weaknesses. But the consistency problem might threaten regular users. Try to imagine: if a Bitcode containing malicious code is embedded in an application A that is submitted to the App Store for review, regular users might download the application with malicious code from the App Store.

We call such attack mode Bitcode Injection. Later there will be more detailed introduction of the implementation of such attacks as well as our test results.

Apple

If a malformed MachO can be uploaded to Apple’ s server, two extra operations are needed compared with the earlier time: to unpack xar; to compile Bitcode.

When something wrong happens in the two processes, the least damage could be a DoS on Apple’ s server, and the most severe outcome could be code execution on Apple’ s server.

In addition, Bitcode is originally a serialized form of LLVM-IR, but LLVM-IR is an intermediate representation, which has never been in direct exposure before. Now it is completely open, and it is a binary format, so it is quite easy to result into problems. The process of Bitcode generating an executable is mainly composed of the following several sub-processes:

1. Code optimization of Target-independent IR
2. Transforming to Target-dependent IR, and Legalizing it
3. Optimization and code generation relevant with platforms

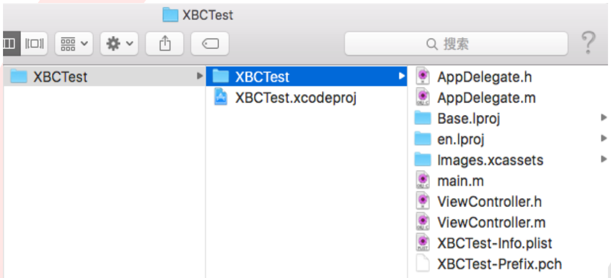
These are supposed to be the inner processes of a compiler. Due to various reasons, traditional tests for compilers are mainly focused on the front-end Parser and Lexer. Now some of the above intermediate or back-end processes are also exposed, thanks to Bitcode. If any problems happen to the above process, the worst outcome would be the code generation of a compiler is controlled.

This is the analysis on attack surfaces, and later we will introduce the lines of thinking to test xar and Bitcode, as well as problems uncovered.

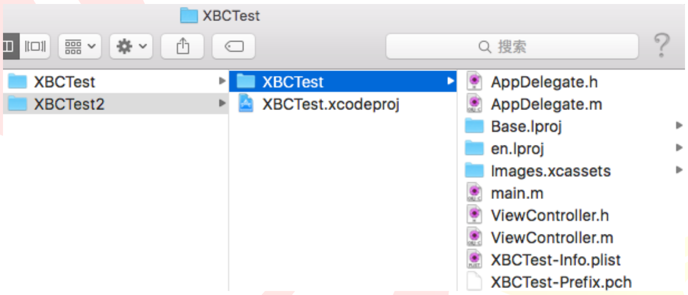
Bitcode Injection

Methods to conduct Bicode injection have been introduced during the description of Bitcode workflow, but they are not concise enough. Here we bring in a simpler approach, and its idea is to make full use of Xcode. The specific realization of such approach includes:

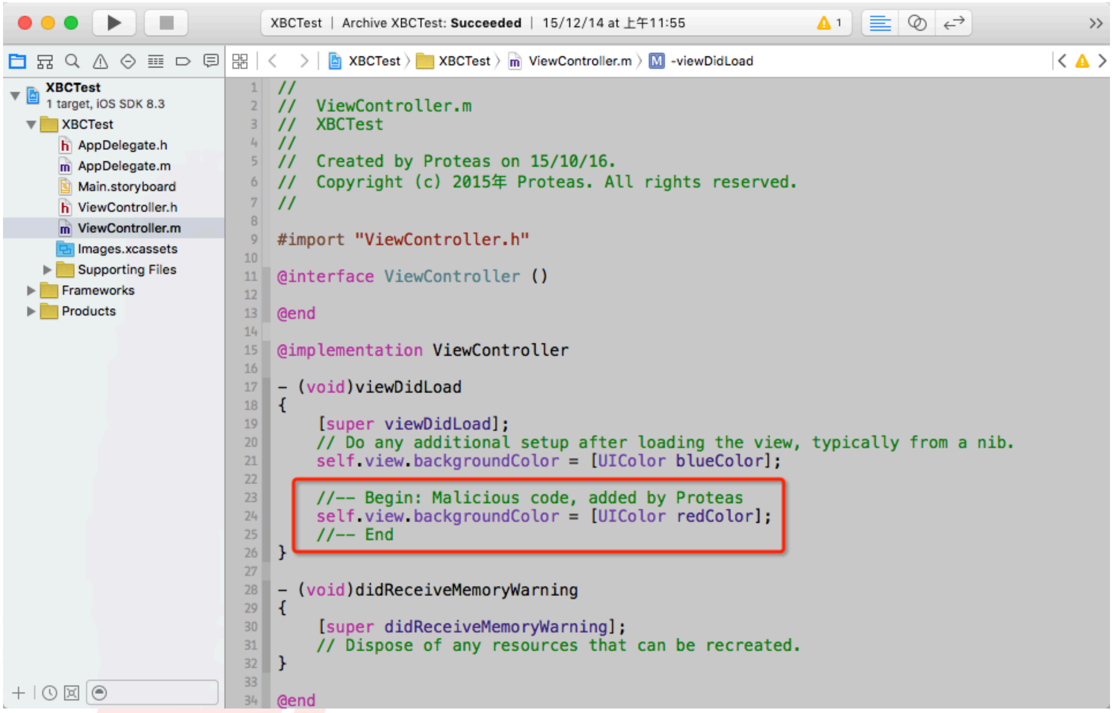
1. Using Xcode to set up a project XBCTest:



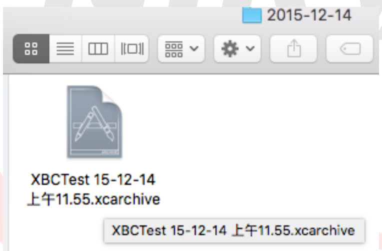
2. Coping project XBCTest, to get project XBCTest2:

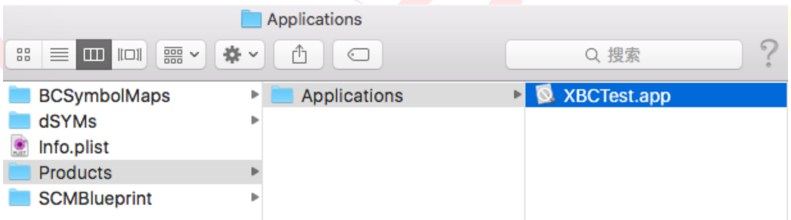


3. Editing the source code of project XBCTest2, and embed in it malicious code:

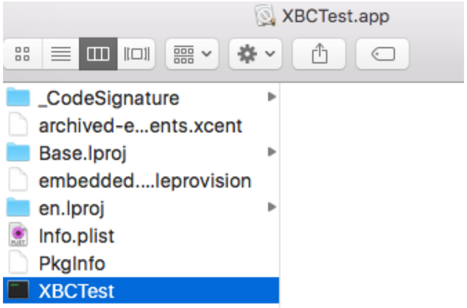


4. Archiving project XBCTest2:



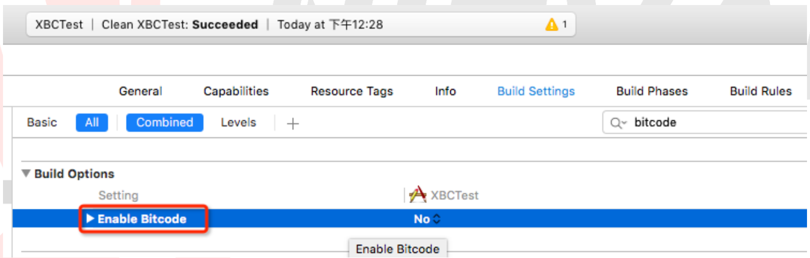


5. Acquiring MachO, and applying segedit to extract xar containing bitcode from MachO:



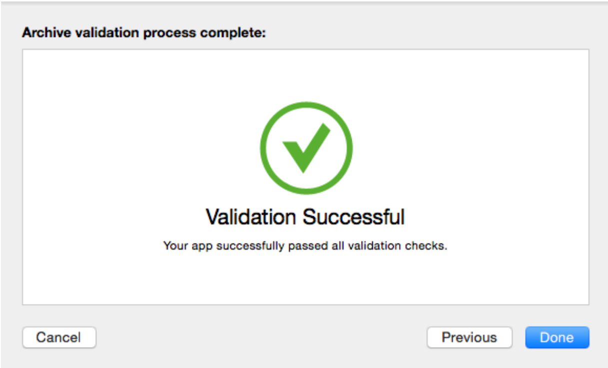

```
提取 Extracting xar: segedit ./XBCTest -extract "__LLVM" "__bundle" BC.xar
```

6. Modifying the linker flag in project XBCTest, and embedding the extracted xar--`BC.xar` into the MachO of project XBCTest.
7. Disabling the Bitcode feature of project XBCTest, and archiving and uploading it to the App Store:

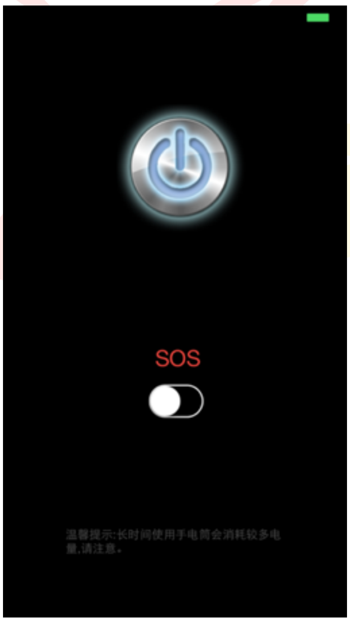


In our test, we have not embedded any malicious code, but instead we found two completely different applications from the internet, and embedded Bitcode of one of them to the MachO of the other, and then submit it to the App Store.

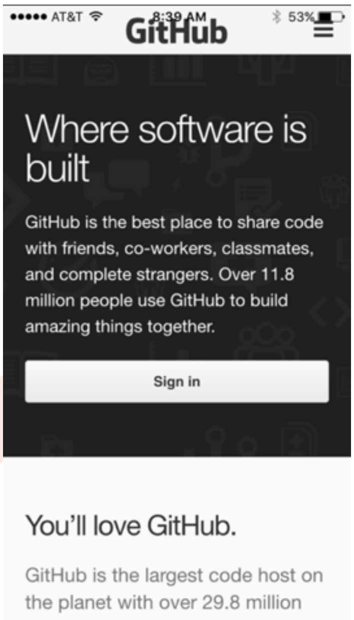
When an application is submitted to the App Store, two aspects are primarily checked: Xcode will do local static parsing; upon submission, Apple’ s server will double-check. However, the applications constructed with Bitcode Injection can pass both checks:



After a long-time verification, our application was rejected. The reason was: our application was not in accordance with the description. In our description, our application should look like this:



But when the verifier of Apple finished installation, the program looked like this:



This proved at least three points:

1. The Bitcode injection method we used is of no problem.
2. The verifier of Apple verifies the program compiled from Bitcode.
3. Consistency is reviewed by human. If the embedded malicious code has not affected UI, there are still possibilities that verification is passed.

Testing xar

Line of Thoughts

Fuzzing xar, and the method to generate data is variation from standard xar files.

Test results

At present, we have mainly fuzzed out some null pointer dereference problems.

Testing clang

Line of Thoughts

Fuzzing the functions from Bitcode to Object in clang, which is also to adopt variation to generate test data.

Test results

Fuzzing clang helps us to uncover some problems in relation with heap corruption.

Summary

1. The Xcode 7 bitcode feature has opened up new huge attack surfaces. Apple should do something to narrow them down, for example: to check whether Bitcode is identical to its MachO file.
2. In the report we have explained in depth the attack surfaces and the test line of thought for each attack surface. We hope they will be helpful for your research on attack surfaces and security associated with Bitcode.

References

[1] What’ s New in Xcode

[2] LLVM Bitcode File Format