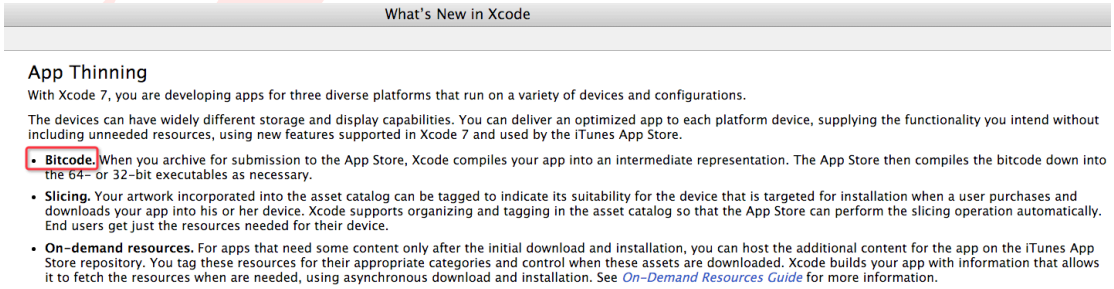
Xcode 7 Bitcode 的工作流程及安全性评估

Proteas of Qihoo 360 Nirvan Team

2015-12-10

简介

随着 Xcode 7 的发布，苹果为 Xcode 增加了一个新的特性 Bitcode [1]：



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.

新的特性往往意味着新的攻击面。本文首先介绍什么是 Bitcode 及 Bitcode 相关的工作流程，在熟悉了 Bitcode 的工作流程后，接下来是评估 Bitcode 相关的攻击面，最后介绍针对各个攻击面的测试方法及目前的测试结果。

什么是 Bitcode

简单来说，Bitcode 是 LLVM-IR 在磁盘上的一种二进制表示形式。关于 Bitcode 详细描述，请参考[2]，这里会用例子来让大家对 Bitcode 有个感性认识。

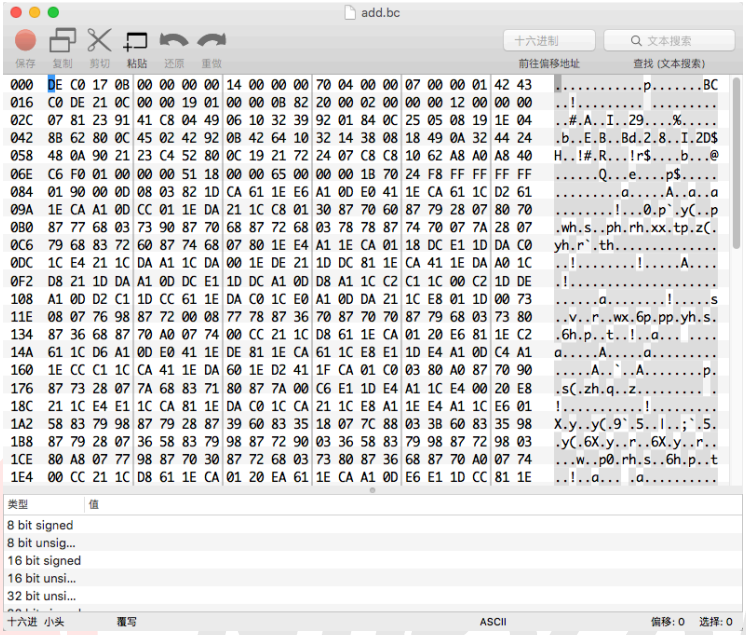
先写一个简单的 C 程序，功能是计算两个数的和，代码如下：

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

将如上程序保存为 add.c，然后我们将源程序编译成 Bitcode：

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

执行如上命令会生成 add.bc，我们使用二进制编辑器打开生成的文件，查看文件内容：



由于 Bitcode 是 LLVM-IR 的二进制表示形式，如上图，在不了解编码方式的前提下基本不可读。下面我们把 Bitcode 转换成文本形式：

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

用文本编辑器打开 add.ll，可以看到 add 函数的 LLVM-IR 内容如下：

```
; 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
; 如下是 add() 对应的 LLVM-IR
; 可以注意到这种表示形式会申请很多变量，
; 感兴趣的同学可以了解下  Static Single Assignment (SSA)
define i32 @add(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4          ; 变量 1，4 字节空间，后续用来存放参数 a
  %2 = alloca i32, align 4          ; 变量 2，4 字节空间，后续用来存放参数 b
  %c = alloca i32, align 4          ; 变量 c，4 字节空间，后续用来存放结果 c
  store i32 %a, i32* %1, align 4    ; 将 a 保存到变量 1 中
  store i32 %b, i32* %2, align 4    ; 将 b 保存到变量 2 中
  %3 = load i32, i32* %1, align 4   ; 将立即数 1 保存到变量 3 中
  %4 = load i32, i32* %2, align 4   ; 将立即数 2 保存到变量 4 中
  %5 = add nsw i32 %3, %4           ; 将变量 3 与变量 4 的和保存到变量 5 中
```

```
  store i32 %5, i32* %c, align 4    ; 将变量 5 保存到结果 c 中
  %6 = load i32, i32* %c, align 4   ; 将结果 c 保存到变量 6 中
  ret i32 %6                        ; 返回变量 6
}
```

对比源码与已经注释过的 add() 函数的 LLVM-IR 表示，大家应该对 LLVM-IR 有个感性认识了，下面我们一起看下 Bitcode 的工作流程。

工作流程

苹果关于工作流程的描述：" 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."

如上的工作流程可以分为两个阶段：

1. 在将应用上传到 AppStore 时，Xcode 会将程序对应的 Bitcode 一起上传。
2. AppStore 会将 Bitcode 重新编译为可执行程序，供用户下载。

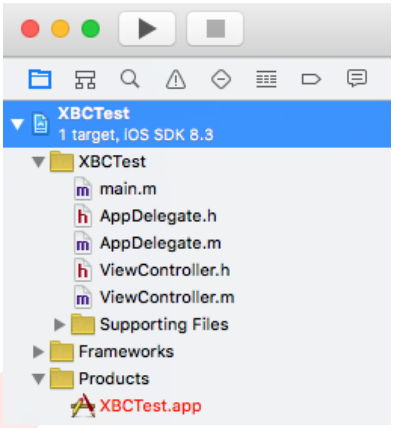
下面会将 Bitcode 相关的完整的工作流程分解为如下几个问题或子过程并分别做说明：

1. Where is the Bitcode？
2. 嵌入 Bitcode 的方法
3. 从 Bitcode 生成可执行程序的方法

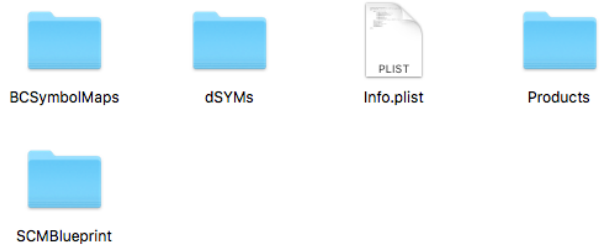
Where is the Bitcode？

参考苹果的描述，只有在 Archive 时才会生成 Bitcode，于是建立了一个测

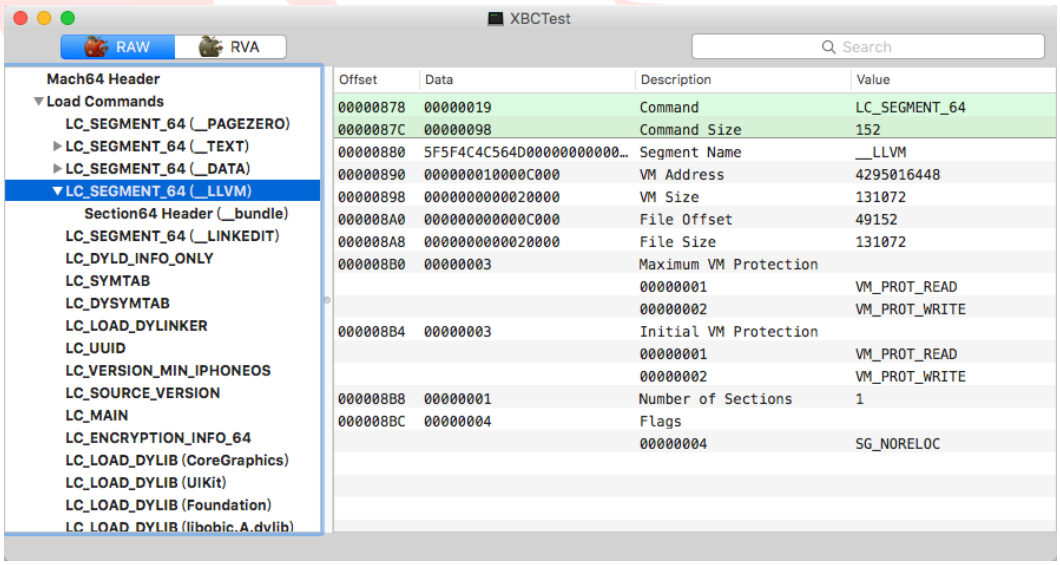
试工程：



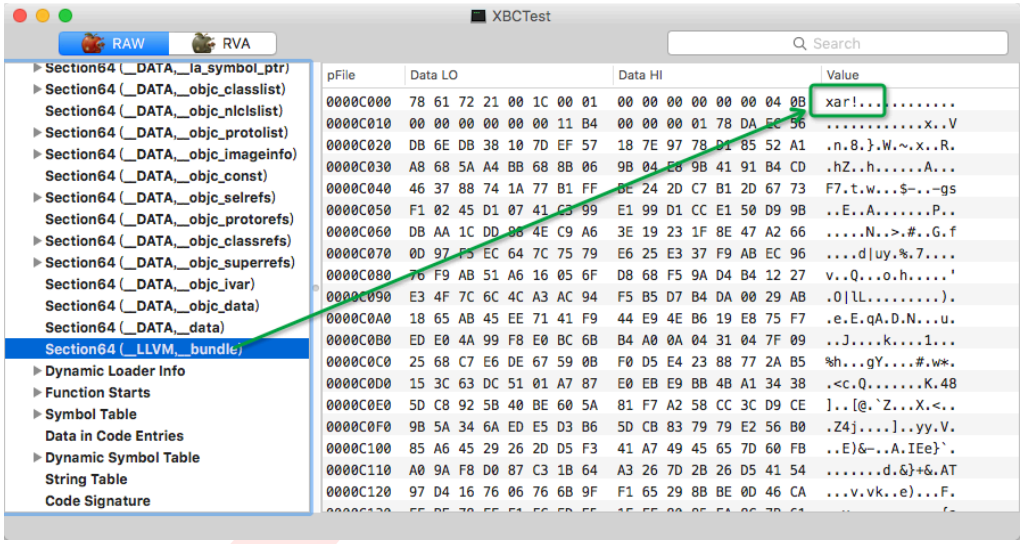
执行 Archive，然后查看生成的包结构：



经过分析在如上的目录中并没有直接找到 Bitcode，接下来检查生成的 MachO。使用 MachOView 加载生成的 MachO，结果如下图：



从上图可以看到最终的可执行程序中多了 LLVM 相关的 Segment 与 Section。继续查看对应的 Section 的信息：



如上图，Section __bundle 中保存的是一个 xar 文档，提取出 xar 文档，然后使用如下命令解开文档：

```
解开：xar -x -f XXX.xar
```

解开后，可以看到 Bitcode 文件。

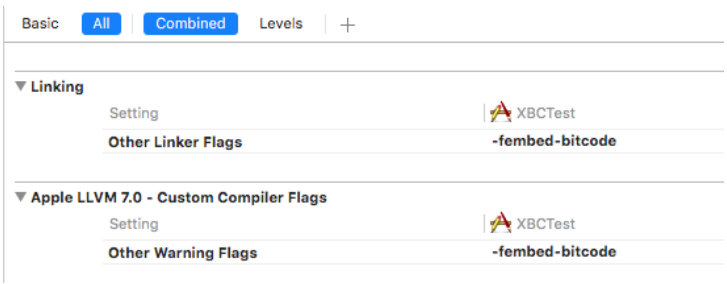
总结 :程序对应的 Bitcode 被 Xcode 打包成 xar 文档，嵌入的 MachO 中。

下面我们看下在 MachO 中嵌入 Bitcode 的方法。

嵌入 Bitcode 的方法

方法一

通过对比 Archive 与非 Archive 时的编译参数，发现只要在如下图所示的位置添加编译参数：-fembed-bitcode，即可让 Xcode 普通编译时也在 MachO 中嵌入 Bitcode：



方法二

方法一虽然很方便，但是 IDE 做了太多工作，不便于理解具体过程，接下来我们自己编译可执行文件。从源代码生成可执行程序主要分为：编译、链接两个过程，为了控制这两个过程，下面会讲解 Makefile 的配置，及这两个过程用到的参数。

在使用 Makefile 编译 iOS 程序时，有些通用的配置，如下的通用配置，供大家参考：

```
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
```

以 main.m 为例说明编译需要的参数：

```
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
```

将 main.o，AppDelegate.o，ViewController.o 链接成可执行程序的参数：

```
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
```

大家把如上的 Makefile 片段稍加修改，整理到一个 Makefile 文件中，就可

以通过 make 命令嵌入 Bitcode 到可执行程序。

方法三

在这个方法中我们会将上面的步骤进一步分解，具体过程为：

源码→Bitcode→xar→可执行程序

源码→Bitcode

在这个过程中我们将 iOS 应用的源码编译成 Bitcode，下面会 main.m 为例来说明使用的参数：

```
CC_FLAGS_COMMON_BC = $(CC_FLAGS_COMMON)
COMPILE_iOS_32_BC = $(CC_iOS) -cc1 -x objective-c $(CC_FLAGS_COMMON_BC) -triple thumb
v7-apple-ios8.0.0 -disable-llvm-optzns -target-abi apcs-gnu -mfloat-abi soft $(SYS_RO
OT)
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
```

完成这个过程后，我们可以得到三个 Bitcode 文件：

1. main.bc
2. AppDelegate.bc
3. ViewController.bc

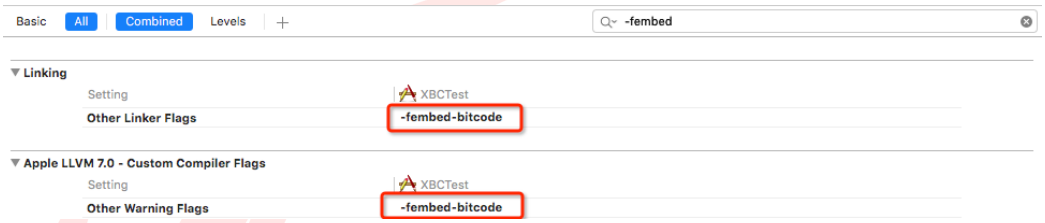
Bitcode→xar

在这一步我们会将如上得到的三个 Bitcode 文件打包到一个 xar 文档中。打包没什么特别，需要注意的是需要与 Xcode 生成的 xar 保持兼容，具体参数如下：

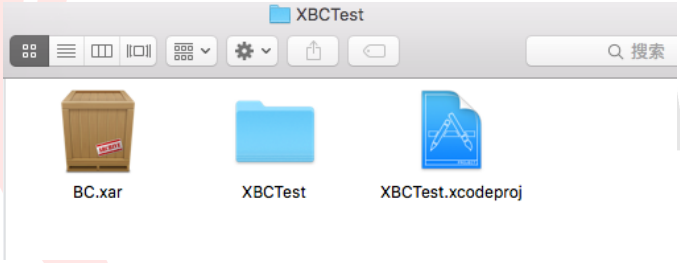
```
生成：xar --toc-cksum none -c -f BC.xar main.bc AppDelegate.bc ViewController.bc
```

xar→可执行程序

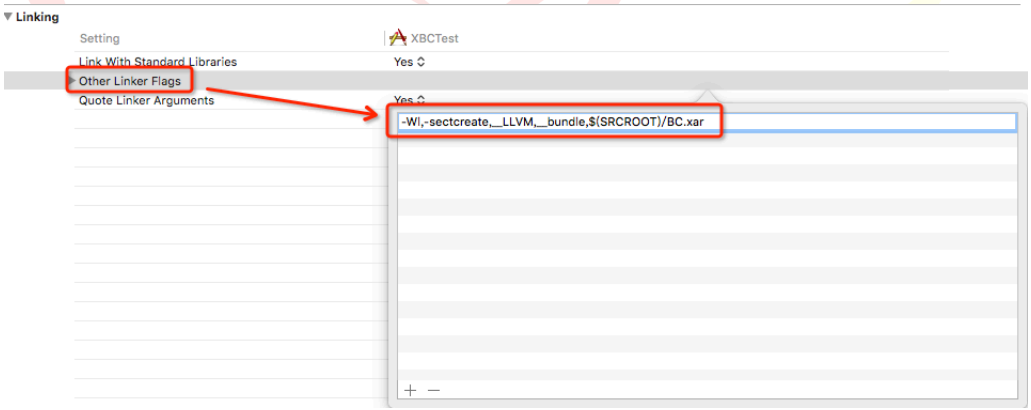
为了简化过程，这里我们会跳出 Makefile，使用 Xcode，首先清除如下的编译参数：



将刚刚生成的 BC.xar 拷贝到测试工程的根目录：



编辑工程设置的 Other Linker Flags，添加：-Wl,-sectcreate,__LLVM,__bundle,$(SRCROOT)/BC.xar，如下图：



编译程序，查看生成的 MachO 文件，可以看到 Bitcode 已经被添加到了 MachO 中。

如上我们介绍了在 MachO 中嵌入 Bitcode 的方法，对应的是第一个过程：

“**在将应用上传到 AppStore 时，Xcode 会将程序对应的 Bitcode 一起上传**”，下面我们说明第二个过程。

从 Bitcode 生成可执行程序的方法

第二个过程为：“**AppStore 会将 Bitcode 重新编译为可执行程序，供用户下载**”。第二个过程是在苹果的 Server 上进行的，我们没法直接获得细节，但是应该都是基于相同的工具链，我们可以模拟这个过程。

从 MachO 中提取 Bitcode

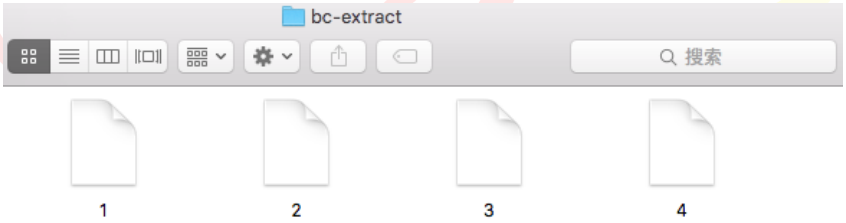
AppStore 拿到我们上传的 IPA 后，首先需要从 IPA 内的 MachO 文件中提取出包含 Bitcode 的 xar 文档。在 Xcode 的工具链中有个工具 segedit 可以用来从 MachO 提取 Section，提取 xar 的具体参数如下：

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

提取到 xar 后，解开 xar：

```
解开：xar -x -f Embedded-BC.xar
```

得到如下几个 Bitcode 文件：



还可以使用 llvm-dis 工具将如上文件处理成可读形式，从而了解每个文件的内容。

生成可执行程序

在有了 Bitcode 后，接下来需要将 Bitcode 编译成可执行程序，分为两个

过程：将 Bitcode 编译成 Object 文件；链接 Object 文件到可执行程序。

将 Bitcode 编译成 Object 文件

Makefile 片段如下：

```
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
```

链接 Object 文件到可执行程序

Makefile 片段如下：

```
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
```

如上我们已经从 Bitcode 重新生成了可执行程序 XBCTest。

攻击面

我们先回顾下 Bitcode 在本地的工作流程：**Xcode** 将嵌入了 Bitcode 的 MachO 上传到 AppStore。通过分析可以发现这里存在两个问题：

1. MachO 与其中嵌入的 Bitcode 的一致性问题。即：能否把 **程序 B** 的 Bitcode 嵌入到 **程序 A** 中。
2. AppStore 是否信任了 Xcode，而没有检查一致性问题，从而允许将

Malformed MachO 上传到 AppStore。

在分析了可能存在的问题后，我们认为如果 Bitcode 流程与功能存在缺陷，便可以对两个目标形成威胁：普通用户、苹果。

普通用户

由于 Bitcode 对普通用户是透明的，因此无法通过其弱点直接攻击用户。但是一致性问题是可能对普通用户造成威胁的，试想：如果提交 AppStore 审核的程序 A 中嵌入了含有恶意代码的 Bitcode，普通用户就有可能从 AppStore 下载到含有恶意代码的程序。

对于这种攻击方式我们将其叫做 Bitcode Injection，下文会详细介绍这种攻击的实施方法，及我们的测试结果。

苹果

如果 Malformed MachO 可以被上传到苹果的服务器，苹果的服务器相较于之前，主要需要进行两个额外的操作：解开 xar；编译 Bitcode。如果这两个过程出现问题，轻则可以在苹果的服务器上造成 DoS，重则可以在苹果的服务器上造成任意代码执行。

另外，Bitcode 原本是 LLVM-IR 的一种序列化形式，而 LLVM-IR 是一种中间形式，之前没有被直接暴露出来，现在完全开放了，而且又是二进制格式，这是很容易出问题的。从 Bitcode 生成可执行文件的过程主要由如下几个子过程组成：

1. 基于平台无关的 IR 的代码优化。

2. IR 的平台相关化、合法化。

3. 平台相关的优化、代码生成。

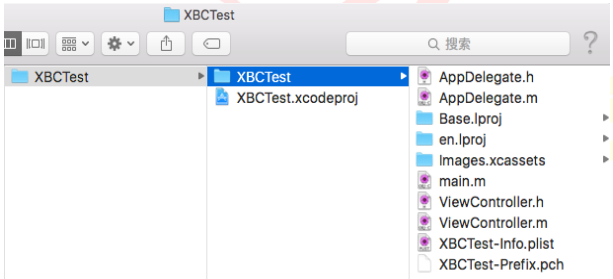
这些原本是编译器的内部过程，由于各种原因，传统的对编译器的测试主要集中在前端的 Parser 与 Lexer，现在借由 Bitcode 如上的一些中间或者后端过程也暴露了出来，如果如上的过程出现问题最糟糕的结果是可以控制编译器的指令生成。

以上是关于攻击面的分析，后文会介绍测试 xar 及 Bitcode 的思路，以及发现的问题。

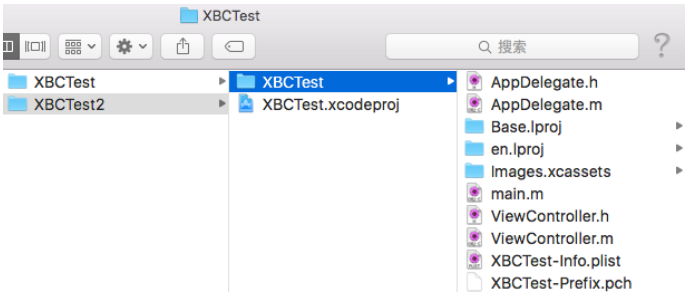
Bitcode Injection

上文在介绍 Bitcode 工作流程时已经介绍了实施 Bitcode Injection 的方法，但是上面提到的方法不够简练，这里我们再介绍一种更简单的方法，主要的思路就是最大限度的利用 Xcode，这个方法的具体实施步骤为：

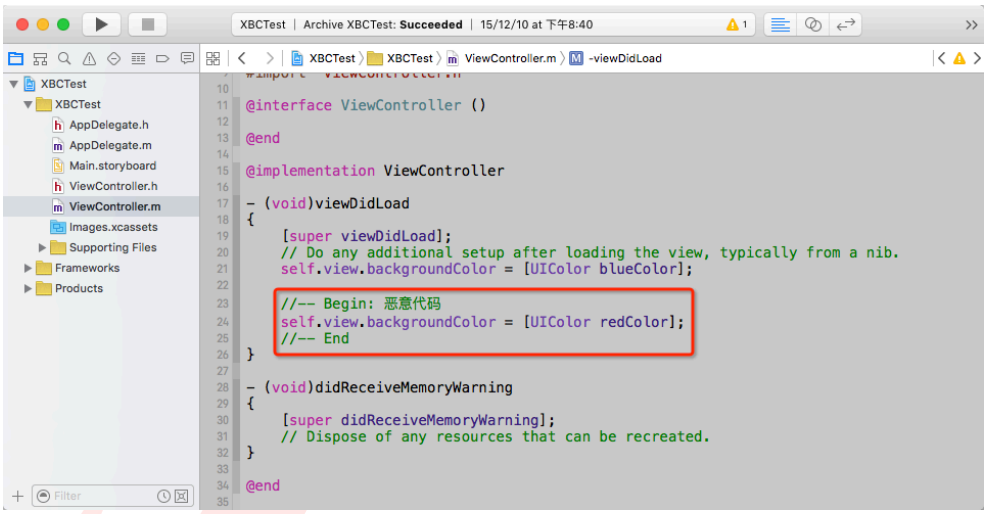
1. 用 Xcode 建立工程 XBCTest：



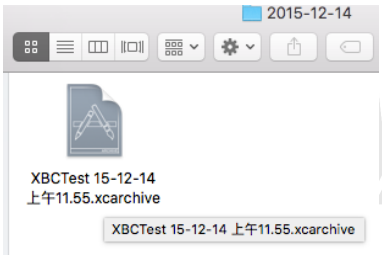
2. 复制工程 XBCTest，得到工程 XBCTest2:

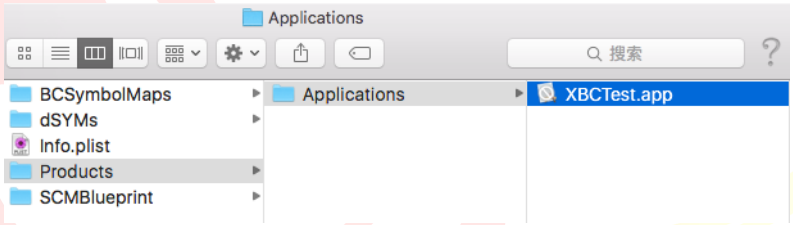


3. 修改工程 XBCTest2 的源码，嵌入恶意代码:

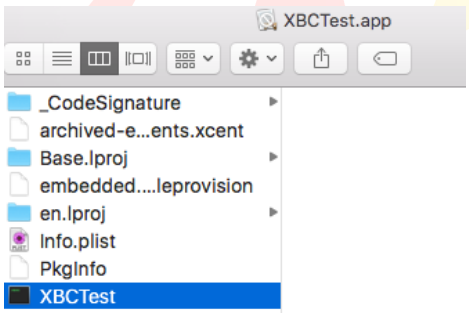


4. Archive 工程 XBCTest2：





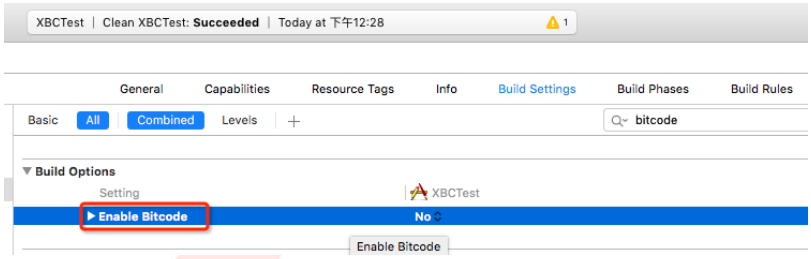
5. 获得 MachO，利用 segedit 从 MachO 中提取出含有 Bitcode 的 xar:



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

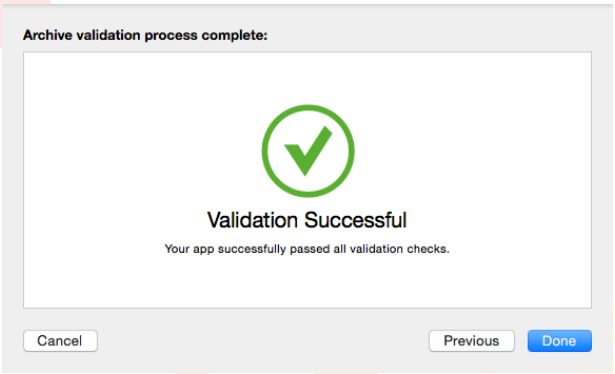
6. 修改工程 XBCTest 的链接标记，将提取的 xar: `BC.xar` 嵌入到工程 XBCTest 的 MachO 文件中。

7. 禁用工程 XBCTest 的 Bitcode 特性，Archive 并上传 AppStore:



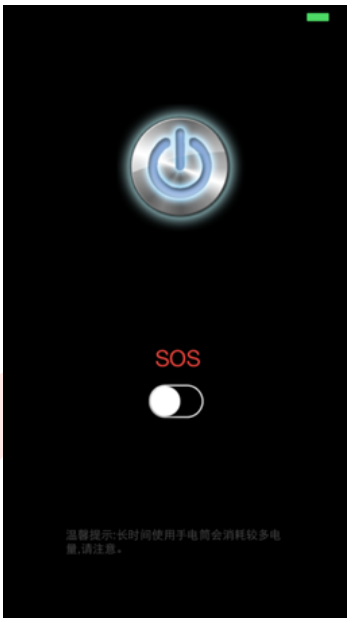
我们在测试的过程中并没有嵌入恶意代码，而是从网上找个两个完全不同的应用，将其中一个的 Bitcode 嵌入到另一个的 MachO 中，并提交到 AppStore。

在将应用提交到 AppStore 的过程中主要会进行两个方面的检查：Xcode 在本地进行静态分析；提交后，苹果的服务器还会进行检查。但是使用 Bitcode Injection 构造的应用可以通过这两项检查：

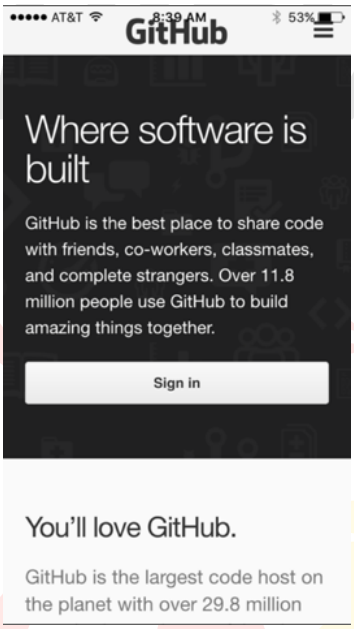


经过漫长的审核后，我们的应用被拒了，理由是：我们的应用与描述不符。

在描述中我们的应用应该长成如下样子：



但是苹果的审核人员安装后，程序却长成这个样子：



这至少可以说明三个问题：

1. 我们使用的 Bitcode Injection 方法没有问题。
2. 苹果的审核人员审核的是从 Bitcode 编译出来的程序。
3. 一致性是靠人肉区分的。如果嵌入对 UI 没有影响的恶意代码，还是有可能绕过审核的。

测试 xar

思路

对 xar 进行模糊测试，生成数据的方法是基于标准的 xar 文档进行变异。

测试结果

目前主要 Fuzz 出一些空指针解引用问题。

测试 clang

思路

对 clang 中的 Bitcode 到 Object 的功能进行模糊测试，也是采用变异的方法生成测试数据。

测试结果

通过对 clang 的 Fuzz 我们发现了一些堆损坏相关的问题。

总结

1. The Xcode 7 bitcode feature has opened a huge attacking surface, Apple should do something to narrow it, for example: checking the bitcode is identical to the related MachO file.
2. 在上文中我们详细介绍了所考虑到的攻击面，及针对每个攻击面的测试

思路，希望这些对大家研究 Bitcode 相关的攻击面及安全性有帮助。

参考资料

[1] What’ s New in Xcode

[2] LLVM Bitcode File Format