

iOS 二进制文件补丁技术简介（第一部分）

Email:root@obaby.org.cn

翻译：obaby

<http://www.h4ck.org.cn>

作为我的另外一个身份—法政科学家，需要经常来破解一些程序。当在处理一些高等级的案子的时候，并不是总是那么简单的可以从文件系统中获取到相关的数据；这将会变得更加困难，尤其是数据经过加密或者混淆的时候。因而，在这种情况下，对整个文件系统进行克隆，并且执行一个法律允许的破解（forensic hacking）会使情况变得更加简单；这通常会有很多种办法来是一个应用程序展开自己的内部数据并且暴露出来。我们同样会作为一个安全审计院，测试员或者黑客来测试金融/银行、政府、或者其他客户的一些隐私数据安全，我们会针对这些数据或者应用程序（有时候是服务器）进行攻击（可控情况下）来看系统是不是最终安全，多半情况下，我们会在被测试程序中发现一些非常严重的安全漏洞。有时候我会发现许多程序的加密算法实现千疮百孔，允许我攻击这个加密算法的实现方式，而不必对整个应用进行解密（这个将会更加困难）。

这里有很多的不同的方法来篡改一个 iOS 应用。我在我的一本书中（<<[Hacking and Securing iOS Applications](#)>>）介绍过其中的一些方法。最常用的（简单的）方法就是使用 gdb 或者 cycript 之类的工具来在执行过程中对 Object-C 的执行代码进行篡改，这个我已经在 Black Hat 2k12 上进行了阐述（[幻灯片](#)）。这个是非常容易实现的，因为整个运行时只经过极少数的 C 函数。这样就变得可以非常简单的来劫持一个应用程序的执行流程，或者创建自己的对象，或者替换程序内的方法。很多的时候单独的考虑运行时状态并不能够让你从一个应用程序中得到想要的结果。我在我的书中写到的安全性最差的例子，在启动的时候在登录界面通过一个简单的调用 [OneSafeAppDelegate userIsLogged:]，就加载了所有的数据，并且加密方法非常简单。因而单独的考虑运行时只能将你带到这么远，cycript 只能在方法级别使用。如果你需要重载一个方法内部的一些逻辑，你需要借助一个调试器来实现。调试一个程序可以给你更多的控制权，但是这同样是一个交互的过程；你需要在每次进行分析的时候都要准备好自己的调试器和调试进程，并且处理程序的相关修改（或者可以写一些脚本来实现）。开发者现在也同样在检测越狱和反调试技巧上变得更加的聪明，也就是说，你需要去穿越这些保护才能到达你想要的地方。

这就是为什么今天要讲的二进制文件修改的意义。其中好处是二进制文件的修改可以直接修改程序的内部逻辑，而不必在调试过程中每次都要修改。通过修改程序自身的代码，你可以重写整个应用。这同样可以使你达到一个机器指令级别，并且可以控制寄存器，参数，比较操作，或者一些其他的逻辑。二进制文件修改在反盗版系统领域已经有很长的历史了，但是这同样适用于司法取证或者安全审计领域。如果我能找到一个方法修改程序来提供给我访问特定证据的权限，那么我可以拷贝这个二进制文件到原来的系统（如果需要）来释放相关的证据，或者将携带已经修改过的设备交给调查员。对于安全审计来说，我可以提供一个修改过的二进制文件，并且附带一份报告来告诉他们怎么破解他们的应用以及怎么样来加强安全性。

修改二进制程序通常需要对程序进行二次签名，因为在修改的过程中会破坏苹果已经添加的签名信息。

通常人们会将程序拷贝到一个已经越狱的设备上，一些越狱的设备并没有完全的禁用签名，这时候就需要使用 `ldid` 或者 `codesign_allocate` 之类的工具进行重签名。将一个开发者的二进制文件进行修改之后重新使用另外的密钥进行签名也是可以的。这是一个黑客最常用的修改方法，来使应用程序运行在越狱的设备上。

现在有很多的工具可以用来修改一个二进制文件，你可以使用免费的工具或者可以花\$5,000 来使用一个非常好的工具。这篇文章将会介绍怎么来完成这样工作，并且也会涉及到怎么使用一系列工具来使工作变得更简单。文章中的大部分内容也适用于修改其他应用程序（例如 x86 架构下的 Mac OSX 应用），因为他们同样是 Mach-O 文件结构（但是指令集是不一样滴）。

使用下面的代码作为例子，这段代码调用了另外一个函数并且随后判断返回值。这类函数在应用中随处可见。它可以用来检测设备是否越狱，或者确保帐户有足够的余额进行转账，或者验证用户的密码（最好不要这样）。这是一个草莓味的例子，(HOHOHO)：

```
#include <stdlib.h>
int test_condition() {
    return 0xff;
}
int main() {
    int result = test_condition();
    if (result) {
        exit(0);
    }
}
```

代码非常简单，调用了一个 `test_condition()` 函数，并且返回了 `0xff` (255)。然后检测返回的数值是不是 0。这个和应用检测是否越狱之类的返回值结果检测是类似的。

如果要编译这个 iOS 应用你可以使用 `llvm-gcc`。在执行之前确保你已经安装 `xcode` 和命令行工具：

```
$export
SDK_PATH=/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS6.1.sdk
$export
ARM_PATH=/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/
$ ${ARM_PATH}/arm-apple-darwin10-llvm-gcc-4.2 -o test1 test1.c -isysroot ${SDK_PATH}
```

现在来看下生成的二进制文件，可以通过 `otoolmingl` ”

```
$ otool -tV test1
```

```
_main:
 00002fa8 e92d4080 push {r7, lr}
 00002fac e1a0700d mov r7, sp
 00002fb0 e24dd00c sub sp, sp, #12
-00002fb4 e59f0034 ldr r0, [pc, #52]
| 00002fb8 e58d0000 str r0, [sp]
| 00002fbc ebfffff0 bl _test_condition <- test_condition()
| 00002fc0 e58d0004 str r0, [sp, #4]
| 00002fc4 e59d0004 ldr r0, [sp, #4] <- result (0xff)
| 00002fc8 e59d1000 ldr r1, [sp] <- comparison (0)
| 00002fcc e1500001 cmp r0, r1 <- subtract the two
| 00002fd0 1a000000 bne 0x2fd8 <- go if nonzero
| 00002fd4 ea000001 b 0x2fe0 <- otherwise, go here
| 00002fd8 e3a00000 mov r0, #0 <- load exit value
| 00002fdc eb000004 bl 0x2ff4 @ symbol stub for: _exit
| 00002fe0 e5170004 ldr r0, [r7, #-4]
| 00002fe4 e1a0d007 mov sp, r7
| 00002fe8 e8bd4080 pop {r7, lr}
| 00002fec e12ffff1e bx lr
-00002ff0 00000000 andeq r0, r0, r0
```

输出的内容非常的直观，因为我没有使用任何的优化，也没使用 thumb 指令。在 `test_condition` 调用之前，一个 0 已经被存在了栈上。这就是我们在下面的返回值要比较的数值。从 `0x2fbc` 开始主函数跳转到 `test_condition` 分支。这个和 x86 架构下的 `call` 是等价的。在返回后，在 `0x2fcc` 进行返回值和 `r1` 进行比较。可以通俗的这么理解，compare 0xff and zero，下一条指令就是 what to do if... 实际情况是在比较过程中将两个寄存器的值相减。Bne 指令(branch if not equal)会紧随上面的比较，并且依法一个跳转到 `0x2fd8`，如果比较的结果是非零值（例如 return code!=0）。在地址 `0x2fd8`，可以看到 `r0` 加载了一个退出代码，并且执行退出。紧跟 bne 指令之后只另外一个分支 `0x2fe0`，这个分支对应的是 `test_condition` 返回值为 0 的情况。

现在可以来看下校验函数：

```
_test_condition:
 00002f84 e24dd008 sub sp, sp, #8
- 00002f88 e59f0014 ldr r0, [pc, #20] <- load 0xff into r0
| 00002f8c e58d0000 str r0, [sp]
| 00002f90 e59d0000 ldr r0, [sp]
| 00002f94 e58d0004 str r0, [sp, #4]
| 00002f98 e59d0004 ldr r0, [sp, #4]
| 00002f9c e28dd008 add sp, sp, #8
| 00002fa0 e12ffff1e bx lr <- return
- 00002fa4 000000ff .long 0x000000ff <- our data (0xff)
```

这个同样是非常直接的代码，因为在编译过程中没有任何的优化。在 `0x2f88` 指令加载了内存地址 `[program counter + 20]` 处的数值到 `r0`。这个值就是我们直接保存在二进制文件中的数据 0xff。在几条

指令之后，地址 **0x2fa0** 处，执行到 lr 分支 (link register)，在这个寄存器中保存了调用方的返回值。剩余的脏指令，都是编译器自动生成的，并且这些指令对我们来说是没有用的。它首先会在堆上申请 8 byte 内存，并且对它进行读写操作，然后这个堆在返回之前会被清除掉。

值得注意到东西：ldr 指令在 main 和 test_condition 函数中都是应用的 program counter (pc)，在此基础之上加上偏移 (offset)。在这条指令执行的时候，pc 等价于函数的起始地址+8 byte (两条指令长度)。如果二进制文件在编译的时候使用-mthumb 参数，那么 pc 等价于函数起始地址+4 byte (也是两条指令长度)。在我们的实例中，test_condition， $pc + \#20 = 0x2f88 + 8 + 20 = 0x2fa4$ (我们没有使用 thumb 进行编译)。

如果理解了上面的内容那么要想进行代码攻击就变得非常简单了。下面的可能方法会立马显示在脑海中：

- 因为我的的测试值是硬编码的，我们可以在二进制文件中将 0xff 修改为 0x00
- 我们可以修改 test_condition 函数返回值为 0
- 我们可以逆转在 main 函数中的逻辑校验
- 我们可以 nop 掉第一个分支 (到 exit)，让程序执行到我们想要的分支

同样还有其他的方法来攻击和修改这些代码，但是上面的四个是对于攻击这样的代码逻辑最常用的修改方式。我们将会对此逐一介绍：

计算文件偏移

不管我们用哪种方式修改文件，在这之前我们都需要知道要修改的二进制文件的地址。虽然 otool 可以给我们一个偏移量，但是这些偏移量并不是磁盘上的文件的真正地址。他们反映的是加载到内存中的二进制文件的偏移量。为了将这个偏移量转换为文件偏移量，我们需要将文件偏移 (file offset) 映射到内存偏移 (memory offset)。

为了创建这个映射，还是要用到 otool。Mach-o 文件格式包含了一系列的加载命令。这些加载命令告诉动态连接器将二进制文件的区段映射到虚拟内存的什么地址空间。这个程序的代码可以在二进制文件的 `__text` 区段找到。为了列出区段不同的的加载命令，使用 `otool -l` 参数

```
$ otool -l test1
```

```
Section
```

```
sectname __text
segname __TEXT
addr 0x00002f0c
size 0x000000e8
offset 7948
align 2^2 (4)
reloff 0
nreloc 0
flags 0x80000400
reserved1 0
reserved2 0
```

```
...
```

文件的偏移量是 **7948**，区段的起始地址是 **0x2f0c**。现在回到上面的反汇编内容中看看 **0xff** 在内存中的地址：

```
00002fa4 000000ff .long 0x000000ff <- our data (0xff)
```

如果你将调试器挂载到这个程序上，你将可以刚好看到这个数值。

```
# gdb -q ./test1
Reading symbols for shared libraries . done
(gdb) b main
Breakpoint 1 at 0x2fb4
(gdb) r
Starting program: /private/var/root/test1
Reading symbols for shared libraries + done
Reading symbols for shared libraries .....
```

```
Breakpoint 1, 0x00002fb4 in main ()
(gdb) x/w 0x2fa4
0x2fa4 <test_condition+32>: 0x000000ff
(gdb)
```

并且如果你喜欢的话可以修改这个数值：

```
(gdb) set *(char *) 0x2fa4 = 0
(gdb) x/w 0x2fa4
0x2fa4 <test_condition+32>: 0x00000000
```

为了计算 **0x2fa4** 在文件中的偏移量，你可能需要会那么一点点的数学。从我们要获取的数据的起始地址减去区段的起始地址，然后加上文件偏移就得到了对应的文件中的地址。所以：

$$\begin{aligned} \text{File Offset} &= (\mathbf{0x2fa4} - \mathbf{0x2f0c}) + 7948 \\ &= 152 + 7948 \\ &= \mathbf{8100} (\mathbf{0x1fa4}) \end{aligned}$$

```
$ hexdump -s 0x1fa4 -n 4 test1
0001fa4 ff 00 00 00
```

这就是我们想要的地址了，并且由于 iOS 是 little endian 模式，数值是被倒序存放的。在十六进制编辑器中修改文件偏移 **0x1fa4**，然后保存。

```
$ hexdump -s 0x1fa4 -n 4 test1
0001fa4 00 00 00 00
```

现在当程序运行的时候就会加载这个数值而不会加载 **0xff**。后面的比较指令将会为 **0**，而不是以前的非零值。当这个发生之后后面的 **bne** 指令就会失败，所以程序就会执行第二个分支。恭喜！你刚刚完成了修改一个逻辑校验的二进制文件！

控制返回值

现在你已经熟悉怎么对磁盘上的文件进行修改，以及怎么通过区段地址和内存地址来计算文件偏移了。但是大多数的程序不会使用硬编码数据，所以，单是知道怎么修改硬编码的数据并不能让你走的更远。

我们讨论的第二个技术是攻击并且修改 **test_condition** 的返回值。首先不用考虑他是怎么得到这个数值的，我们只需要让这个函数始终返回 **0**，并且程序不会立即退出即可。让我们再来看一下这个 **test_condition** 函数：

```

_test_condition:
00002f84 e24dd008 sub sp, sp, #8
- 00002f88 e59f0014 ldr r0, [pc, #20] <- load 0xff into r0
| 00002f8c e58d0000 str r0, [sp]
| 00002f90 e59d0000 ldr r0, [sp]
| 00002f94 e58d0004 str r0, [sp, #4]
| 00002f98 e59d0004 ldr r0, [sp, #4]
| 00002f9c e28dd008 add sp, sp, #8
| 00002fa0 e12fff1e bx lr <- return
- 00002fa4 000000ff .long 0x000000ff

```

在 arm 处理器架构下，函数的返回值保存在 R0 中，这个操作是由 **0x2f88** 完成的。就像我们讨论的其余的代码都是白痴处理器自动生成的代码，可以不用考虑。在地址 **0x2f84**，在堆上 8 byte 内存被申请，并且在 **0x2f9c** 被释放。我们需要确保如下的情形：我们在替换开始的 sub 函数的时候既不会释放堆，也不会在开始的时候申请堆。我们想要的目标就是将 R0 寄存器值修改为 0，然后通过 lr 寄存器返回即可：

```

mov r0, 0
bx lr

```

虽然我们知道指令是这个样子，但是我们却不能直接将这些指令写入到二进制文件中。下一步就是获取这些指令的十六进制编码。为了完成这个工作，我们需要用到 llvm-mc。这个东西是和 llvm 编译器一起的，并且可以通过 llvm 项目网站 (llvm.org) 免费下载。通过 llvm-mc 你可以轻易的获取到你想要的任何指令的编码：

```

$ echo "mov r0, 0; bx lr" | llvm-mc -assemble -triple=armv7 -show-encoding
.text
mov r0, #0 @ encoding: [0x00,0x00,0xa0,0xe3]
bx lr @ encoding: [0x1e,0xff,0x2f,0xe1]

```

你同样可以获取 thumb 指令对应的编码：

```

$ echo "mov r0, 0; bx lr" | llvm-mc -assemble -triple=thumbv7 -show-encoding
.text
mov.w r0, #0 @ encoding: [0x4f,0xf0,0x00,0x00]
bx lr @ encoding: [0x70,0x47]

```

现在可以使用我们的新指令重写 test_condition 函数了。你已经非常熟悉怎样计算文件偏移了。Test_condition 起始于 **0x2f84**，使用同样的方法计算：

$$\begin{aligned}
 \text{File Offset} &= (0x2f84 - 0x2f0c) + 7948 \\
 &= 120 + 7948 \\
 &= 8068 (0x1f84)
 \end{aligned}$$

```
$ hexdump -s 0x1f84 -n 8 test1
0001f84 08 d0 4d e2 14 00 9f e5
```

你可以看到在 0x1f84 有 8byte 完美的匹配到了反汇编数据的内容,但是,这个东西是 little endian 编码的。

```
00002f84 e24dd008 sub sp, sp, #8
00002f88 e59f0014 ldr r0, [pc, #20]
```

使用十六进制编辑器,将生成的数据小心的替换到对应的地址。这个函数剩下的指令就没什么意义了,因为在下面的代码之前我们的代码已经返回了,永远不会执行到。关键是要有足够的空间来防止我们的替换代码。

提示: llvm-mc 提供的编码已经按照处理器的编码格式转换,所以这里没有必要进行倒序了。方便!

现在你的二进制文件看起来应该是这样的:

```
$ hexdump -s 0x1f84 -n 8 test1
0001f84 00 00 a0 e3 1e ff 2f e1
```

如果你使用 otool 再次进行反汇编,那么 test_condition 函数的前两行指令应该是下面的样子:

```
_test_condition:
00002f84 e3a00000 mov r0, #0
00002f88 e12ffffe bx lr
```

再次说明,我们无需关心那些剩余的执行不到的指令。现在当程序执行的时候他会立即加载一个 0,然后返回。下面的比较函数将会等于零而导致 bne 指令被忽略掉,并且逆序执行后面的指令!

逻辑反转

修改函数的返回值在当有非常多的函数调用到这个函数的时候是非常有效的,并且你希望所有的返回值都是一样的。在有些情况下,你会发现你只是希望一部分函数的返回值被改掉,而剩余的则能继续按照原来的方式运行。在这种情况下,更好的方式就是改变调用方的逻辑。例如我们的代码是这个样子:

```
if (result) {
    exit(0);
}
```


这个是由 `cmp` 和 `bne` 指令来实现的，通过比较返回值和 0，如果不相等就跳到分支执行。

```
if (result) {
    exit(0);
}
```

```
00002fcc e1500001 cmp r0, r1 <- subtract the two
00002fd0 1a000000 bne 0x2fd8 <- go if nonzero
```

`Bne` 指令有条相反的指令 `beq`。`Beq` 指令 (branch if equal to zero) 导致程序在比较结果为 0 的情况下跳转到分支执行，类似于：

```
if (! result) {
    exit(0);
}
```

换句话说，如果我们知道比较会失败，那么我们就能通过修改比较指令来越过这个检测。这个同样可以通过十六进制编辑器来完成，并且技术你已经会了。就是简单的修改 `bne` 指令。

提示：*`bne` 和 `beq` 指令都是通过检测 `z` 标志位来实现的，`cmp` 指令会设置或者清除 `z` 标志位的状态*

获取正确的指令有一个小技巧，在 `0x2fd8` 处你看到的刚好是程序反汇编之后计算出来的偏移。这条指令时及时上是 `bne 0`，因为 `pc` 正好在前面的两条指令之前。

```
_main:
 00002fa8 e92d4080 push {r7, lr}
 00002fac e1a0700d mov r7, sp
 00002fb0 e24dd00c sub sp, sp, #12
-00002fb4 e59f0034 ldr r0, [pc, #52]
| 00002fb8 e58d0000 str r0, [sp]
| 00002fbc ebfffff0 bl _test_condition <- test_condition()
| 00002fc0 e58d0004 str r0, [sp, #4]
| 00002fc4 e59d0004 ldr r0, [sp, #4] <- result (0xff)
| 00002fc8 e59d1000 ldr r1, [sp] <- comparison (0)
| 00002fcc e1500001 cmp r0, r1 <- subtract the two
| 00002fd0 1a000000 bne 0x2fd8 <- go if nonzero
| 00002fd4 ea000001 b 0x2fe0 <- otherwise, go here
| 00002fd8 e3a00000 mov r0, #0 <- load exit value
| 00002fdc eb000004 bl 0x2ff4 @ symbol stub for: _exit
| 00002fe0 e5170004 ldr r0, [r7, #-4]
| 00002fe4 e1a0d007 mov sp, r7
| 00002fe8 e8bd4080 pop {r7, lr}
| 00002fec e12ffff1e bx lr
-00002ff0 00000000 andeq r0, r0, r0
```

你可以通过 `llvm-mc` 来确认：

```
echo 'bne 0' | llvm-mc -assemble -triple=armv7 -show-encoding  
.text  
bne #0 @ encoding: [0x00,0x00,0x00,0x1a]
```

逆转这个逻辑控制，将 `bne` 修改为 `beq`：

```
$ echo 'beq 0' | llvm-mc -assemble -triple=armv7 -show-encoding  
.text  
beq #0 @ encoding: [0x00,0x00,0x00,0x0a]
```

现在来计算合适的地址将 `0x1a` 替换为 `0x0a`。同样进行再次反汇编来确认是否修改正确：

```
00002fcc e1500001 cmp r0, r1  
00002fd0 0a000000 beq 0x2fd8 <- Ha ha! h4x0r3d j00!
```

现在你的程序就会将返回值和 `0` 比较，并且在失败之后跳转到退出而不是越过。

Nop 指令

`Nop` 是 `no operation` 简写，意思是没有任何操作。等价于将指令用空白指令替代。在 `arm` 架构下 `nop` 占用 `4bytes`。

```
nop @ encoding: [0x00,0xf0,0x20,0xe3]
```

Thumb 指令集下为 `2bytes`

```
nop @ encoding: [0x00,0xbf]
```

在最后的实例中我们通过修改 `bne` 来达到目的，同样可以将这些代码 `nop` 掉，从而阻止程序退出。因为在 `bne` 之后的分支会直接退出，而程序则认为测试已经结束而将会继续执行。为了实现这个效果，简单的看下 `bne` 附近的代码：

```
00002fcc e1500001 cmp r0, r1 <- subtract the two  
00002fd0 1a000000 bne 0x2fd8 <- go if nonzero  
00002fd4 ea000001 b 0x2fe0 <- otherwise, go here
```

然后通过 `nop` 来替换：

```
00002fcc e1500001 cmp r0, r1
00002fd0 e320f000 nop <- Ha ha!
00002fd4 ea000001 b 0x2fe0
```

```
$ hexdump -C -s 0x1fd0 -n 4 test1
00001fd0 00 f0 20 e3
```

你现在已经完全抹杀原来的那些指令了！

练习：

1. 不替换 bne 指令，尝试重写 bne 后面的分支指令来达到目的
2. 让程序返回 0xff，但是返回值要和 0xff 比较，从而通过检测
3. 尝试另外一种攻击：nop 掉退出指令。幸运的是下面的指令都是紧跟着退出指令的，但是如果不是呢？会发生什么？

额外的一些东西：

1. 这个教程针对的是只针对一个架构的二进制文件。对于那些适应多架构的二进制文件，你需要反汇编并且修改所有相关的架构中的对应代码。使用 otool -f 来查看文件头，你将会得到一系列不同架构的文件偏移：

```
$ otool -f my_universal_binary
Fat headers
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
cputype 16777223
cpusubtype 3
capabilities 0x80
offset 4096 <- Add this offset
size 767344
align 2^12 (4096)
architecture 1
cputype 7
cpusubtype 3
capabilities 0x0
offset 774144 <- And later on this one
size 606112
align 2^12 (4096)
```

在修改每个架构的代码的时候要确保将每个架构下的文件偏移带入到你的计算过程中。并且在

修改的时候要确保修改的代码在十六进制模式下是吻合的。

2. 如果你愿意花费\$60 那么你可以得到一个非常好的工具叫做 Hopper。通过它可以修改文件，反编译，并且重新生成文件。你可以通过 Hopper' s Modify -> Assemble Instruction 来修改指令，然后通过 File -> Product New Executable 来创建一个新的二进制文件。这将可以帮助你减少大量的体力劳动。但是徐汇怎么来手工计算这个东西才是本文的目的。
3. 如果你是一个非常专业的测试人员，按么建议你考虑下 IDA (The Interactive Disassembler)，The starter edition 需要\$500 美金，但是这个是非常值得。当然你可以用一个免费的演示版，在 [这里](https://www.hex-rays.com/products/ida/support/download_demo.shtml)：https://www.hex-rays.com/products/ida/support/download_demo.shtml

原文链接：

<http://www.zdziarski.com/blog/?p=2172>