
iOS 应用安全开发规范

序号	版本号	修订日期	修订概述	修订人	审核人	备注
1	V1.0	2013-01-01	初稿	白开心	白开心	
2	V1.1	2013-07-11	补充第 6 章反调试、反编译等内容	白开心	独归	
3						

前言

虽然 iOS 提供了良好的安全架构和 app store 环境，但是由于系统越狱、应用逆向破解、通信窃听的存在，所以开发同学请在系统设计和代码实现时规避可能引入的安全风险。

关于文档结构如下。

第一章介绍 iOS 的应用开发框架和安全机制。

第二章描述敏感数据的本地存储、文件操作、日志输出的安全风险及相应的安全防范措施。

第三章阐述了网络通信安全 and 安全要求。

第四章关注代码实现时可能存在的传统 C 代码的安全漏洞和开源的白盒代码安全扫描工具。

第五章指出传统 web 安全漏洞在 iOS 应用上依然存在，包括 XSS 和 XML 解析漏洞的攻击和防御。

第六章概述了 iOS 应用的逆向破解风险和开发过程需要注意的事项。

1.iOS 应用框架和安全机制

1.1 应用框架

iOS 应用框架分为 4 层,分别是 Cocoa Touch、Media、Core Service 和 Core OS。
开发 iPhone OS 应用,需要在 MAC OS X 运行 Xcode 开发工具。
iPhone 主要开发语言是 Objective-C,是扩充 C 的面向对象的编程语言。

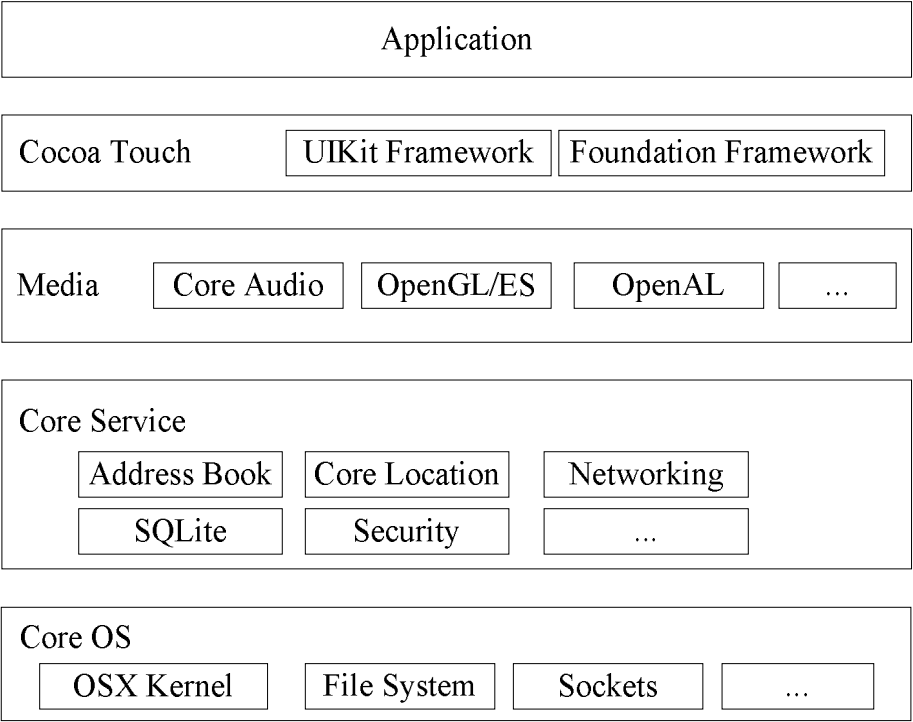


图 1.1 iOS 应用开发框架

1.2 安全机制

iOS 在设计时就充分考虑了安全需求,提供了简单易用且富有强度的安全保护功能, iOS 的安全架构如图 1.2 所示。设备底层的硬件、固件中包含有内嵌的设备密钥、用于签名的根证书、用于加密的 AES 加密引擎,能够在底层抵御恶意软件,而上层 OS 中的代码签名、文件加密和数据保护、应用沙盒等机制能够保证用户数据的完整性和机密性,防止设备、数据的未授权访问和其他类型的攻击。

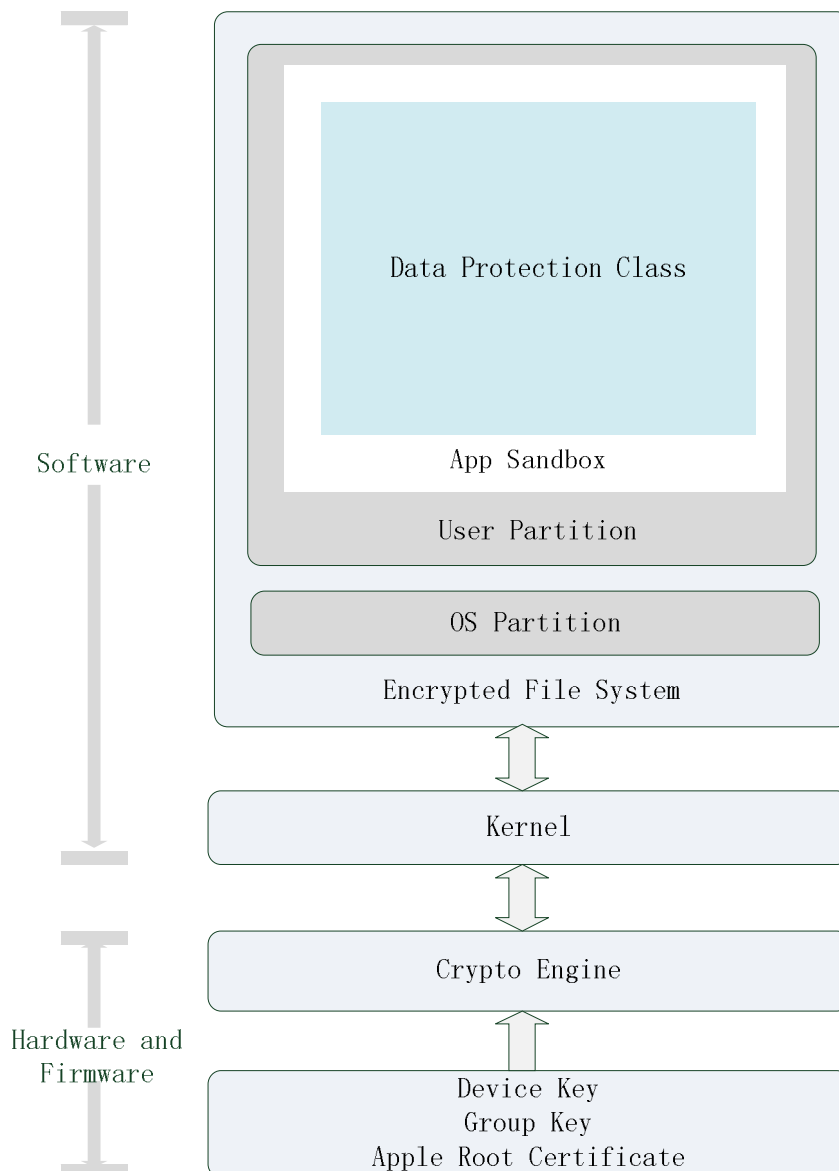


图 1.2 iOS 安全框架

iOS 的主要安全特性包括减少的受攻击平面、设备访问控制、ASLR、DEP、应用沙盒、数据保护、网络安全传输协议等，具体见下面小节内容。

1.2.1 减少的受攻击平面

攻击平面是指处理用户输入的代码部分。减小攻击平面就减小了攻击者的攻击范围。与 Mac OS X 相比，iOS 从以下几方面降低了攻击平面。

- 1) 减少可能被攻击者使用的应用，比如 shell 及命令 (/bin/sh)、flash、java、远程登录服务等，不给攻击者提供可能会被利用的攻击工具。
- 2) 不支持对某些文件类型的处理，比如，iOS 中不会对 .psd、.mov 等类型的文件进行处理；只支持 pdf 文件类型的部分特征。
- 3) 权限隔离。利用用户、组等 unix 的传统权限机制来将进程之间互相隔离。
- 4) 代码签名。所有在 iOS 中运行的应用必须拥有 apple 的签名才能运行。

1.2.2 设备访问控制

利用设备密钥和用户密码来阻止未授权的用户使用设备，在设备丢失时，用户能够远程抹除设备上的数据。

1.2.3 通用的攻击缓解措施

- 1) ASLR (Address Space Layout Randomisation, 随机化地址布局)。这个安全特性是将数据和代码在进程空间的映射地址随机化，目的是增加恶意攻击者获取进程内存布局信息的复杂度，减少内存漏洞被利用的可能性。需要应用在编译时加上“-Fpie -pie”标记，否则编译好的应用会加载到一个固定的地址。这也可以用 `otool -hv app_name` 来查看一个应用编译时是否带有-Fpie 标记，iOS 5 内置的应用编译时都带有该标志。
- 2) DEP (Data Execution Prevention, 数据执行阻断)，DEP 不允许内存中的数据被当作代码执行。DEP 是为了防止恶意应用加载并执行攻击 payload，当恶意应用加载 payload 执行时，payload 会被识别为数据，而不是代码。比如 ARM 处理器的 XN (Execute Never) 特性能够标记内存页面为不可执行。被标记为既可写又可执行的内存页面只对那些在严格控制条件（比如系统内核会检查应用是否存在 apple 的动态签名权利）下的应用才可使用。
- 3) Stack Smashing Protection—栈溢出保护，是指为了防止缓冲区溢出，在栈中函数的局部变量之前放置一个随机数，等函数返回时，首先检查该随机数是否正确，如果正确就继续执行；这个保护策略的原理是如果存在溢出的话，这个随机数会被覆盖破坏。为了利用该保护策略，只需要在编译应用时添加-fstack-protector-all 标记。可利用 `otool -I -v app_name | grep stack`，查看应用的符号表中是否包含有“__stack_chk_fail”和“__stack_chk_guard”标记。
- 4) 在 iOS SDK 5.0 中引入 ARC (Automatic Reference Counting, 自动引用记数) 来进行自动内存管理，减少开发者管理内存时可能导致的内存错误。可使用 `otool -I -v DummyApp-ARC | grep "_objc_release"` 等命令查看应用符号表中与 ARC 相关的符号，以此判断应用是否开启了该特性，开发者在编译时可通过-fno-objc-arc 标记将它禁用。与 ARC 相关的符号如下：
 - `_objc_retainAutoreleaseReturnValue`
 - `_objc_autoreleaseReturnValue`
 - `_objc_storeStrong`
 - `_objc_retain`
 - `_objc_release`
 - `_objc_retainAutoreleasedReturnValue`

1.2.4 进程级的沙盒机制

应用之间、应用和 OS 之间的资源都是隔离的。

iOS 利用 user、group 和其他 UNIX 的文件权限机制来进行进程隔离，比如浏览器、邮件客户端和第三方应用以 mobile 用户身份运行，最重要的系统进程以 root 用户运行，其他系统进程以_wireless 或_mdnsresponder 用户身份运行。一个进程不能直接访问其他进程的资源（包括文件、内存等）即使某个进程被攻击者获取全部控制，也只能获取到该应用所具有的权限和相应的资源。

第三方的应用对用户信息和特性（比如 iCloud）的访问是通过声明的权利（declared entitlements）进行控制的，entitlements 是包含应用对资源访问能力的键值对的属性列表文件。

1.2.5 代码签名

为保证应用来源已知、可信且应用本身未被篡改，在未越狱的 iOS 设备上运行的所有应用必须由苹果颁发的证书进行签名。在应用被加载运行时，代码签名机制会检查所有可执行的内存页面，保证应用从上次安装或升级后没有被篡改。

代码签名能够：

- 1) 判断代码片段是否被篡改，防止未授权的应用运行。
- 2) 识别代码的来源（定位到特定的开发者或者签名者）；便于应用更新。
- 3) 判断代码的特定目的是否可信，比如访问一个 keychain 项。

iOS 从系统底层到上层应用都采用了代码签名机制来保证应用的完整性。比如系统在正常模式下的安全启动链，如图 1.3 所示。从 Boot ROM（存储有 Apple Root CA 公钥，对下一阶段的代码进行验签）读取，LLB（Low-Level Bootloader）、iBoot 和内核的加载，每一步都会对代码的完整性做校验，签名校验通过后会才会进行下一步的处理。

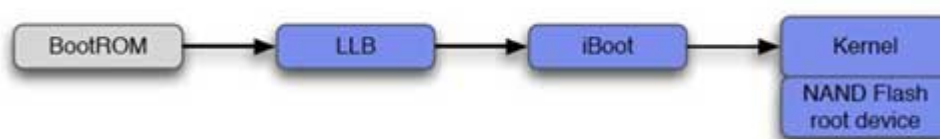


图 1.3 iOS 正常模式的启动加载过程

1.2.6 数据加密

iOS 从框架（包括底层硬件加密和数据保护 API）上来保证用户的数据安全，默认情况下，iOS 将所有存储在文件系统的数据用 AES 算法加密，加密密钥是文件系统的密钥（File System Key），存储在 flash 中。文件系统只在静止时被加密，等设备开启后，基于硬件的密码加速器就会对文件系统解锁。

除了硬件加密，iOS 还提供了数据保护的 API 来加密单个文件或 keychain 项，加密密钥从设备密码生成导出。这种加密方式只能在设备开启的时候才能运行，当设备处于 locked 状态时，Data Protection API 就不能被访问了（除非缓存

在内存中)。

1.2.7 网络安全

提供了安全协议来进行身份认证和传输数据加密，iOS 支持 SSL V3 和 TLS V1.1、TLS V1.2，Wifi (WAP2、EAP-TLS、EAP-FAST 等)、蓝牙、同时还能够与支持 L2TP、PPTP、IPSec 协议的 VPN 服务器进行保密通信。

2.敏感数据保护

2.1 敏感数据的本地存储风险

本章提到的敏感数据，包括定义、录入、传输、展示等，请详细参考《[SofaMVC 安全编码规范](#)》第 5 章、《[支付宝会员信息展示规范.docx](#)》。

如果在 iOS 设备的本地配置文件、数据库中存储用户的敏感数据，比如永久的会话 token、用户明文密码、银行卡信息，可能造成数据的泄露，同时也违反央行、PCI 合规。

- 1) 比如 facebook 的 iOS 客户端曾经在本地配置文件中存储用户的永久会话 token，如果该 token 被恶意获取，就可以在其他设备登录。
<http://www.securitylearn.net/2012/06/21/facebook-ios-app-does-not-expire-the-session-on-logout/>。
- 2) 类似的敏感数据存储漏洞也发生在花旗银行的 iOS 客户端应用，具体可参考 <http://www.cvedetails.com/cve/CVE-2010-2913/>。

为了保护本地存储的重要数据，建议采用 iOS 提供的数据保护 API 和加密机制。

2.2 iOS 加密机制

应用能够以文件、数据库、系统日志、Keychain、Cookie 等方式存储数据，为了保证用户的数据安全，iOS 提供了数据保护的 API，Data Protection API 主要是通过层次化的密钥串来对数据进行加解密。

密钥的层次结构如图 2.1 所示，最顶层的密钥是设备的 UID key 和用户的口令，其中 UID key 是一个 AES-256 位的设备密钥，它是生产时嵌入到设备的应用处理器中的，每个 iOS 设备的 UID key 都是不同的。

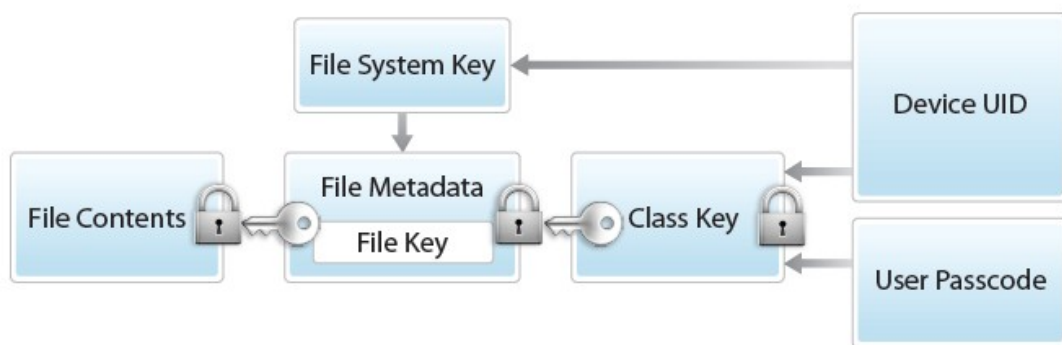


图 2.1 iOS 的层次化的密钥

当文件被创建时，Data Protection API 创建一个 256 位的 per-file 密钥，并传递给硬件的 AES 引擎（每个 iOS 设备都有一个专用的 AES 256 加密引擎，作为硬件内嵌在闪存和内存之间的 DMA 通道中），AES 引擎使用这个文件加密密钥来将数据加密并写入闪存。

Per-file 密钥是由 Class 密钥加密保护，并存储在文件的元数据中（File Metadata）。所有文件的元数据是由文件系统密钥（File System Key）加密的。File System Key 是一个在 iOS 首次被安装或者设备被用户擦除时创建的随机密钥，存储在可擦除的闪存区域，它不是被设计用于保密数据，而是用来快速擦除数据。

当文件被打开时，它的元数据被文件系统密钥解密，得到里面存储的 per-file 密钥和一个标识哪个类保护该密钥的标记。Per-file 密钥由 class 密钥解密出来并提供给 AES 引擎，当从 flash 中读取文件时进行解密。

在 iOS4 中，密钥的具体存储、生成算法和数据加解密过程如图 2.2 所示，图 2.2 中的标注解释如下。

- Keybags: 文件和 keychain 数据保护类的密钥存储、管理的地方。
- Passcode: 用户输入解锁的密钥。
- EMF: 数据分区的加密密钥，由 0x89B 加密，格式为长度 (0x20) + AES (Key89B, EMF key)。
- Dkey: NSProtectionNone class key, 由 0x835 密钥封装生成，格式为 AESWRAP(key835, Dkey)。
- BAG1: System keybag 的密钥，由 keybagd 守护进程从用户区读取来解密 systembag.kb。

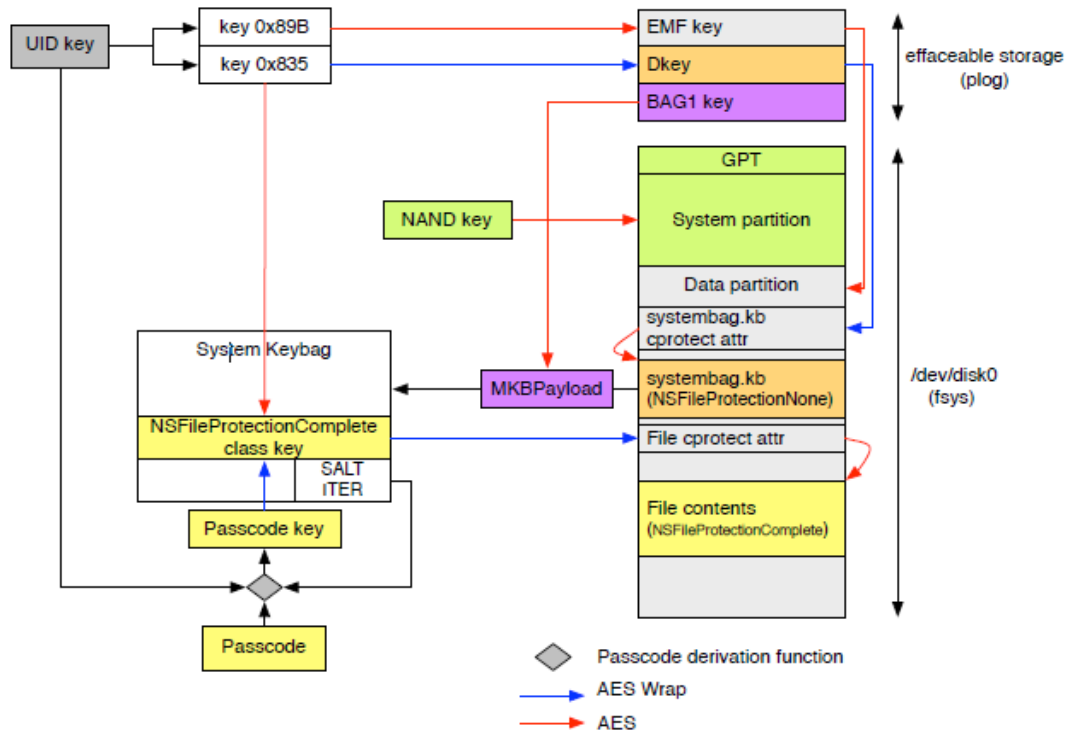


图 2.2 iOS 4 中密钥的生成、存储和数据加解密总图

2.3 iOS 数据保护 API

iOS 提供的 Data Protection API，即使在设备锁屏、断电的情况下，也能保护存储数据的机密性。当文件在设备上被创建时，它被创建它的应用指派了一种类别。每个类别采用不同的策略来确定数据何时能够被访问。

数据的可用性设计时根据设备的状态可划分为 when unlocked、while locked、after first unlocked、always。不同状态下使用的数据保护 API 如表 2.1 所示。

表 2.1 数据保护 API

可用性	文件数据保护	Keychain 数据保护
When unlocked	NSProtectionComplete	kSecAttrAccessibleWhenUnlocked
While locked	NSFileProtectionCompleteUnlessOpen	N/A
After first unlock	NSFileProtectionCompleteUntilFirstUserAuthentication	kSecAttrAccessibleAfterFirstUnlock
Always	NSProtectionNone	kSecAttrAccessibleAlways

2.3.1 文件数据保护

文件的 Data Protection API 根据传递给 NSData、NSFileManager 类的一个属性值（例如 NSFileProtectionKey 属性）的不同，划分为 4 个层级，如表 2.2 所示。

表 2.2 Data Protection API 保护层次

NSData	NSFileManager	描述
NSDataWritingFileProtectionNone	NSFileProtectionNone	文件、数据未做加密
NSDataWritingFileProtectionComplete	NSFileProtectionComplete	文件被加密，并且在设备处于锁定状态时无法访问
NSDataWritingFileProtectionCompleteUnlessOpen	NSFileProtectionCompleteUnlessOpen	文件被加密且锁定状态下是不可访问的，当一个应用在设备处于未锁定状态时获取了一个打开文件的句柄，即使稍后设备锁定，该句柄仍然可用，只是这时文件不会被加密。
NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication	NSFileProtectionCompleteUntilFirstUserAuthentication	在设备没有重启并且首次解锁前，文件都是加密的，并且不能被访问。当用户首次对设备解锁后，应用就可以访问文件了，即使后续设备处于锁定状态时仍然能够被访问。目的是用来防护需要设备重新启动的攻击。

默认情况下，所有的文件都具有 NSFileProtectionNone 值，即可以在任何时刻被读写。下面的代码展示了怎么利用 NSFileManager 类来设置一个文件的 NSFileProtectionKey 属性。

```
//创建一个 NSProtectionComplete 属性
NSDictionary *protectionComplete = [NSDictionary
dictionaryWithObject:NSFileProtectionComplete forKey:NSFileProtectionKey];
//设置文件的属性
[[[NSFileManager] defaultManager] setAttributes:protectionComplete ofItemAtPath:filePath
error:nil];
```

当一个应用需要保存数据到文件，但在设备锁住时不需要访问文件，这时就可以使用 NSDataWritingFileProtectionComplete 或者 NSFileProtectionComplete 属性。

```
-(BOOL) getFile
{
    NSString *fileURL = @"http://www.mdsec.co.uk/training/wahh-live.pdf";
    NSURL *url = [NSURL URLWithString:fileURL];
    NSData *urlData = [NSData dataWithContentsOfURL:url];
    if ( urlData )
    {
        NSArray *paths =
        NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask,
        YES);
        NSString *documentsDirectory = [paths objectAtIndex:0];
```

```
NSString *filePath = [NSString stringWithFormat:@"%s/%s",
documentsDirectory,@"wahh-live.pdf"];
NSError *error = nil;
[urlData writeToFile:filePath options:NSDataWritingFileProtectionComplete
error:&error];
return YES;
}
return NO;
}
```

2.4 文件操作风险与防护

2.4.1 目录遍历风险

NSFileManager 和 NSFileHandle 类,前者主要用于文件系统,后者用于 socket、pipe 和设备的访问。NSFileHandle 类通过文件描述符提供了更强大的方法来操作文件,类似于 C 语言中的文件操作。如果攻击者能够控制文件名的部分内容,那么 NSFileManager 和 NSFileHandle 都可能遭受目录遍历的攻击 (Directory Traversal)。

比如用两个类读取一个文件的代码如下所示,由于开发者没有对用户传入的字符串作格式过滤 (过滤 ../), 如果此时攻击者传入的文件名是类似 “.././Documents/secret.txt”, 就会遭受目录遍历攻击, 可能导致敏感数据泄露。

```
- (NSData*) readContents:(NSString*)location
{
    NSFileManager *filemgr;
    NSData *buffer;
    filemgr = [NSFileManager defaultManager];
    buffer = [filemgr contentsAtPath:location];
    return buffer;
}

- (NSData*) readContentsFH:(NSString*)location
{
    NSFileHandle *file;
    NSData *buffer;
    file = [NSFileHandle fileHandleForReadingAtPath:location];
    buffer = [file readDataToEndOfFile];
    [file closeFile];
    return buffer;
}
```

测试代码:

```
NSString *fname = @".././Documents/secret.txt";
NSString *sourcePath = [[NSString alloc] initWithFormat:@"%s/%s", [[NSBundle mainBundle] resourcePath], fname];
```

```
NSLog(@"##### PATH = %@", sourcePath);
NSString *contents = [[NSString alloc] initWithData:[fm readContentsFH:sourcePath]
encoding:NSUTF8StringEncoding];
NSLog(@"##### File contents: %@", contents);
```

2.4.2 文件操作风险防护

对程序中接受用户输入的数据进行严格的格式、内容（包括长度、类型、是否包含非法字符等）过滤，并在页面上展示时进行相应的编码。对敏感信息的展示采用部分截断展示的方式，比如信用卡号、身份证号等只展示前 6 后 4 位。

关于敏感数据的输入、展示可参考《支付宝敏感信息及日志打印规范》。

2.5 keychain 风险与防护

iOS keychain 是一个 sqlite 数据库（包括 genp、inet、cert、keys 四个表），用来存储敏感数据，比如证书、密钥等。开发者可以利用苹果提供的 keychain API 来加密存储关键数据。

Keychain 是被一个设备密钥加密的，每个设备的硬件密钥都不一样，即使攻击者能够获取到 keychain 数据库，也无法获取到原来的明文数据。Keychain 服务能够基于应用的 GUID 来限制应用访问的设备上的哪些数据，默认情况下应用只能访问与应用的 GUID 相同的数据。现在苹果引入了 keychain group，属于同一个组的应用能够共享 keychain 中的数据项。

应用对 keychain 中每一项的访问受限于他们被授予的权利（entitlements）。Keychain 使用存储在“keychain-access-group”权利规范文件中的应用的标识来识别应用。

2.5.1 攻击 keychain

（1）方法 1—基于越狱应用的破解

在一个越狱过的 iOS 设备上，可以写一个应用加入到所有应用的组中，这样它就可以访问所有的 keychain 项，[Keychain-Dumper](http://labs.neohapsis.com/2012/01/25/keychain-dumper-updated-for-ios-5/) 就是这样一个工具，<http://labs.neohapsis.com/2012/01/25/keychain-dumper-updated-for-ios-5/>。

将 keychain-dumper 拷贝到 iOS 设备，进入目录后可执行“./keychain-dumper”，就会将 keychain 中的通用密码和网络密码数据项给 dump 出来。

另外还可以在越狱过的 iOS 设备上通过 Cydia 市场安装一个 KeychainViewer 的工具，该软件提供了图形化的界面来展示 keychain 中的数据项。

（2）方法 2—针对特定设备的破解

利用 DFU mode 漏洞，挂载自制的 ramdisk 对 apple A4 芯片的设备进行暴力破解用户密码和 keychain 数据项的方法可参考：<http://code.google.com/p/iphone-dataprotection/wiki/README>。

2.5.2 keychain 数据保护

和文件系统类似，根据传递给 `SecItemAdd`、`SecItemUpdate` 函数的 `kSecAttrAccessible` 属性值的不同，keychain 中数据保护的层级也可以划分为 4 类，见表 2.3。Keychain 中的条目创建时默认的保护层级是 `kSecAttrAccessibleAlways`，即在任何时刻都可以被解密并移植到其他设备上。如果使用了 `-ThisDeviceOnly` 保护类，keychain 中条目将会被一个从设备密钥衍生出的密钥加密保存，这保证了只有在该设备上才能对该条目进行解密。

表 2.3 Keychain Item Protection Attributes

保护属性	描述
<code>kSecAttrAccessibleAlways</code>	keychain 中的条目总是能够被访问。
<code>kSecAttrAccessibleWhenUnlocked</code>	keychain 中的条目只有设备处于解锁时才能被访问。
<code>kSecAttrAccessibleAfterFirstUnlock</code>	Keychain 中的条目只有在设备重启，第一次解锁后才能访问。
<code>kSecAttrAccessibleAlwaysThisDeviceOnly</code>	只有在本设备能够随时访问（不能迁移到其他设备上使用）。
<code>kSecAttrAccessibleWhenUnlockedThisDeviceOnly</code>	只有在本设备的解锁状态下可以访问。
<code>kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly</code>	设备重启后第一次解锁成功后才能访问，并且仅限本设备（不能迁移到其他设备上使用）。

下面的例子中，应用先创建一个 key-value 的字典，用来存储 keychain 中的配置属性和值，设置了 keychain 的访问属性为 `kSecAttrAccessibleWhenUnlocked`，只要设备处于解锁状态就可以访问。另外，应用利用 `SecItemAdd` 方法向 keychain 中添加了一个属性为 `kSecValueData` 的数据项，值为传入的参数 `licenseData`。

```
- (NSMutableDictionary *)getkeychainDict:(NSString *)service {
    return [NSMutableDictionary dictionaryWithObjectsAndKeys:
        (id)kSecClassGenericPassword, (id)kSecClass,
        service, (id)kSecAttrService, service, (id)kSecAttrAccount,
        (id)kSecAttrAccessibleWhenUnlocked, (id)kSecAttrAccessible, nil];
}

- (BOOL) saveLicense:(NSString*)licenseKey {
    static NSString *serviceName = @"my.company.VulnerableiPhoneApp";
    NSMutableDictionary *myDict = [self getkeychainDict:serviceName];
    SecItemDelete((CFDictionaryRef)myDict);
    NSData *licenseData = [licenseKey dataUsingEncoding:NSUTF8StringEncoding];
    [myDict setObject:[NSKeyedArchiver archivedDataWithRootObject:licenseData]
        forKey:(id)kSecValueData];
    OSStatus status = SecItemAdd((CFDictionaryRef)myDict, NULL);
    if (status == errSecSuccess) return YES;
    return NO;
}
```

2.6 sql 注入风险与防护

(1) sql 注入风险

iOS 应用通常会在客户端利用 SQLite 数据库存储一些应用数据，如果 sql 语句采用拼接参数的方式，并且没有对用户输入的数据进行过滤，可能会导致 sql 注入。

比如下面代码中就存在 sql 注入漏洞。

```
NSString *uid = [myHTTPConnection getUID];
NSString *statement = [NSString stringWithFormat:@"SELECT username
FROM users where uid = '%@'",uid];
const char *sql = [statement UTF8String];
```

(2) sql 注入防御

防范 sql 注入的方法是进行变量绑定和参数化的查询。

```
const char *sql = "SELECT username FROM users where uid = ?";
sqlite3_prepare_v2(db, sql, -1, &selectUid, NULL);
sqlite3_bind_int(selectUid, 1, uid);
int status = sqlite3_step(selectUid);
```

2.7 日志输出敏感数据

在开发和测试环境中，NSLog 经常用来输出日志信息来调试应用，但在线上发布的产品，需要在 NSLog 中输出敏感数据。具体可在 Xcode 中添加一个自定义的 Debug 编译标记，同时在代码中定义一个条件编译的宏，具体可参考下面网址。

<http://iphoneincubator.com/blog/debugging/how-to-create-conditional-log-statements-in-xcode>。

<http://iphoneincubator.com/blog/debugging/the-evolution-of-a-replacement-for-nslog>。

2.8 账户信息展示安全

在应用展示用户敏感数据时，包括银行卡号、手机号、身份证号、姓名、密钥相关数据时，需要进行部分隐藏展示或全部隐藏展示。具体账户信息展示规则可参考《[支付宝会员信息展示规范.docx](#)》。

3.网络通信安全

尽量不要使用私有协议进行通信。

为保证数据传输过程中的安全，包括数据的机密性和完整性，客户端与服务端之间的通信应该利用 iOS SDK SSL 库中的 NSURLRequest、NSURLConnection、CFNetwork 等类，采用 https 通信。为避免 Man-in-the-Middle 攻击，**禁止应用与**

使用自签名或无效证书的服务端进行通信，避免出现类似下面的代码。

```
#import "loadURL.h"
@interface NSURLRequest (DummyInterface)
+ (BOOL)allowsAnyHTTPSCertificateForHost:(NSString*)host;
+ (void)setAllowsAnyHTTPSCertificate:(BOOL)allow forHost:(NSString*)host;
@end
@implementation loadURL
-(void) run
{
    NSURL *myURL = [NSURL URLWithString:@"https://localhost/test"];
    NSMutableURLRequest *theRequest = [NSMutableURLRequest requestWithURL:myURL
    cachePolicy:NSURLRequestReloadIgnoringCacheData timeoutInterval:60.0];
    [NSURLRequest setAllowsAnyHTTPSCertificate:YES forHost:[myURL host]];
    [[NSURLConnection alloc] initWithRequest:theRequest delegate:self];
}
@end
```

攻击者还可以向自己的手机安装可信证书，然后利用 Charles 或者 burp suite 进行 https 中间人抓包分析，防御此类分析可采用 [ssl pinning 技术](#)，将服务端证书打包到客户端应用中，在 NSURLConnectionDelegate protocol 的 [connection:willSendRequestForAuthenticationChallenge:](#) 中比较获取的服务端证书与客户端内置的证书是否一致。示例程序可见 <https://github.com/doubleencore/de-SSLPinning>。

除采用 https 协议外，基于以下原因，建议对一些重要数据（如用户的密码、交易订单等）进行业务层的数据加密和签名。

- 1) https 协议在 iOS 实现过程中可能出现漏洞，比如 iOS 4.3.5 曾经存在 SSL 中间人攻击的漏洞，具体可参考下述网址。
<https://www.trustwave.com/spiderlabs/advisories/TWSL2011-007.txt> ;
<http://support.apple.com/kb/HT4824>。
- 2) 存在 sslstrip 等中间人攻击，分析通信协议、客户端与服务端的交互接口，然后进行进一步的攻击（比如扫号、批量领取红包等）。例如，支付宝 iOS 客户端的 ssl 中间人抓包分析如图 3.1 所示，具体环境搭建过程可参考
<http://danqingdani.blog.163.com/blog/static/1860941952012112353515306/>。



图 3.1 支付宝 iOS 客户端查询交易记录的请求和返回数据

4.Object-C 编程

iOS 和 Mac OS X 一起使用 Object-C 和大部分的 Cocoa API，虽然 Object C 进行了托管的内存管理，但是应用中仍可能存在 C 程序的传统安全风险，典型的代码漏洞如下面小节。

4.1 缓冲区溢出

缓冲区溢出包括栈溢出和堆溢出，指堆、栈预留分配的内存不够存储输入的数据，并且没有被截断，将会导致数据溢出，覆盖其他数据。如果输入的数据是精心构造的恶意代码，就可能执行恶意操作。关于缓冲区溢出示例及防范具体可参考下面网址。

<http://developer.apple.com/library/mac/#documentation/security/conceptual/SecureCodingGuide/Articles/BufferOverflows.html>。

- 1) 在编译应用时添加-fstack-protector-all 标记，避免栈溢出。
- 2) 在使用缓冲区时首先计算将要存储对象的大小，避免使用硬编码缓冲区大小（如果代码变更，需要对所有硬编码的数值进行修改，如有遗漏，就可能造成溢出），见下面代码示例。

不规范使用	正确使用
<pre>char buf[1024]; ... if (size <= 1023) { ... }</pre>	<pre>#define BUF_SIZE 1024 ... char buf[BUF_SIZE]; ... if (size < BUF_SIZE) { ... }</pre>

	}
--	---

- 3) String 操作尽量用 NSString 对象, 如果要用 c 风格函数, 应该用 strlen、strn 家族的函数, 例如 strcat、strcpy、[v]snprintf。

4.2 整数溢出

如果 4.1 节中的缓冲区大小由用户决定, 假如恶意用户输入一个 integer 类型无法表示的整数, 可能造成应用崩溃, 或者造成其他问题。比如 32 位的有符号整数 $\text{int } 2147483647 + 1 = -2147483648$ 。

在计算分配缓冲区大小时, 如下代码分别展示了错误、正确的整数使用方法。

(1) 错误的代码

```
size_t bytes = n * m; //可能造成溢出
if (bytes < n || bytes < m) { /* 错误的判断方法 */
    ... /* allocate "bytes" space */
}
```

(2) 正确的代码

```
#define SIZE_MAX 2147483647
size_t bytes = n * m;
if (n > 0 && m > 0 && SIZE_MAX/n >= m) {
    ... /* allocate "bytes" space */
}
```

4.3 格式化字符串

格式化字符串漏洞的主要原因是我们在使用 [v][f]printf 等函数时没有严格检查输入参数的个数, 比如正常的 printf 函数格式应该是: printf(“格式控制字符串”, 变量列表); 而我们经常会省略掉后面的变量列表或者变量个数, 导致跟前面的格式控制符没有一一对应, 如果格式控制的字符串是可以被控制的, 那么可以利用特殊的格式控制符 %x 读取内存数据, 用 %n 写入数据, 甚至经过精心构筑后可以向任意地址写入任意数据。

格式化字符串的漏洞示例和分析可参考下面网址。

<http://seckungfu.com/blog/2012/08/09/ge-shi-hua-zi-fu-chuan-lou-dong-fen-xi-zhi-%5B%3F%5D/>。

在使用以下 Object-C 类和方法进行格式化字符串输出时, 可参照 <https://developer.apple.com/library/ios/#documentation/CoreFoundation/Conceptual/CFStrings/formatSpecifiers.html>。

```
NSLog
[NSString stringWithFormat]
[NSString stringByAppendingFormat]
[NSString initWithFormat]
[NSMutableString appendFormat]
[UIAlertView alertWithMessageText]
[UIAlertView informativeTextWithFormat]
```

```
[NSException format]
[NSMutableString appendFormat]
[NSPredicate predicateWithFormat]
```

4.4 重复释放指针

指针指向的对象的重复释放会造成程序崩溃等。

4.5 Use-after-free

使用已经释放的对象指针，同样会造成程序崩溃或者被恶意攻击利用。

4.6 代码漏洞扫描

为避免 C 编程带来的风险，可以在 iOS 程序开发完之后，采用下面的软件进行静态代码扫描。

1) Flawfinder

对缓冲区溢出、格式化字符串等 C 风险函数（如 strcpy、fprintf）进行分析。

<http://www.dwheeler.com/flawfinder/>。

2) Clang Static Analyzer

Clang Static Analyzer 也是一个开源的代码扫描工具，它既可以独立运行，也可以在 Xcode 中作为插件运行。

<http://clang-analyzer.llvm.org/>。

5 . 传统 web 漏洞与防护

5.1 XSS 漏洞及防护

UIWebView 是 iOS web 页面的展示渲染引擎，支持多种文件格式，比如 HTML、PDF、RTF、Office Documents、iWork Documents；同时还支持 javascript。当用户控制的变量未经过滤和编码在 WebView 展示时，就可能产生 XSS 漏洞，窃取用户的隐私数据，比如会话 token、本地存储的敏感数据等。

比如，产生 XSS 漏洞的示例代码如下。首先，username 被添加到 NSString 对象 javascript，然后 javascript 对象被 stringByEvaluatingJavaScriptFromString 方法添加到了 web view 的 Dom 对象中。

```
NSString *javascript = [[NSString alloc] initWithFormat:@"var myvar=\"%@\";", username];
```

```
[mywebView stringByEvaluatingJavaScriptFromString:javascript];
```

本地的 html 页面中输出 myvar 变量。

```
<html>
```

```
<p>Cross-Site Scripting in UIWebView:</p>
<p>This is an example of XSS:
<script>document.write(myvar);</script>
</p>
</html>
```

Skype V3.0.1 以及之前的版本曾经对用户输入的用户名未做编码展示，存在 XSS 漏洞，导致攻击者构造的 JS 代码可在客户端执行，再加上 skype 对内置 webkit 浏览器使用的 URI scheme 定义不当，该漏洞可以被攻击者用来获取受害用户的通讯录等敏感数据。具体漏洞可参考 <https://superevr.com/blog/2011/xss-in-skype-for-ios/>。

对 XSS 漏洞的防护措施主要有 (1) 对用户输入的非法字符过滤，比如 <、>、' 等。(2) 在页面展示用户输入的变量时，进行相应的编码，比如 html 编码展示。

5.2 XML 解析

XML 文档在应用中广泛使用，iOS 提供两种解析 XML 的方式：NSXMLParser 和 libxml2，另外还有很多第三方的 XML 解析器，比如 TBXML、TinyXML。

5.2.1 攻击 XML 解析

一种常见的攻击方式是让 XML 解析器遍历大量有嵌套循环的元素，进行 DoS 攻击。iOS 提供的两种解析器均对这种攻击方式免疫，当 NSXMLParser 检测到嵌套的元素时，它会抛出一个 NSXMLParserEntityRefLoopError 的异常；而 libxml2 会抛出一个 “Detected an entity reference loop” 的错误。

另外一种攻击方式是让 XML 解析器解析外部的 XML 文档，这个功能对 NSXMLParser 默认是关闭的，开启该功能需要开发者设置一个 setShouldResolveExternalEntities 的选项，该选项会使委托方法 foundExternalEntityDeclarationWithName 在发现外部实体时被调用。但 libxml2 是默认开启的。

下面使用 NSXMLParser 的代码是存在被攻击的风险的，请默认不要开启红色字体标注的功能。

```
#import "XMLParser.h"
@implementation XMLParser
- (void)parseXMLStr:(NSString *)xmlStr {
    BOOL success;
    NSData *xmlData = [xmlStr dataUsingEncoding:NSUTF8StringEncoding];
    NSXMLParser *addressParser = [[NSXMLParser alloc] initWithData:xmlData];
    [addressParser setDelegate:self];
    [addressParser setShouldResolveExternalEntities:YES];
    success = [addressParser parse];
}
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
```

```

attributes:(NSDictionary *)attributeDict {}
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string {}
- (void)parser:foundExternalEntityDeclarationWithName:publicID:systemID {}
- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError {
    NSLog(@"Error %i, Description: %@", [parseError code],
        [[parser parseError] localizedDescription]); }
@end

```

如果传入的参数 `xmlStr` 为如下代码，当解析器尝试解析该 `xml` 实体时，会使 iOS 设备向远程的 web 服务器发送一个 HTTP 请求。

```

NSString *xmlStr = @"<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\
    <!DOCTYPE foo [\
        <!ELEMENT foo ANY >\
            <!ENTITY xxe SYSTEM \"http://192.168.0.7/hello\">\
        ]>\
    <foo>&xxe;</foo>";
XMLParser *xp = [[XMLParser alloc] init];
[xp parseXMLStr:xmlStr];

```

针对 libxml2 的攻击向量如下。

```

#import <libxml/xmlmemory.h>
@implementation LibXml2
- (BOOL) parser:(NSString *)xml {
    xmlDocPtr doc = xmlParseMemory([xml UTF8String], [xml
lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
    xmlNodePtr root = xmlDocGetRootElement(doc);
}
@end

```

5.2.2 防御方法

- 1) 不打开 `NSXMLParser` 的 `setShouldResolveExternalEntities:YES` 选项来解析外部的 `xml` 文档。
- 2) 不使用 `libxml2` 解析外部的 `xml` 文档。

6 . 应用的逆向破解风险

6.1 iOS 应用逆向破解

iOS 应用的破解是指调试、反编译、逆向解密篡改一个 iOS 应用。

苹果 AppStore 中的应用都会被 apple 的二进制加密策略保护，这些应用在运行时由 iOS 内核的 mach loader 解密，iOS 可执行程序格式如图 6.1 所示。

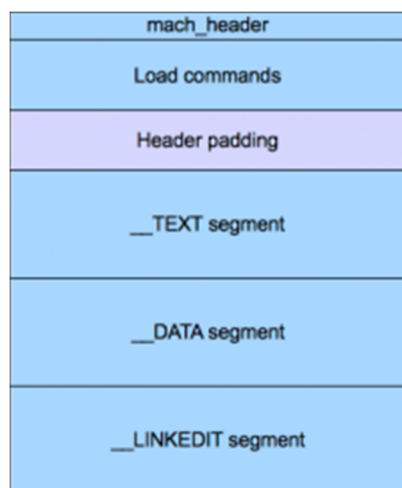


图 6.1 iOS Mach-O 文件格式

因此，只有在应用解密后才能够更好的理解它的工作原理、内部的类的结构等。为了获取解密后的应用，可以使用 GDB 对应用进行调试，等加载器对应用解密完成时，使用 `dd` 命令将解密后的镜像文件 `dump` 出来，然后再进行修改替换、打包成新的破解后的应用。最后再用 `otool`、`strings`、`class-dump`、`IDA Pro` 工具对解密后的应用做进一步的逆向分析。

解密应用的过程可以利用 `Rasticrac`、`PoedCrackMod`、`ClutchPatched`、`Clutch`、`Crackulous`、`AppCrack` 等工具来自动化完成，还可以使用 GDB 和一些破解工具手工完成。具体破解过程可参考《支付宝 iOS 应用安全测试文档》或者以下网址。

- 1) <http://lightbulbone.com/post/27887705317/reversing-ios-applications-part-1>
- 2) http://blog.claudxiao.net/2012/07/decrypt_app_in_ios5/
- 3) <http://hackulo.us/wiki/PoedCrackMod>
- 4) <http://hackulo.us/forums/index.php?/topic/62268-release-pcm-aka-poedcrackmod-the-cracking-shell-script/>
- 5) <http://hackulo.us/wiki/Clutch>

6.2 反调试

禁止 GDB、LLDB、Dbgserver 等调试工具调试(建议仅用于发布版本中，否则会导致程序无法被调试。

6.2.1 检查进程的标记

原理：当进程被调试的时候，内核自动设置进程 `kinfo_proc` 结构中的 `p_flag` 标记选项为 `P_TRACED`。因此可以通过检查该标记来确认应用是否被调试。

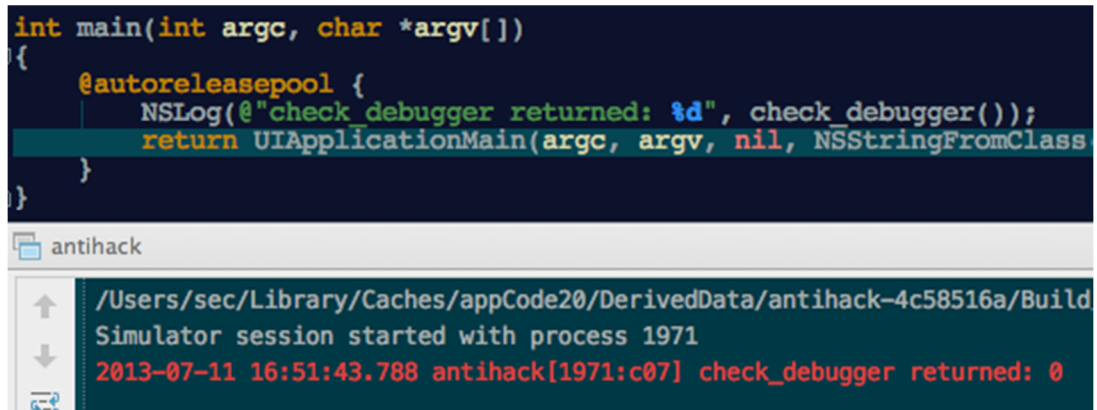
```
#include <unistd.h>
#include <sys/types.h>
#include <sys/systtl.h>
#include <string.h>
static int check_debugger( ) __attribute__((always_inline));
```

```

int check_debugger( )
{
    size_t size = sizeof(struct kinfo_proc);
    struct kinfo_proc info;
    int ret, name[4];
    memset(&info, 0, sizeof(struct kinfo_proc));
    name[0] = CTL_KERN;
    name[1] = KERN_PROC;
    name[2] = KERN_PROC_PID;
    name[3] = getpid();
    if (ret = (sysctl(name, 4, &info, &size, NULL, 0))) {
        return ret; /* sysctl() failed for some reason */
    }
    return (info.kp_proc.p_flag & P_TRACED) ? 1 : 0;
}

```

未使用调试器附加时，返回值为 0，如图 6.2。



```

int main(int argc, char *argv[])
{
    @autoreleasepool {
        NSLog(@"check_debugger returned: %d", check_debugger());
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
    }
}

```

antihack

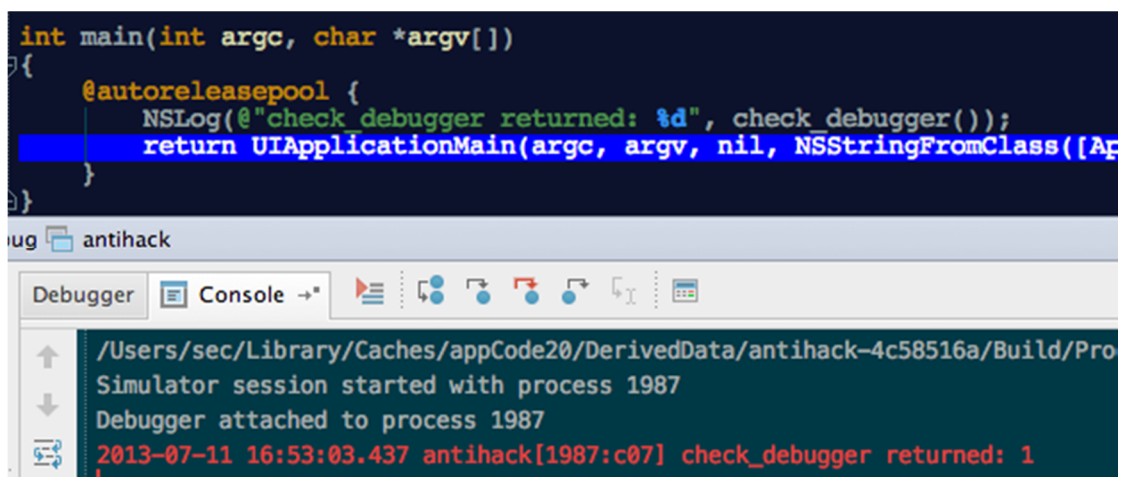
```

/Users/sec/Library/Caches/appCode20/DerivedData/antihack-4c58516a/Build/Products/Debug-iphonesimulator/antihack
Simulator session started with process 1971
2013-07-11 16:51:43.788 antihack[1971:c07] check_debugger returned: 0

```

图 6.2 没有调试器附加时返回结果为 0

当有调试器附加到进程时，返回值为 1，如图 6.3。



```

int main(int argc, char *argv[])
{
    @autoreleasepool {
        NSLog(@"check_debugger returned: %d", check_debugger());
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
    }
}

```

antihack

```

/Users/sec/Library/Caches/appCode20/DerivedData/antihack-4c58516a/Build/Products/Debug-iphonesimulator/antihack
Simulator session started with process 1987
Debugger attached to process 1987
2013-07-11 16:53:03.437 antihack[1987:c07] check_debugger returned: 1

```

图 6.3 有调试器附加时返回结果为 1

6.2.2 调用 ptrace 请求来检查进程是否被调试

调用 ptrace 请求的 PT_DENY_ATTACH 方法。使用该方法后，下面两种情况都会导致调试器崩溃，应用退出。

```
#import <dlfcn.h>
#import <sys/types.h>
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);
#ifdef PT_DENY_ATTACH
#define PT_DENY_ATTACH 31
#else // !defined(PT_DENY_ATTACH)

void disable_gdb() {
    void* handle = dlopen(0, RTLD_GLOBAL | RTLD_NOW);
    ptrace_ptr_t ptrace_ptr = dlsym(handle, "ptrace");
    ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);
    dlclose(handle);
}
```

当程序未被调试时，测试结果如图 6.4。

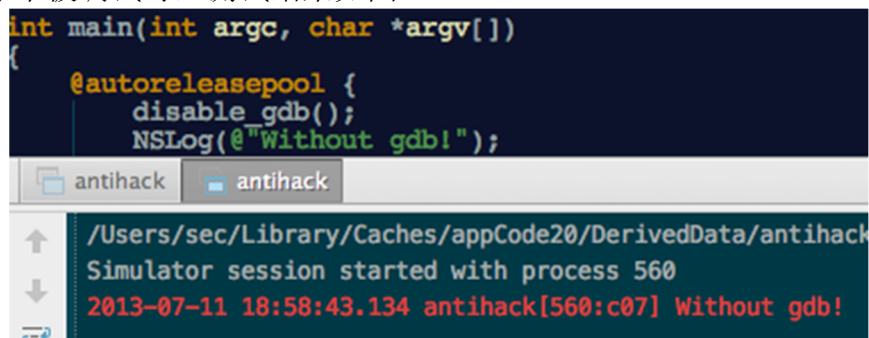


图 6.4 无调试器附加时输出结果

而当程序被调试时，输出结果如图 6.5。

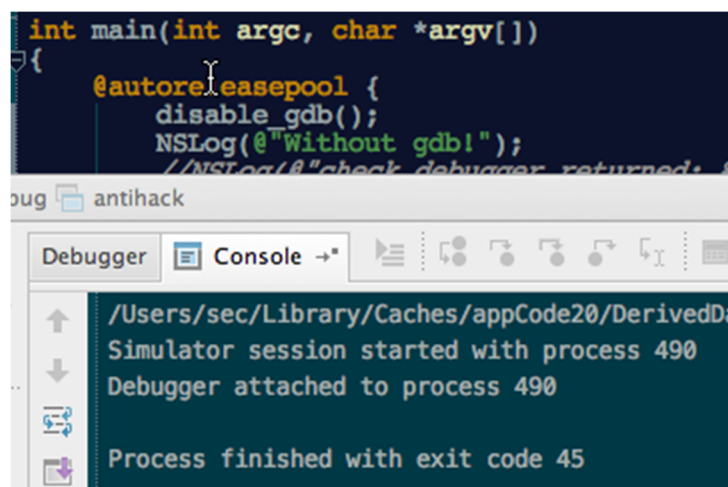


图 6.5 有调试器附加时调试结果

6.3 反一反编译

6.3.1 inline 和 static 属性

为增大攻击者反编译、调试查看应用功能的难度，在编写代码时可通过对函数设置 `inline`、`static` 属性，使得编译后生成的目标代码在反汇编成汇编代码时，不容易阅读清楚程序内在逻辑。因为 `inline` 标记的函数体在编译时，它的汇编代码会多次嵌入到调用函数中；而 `static` 属性会让编译生成的二进制代码中没有清晰的符号表。

```
#include <stdlib.h>
static int is_odd(int n) __attribute__((always_inline));
static inline int is_odd(int n) {
    if (n % 2 != 0) {
        return 1;
    } else {
        return 0;
    }
}
int main( ) {
    int session = 3;
    if (is_odd(session))
        return 1;
    return 0;
}
```

```
$ gcc -o numcheck securitycheck.c -finline-functions -Winline
```

编译时 `-finline-functions` 告诉编译器尽可能的采用 `inline` 编译方式；`-Winline` 告诉编译器在不能使用 `inline` 编译方式编译时报警。但有的函数不能用 `static` 来声明，上面讲的利用 `static` 属性来提高反编译难度的方法就失效了。

<http://danqingdani.blog.163.com/blog/static/18609419520123111202352/>

6.3.2 funroll-loops 编译选项

`-funroll-loops` 编译选项使得程序中的循环代码完全展开，比如 `for`、`while` 等循环。这样会增加汇编代码的长度，攻击者在篡改应用时，需要隔离出每个循环来进行单独的修改，从而增大了攻击难度。

在 `linux` 环境下编译可以完全展开整个循环逻辑，但是在苹果的编译器中目前是不能完全展开。可参考：
<http://danqingdani.blog.163.com/blog/static/186094195201231112854642/>。

另外编译器的 `-O3` 编译选项可以将具体的计算逻辑隐藏起来，直接输出计算结果，使攻击者完全看不出中间的计算逻辑。

6.3.3 Strip 符号表

在应用编译完成后，用 `strip` 命令去除目标文件中的指定符号，避免攻击者轻松获取函数的符号和地址。`Strip` 命令去除可执行应用 `num` 中的符号的前后对比如图 6.6 所示。

```
053438Z50L01682:test shanhushang$ nm num
0000000100001048 S _NXArgc
0000000100001050 S _NXArgv
0000000100001060 S __progname
0000000100000000 T __mh_execute_header
0000000100001058 S _environ
U _exit
0000000100000e20 T _is_odd
0000000100000e50 T _main
U _printf
0000000100001000 s _pvars
U dyld_stub_binder
0000000100000de0 T start
053438Z50L01682:test shanhushang$ strip num
053438Z50L01682:test shanhushang$ nm num
0000000100000000 T __mh_execute_header
U _exit
U _printf
U dyld_stub_binder
053438Z50L01682:test shanhushang$ otool -tv num
num:
(__TEXT,__text) section
0000000100000de0      pushq    $0
0000000100000de2      movq    %rsp, %rbp
```

图 6.6 strip 去除应用的符号表的前后对比

6.4 反注入

<http://danqingdani.blog.163.com/blog/static/186094195201231111625535/>

6.5 编译保护

由于 iOS 应用存在被逆向破解的风险，所以应用开发时不要将重要数据的加密密钥硬编码到代码中。对应用编译打包时，注意使用以下安全标记，如表 6.1。

表 6.1 应用编译时的安全标记选项

编译保护	检查方法
编译时带有 PIE 标记， <code>-Fpie -pie</code>	<code>otool -hv app_name</code> ，查看 <code>pie</code> 标记
编译时带有 <code>stack-protector-all</code> 标记， <code>-fstack-protector-all</code>	<code>otool -I -v app_name grep stack</code> ，查看应用的符号表中是否包含有“ <code>__stack_chk_fail</code> ”和“ <code>__stack_chk_guard</code> ”标记
应用使用了 ARC 特性（Automatic	<code>otool -I -v DummyApp-ARC grep</code>

Reference Counting), 编译时没有使用 -fno-objc-arc 标记	"_objc_release"等命令查看应用符号表中与 ARC 相关的符号是否存在, 详见 1.2.3 节。
---	--

7. 参考资料

- 1) <http://securityxploded.com/demystifying-iphone-forensics-on-ios5.php>
- 2) <http://code.google.com/p/iphone-dataprotection>
- 3) http://developer.apple.com/library/ios/#technotes/tn2250/_index.html
- 4) <http://developer.apple.com/library/mac/#documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html>
- 5) http://hackulo.us/wiki/IOS_Cracking
- 6) <http://developer.apple.com/library/ios/navigation/#section=Topics&topic=Security>
- 7) <http://sqlite.org/sqlite-amalgamation-3071300.zip>
- 8) <http://www.mdsec.co.uk/index.html>
- 9) <https://viaforensics.com/mobile-security-category/secure-mobile-development-42-practices-secure-ios-android-development.html>
- 10) <http://carnal0wnage.attackresearch.com/2010/11/iphone-burp.html>
- 11) <http://www.mdsec.co.uk/index.html>