Through the mach portal

ianbeer

overview

Part I - escaping the iOS app sandbox

- Mach ports and mach port names
- lifetime management of names
- single-send-right rule
- message destruction in userspace
- name allocation and free algorithms
- port rights state transitions

Part II - getting the kernel task port

- kernel port structures
- reference counting
- race conditions
- using no-senders
- zalloc
- finding the kernel task port

mach ports in 30 seconds:

A **mach port** is a *kernel-maintained* message queue

A **mach port** is *multiple sender*, single receiver

mach ports in 30 seconds:

In userspace
mach port names name *rights* a process has over a particular message queue

send right: try to *enqueue* an unlimited number of messages to a particular message queue

send-once right: try to *enqueue* a single messages to a particular message queue

receive right: try to *dequeue* an unlimited number of messages from a particular message queue

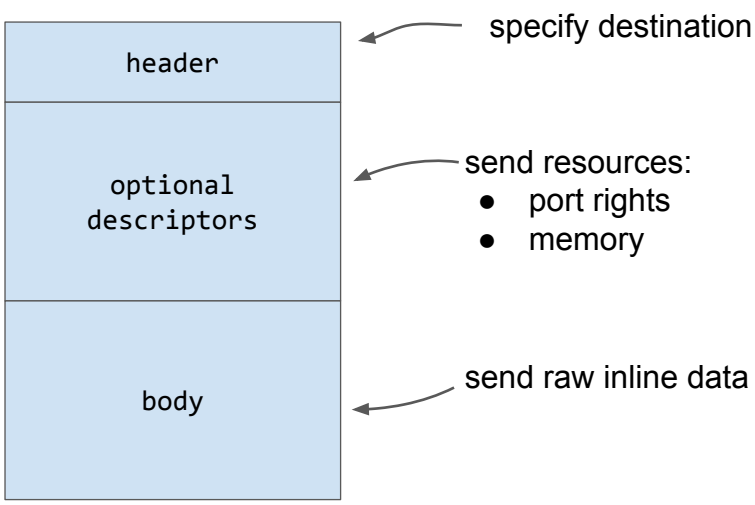
---

portset right: try to *dequeue* an unlimited number of messages from **multiple** message queues

dead-name right: do nothing (the queue no longer has a receiver)

I'm using the term message queue but for kernel owned ports for the kernel MIG apis the messages never get queued; there's a fast path which turns them into synchronous syscalls (see ipc_kobject_server)

mach messages

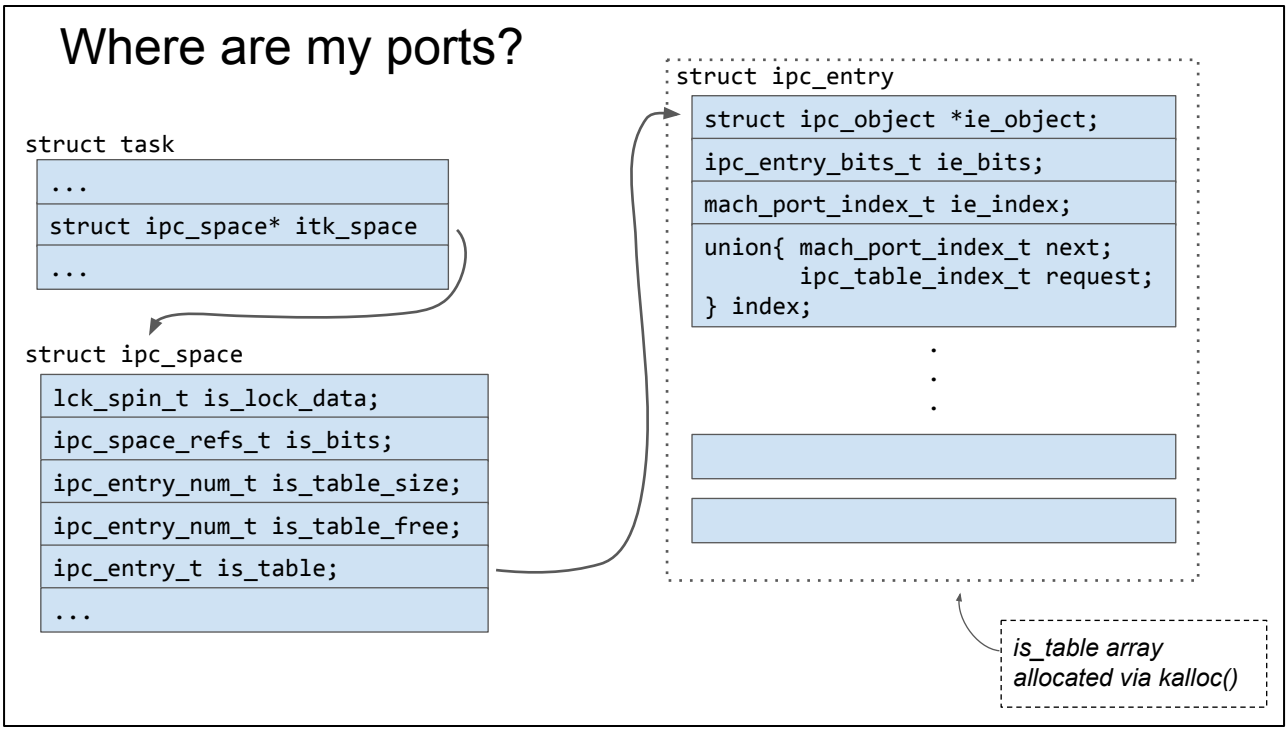


the header also contains port rights; the most important one (at least for IPC/RPC) being the reply port

lsmp - list mach ports

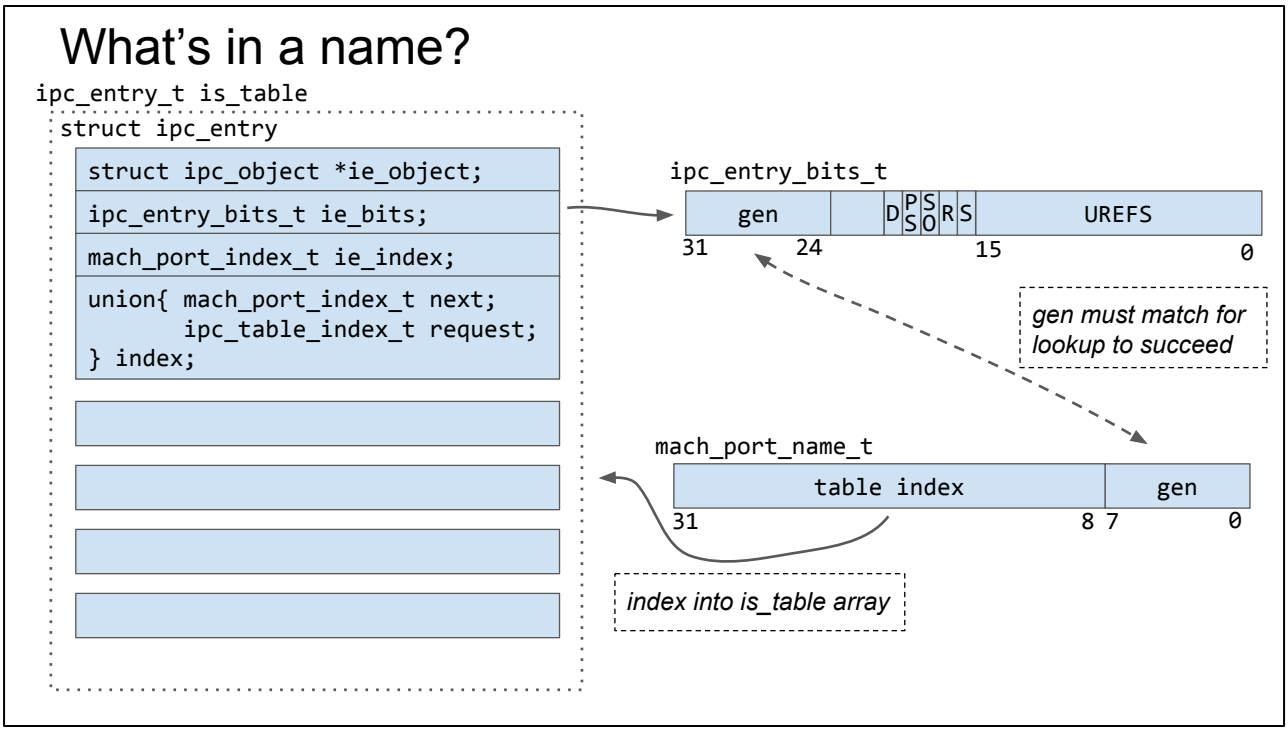
```
Process (5667) : hello_world
  name     ipc-object   rights    send sonce qlimit msgcount context            identifier  type
---------  ---------- ---------- ----- ----- ------ -------- ------------------ ----------- ------------
0x00000103 0x25d412b7 send           2                                          0x00000000  TASK SELF
0x00000203 0x28f58c67 recv                         5        0 0x0000000000000000
0x00000307 0x25d404ef send           1                                          0x00000000  THREAD
0x00000403 0x28f5a35f recv                         5        0 0x0000000000000000
0x00000503 0x28f5b517 recv                         5        0 0x0000000000000000
0x00000607 0x2604f2b7 recv                         5        0 0x0000000000000000
0x00000707 0x2662f0bf send           4             6        0 0x0000000000000000 0x00032603  (1) launchd
0x00000803 0x0d9f55ff send           1                                          0x00000000  CLOCK
0x00000903 0x2604f0bf send           1                                          0x00000000  SEMAPHORE
0x00000a03 0x27fb1057 recv,send      1             5        0 0x0000000000000000
```

This is the output of lsmp for a simple hello_world program on MacOS (the output is slightly truncated to fit the slide.) Notice that there's some kind of pattern to the names.



struct task is the processes abstraction for the Mach side of the XNU kernel. See task_t considered harmful [https://googleprojectzero.blogspot.com/2016/10/taskt-considered-harmful.html] for more discussion about task structs.

Each task has it's own ipc namespace. This is just a way of assigning "names" to ports



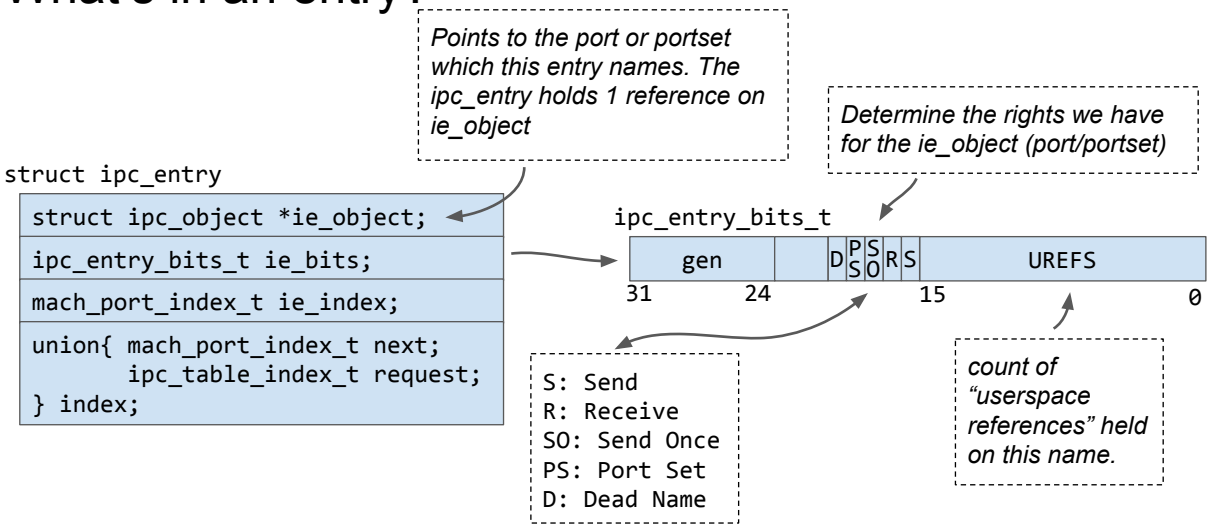
ipc_entry_lookup takes the mach_port_name_t from userspace and splits it up into the table index and generation number fields. Note that XNU doesn't use c bitfields for this, it's all macros and bittwiddling.

That index is checked against the size of the table then used to index the array. Then the generation number from the mach_port_name_t is compared against the generation name stored in the ie_bits field. Only if they match is the entry pointer returned.

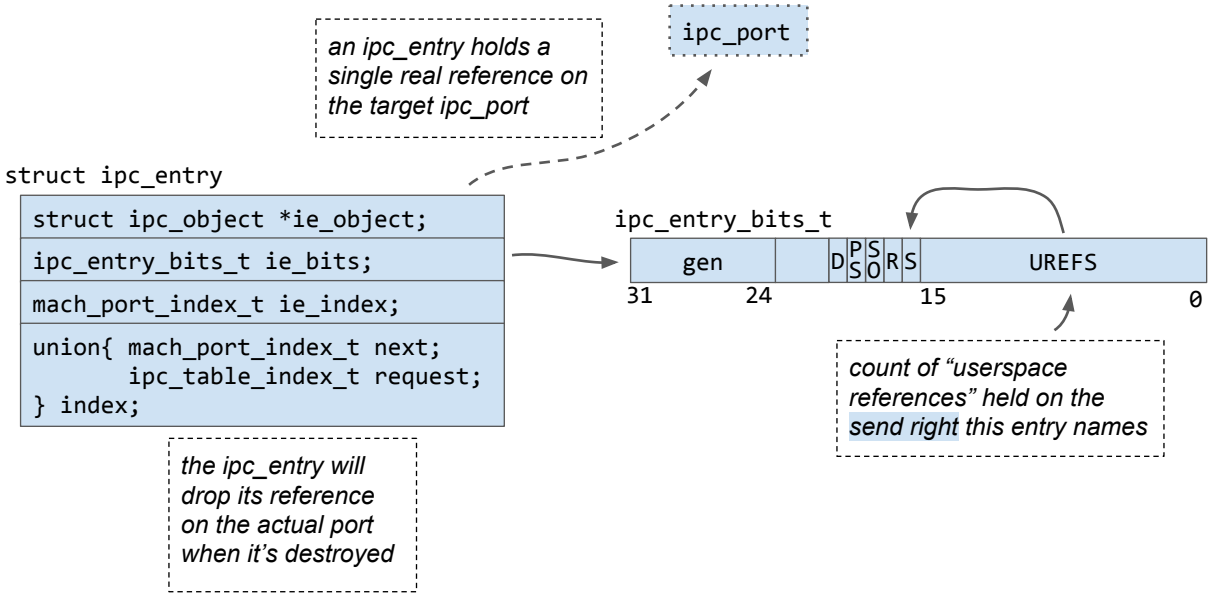
Note that it used to be the case that there was a much more complicated tree structure which held the ipc_entrys which allowed userspace to give ports arbitrary names (so you could do fun stuff like give a port the same name as a pointer to an object in your process that was associated with it (...) Now userspace has far less control over port names.

The generation number should jump out as interesting; why is it there? It's a mitigation :) As we'll see, managing mach port names is complicated and plenty of code gets is wrong.

What's in an entry?



Putting a refcount on your refcount



the ipc_entry holds one reference on the target port
UREFS "refcount" that reference :)

Both these reference count implementations had exploitable bugs.

UREFS are a reference count just for the send rights possessed by an ipc_entry. an ipc_entry can name multiple rights.

It the UREFS go to zero and the entry only names a send right then the entry is freed and the reference the entry held on the ipc_port is dropped.

Managing ipc_entry lifetime

ipc_entry will be freed when it no longer names any rights

Managing ipc_entry lifetime (userspace)

```
mach_port_mod_refs(
        ipc_space_t task,
        mach_port_name_t name,
        mach_port_right_t right,
        mach_port_delta_t delta);
```

if `right` is MACH_PORT_RIGHT_SEND **and** `name` names a send right:
  then: add the signed `delta` value to the name's `UREFS`

else if right is MACH_PORT_RIGHT_RECEIVE **and** `name` names a receive right **and** `delta` is `-1`:
  then: destroy the receive right

```
mach_port_deallocate(
        ipc_space_t task,
        mach_port_name_t name);
```

if `name` names a send right:
  then: drop one `UREF`

```
mach_port_destroy(
        ipc_space_t task,
        mach_port_name_t name);
```

destroy all rights held by `name`

It's actually a very complicated web of APIs and conventions.

These are the three userspace APIs which code can use to manipulate mach port names.
They all have subtly different meanings and as we'll see later the kernel has mitigations (but only mitigations) against their incorrect usage.

An interesting comment when manipulating UREFS

```
case MACH_MSG_TYPE_PORT_SEND:
  assert(port->ip_srights > 0);
  if (bits & MACH_PORT_TYPE_SEND) {
    mach_port_urefs_t urefs = IE_BITS_UREFS(bits);

    assert(port->ip_srights > 1);
    assert(urefs > 0);
    assert(urefs < MACH_PORT_UREFS_MAX);

    if (urefs+1 == MACH_PORT_UREFS_MAX) {
      if (overflow) {
        /* leave urefs pegged to maximum */
        port->ip_srights--;
        ip_unlock(port);
        ip_release(port);
        return KERN_SUCCESS;
      }
```

both userspace and the kernel can manipulate the UREFS count

/* leave urefs pegged to maximum */

This code is called when the kernel is copying out any rights contained in a message to the receive

This means that UREFS are capped at MACH_PORT_UREFS_MAX
note that if overflow is true (which is was) then the copyout succeeds and the name will still be given to userspace

Is this an exploitable bug though?

An interesting comment when manipulating UREFS

```
case MACH_MSG_TYPE_PORT_SEND:
  assert(port->ip_srights > 0);
  if (bits & MACH_PORT_TYPE_SEND) {
    mach_port_urefs_t urefs = IE_BITS_UREFS(bits);

    assert(port->ip_srights > 1);
    assert(urefs > 0);
    assert(urefs < MACH_PORT_UREFS_MAX);

    if (urefs+1 == MACH_PORT_UREFS_MAX) {
      if (overflow) {
        /* leave urefs pegged to maximum */
        port->ip_srights--;
        ip_unlock(port);
        ip_release(port);
        return KERN_SUCCESS;
      }
```

what does pegged mean?

The bug is that the comment which says that this leaves urefs pegged to maximum is wrong!

Let's look at the mach_port_deallocate case:

ipc_right_dealloc

```
case MACH_PORT_TYPE_SEND: {
   port = (ipc_port_t) entry->ie_object;
   ... // handle dead names
   if (IE_BITS_UREFS(bits) == 1) {
     ... // handle dropping the last UREF
   } else {
     ip_unlock(port);
     entry->ie_bits = bits-1; /* decrement urefs */
     ipc_entry_modified(space, name, entry);
     is_write_unlock(space);
  }
```

This method should be the inverse of the previous function but it has no mention of "pegged UREFS"

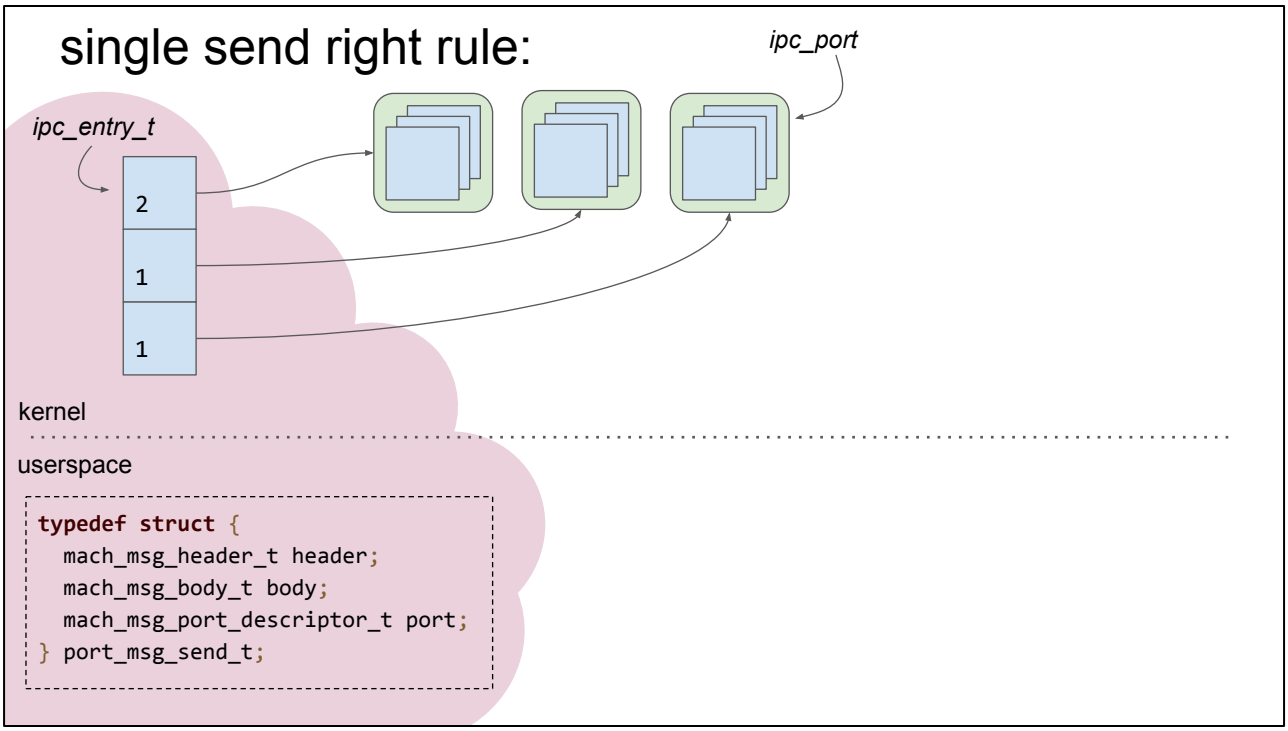
CVE-2016-7637

How is that bug relevant?

Can we actually exploit it to do anything useful?

was that those two things didn't match up.

The interesting part is how we can turn this into a privilege escalation

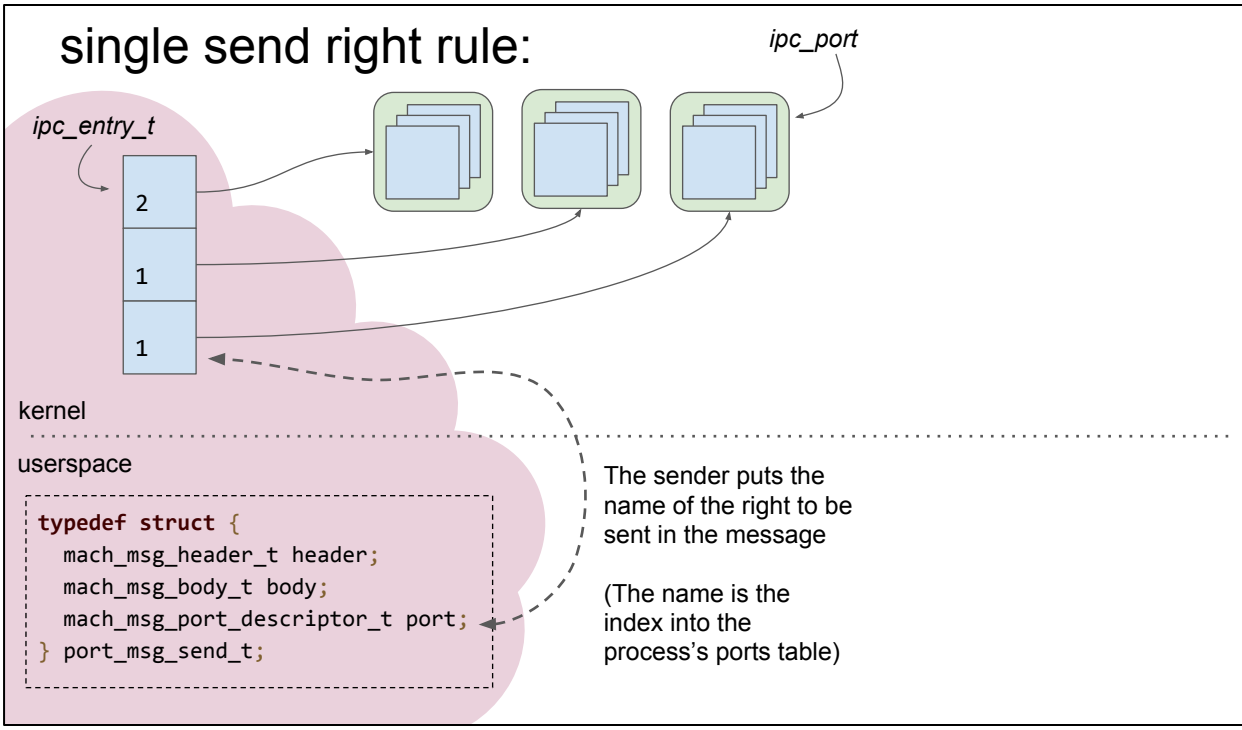


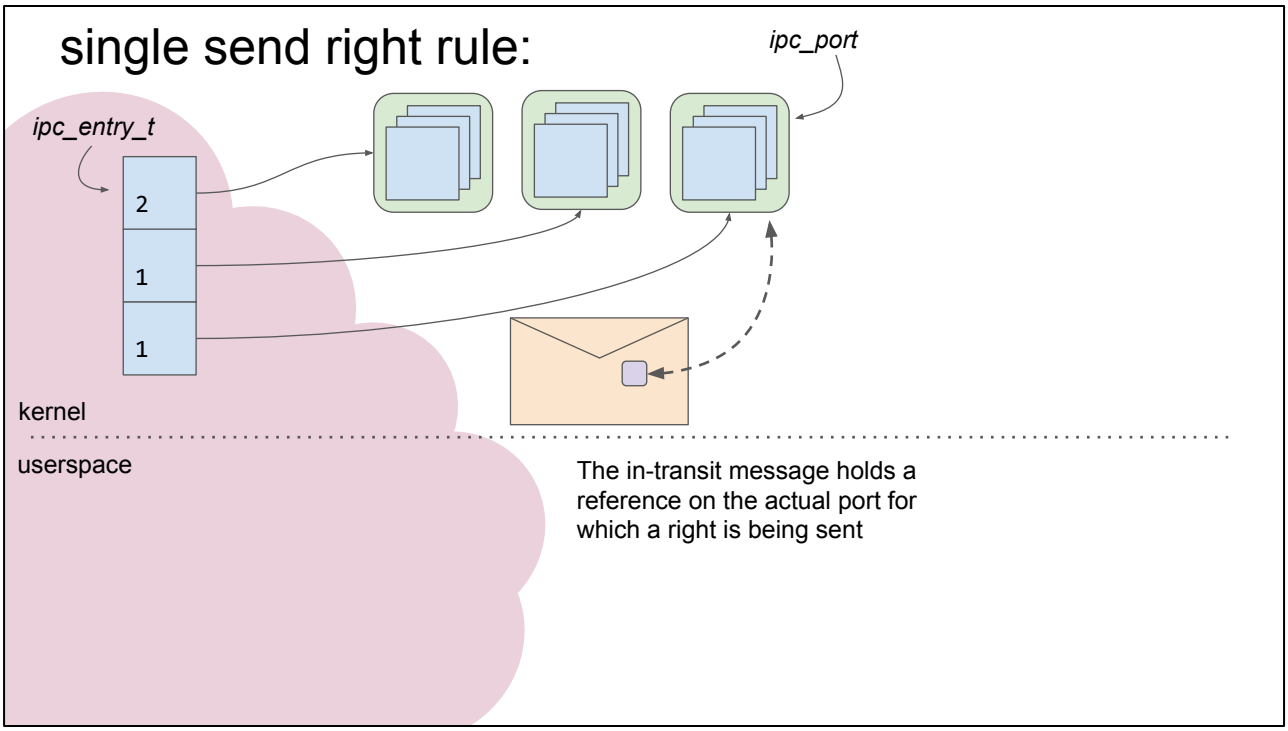
If we can send a mach message to another process then we have a LOT of control over it; this is how we will exploit the bug.

Whilst the bug is technically in the kernel it really just creates exploitable situations in userspace.

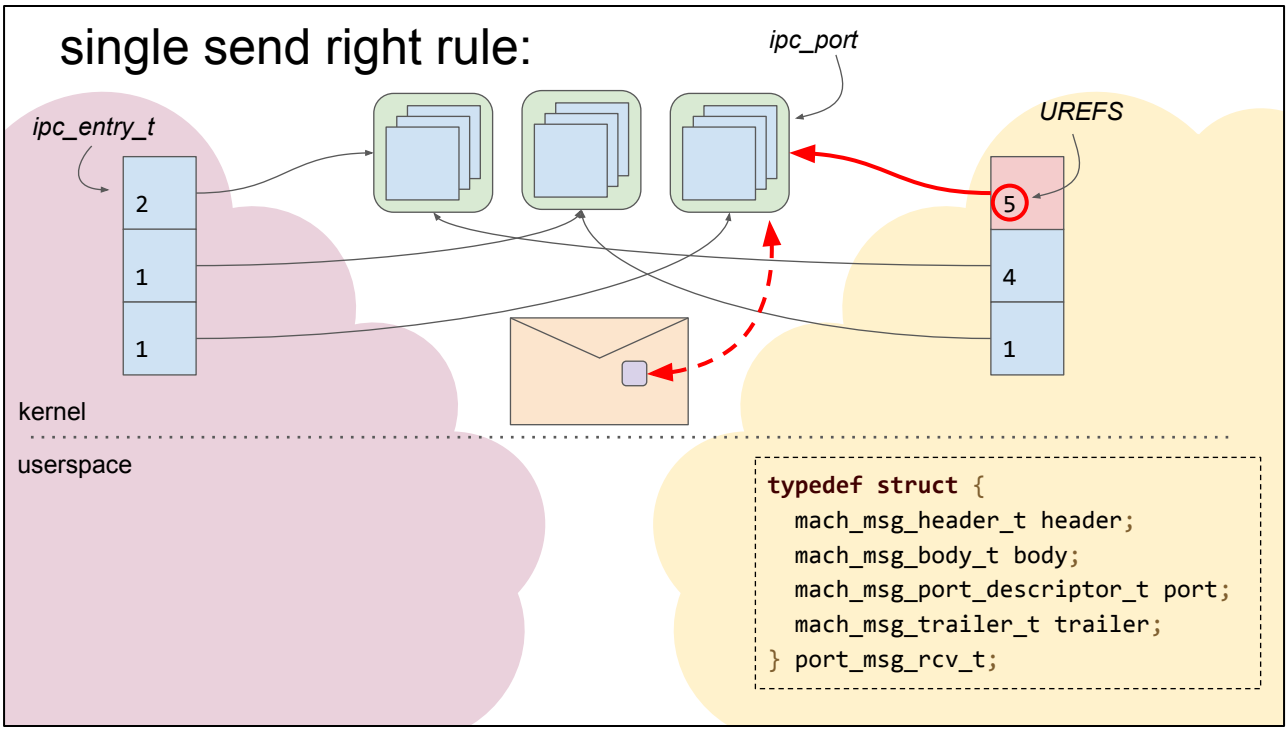
Our exploit will use a fundamental invariant of mach ports namespaces: you will only ever have one name for a send right to a particular mach port. If you ever get sent another send right for that port then you won't get a new name but instead the kernel will bump up the UREF of that name for you.

single send right rule:



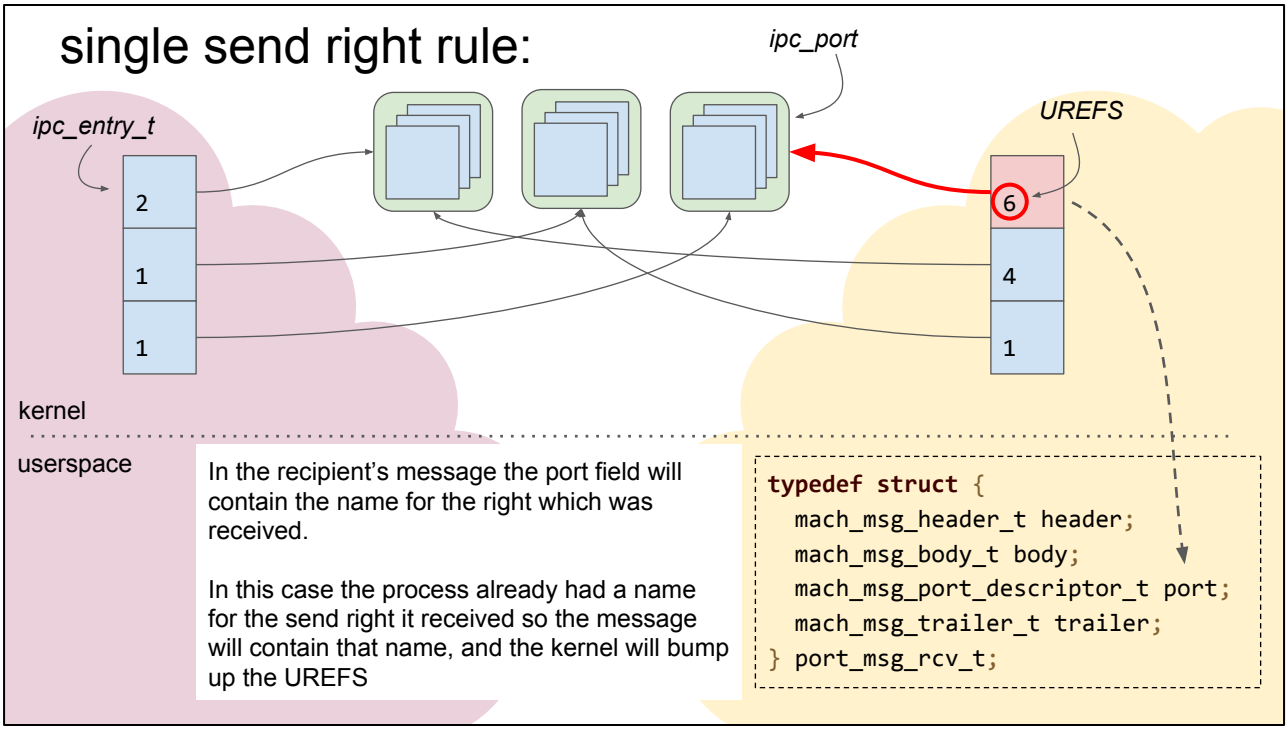


the in-transit port hold a reference on the port being sent
it also contains the rights which are being transferred. The rights were checked when the sender sent the message.



On the receive side the kernel will do a reverse lookup from the target ipc_port to see if that receiving process already has a name for that port (if it's being sent a send right)

single send right rule:



In the recipient's message the port field will contain the name for the right which was received.

In this case the process already had a name for the send right it received so the message will contain that name, and the kernel will bump up the UREFS

```
typedef struct {
  mach_msg_header_t header;
  mach_msg_body_t body;
  mach_msg_port_descriptor_t port;
  mach_msg_trailer_t trailer;
} port_msg_rcv_t;
```

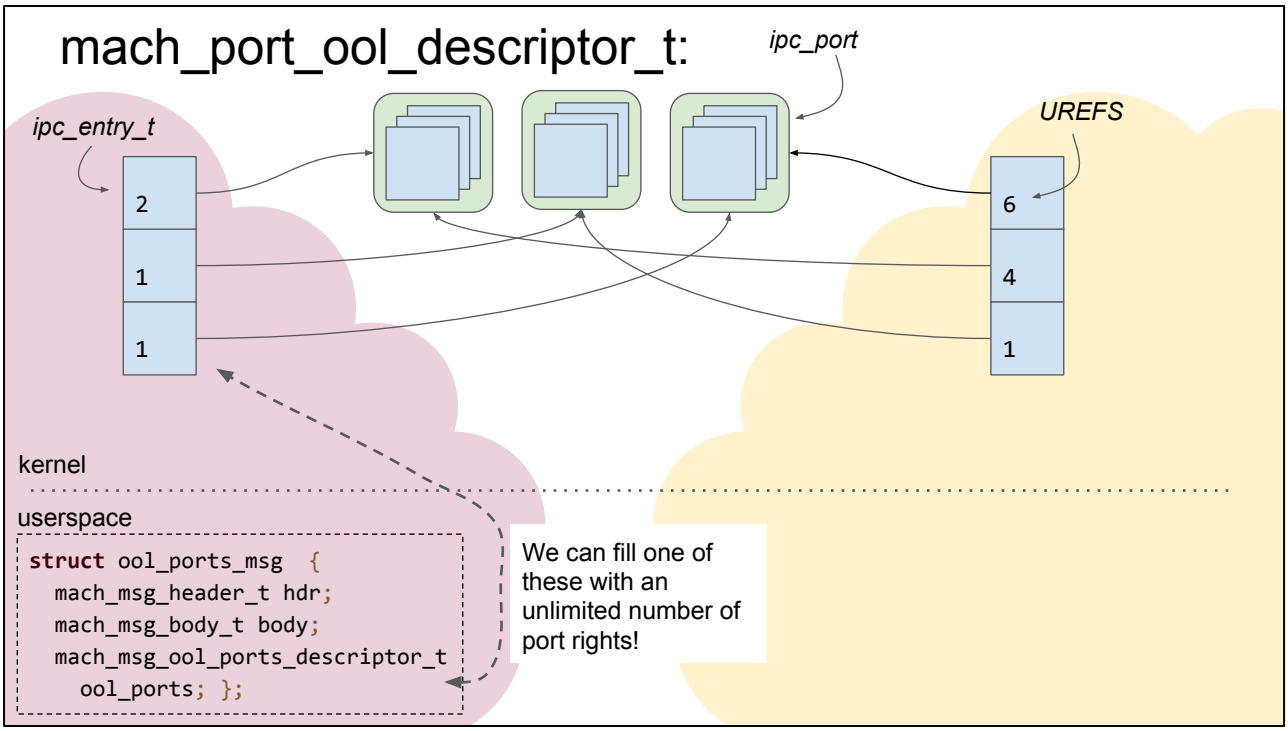
single send right rule:

Allows us to manipulate the UREFS of ipc_entries in remote processes

Requirements:

- we have a send right to the remote process
- we have a send right to the same port in the remote process that we want to manipulate

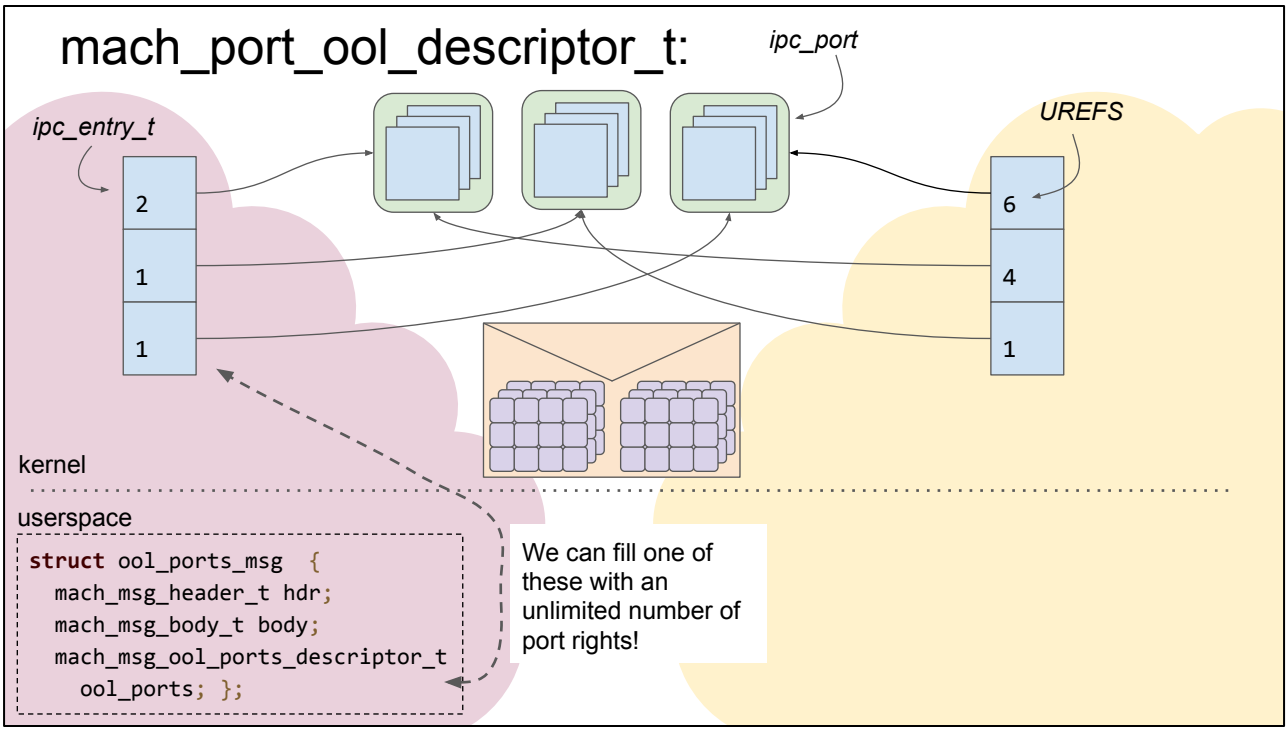
Couple more tricks to build a useful primitive...



We don’t just have to send one send right; we can send a lot at once in one message!

Note that now there is a sensible but still quite high limit imposed on the number of ports you can send like this.

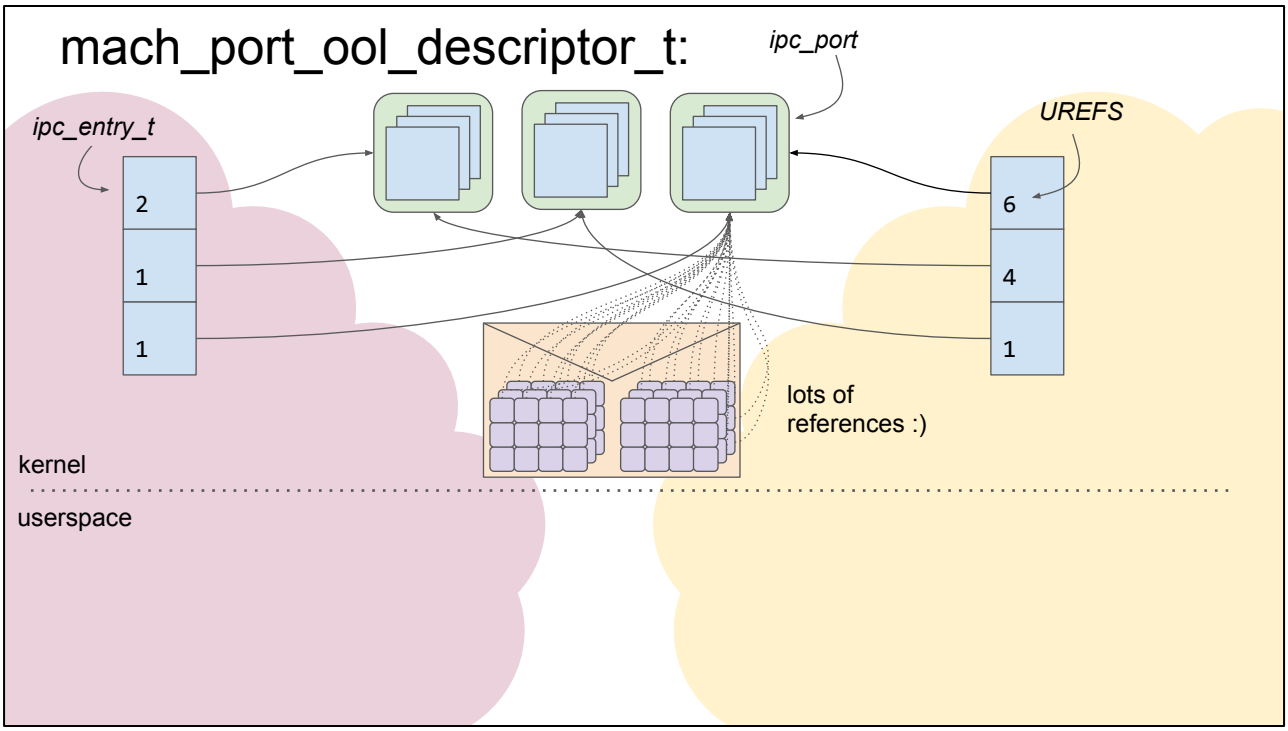
We’ll revisit exactly what that envelope looks like later because it gives you a very nice heap grooming/control primitive

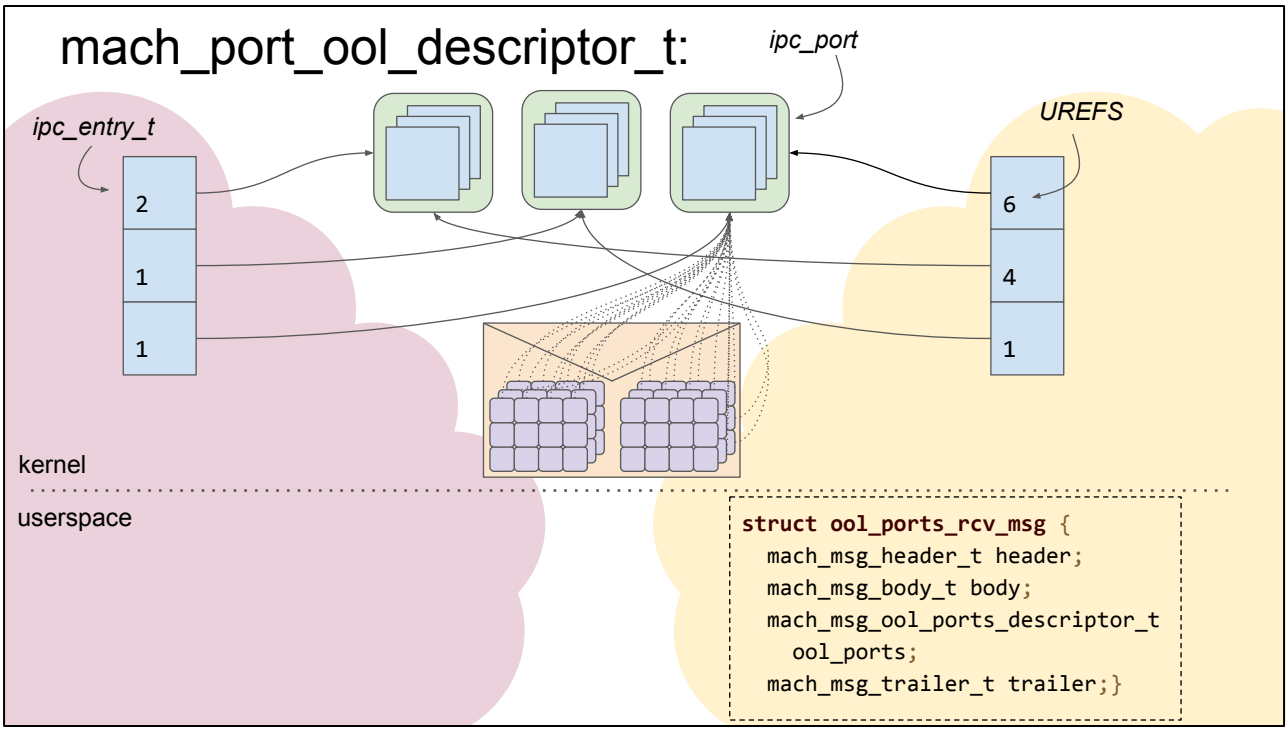


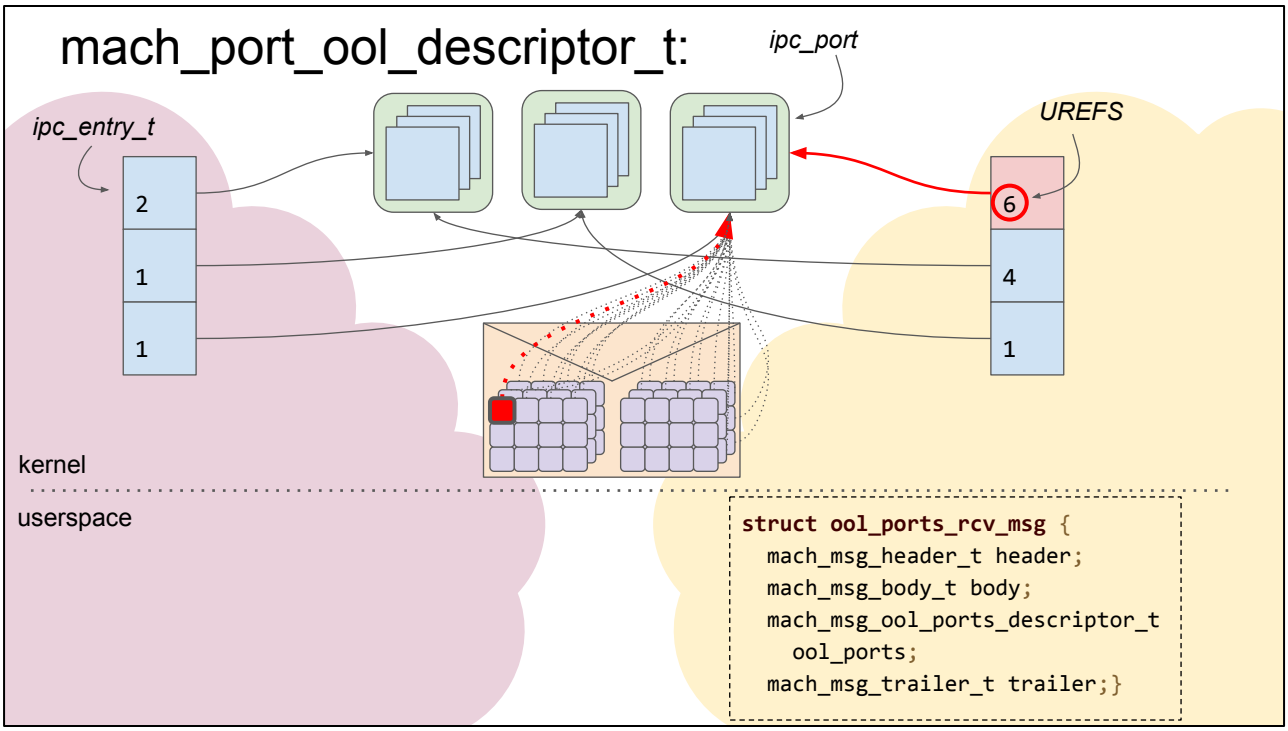
We don’t just have to send one send right; we can send a lot at once in one message!

Note that now there is a sensible but still quite high limit imposed on the number of ports you can send like this.

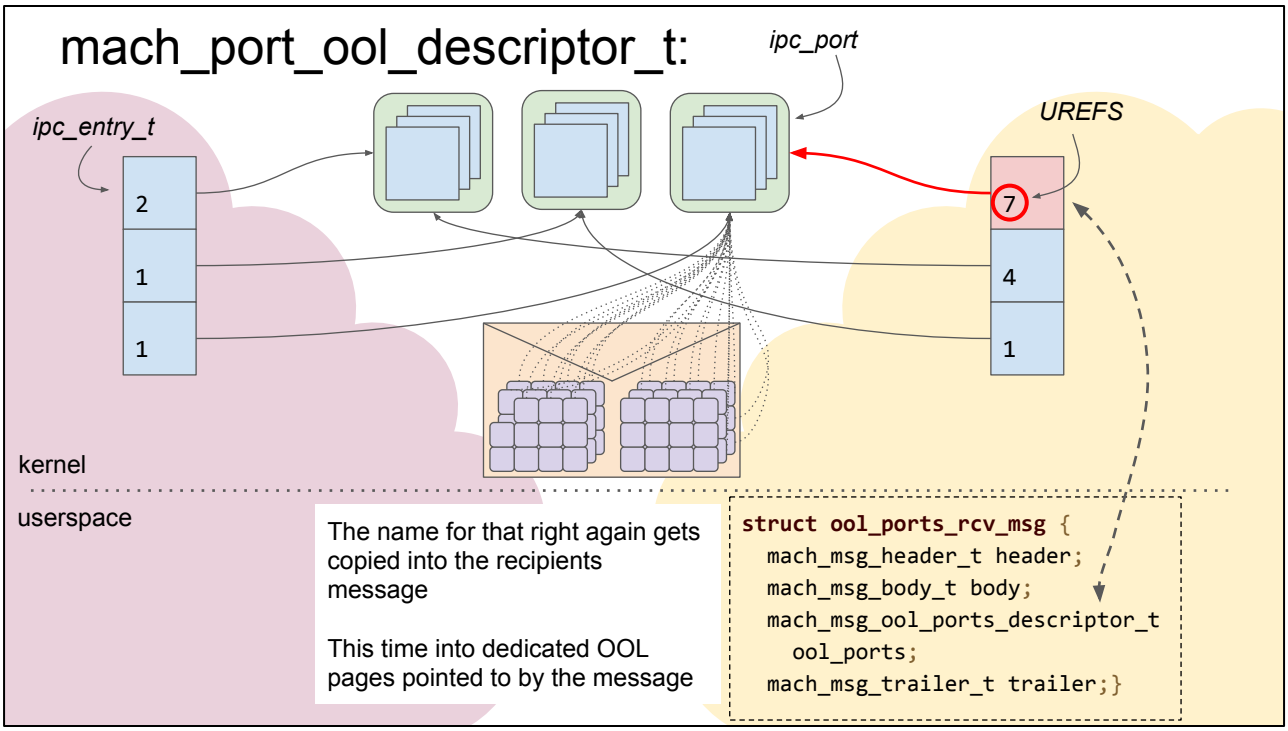
We’ll revisit exactly what that envelope looks like later because it gives you a very nice heap grooming/control primitive

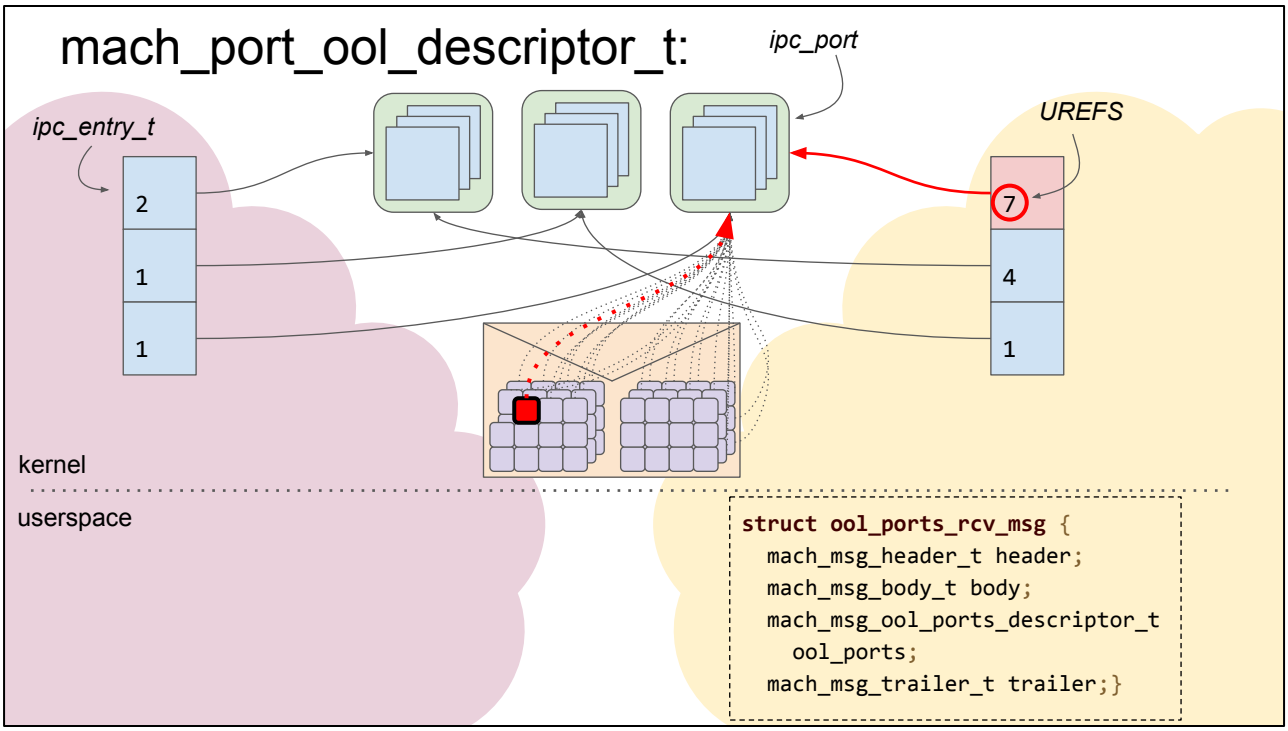

mach_port_ool_descriptor_t:
ipc_port
ipc_entry_t
UREFS
2
1
1
6
4
1
lots of
references :)
kernel
userspace

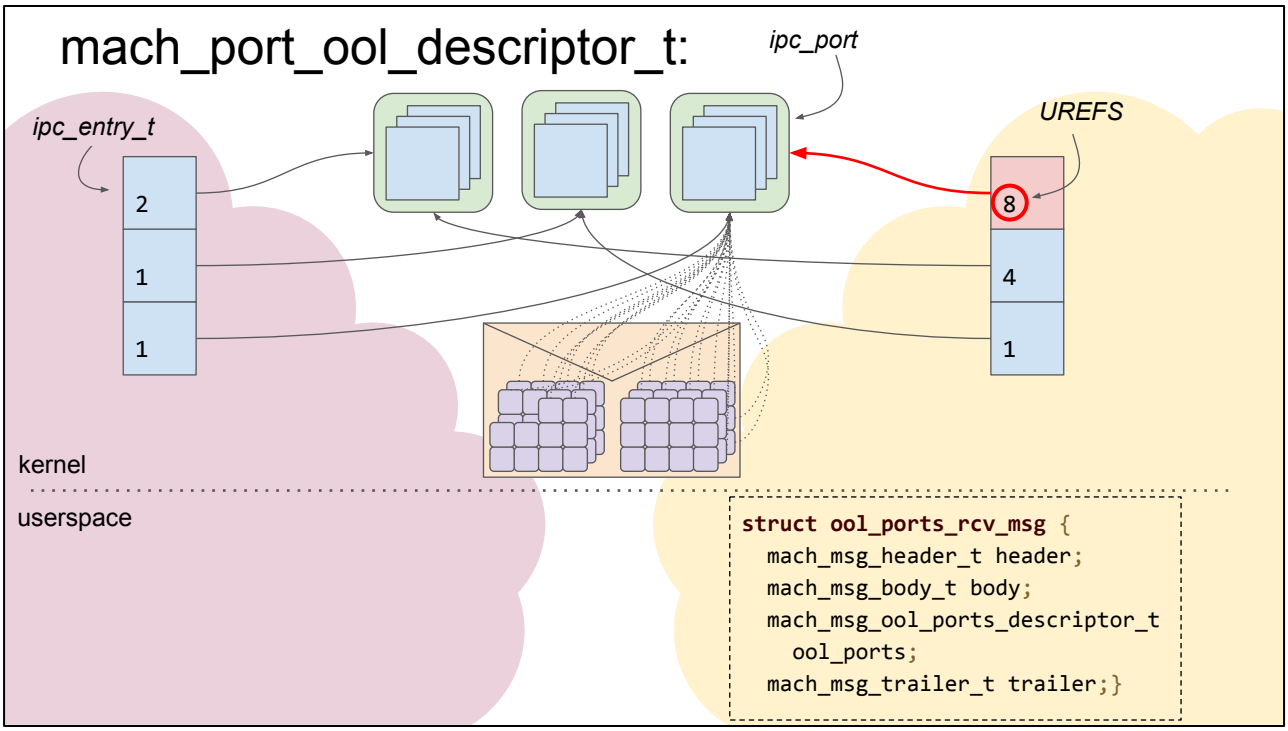

mach_port_ool_descriptor_t:
ipc_port
ipc_entry_t
UREFS
2
1
1
6
4
1
kernel
userspace
struct ool_ports_rcv_msg {
mach_msg_header_t header;
mach_msg_body_t body;
mach_msg_ool_ports_descriptor_t
ool_ports;
mach_msg_trailer_t trailer;}

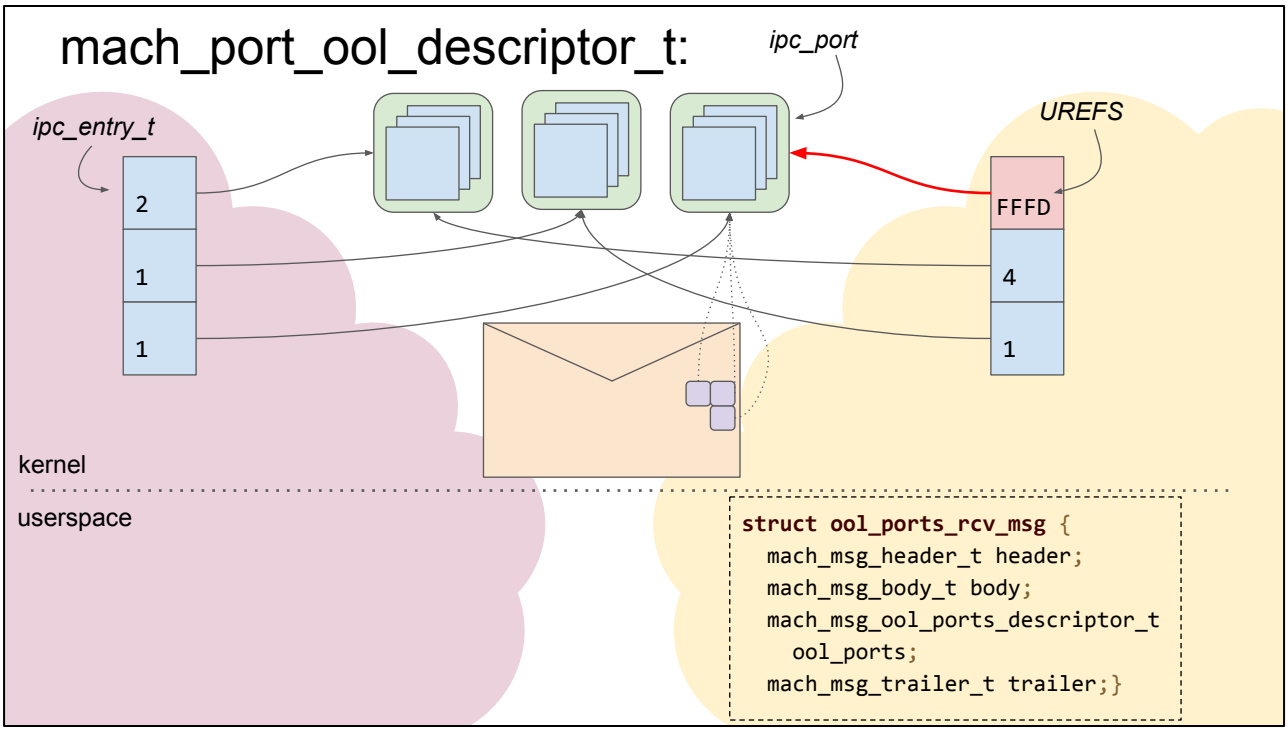


On receipt of the message each of those port references contained in the message will be converted to a UREF because we already have a name for the port (and that name names a send right and the message contains send rights)


mach_port_ool_descriptor_t:
ipc_port
ipc_entry_t
UREFS
2
1
1
7
4
1
kernel
userspace
The name for that right again gets copied into the recipients message
This time into dedicated OOL pages pointed to by the message
struct ool_ports_rcv_msg {
mach_msg_header_t header;
mach_msg_body_t body;
mach_msg_ool_ports_descriptor_t
ool_ports;
mach_msg_trailer_t trailer;}

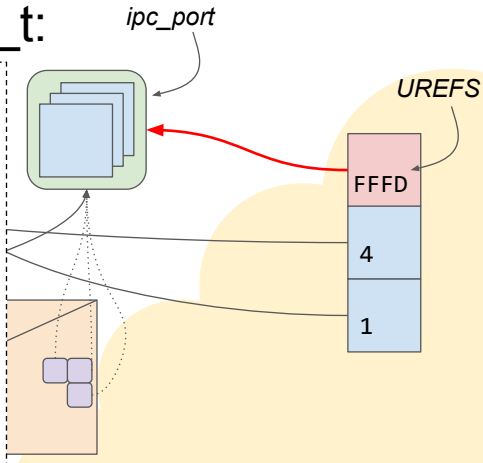

mach_port_ool_descriptor_t:
ipc_port
ipc_entry_t
UREFS
2
1
1
7
4
1
kernel
userspace
struct ool_ports_rcv_msg {
mach_msg_header_t header;
mach_msg_body_t body;
mach_msg_ool_ports_descriptor_t
ool_ports;
mach_msg_trailer_t trailer;}


mach_port_ool_descriptor_t:
ipc_port
ipc_entry_t
UREFS
2
1
1
8
4
1
kernel
userspace
struct ool_ports_rcv_msg {
mach_msg_header_t header;
mach_msg_body_t body;
mach_msg_ool_ports_descriptor_t
ool_ports;
mach_msg_trailer_t trailer;}


mach_port_ool_descriptor_t:
ipc_port
ipc_entry_t
UREFS
2
1
1
FFFD
4
1
kernel
userspace
struct ool_ports_rcv_msg {
mach_msg_header_t header;
mach_msg_body_t body;
mach_msg_ool_ports_descriptor_t
ool_ports;
mach_msg_trailer_t trailer;}

mach_port_ool_descriptor_t:

```
#define   MACH_PORT_UREFS_MAX
((mach_port_urefs_t) ((1 << 16) - 1))

if (urefs+1 == MACH_PORT_UREFS_MAX) {
      if (overflow) {
        /* leave urefs pegged to maximum */
        port->ip_srights--;
        ip_unlock(port);
        ip_release(port);
        return KERN_SUCCESS;
      }
```

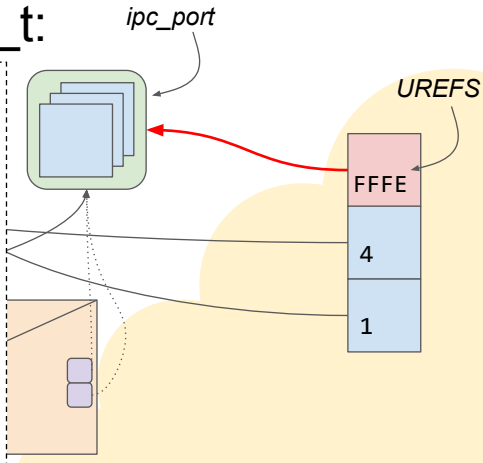



```
struct ool_ports_rcv_msg {
  mach_msg_header_t header;
  mach_msg_body_t body;
  mach_msg_ool_ports_descriptor_t
    ool_ports;
  mach_msg_trailer_t trailer;}
```

what happens when the UREFS approaches MACH_PORT_UREFS_MAX?

mach_port_ool_descriptor_t:

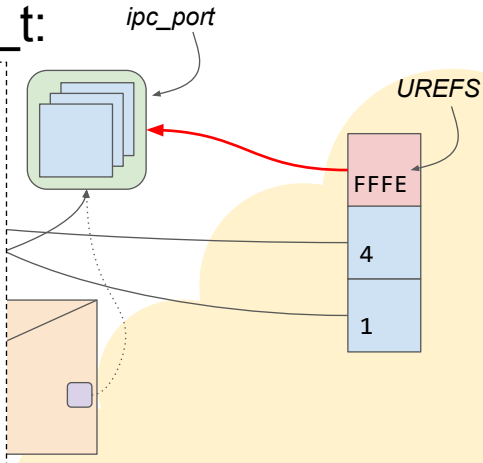
```
#define   MACH_PORT_UREFS_MAX
((mach_port_urefs_t) ((1 << 16) - 1))

if (urefs+1 == MACH_PORT_UREFS_MAX) {
      if (overflow) {
        /* leave urefs pegged to maximum */
        port->ip_srights--;
        ip_unlock(port);
        ip_release(port);
        return KERN_SUCCESS;
      }
```




```
struct ool_ports_rcv_msg {
  mach_msg_header_t header;
  mach_msg_body_t body;
  mach_msg_ool_ports_descriptor_t
    ool_ports;
  mach_msg_trailer_t trailer;}
```

mach_port_ool_descriptor_t:

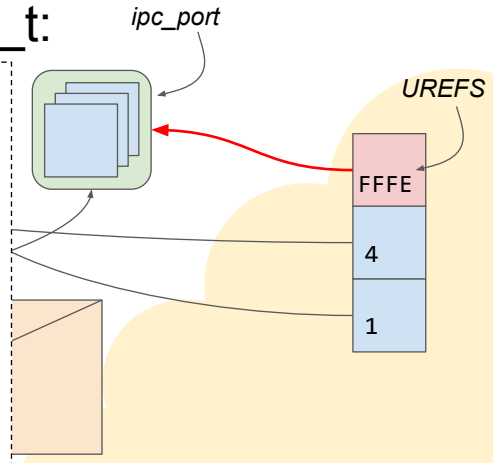
```
#define   MACH_PORT_UREFS_MAX
((mach_port_urefs_t) ((1 << 16) - 1))

if (urefs+1 == MACH_PORT_UREFS_MAX) {
      if (overflow) {
        /* leave urefs pegged to maximum */
        port->ip_srights--;
        ip_unlock(port);
        ip_release(port);
        return KERN_SUCCESS;
      }
```




```
struct ool_ports_rcv_msg {
  mach_msg_header_t header;
  mach_msg_body_t body;
  mach_msg_ool_ports_descriptor_t
    ool_ports;
  mach_msg_trailer_t trailer;}
```

mach_port_ool_descriptor_t:

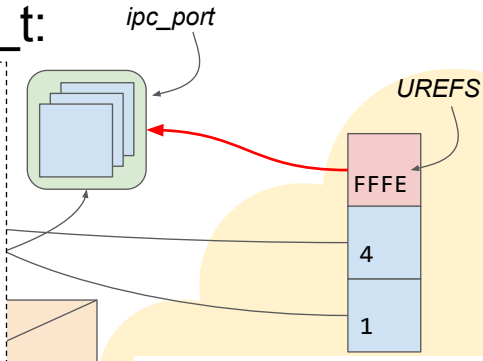
```
#define    MACH_PORT_UREFS_MAX
((mach_port_urefs_t) ((1 << 16) - 1))

if (urefs+1 == MACH_PORT_UREFS_MAX) {
      if (overflow) {
        /* leave urefs pegged to maximum */
        port->ip_srights--;
        ip_unlock(port);
        ip_release(port);
        return KERN_SUCCESS;
      }
```




```
struct ool_ports_rcv_msg {
  mach_msg_header_t header;
  mach_msg_body_t body;
  mach_msg_ool_ports_descriptor_t
    ool_ports;
  mach_msg_trailer_t trailer;}
```

mach_port_ool_descriptor_t:

```
#define   MACH_PORT_UREFS_MAX
((mach_port_urefs_t) ((1 << 16) - 1))

if (urefs+1 == MACH_PORT_UREFS_MAX) {
      if (overflow) {
        /* leave urefs pegged to maximum */
        port->ip_srights--;
        ip_unlock(port);
        ip_release(port);
        return KERN_SUCCESS;
      }
```



These last two port rights in the message will still result in a name being copied into the ool descriptor in the recipient process, but the UREFS for that name will no longer increase

Implications:

Nothing bad has actually happened yet!

But the UREFS count is now out-of-sync with the number of actual user references the process believes it has

An observation:

The sender of the message was able to define the format of the message they sent.

The kernel does not enforce any particular format or schema on mach messages; only that they are valid

The mere act of receiving a message gives your process all the rights contained in it, wanted or not

mach_msg_destroy:

```
/*
 *   Routine:  mach_msg_destroy
 *   Purpose:
 *        mach_msg_destroy is useful in two contexts.
 *
 *        First, it can deallocate all port rights and
 *        out-of-line memory in a received message.
 *        When a server receives a request it doesn't want,
 *        it needs this functionality.
 *
 *        Second, it can mimic the side-effects of a msg-send
 *        operation.  The effect is as if the message were sent
 *        and then destroyed inside the kernel.  When a server
 *        can't send a reply (because the client died),
 *        it needs this functionality.
 */
void
mach_msg_destroy(mach_msg_header_t *msg);
```

part of libsyscall

pretty much every service uses this to handle unrecognised messages (including every XPC + MIG service)

There's an equivalent implementation in the kernel

performs the inverse operation of receiving an arbitrary message

mach_msg_destroy:

```
case MACH_MSG_OOL_PORTS_DESCRIPTOR: {
  mach_port_t                       *ports;
  mach_msg_ool_ports_descriptor_t   *dsc;
  mach_msg_type_number_t            j;

  /* Destroy port rights carried in the message */
  dsc = &daddr->ool_ports;
  ports = (mach_port_t *) dsc->address;
  for (j = 0; j < dsc->count; j++, ports++)  {
    mach_msg_destroy_port(*ports, dsc->disposition);
  }

  /* Destroy memory carried in the message */
  if (dsc->deallocate) {
    mach_msg_destroy_memory((vm_offset_t)dsc->address,
                            dsc->count * sizeof(mach_port_t));
  }
```

if this port is a send right, each destroy call will drop a UREF

Here's the implementation of mach_msg_destroy when it destroys an OOL_PORTS descriptor

freeing port names

UREFS hit an upper limit of 0xFFFE

What if we send a message with 0xFFFE copies of the same send right?

If we send a mach message with 0xFFFE copies of the same send right; it will reach the limit, regardless of how many UREFS the name had to begin with

mach_msg_destroy will call mach_port_deallocate 0xFFFE times; the last time will free the name, again regardless of how many UREFS the name had to begin with!

doing something useful with that:

- need a target tuple (target_process, target_send_right) where freeing target_send_right would be interesting
- probably need to get the exact name reused in a controlled way

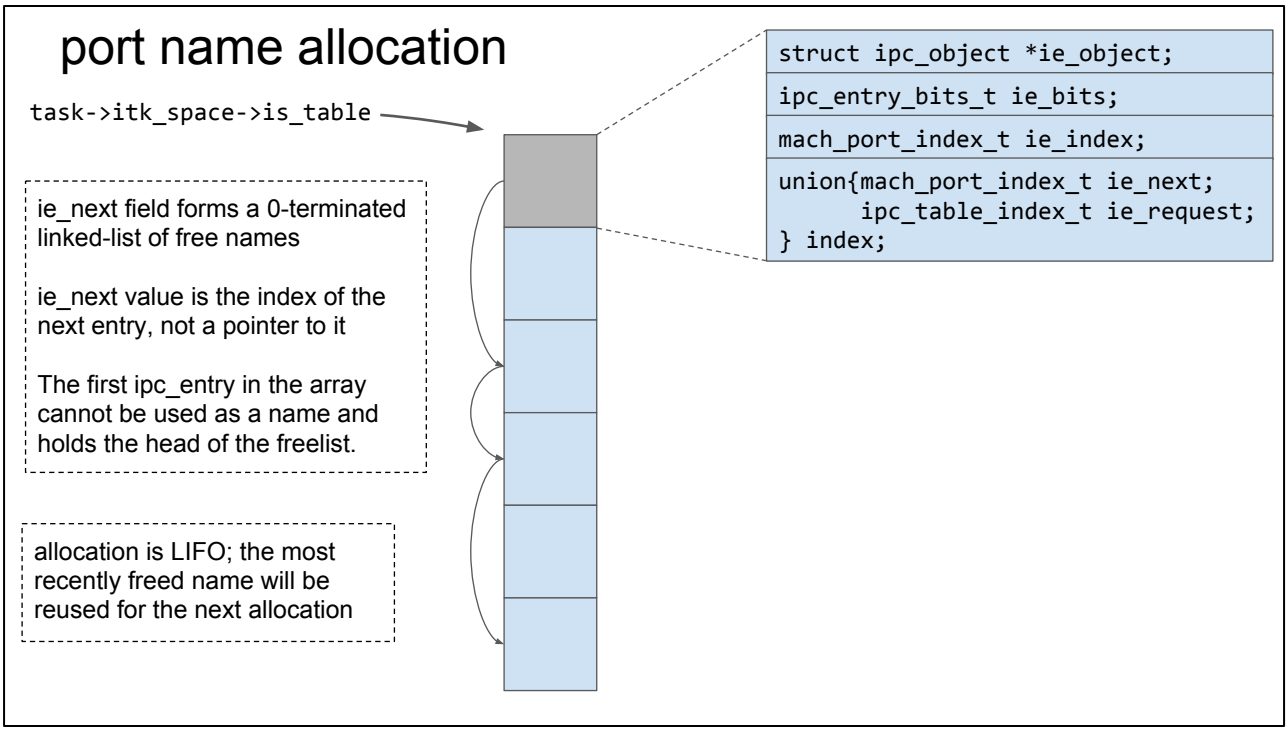
recall the mitigation:

```
/*
 *   For kernel-selected [assigned] port names, the name is
 *   comprised of two parts: a generation number and an index.
 *   This approach keeps the exact same name from being generated
 *   and reused too quickly [to catch right/reference counting bugs].
 */
```

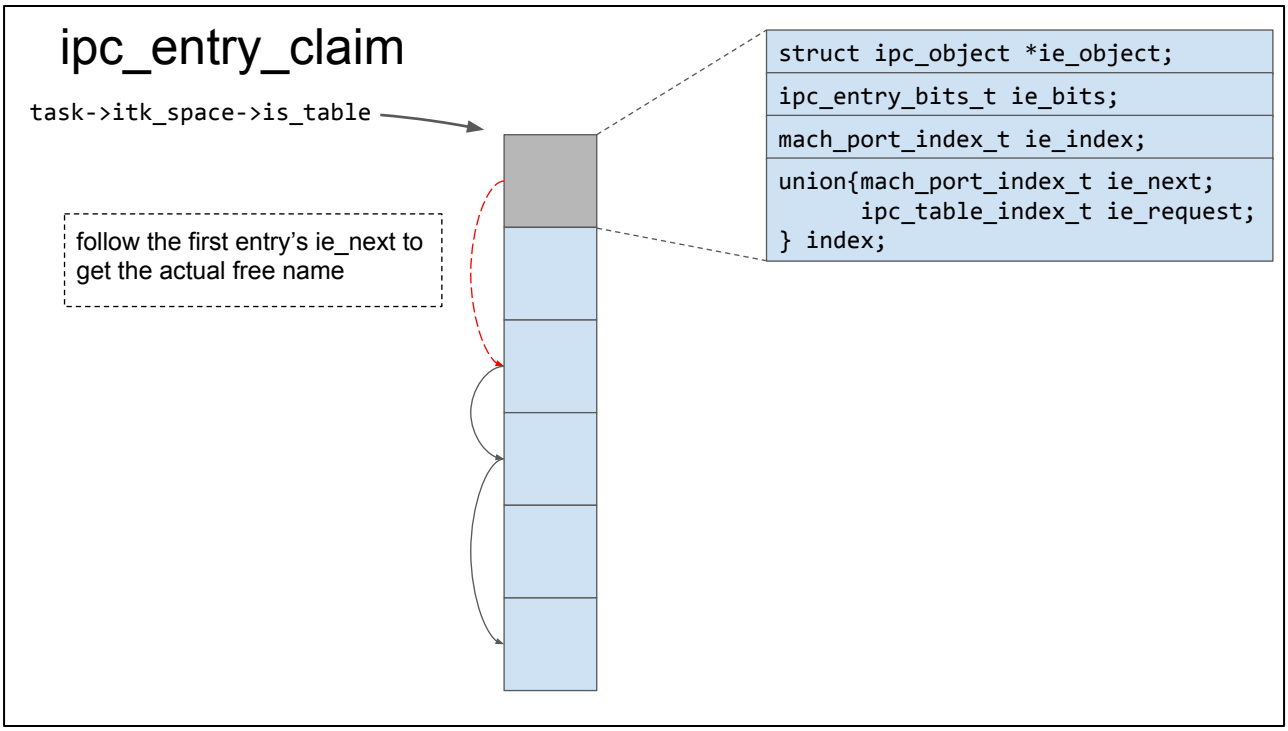
How quickly is too quickly?

Let's look at how names are allocated...

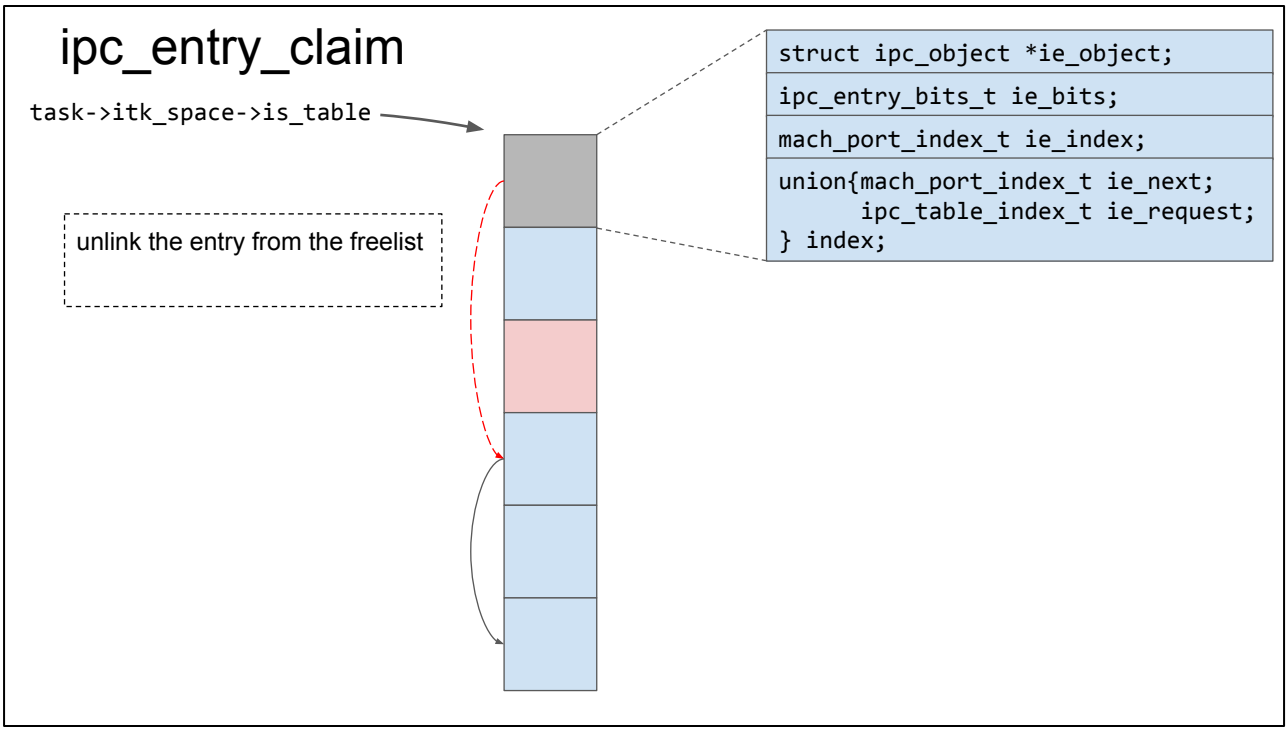
This mitigation was probably added not as a security mitigation but to improved stability.



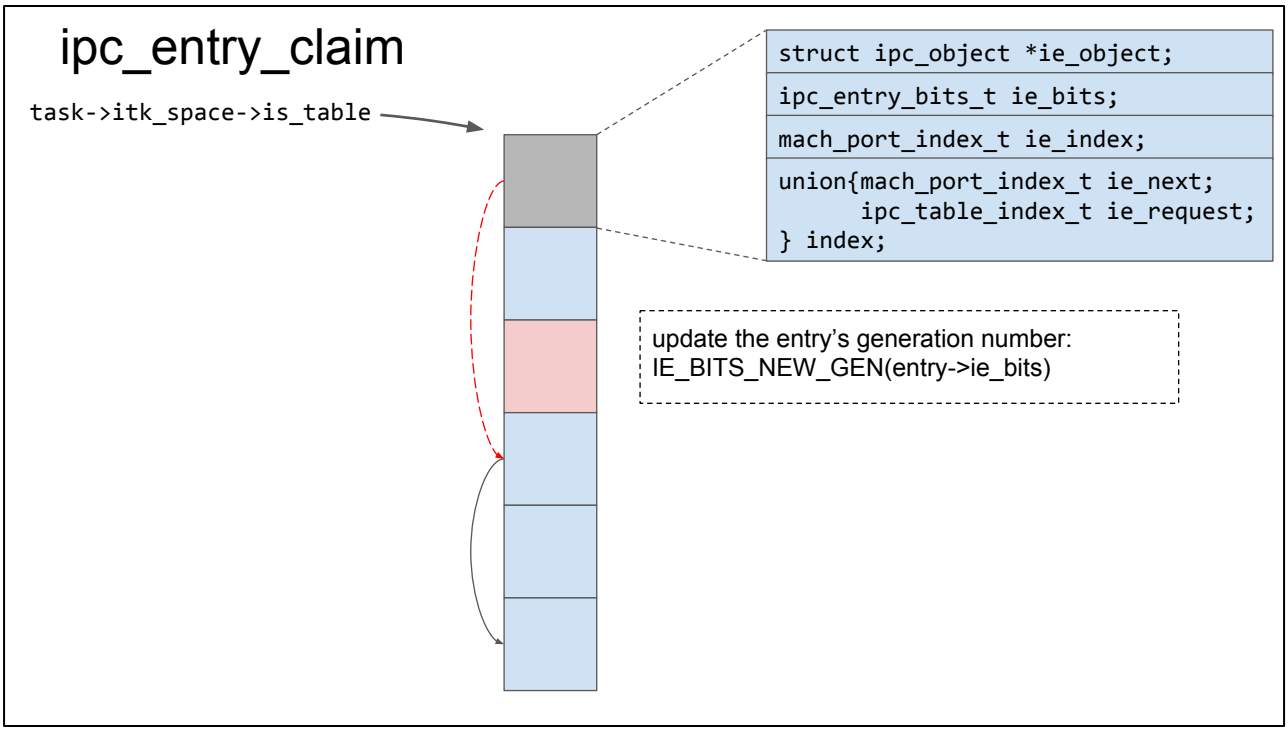
look at it like a weird heap. The generation number is a uaf mitigation. We can defeat that mitigation.
We have a *lot* of control by being able to send mach messages.



ipc_entry_claim will actually do the allocation of a new name.
Follow the first ie_next value to get the actual first_free name



ipc_entry_claim will actually do the allocation of a new name.
Follow the first ie_next value to get the actual first_free name



ipc_entry_claim will actually do the allocation of a new name.
Follow the first ie_next value to get the actual first_free name

Generation numbers

generation number of an entry is incremented each time it's reallocated (on allocation, not free)

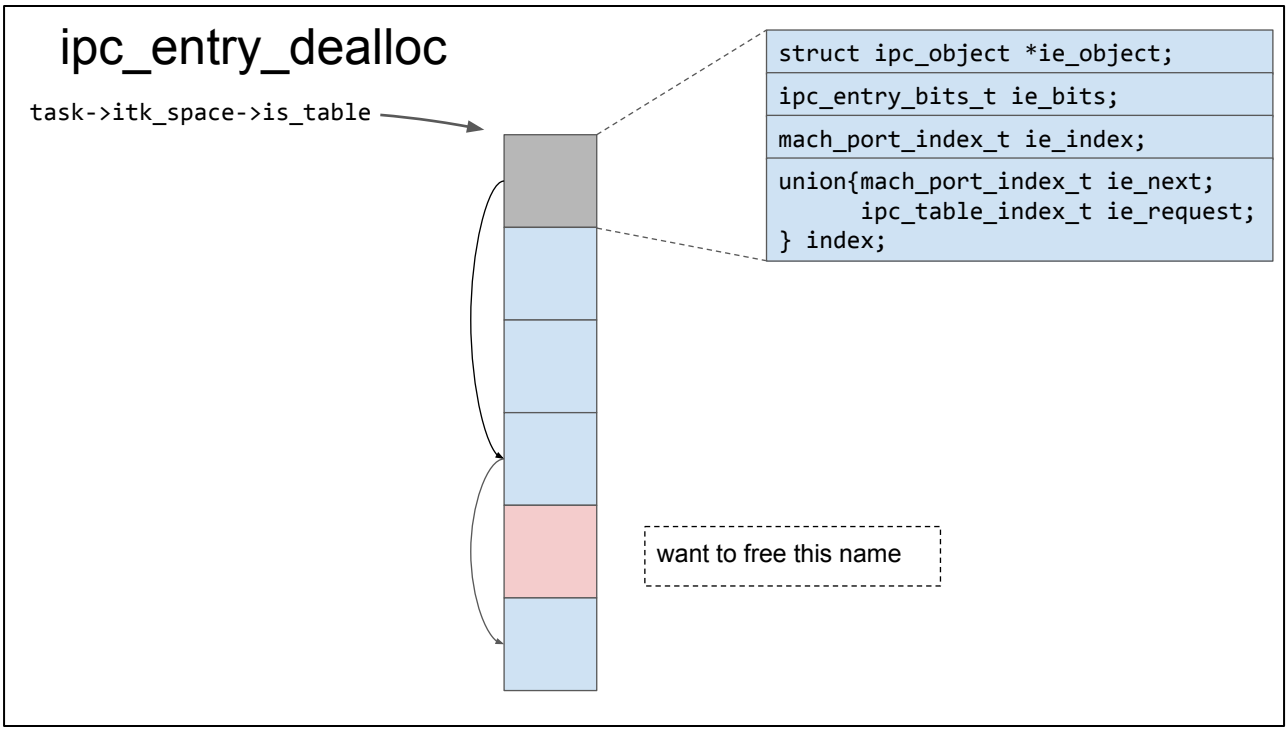
```
#define  IE_BITS_GEN_MASK      0xff000000  /* 8 bits for generation */
#define  IE_BITS_GEN(bits)     ((bits) & IE_BITS_GEN_MASK)
#define  IE_BITS_GEN_ONE       0x04000000  /* low bit of generation */
#define  IE_BITS_NEW_GEN(old)  (((old) + IE_BITS_GEN_ONE) & IE_BITS_GEN_MASK)
```

gives 6 bits for the generation number -> 64 generations before overflow

```
table[i].ie_bits = IE_BITS_GEN_MASK;
```

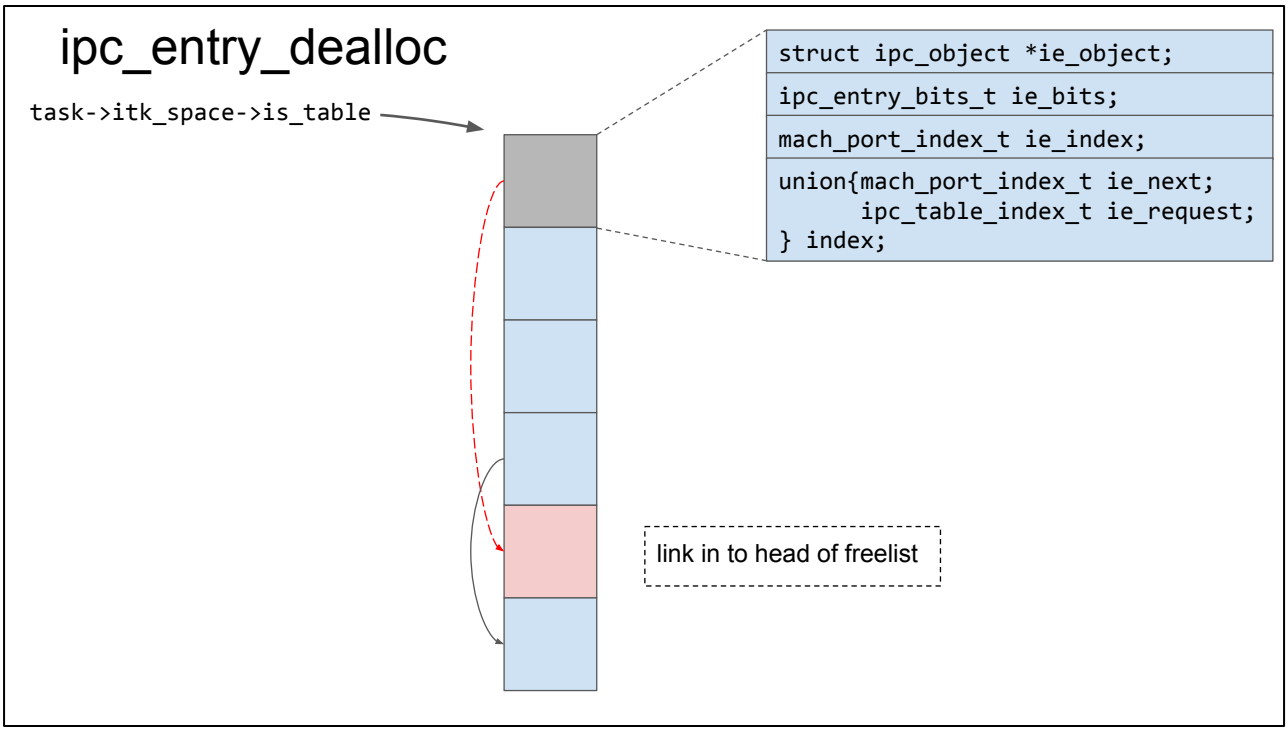
The generation number is initialized by setting it to the mask; not using the mask to clear the right bits to 0. Hence the first generation number is 3, not 4 :)

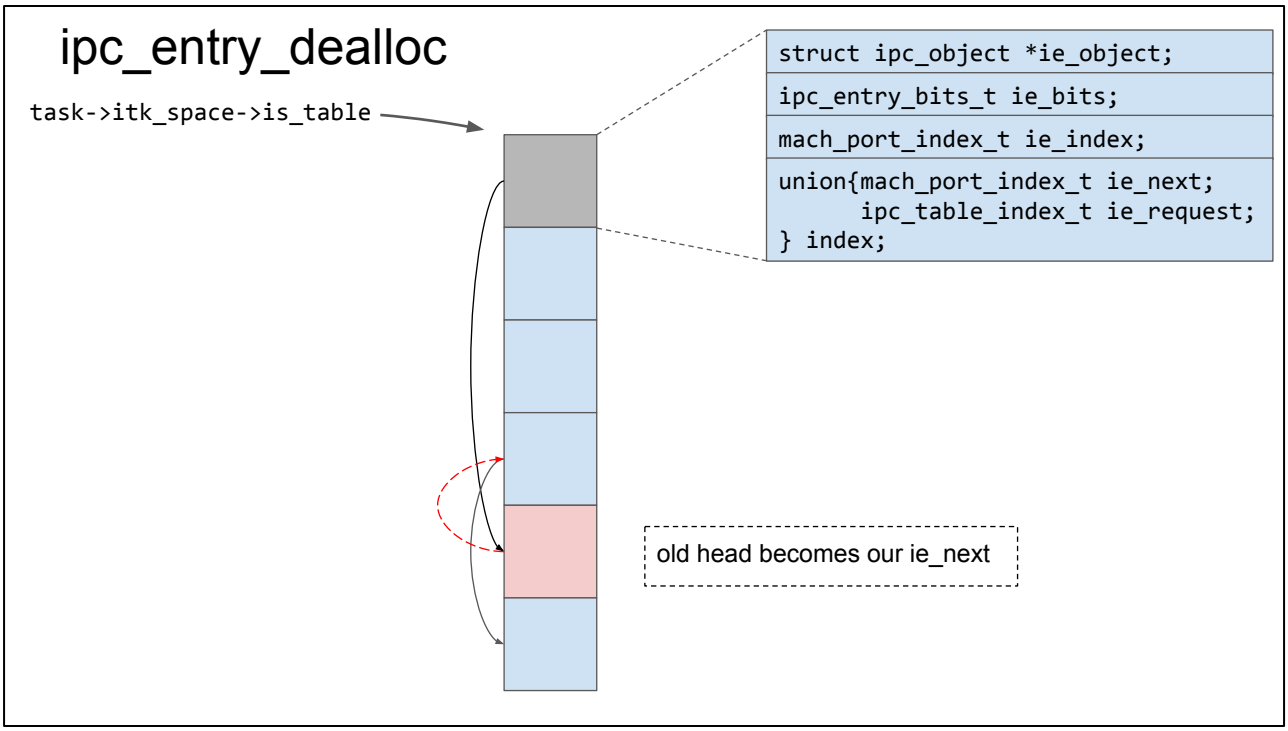
These generation numbers are checked on the userspace/kernel boundary when doing the conversion between names and ports

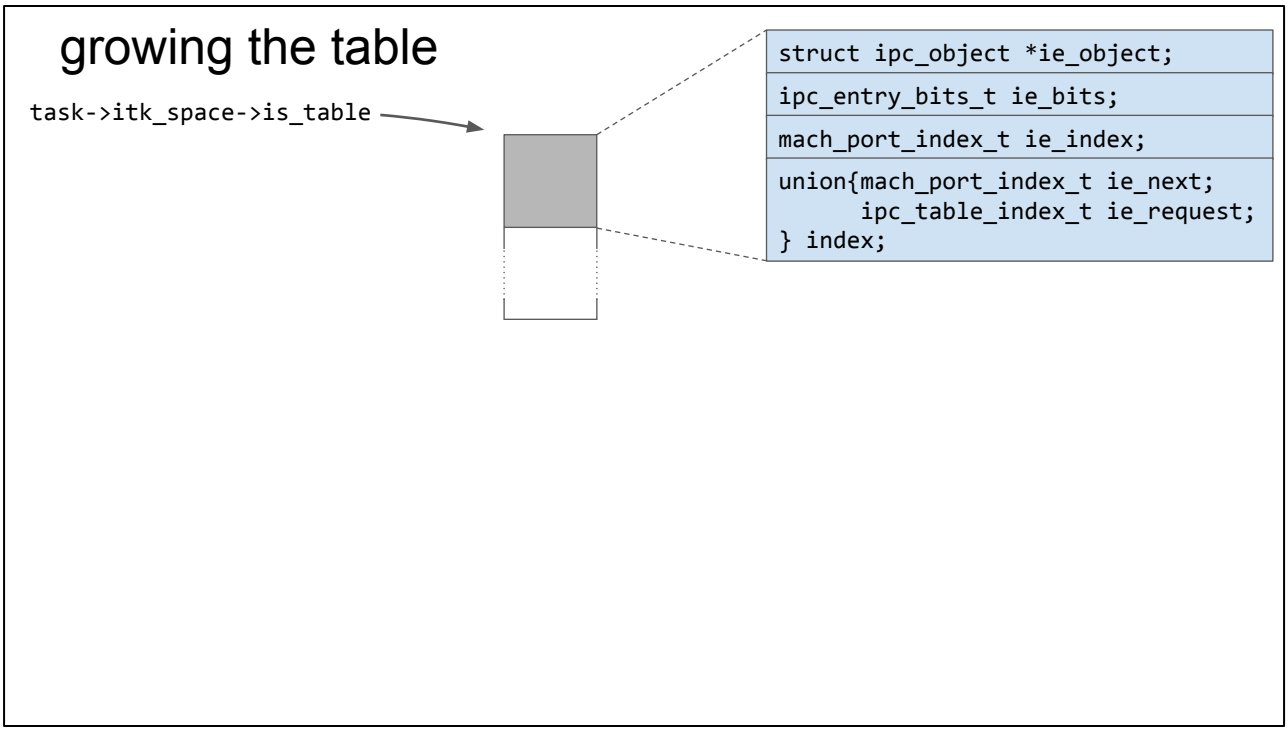


this function is responsible for freeing a name; we want to free this name.

The generation number has already been checked by the caller

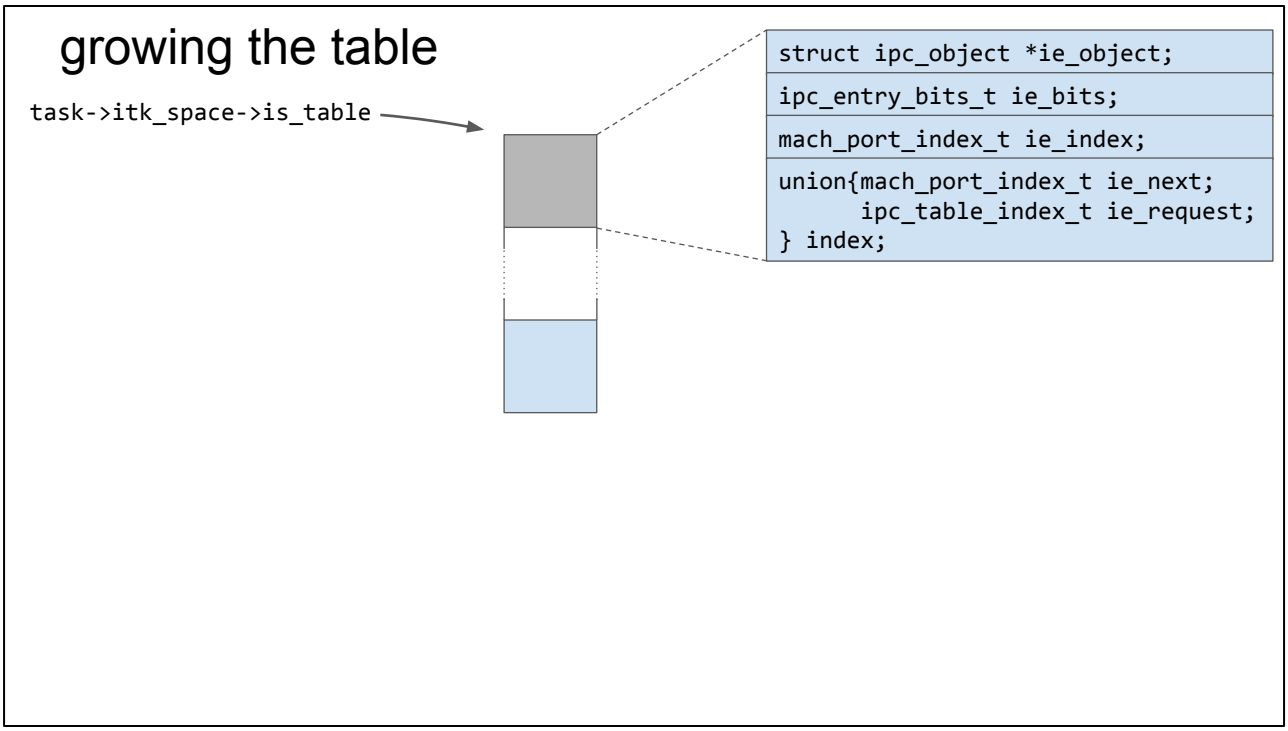

ipc_entry_dealloc
task->itk_space->is_table
struct ipc_object *ie_object;
ipc_entry_bits_t ie_bits;
mach_port_index_t ie_index;
union{mach_port_index_t ie_next;
ipc_table_index_t ie_request;
} index;
link in to head of freelist


ipc_entry_dealloc
task->itk_space->is_table
struct ipc_object *ie_object;
ipc_entry_bits_t ie_bits;
mach_port_index_t ie_index;
union{mach_port_index_t ie_next;
ipc_table_index_t ie_request;
} index;
old head becomes our ie_next



There’s only one more thing to look at and then we’ve covered *all* of the behaviour of the port name table; what happens when the table grows?

Before calling ipc_entry_claim callers have to call ipc_entry_hold which actually ensures that the table has enough free entries and tries to grow the table if not.

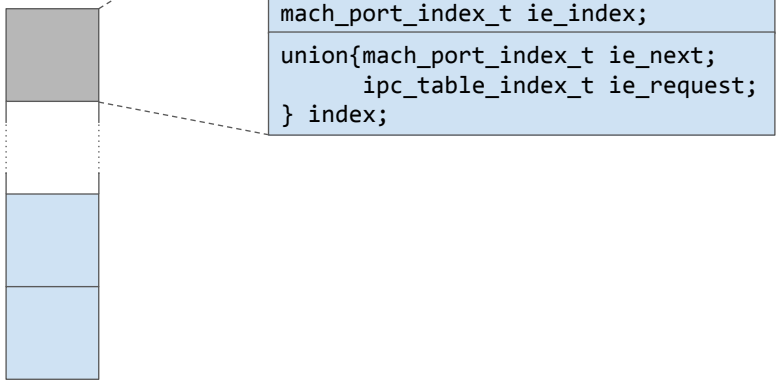


The actual behaviour of the linked list is simple though.

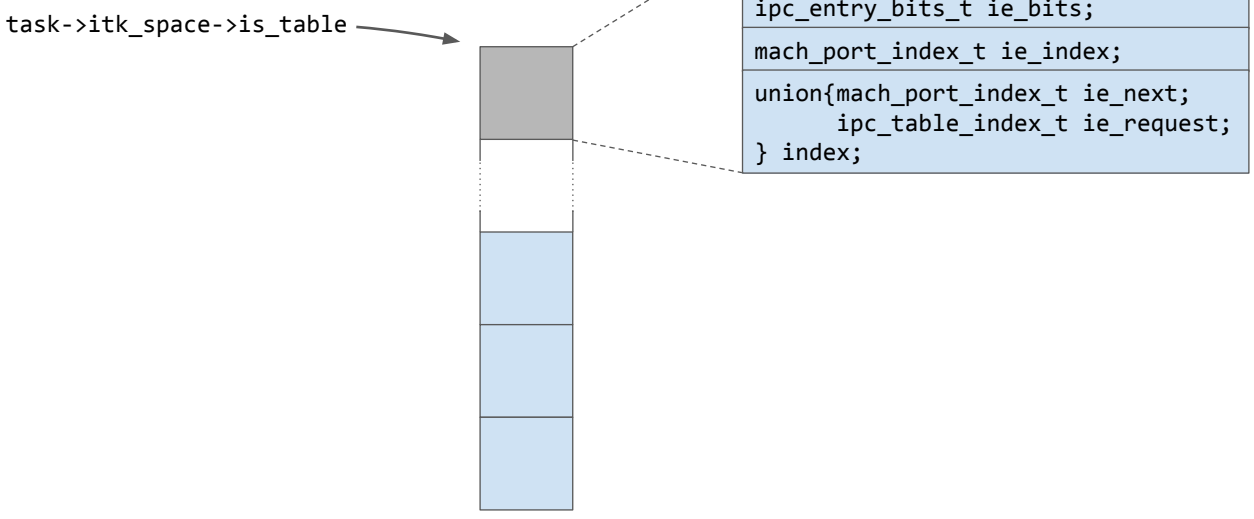
It just reallocates to a bigger heap chunk so we get more ipc_entry structures on the end.

growing the table

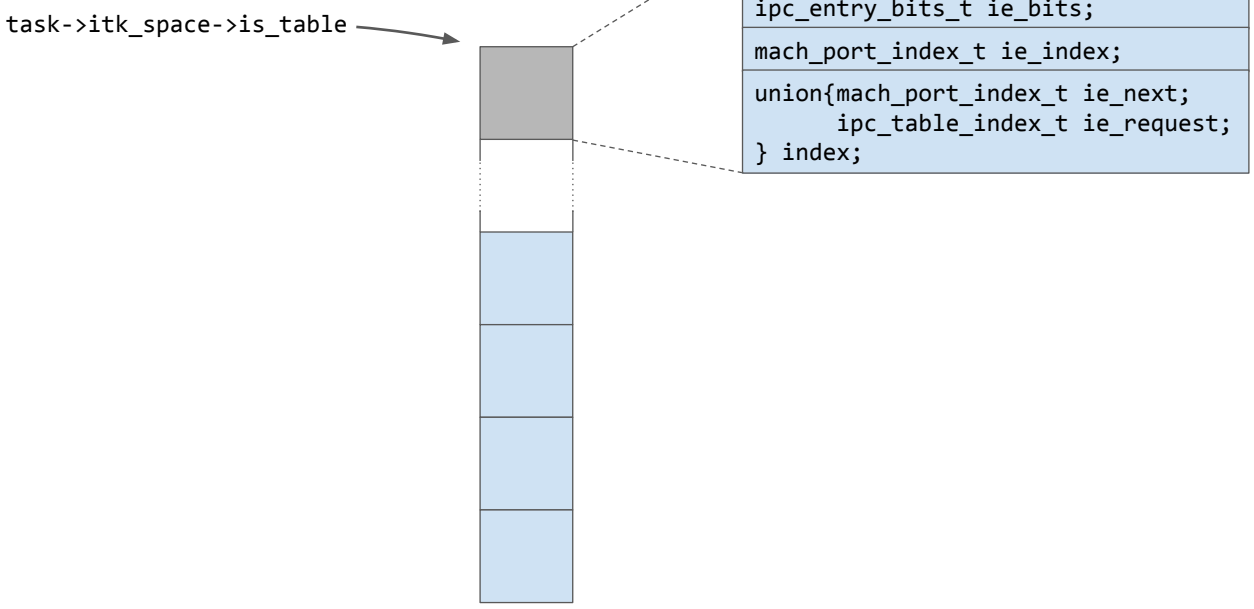
task->itk_space->is_table

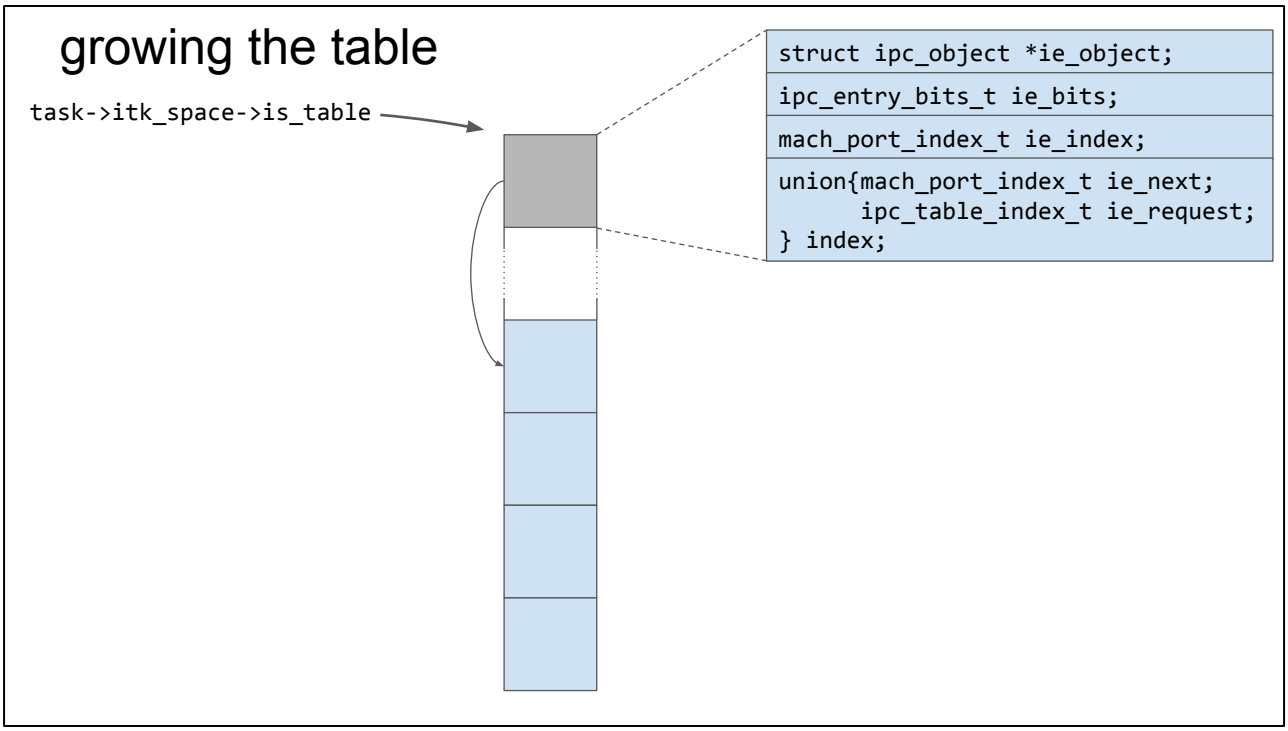


growing the table



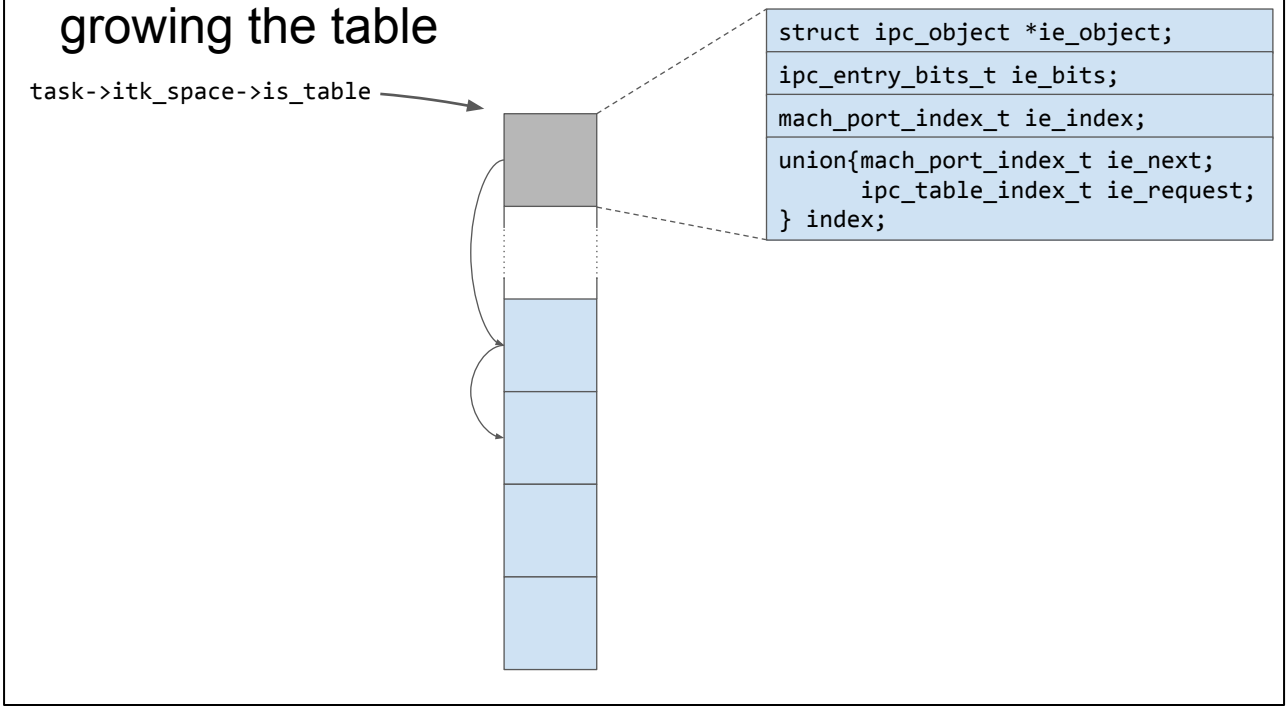
growing the table



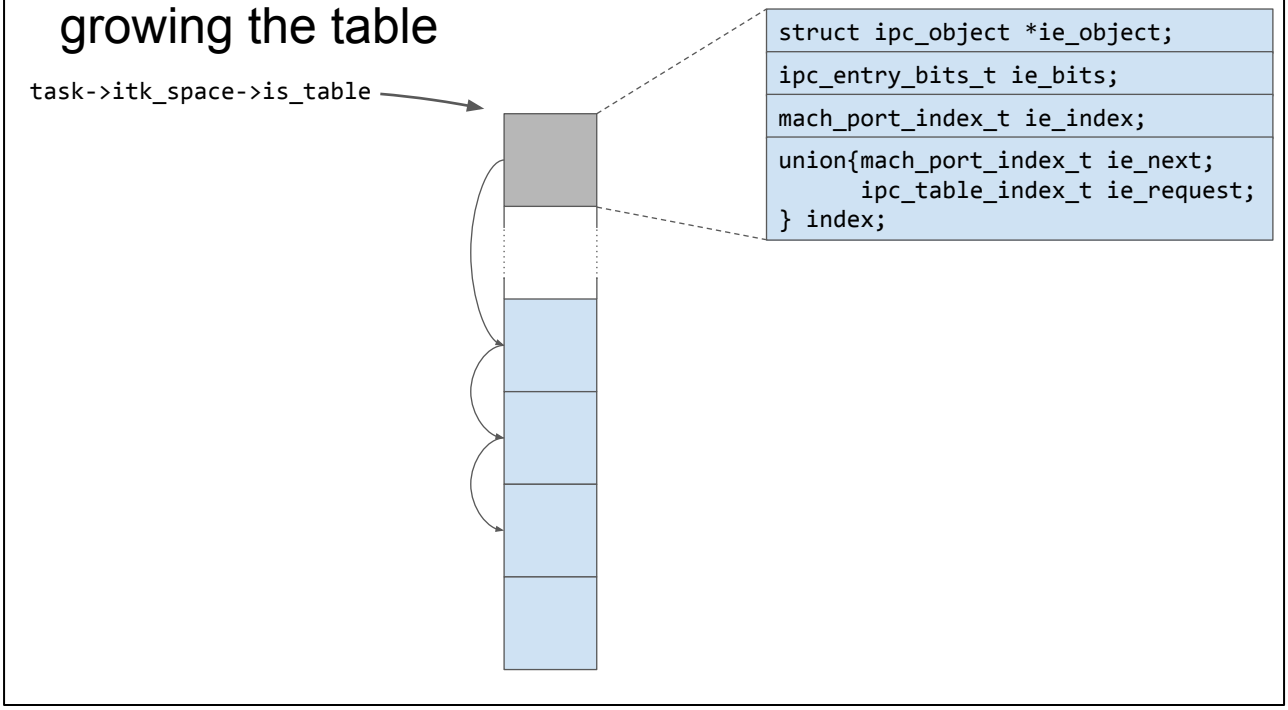


Then they’re linked in to the freelist in order

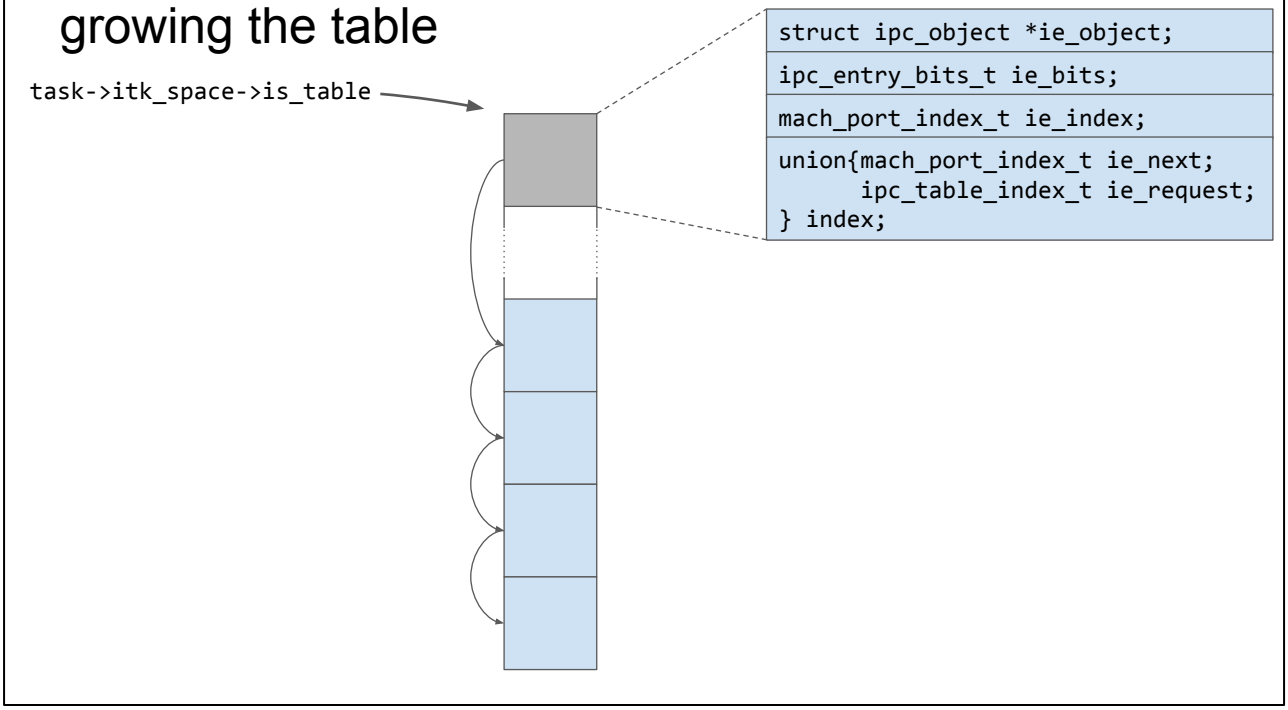
growing the table



growing the table



growing the table



looping port names

As soon as we send our port_free message the target name will be freed and become the head of the freelist

We need that name to be reallocated and freed exactly 63 times

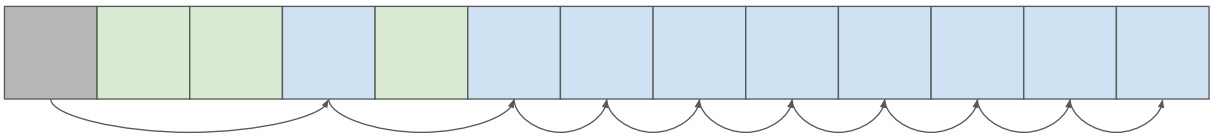
Then reallocated a 64th time as the target in order to get a replacement with exactly the same name (including generation number)

Needs to be exactly the 64th time otherwise generation number won't match

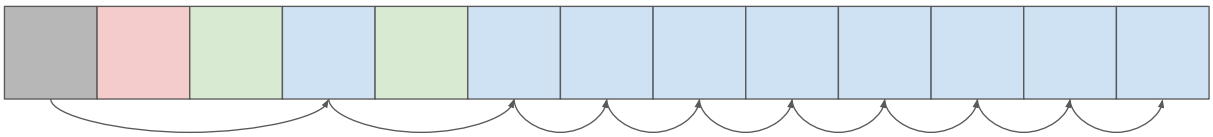
Don't want to assume that no other process sends mach messages

what primitives can we build to reliably get a name reallocated:
moving a name into the middle of the freelist
looping round
reallocating

freelist behavior



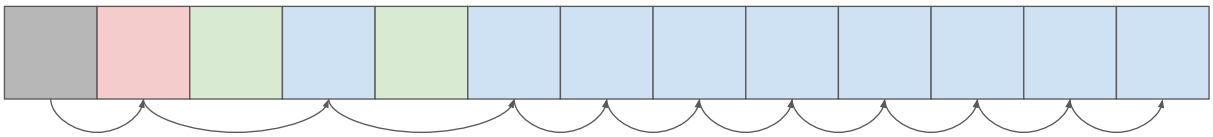
freelist behavior



use UREFS bug
to free this name

will become head
of the freelist;
reused by next
name allocation

freelist behavior

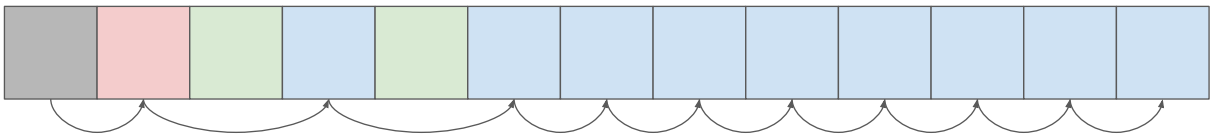


Allocation (by the kernel) and freeing (by mach_msg_destroy in userspace) of OOL port descriptors allocate and free in the same order

the freelist is LIFO

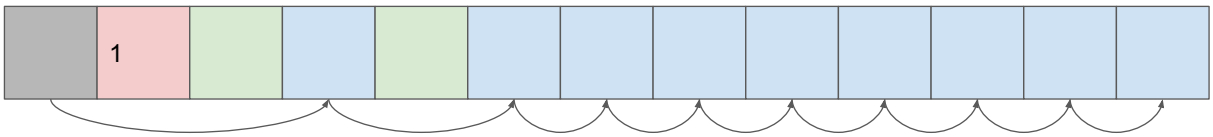
This gives us a primitive to reverse parts of the freelist which will allow us to move our target name within it

freelist behavior



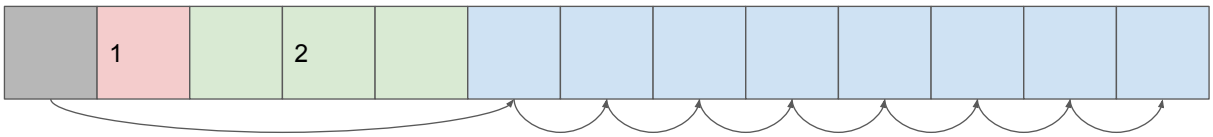
sending a message with 5 ports in an OOL descriptor

freelist behavior



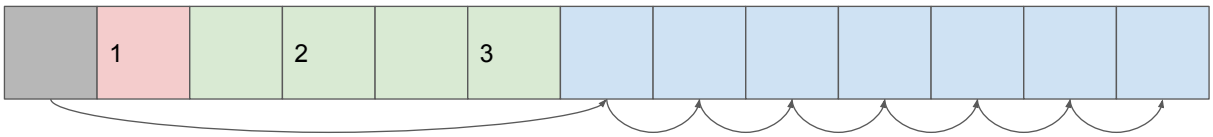
sending a message with 5 ports in an OOL descriptor

freelist behavior



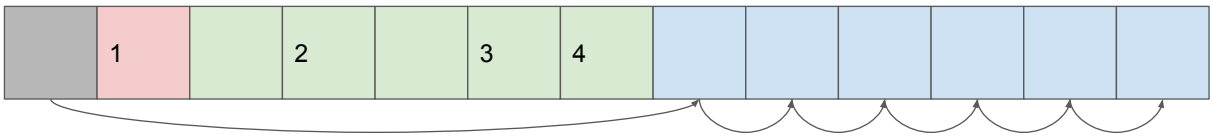
sending a message with 5 ports in an OOL descriptor

freelist behavior



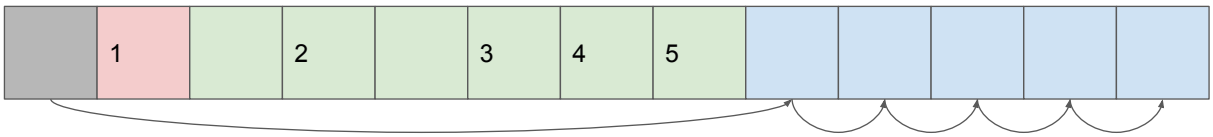
sending a message with 5 ports in an OOL descriptor

freelist behavior



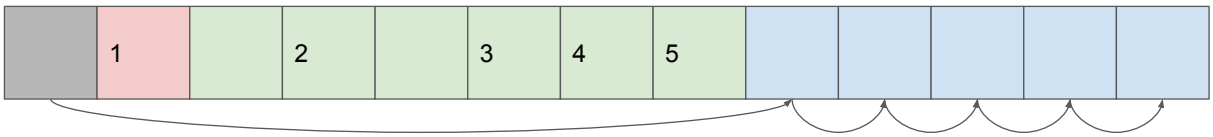
sending a message with 5 ports in an OOL descriptor

freelist behavior



sending a message with 5 ports in an OOL descriptor

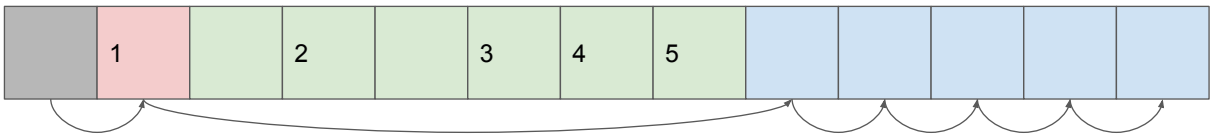
freelist behavior



if the message was invalid it will be passed to mach_msg_destroy which will free the names in the same order as they appear in the message (which was also the order they were allocated.

Let see the effect this has on the freelist

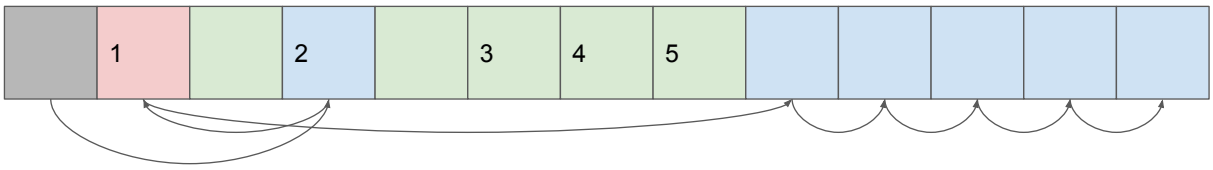
freelist behavior



if the message was invalid it will be passed to mach_msg_destroy which will free the names in the same order as they appear in the message (which was also the order they were allocated.

Let see the effect this has on the freelist

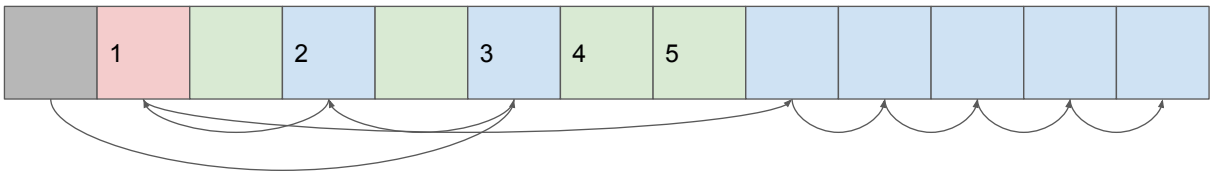
freelist behavior



if the message was invalid it will be passed to mach_msg_destroy which will free the names in the same order as they appear in the message (which was also the order they were allocated.

Let see the effect this has on the freelist

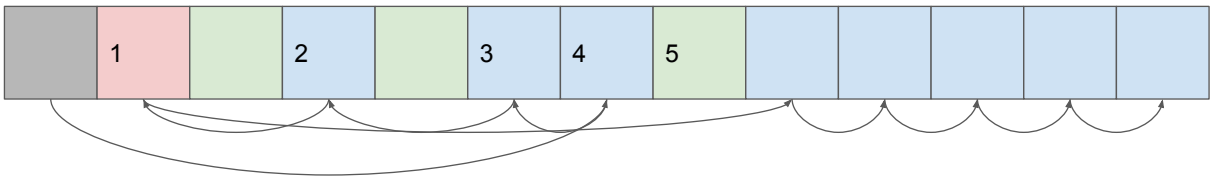
freelist behavior



if the message was invalid it will be passed to mach_msg_destroy which will free the names in the same order as they appear in the message (which was also the order they were allocated.

Let see the effect this has on the freelist

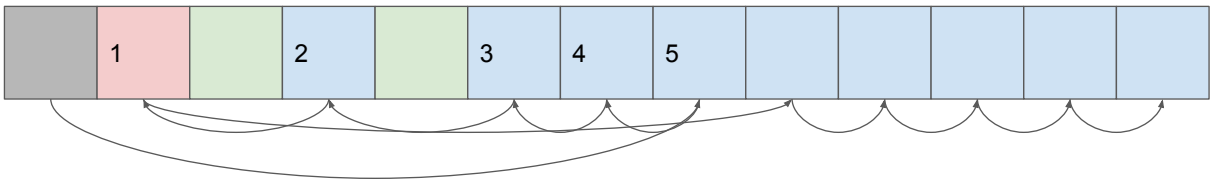
freelist behavior



if the message was invalid it will be passed to mach_msg_destroy which will free the names in the same order as they appear in the message (which was also the order they were allocated.

Let see the effect this has on the freelist

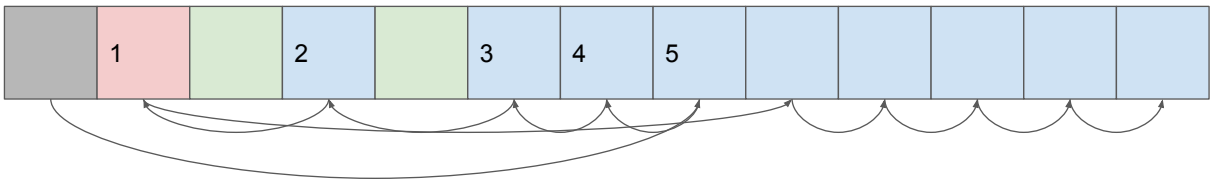
freelist behavior



if the message was invalid it will be passed to mach_msg_destroy which will free the names in the same order as they appear in the message (which was also the order they were allocated.

Let see the effect this has on the freelist

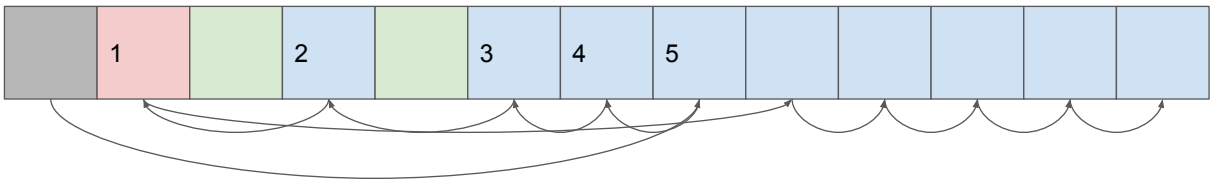
freelist behavior



We’ve moved our target name into the middle of the freelist!

If we ensure that this freelist is long enough then this gives us a good chance to control the reuse

freelist behavior



moving the name into the middle of the freelist used 1 generation number

need to send another 62 invalid messages

send double the number of (different) send rights each as the depth of the target name in the freelist to keep it halfway along

64th time lucky

The next time the target name is allocated it will have the same generation number

This time we don’t want to just bump the generation number; we want the port to be permanently replaced

- Find a way to get the message leaked (pretty common bug)
- Focus on application specific behaviour use a valid message

finding an interesting target

- We have to be able to get a send right to the target service
- We have to be able to get a send right to the same port we want to target in the target service

The target service needs to rely on using a cached copy of the name

This means most kernel-owned ports are off the table; generally they're looked up each time

(eg IOService ports)

launchd

service manager

every process has a send right to launchd

orchestrates mapping from human-readable service names (“`com.apple.iohideventsystem`”) to mach port send right

enforces service sandbox

idea:

free the name launchd has for a service

loop the freed name

register a local restricted service (can do this from iOS app sandbox)

mitm traffic to real service!

what's a task port?

each task has a task port

send rights to the task port == full control over the task

- create a thread
- read/write memory
- manipulate mach port namespace

Don't send other processes your task port!

lsmp shows us who has done this!

it's possible that those tasks have acquired those task ports via task_for_pid but that's a very restricted kernel API. There's no way to restrict processes voluntarily sending each other their task ports.

lsmp - task ports

lsmp tool doesn't show who the task port belongs to; they all show up as TASK_SELF

single send right rule means they can't be the same process; they're actually other task's task ports

```
0x00007803  0x24341e5f  send  0x00000000  TASK SELF (143) hidd
0x00007a13  0x28ce0b17  send  0x00000000  TASK SELF (143) hidd
0x00007c03  0x2427924f  send  0x00000000  TASK SELF (143) hidd
...
```

clients of "com.apple.iohideventsystem" send their task ports to it! How convenient!

building the primitives for launchd exploit

- get send right to service to MITM
- get send right to launchd
- send UREF_OVERFLOW message freeing service name in launchd
- send freelist_reverse message to move name down the freelist
- send 62 looper messages to bump generation number
- register lots of new (restricted) services reusing the target service's name
- crash a client of the target service
- receive task port!

multiplexing services is dangerous

```
boolean_t pm_mig_demux(mach_msg_header_t * request, mach_msg_header_t * reply) {

  mach_dead_name_notification_t *deadRequest = (mach_dead_name_notification_t *)request;

  boolean_t processed = powermanagement_server(request, reply);

  if (processed)
    return true;

  if (MACH_NOTIFY_DEAD_NAME == request->msgh_id) {
    PMConnectionHandleDeadName(deadRequest->not_port);

    mach_port_deallocate(mach_task_self(), deadRequest->not_port);

    reply->msgh_bits        = 0;
    reply->msgh_remote_port = MACH_PORT_NULL;

    return TRUE;
  }
```

is the request handled by powermanagement subsystem?

if not, assume it must be a notification message

drop a UREF on a controlled port name!

What's wrong here? This code is actually implementing the server for *two* services; the powermanagement_server and also the notify_server generated from notify.defs.

The client of the notification server is the kernel but there's nothing stopping a process with a send right to the powermanagement server sending messages with the msgh_id's for the notification server. It's much safer to request those notifications on a seperate, dedicated port.

multiplexing services is dangerous

Gives a similar primitive to the UREFS bug, except now targeting an arbitrary mach port name

No longer need send right to the same port to target it's name in the process, but do need to be able to guess the name

Made a simple crasher by targeting _mach_task_self, the cached name of the task's own task port.

PART II.

We have root.

Well, we have a send right to a task port for a process running with uid 0, no need to actually execute code as root.

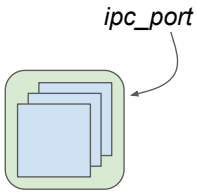
We'll now move to the kernel and look at the mach port abstractions there.

What's in a port?

struct ipc_port

io_bits	io_references
lck_spin_t io_lock_data;	
struct ipc_mqueue ip_messages;	
natural_t bits	
struct ipc_space *receiver;	
ipc_kobject_t kobject;	
...	
mach_vm_address_t ip_context;	
ip_srights	ip_srights
...	

(actually has way more fields than this)



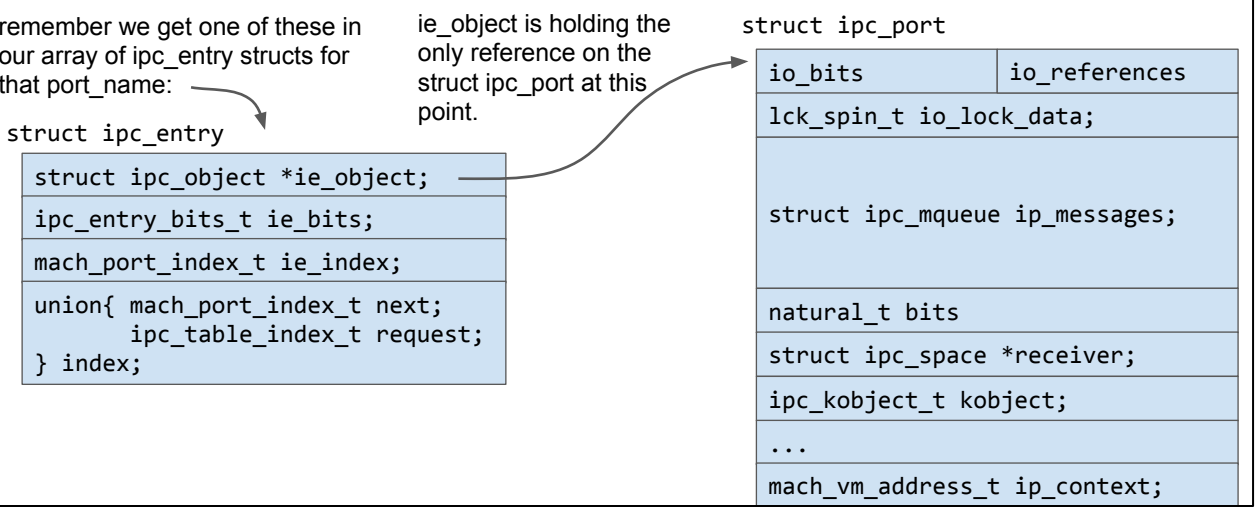
follow the refs

let's take as a base case a freshly allocated user-owned port:

```
mach_port_t port_name = MACH_PORT_NULL;
mach_port_allocate(mach_task_self(),
                   MACH_PORT_RIGHT_RECEIVE,
                   &port_name);
```

remember we get one of these in our array of ipc_entry structs for that port_name:

ie_object is holding the only reference on the struct ipc_port at this point.



We can follow the path of a mach port, tracing who has pointers and hold references. When we pass that mach port name to the kernel we see a ref being taken then if that kernel API holds on to that pointer we'll see it return success from the MIG method; the reference has now been transferred to that API and it's responsible for exactly one reference.

MIG semantics:

```
kern_return_t set_dp_control_port(host_priv_t host_priv,
                                  ipc_port_t  control_port)
{
  if (host_priv == HOST_PRIV_NULL)
    return (KERN_INVALID_HOST);

  if (IP_VALID(dynamic_pager_control_port))
    ipc_port_release_send(dynamic_pager_control_port);

  dynamic_pager_control_port = control_port;
  return KERN_SUCCESS;
}
```

mach port send code converted the name provided by the sender to a port pointer and took a ref on the port

MIG code parsed the message and extracted the port; passed to this method with the extra ref

port pointer assigned to global dynamic_pager_control_port

returning KERN_SUCCESS implies the reference is passed to `dynamic_pager_control_port`

This is a MIG method; we can call it directly from userspace via kobject_server - the hook inserted in the message receive path.

From examining the semantics of MIG code we can see that this method is transferring a reference from the sender to the dynamic_pager_control_port pointer. The previous value of dynamic_pager_control_port is released.

ipc_port_release_send really just drops a reference on the port and adjusts the count of send rights (subtlely but completely different to UREFS.)

ipc_port_release_send drops the reference atomically; that is each port object provides a lock and the reference dropping code will take that lock to ensure that its update is atomic.

But remember the dynamic_pager_control_port pointer only holds one reference on the old value of control port yet there are no locks surrounding the replacement of dynamic_pager_control_port. That means that two threads of control can both call this method and both see the same value for dynamic_pager_control_port even though there's only one reference. If we can get the threads to align correctly we can get two reference dropped where only one is held.

unlocked atomics

```
kern_return_t set_dp_control_port(host_priv_t host_priv,
                                  ipc_port_t  control_port)
{
  if (host_priv == HOST_PRIV_NULL)
    return (KERN_INVALID_HOST);

  if (IP_VALID(dynamic_pager_control_port))
    ipc_port_release_send(dynamic_pager_control_port);

  dynamic_pager_control_port = control_port;
  return KERN_SUCCESS;
}
```

this will atomically drop one reference using a lock provided by the port

`dynamic_pager_control_port` only holds one reference on the port it points to.

The update of dynamic_page_control_port isn't atomic!

why did we need the sandbox escape?

```
kern_return_t set_dp_control_port(host_priv_t host_priv,
                                  ipc_port_t  control_port)
{
  if (host_priv == HOST_PRIV_NULL)
    return (KERN_INVALID_HOST);

  if (IP_VALID(dynamic_pager_control_port))
    ipc_port_release_send(dynamic_pager_control_port);

  dynamic_pager_control_port = control_port;
  return KERN_SUCCESS;
}
```

this will atomically drop one reference using a lock provided by the port

dynamic_pager_control_port only holds one reference on the port it points to.

The update of dynamic_page_control_port isn't atomic!

need a valid send right to the host_priv port to pass this check!

All the UREFS exploit was used for was to gain access to a mach port namespace with a send right to the host_priv port then give the attacking process that send right

analyzing the race

- How easily can you get threads to align properly?
- What happens when you lose the race?
- How do you know when you won?
- Do you have to completely set up the next step of the exploit each time you try?

Exploiting race conditions can be tricky:
How easily can you get threads to align properly?
What happens when you lose the race?
How do you know when you won?
Do you have to completely set up the next step each time you try?

The last one especially can be quite a burden; we don't want to have to do a complete heap groom each time.

We want to build a primitive that gets around all of those points.

Aligning threads

avoid having to care too much about doing this accurately by ensuring:

- nothing bad happens if we lose the race
- we know when we won the race and don't try again

How easily can you get threads to align properly?
Focus on making sure nothing bad happens if you lose. Then just try a lot :)

What happens when you lose the race?
if the two racing threads set the new control port to be NULL then nothing bad will happen no matter what the interleaving

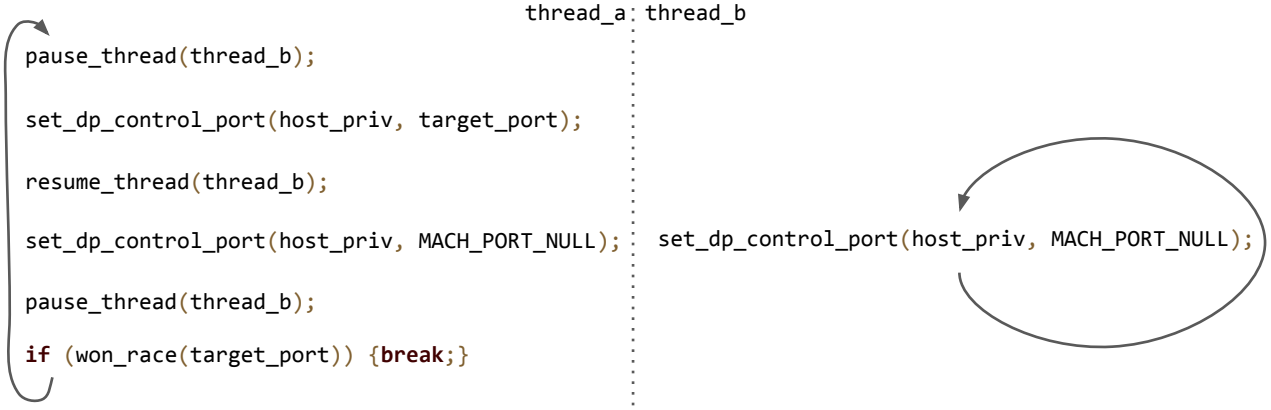
How do you know when you won?
Do you have to completely set up the next step each time you try?

We can use some nice features of mach ports to help us with this

safe racing

```
kern_return_t set_dp_control_port(host_priv_t host_priv,
                                  ipc_port_t  control_port)
{
  if (IP_VALID(dynamic_pager_control_port))
    ipc_port_release_send(dynamic_pager_control_port);
  dynamic_pager_control_port = control_port;
}
```

thread_a	thread_b
pause_thread(thread_b);	
set_dp_control_port(host_priv, target_port);	
resume_thread(thread_b);	
set_dp_control_port(host_priv, MACH_PORT_NULL);	set_dp_control_port(host_priv, MACH_PORT_NULL);
pause_thread(thread_b);	
if (won_race(target_port)) {break;}	



detecting the win

insight: don’t use the bug to free the port!

instead use it to drop an extra ref so we can create a dangling port later under much more controlled circumstances

rather than detecting that the port was freed we can exactly control the number of send rights and use no-senders notifications

no senders

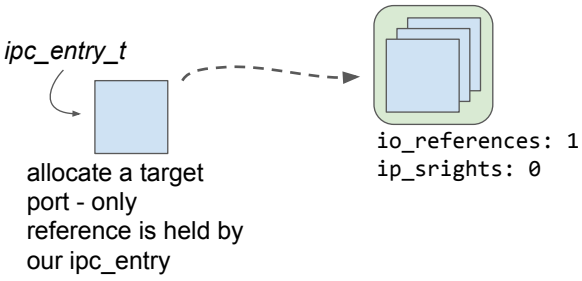
```
void ipc_port_release_send(ipc_port_t port) {
  ipc_port_t nsrequest = IP_NULL;
  mach_port_mscount_t mscount;

  if (!IP_VALID(port)) {return;}

  ip_lock(port);
  port->ip_srights--;

  if (port->ip_srights == 0 && port->ip_nsrequest != IP_NULL) {
    nsrequest = port->ip_nsrequest;
    port->ip_nsrequest = IP_NULL;
    mscount = port->ip_mscount;
    ip_unlock(port);
    ip_release(port);
    ipc_notify_no_senders(nsrequest, mscount);
  } else {
    ip_unlock(port);
    ip_release(port); } }
```

Each ipc_port object does keep a total count of the number of senders - a 32 bit counter ip_srights

ipc_port_release_send as well as dropping a reference also decrements this total count of senders.

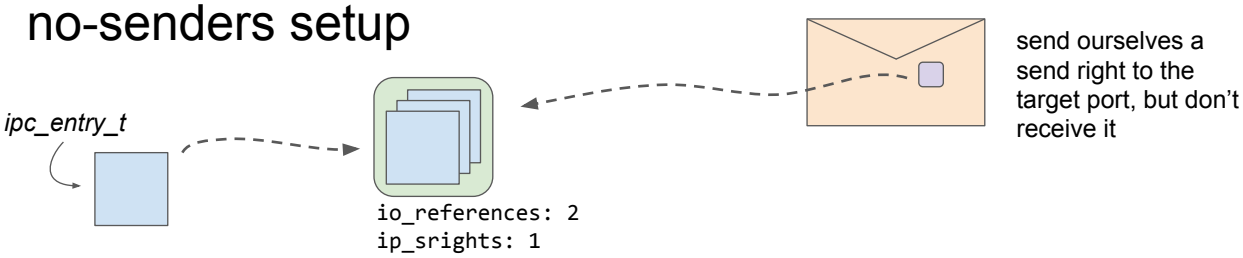
whenever ip_srights hits zero ipc_notify_no_senders will be called on the nsrequest port which hangs off of the port.

no-senders setup

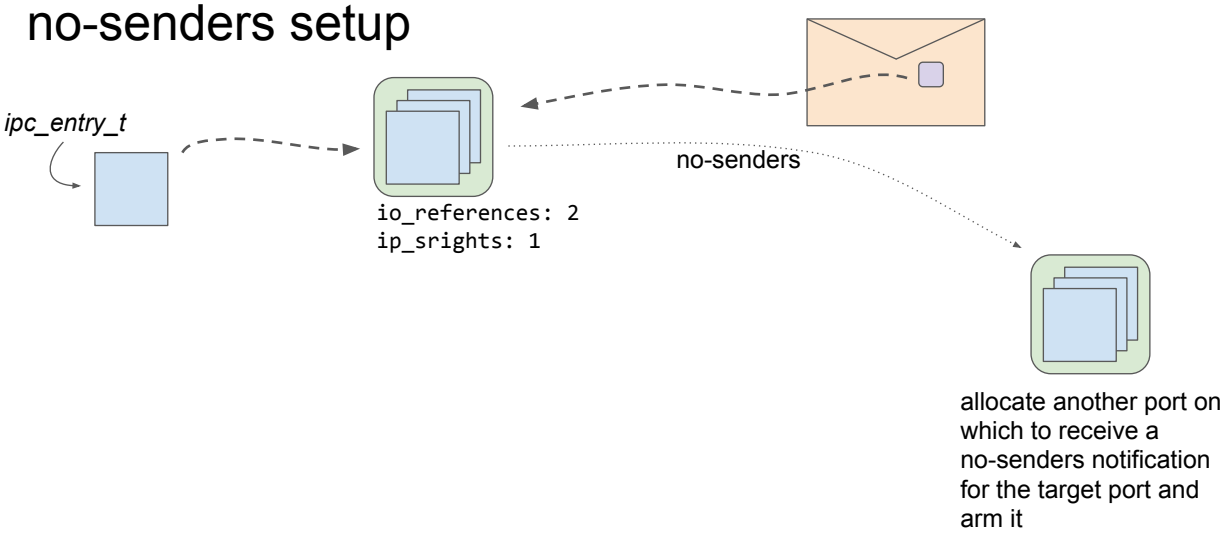


allocate a target port - only reference is held by our ipc_entry

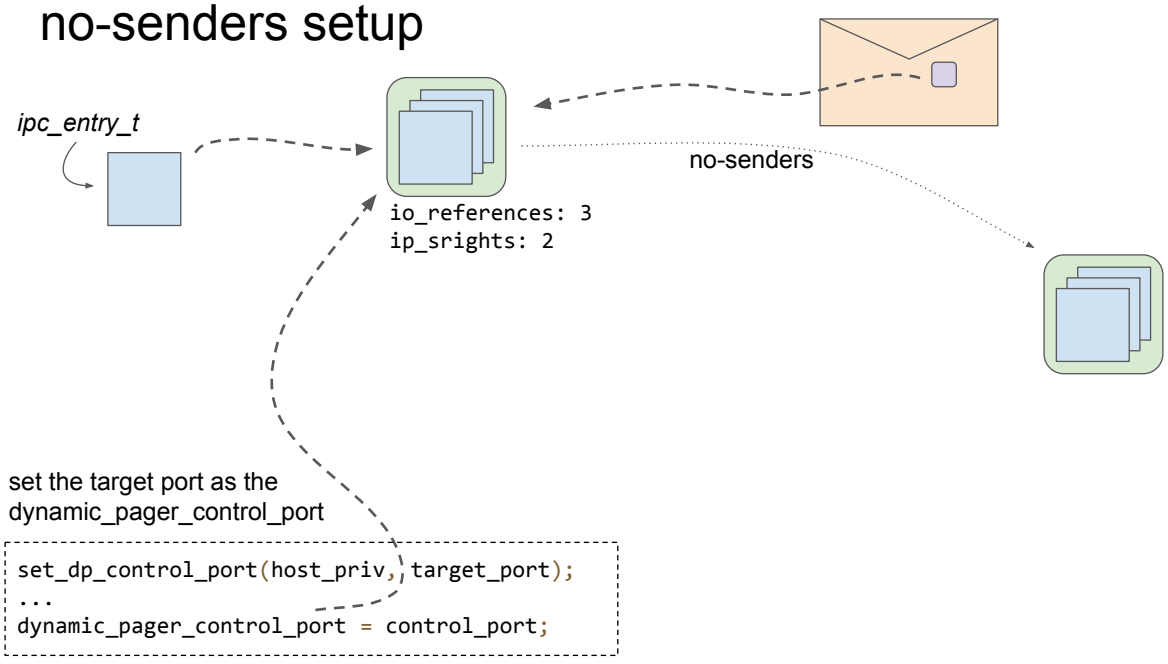
no-senders setup



no-senders setup



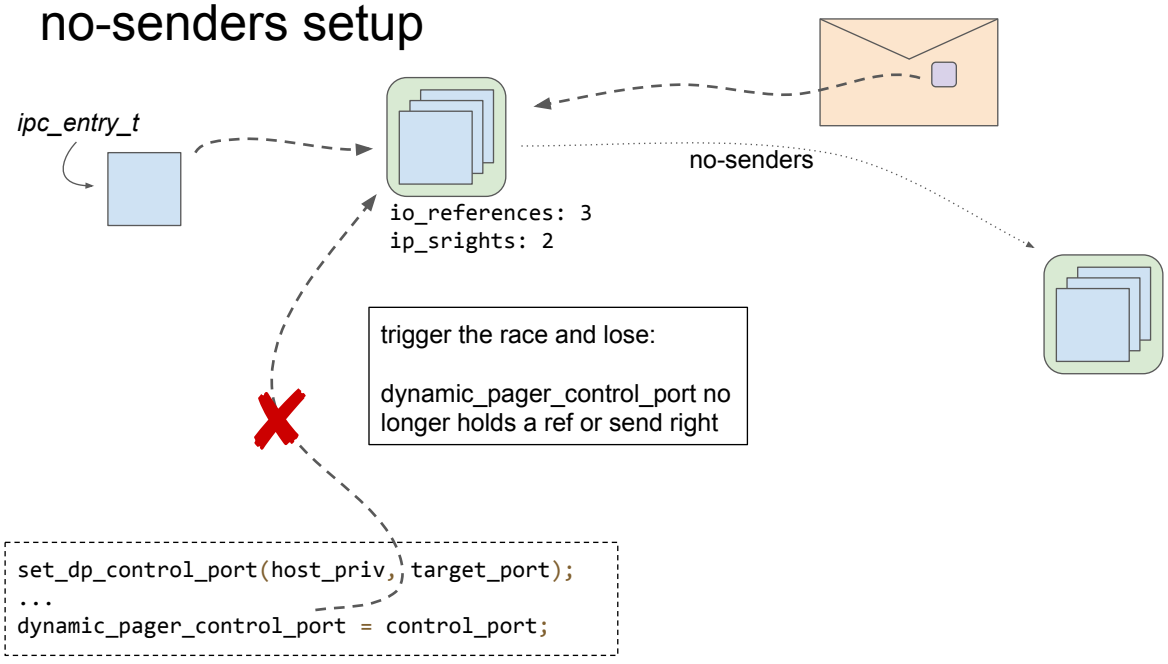
no-senders setup



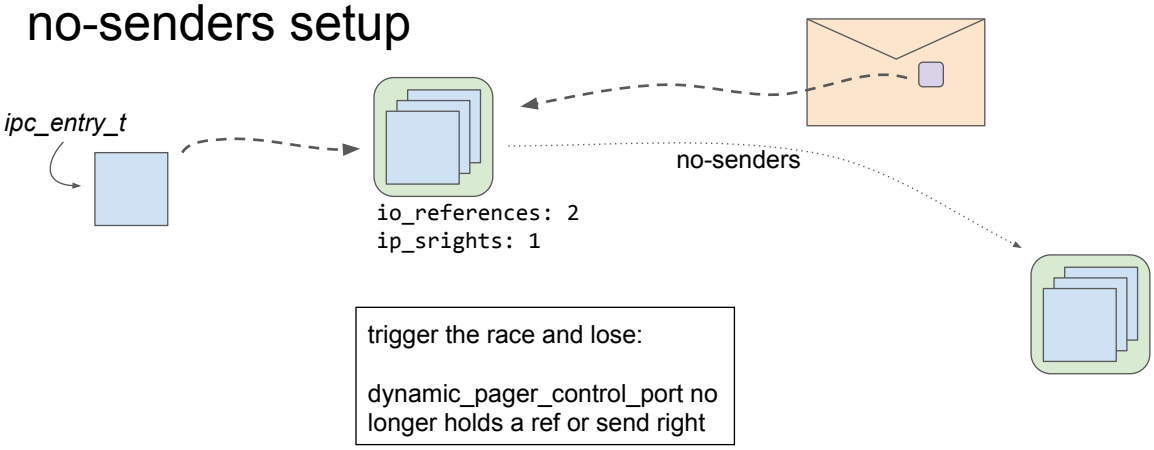
set the target port as the
dynamic_pager_control_port

```
set_dp_control_port(host_priv, target_port);
...
dynamic_pager_control_port = control_port;
```

no-senders setup



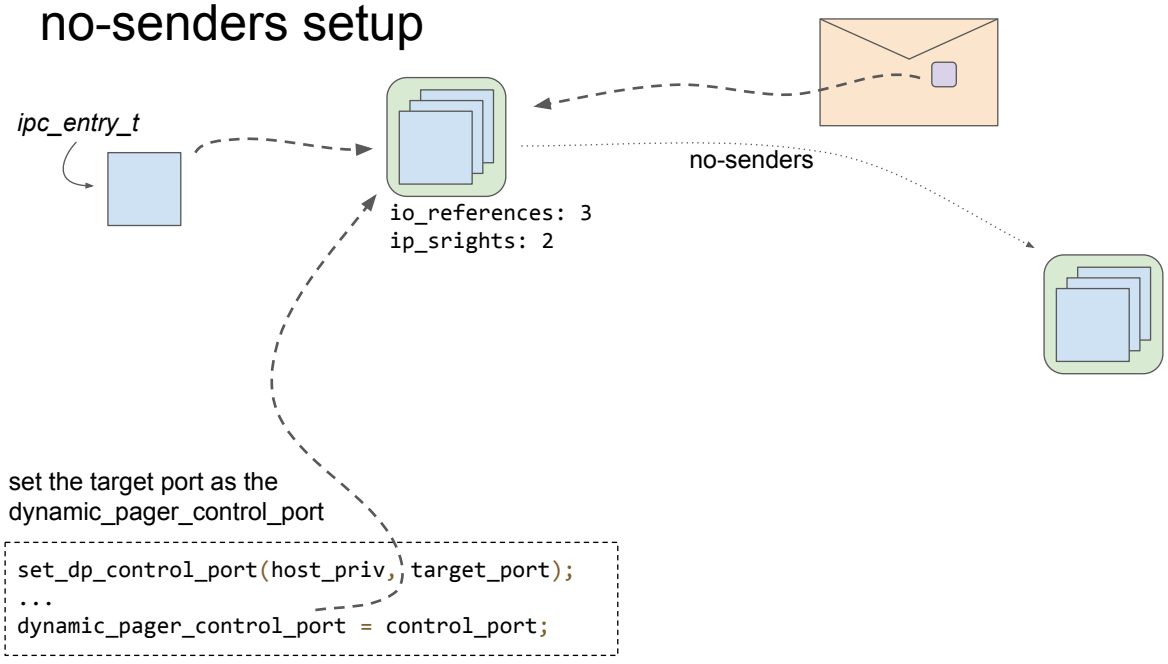
no-senders setup



trigger the race and lose:

dynamic_pager_control_port no longer holds a ref or send right

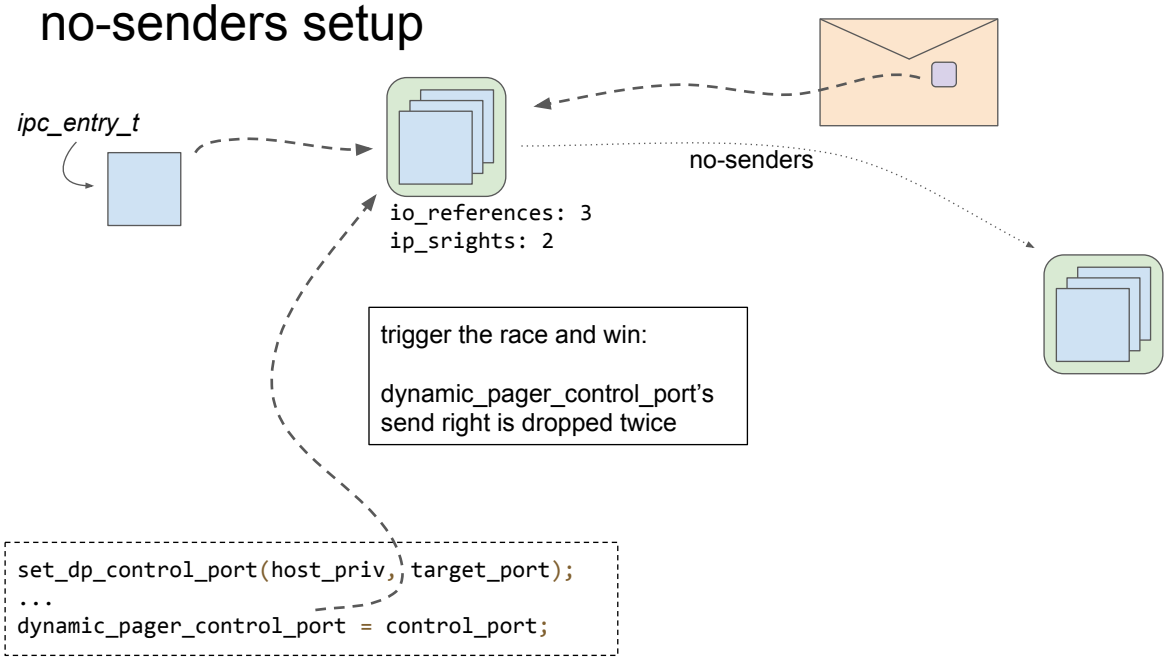
no-senders setup



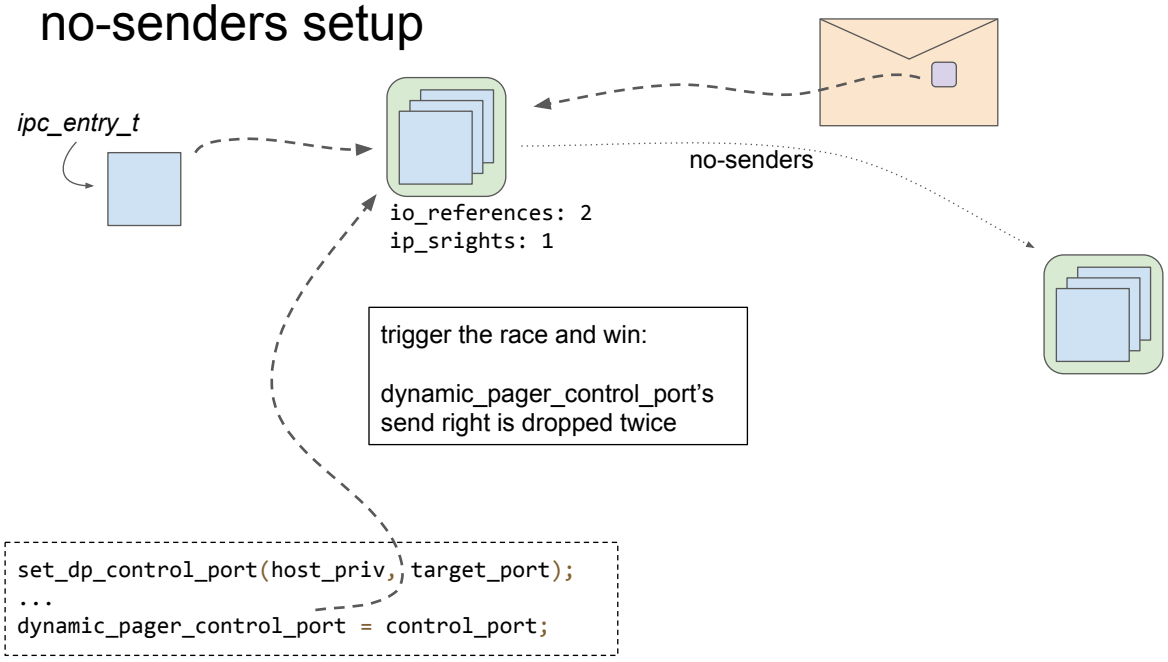
set the target port as the
dynamic_pager_control_port

```
set_dp_control_port(host_priv, target_port);
...
dynamic_pager_control_port = control_port;
```

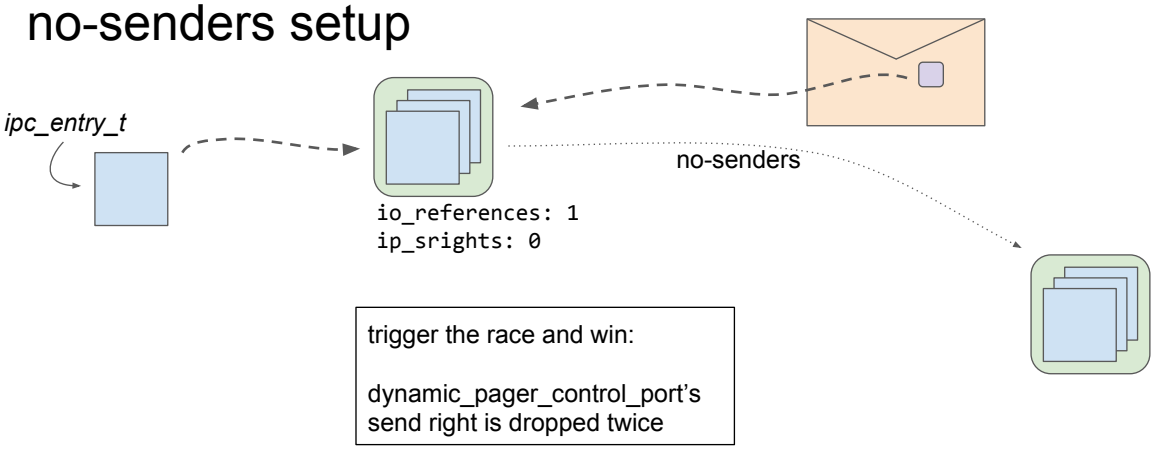
no-senders setup



no-senders setup



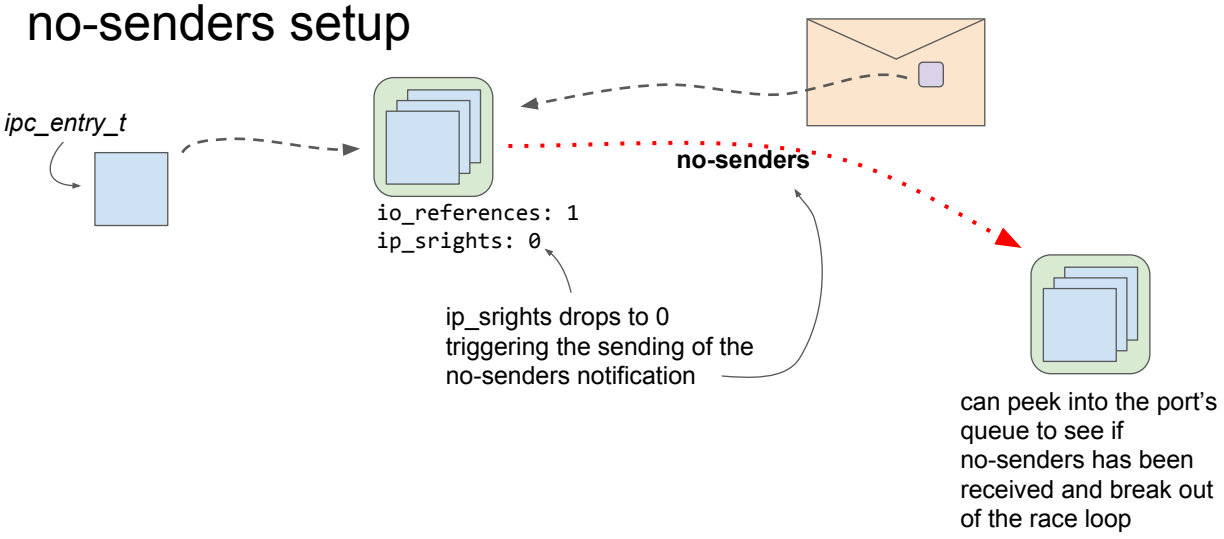
no-senders setup



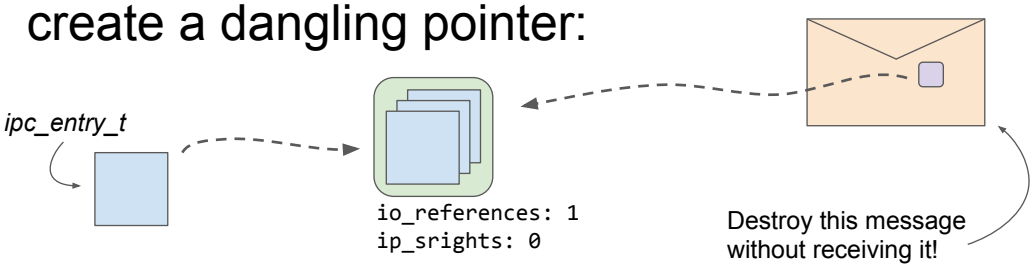
trigger the race and win:

dynamic_pager_control_port's send right is dropped twice

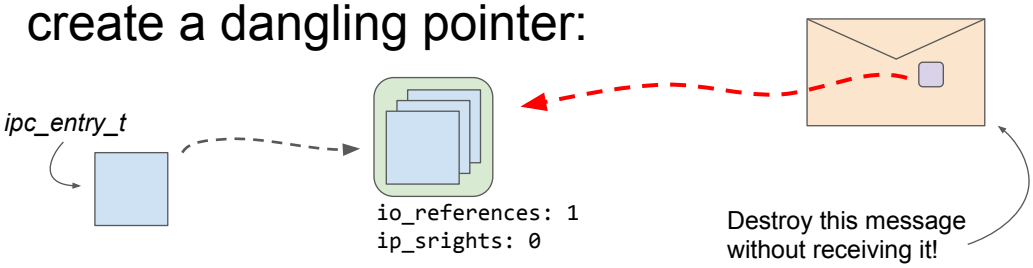
no-senders setup



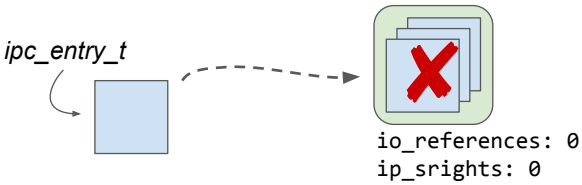
create a dangling pointer:



create a dangling pointer:



create a dangling pointer:



create a dangling pointer:

now have an ipc_entry with
receive rights with a dangling
ie_object pointer

Abstraction

`alloc_port()`
Allocate a new port

`prepare_port()`
stash an extra reference with the kernel and trigger the bug

`free_port()`
drop the extra stashed reference leaving us with a dangling receive right

kernel task port

The goal of all iOS kernel exploitation :)

a send right to the kernel task port == kernel memory read/write

by design

Let’s just go directly for this; no fiddly ROP or shellcode!

It’s always useful to have a goal in mind. Generally iOS kernel exploitation has one goal in mind: get a send right to the kernel task port.

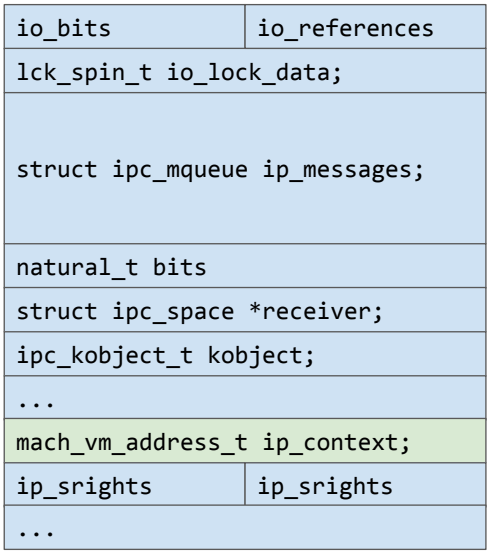
Often they achieve this by first getting code execution in the kernel. But we don’t need to do that :)

Keep in mind that the kernel task port is the ultimate goal and we’ll try to take some shortcuts to get there

I’ve heard this exploitation strategy has been mitigated in iOS 10.3 but I haven’t had a chance to investigate that yet.

mach_port_{set,get}_context

`struct ipc_port`



can get and set `ip_context` freely from userspace

doesn't manipulate io_references, only takes and drops port's lock

Let's use that to read and write in-transit mach port pointers!

if you hold a receive right for a port you can set and retrieve a "context" value. This is simply a 64-bit integer value which is stored in the port object. It's used to for example associate a userspace pointer with a port. You can see this with lsmp (libdispatch associates a heap pointer with dispatch sources for example)

zalloc

see iOS 10 - Kernel Heap Revisited by Stefan Esser for far more detail

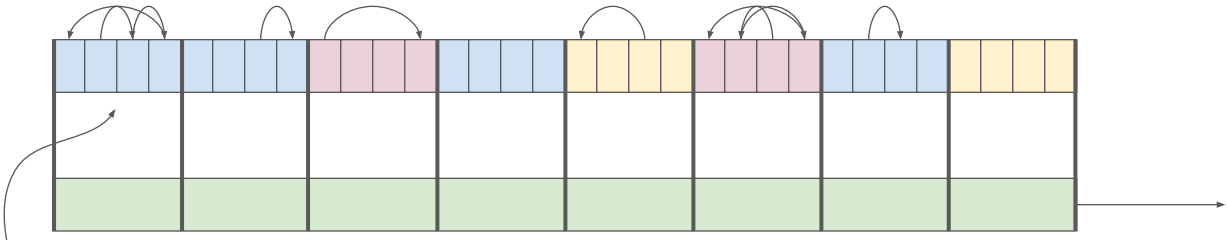
ipc.ports

kalloc.4096

kalloc.1024

zalloc zones request pages from lower level allocator and build page-local randomized freelists

all-free pages in a zone will be returned to lower-level allocator under memory pressure or forced-gc



pages will actually get split up to fit maximum number of zone elements per page

lower level allocator:

- best fit
- grows linearly upwards

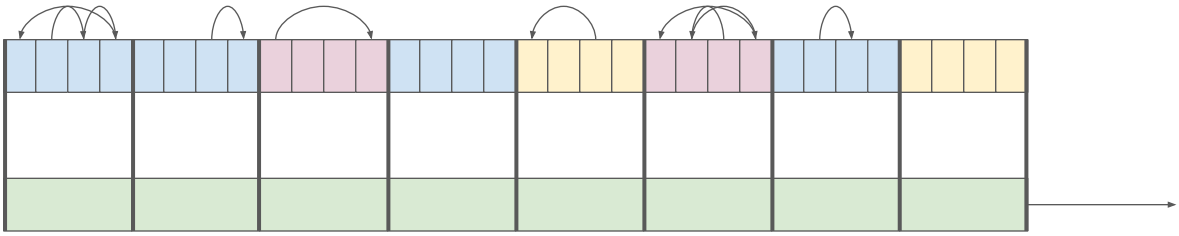
The one problem with that exploitation path is that we need to be able to reallocate the free'd memory with memory representing a different type.

ipc ports are allocated via the zalloc zone allocator and they come from their own zone, from which only ipc objects are allocated.

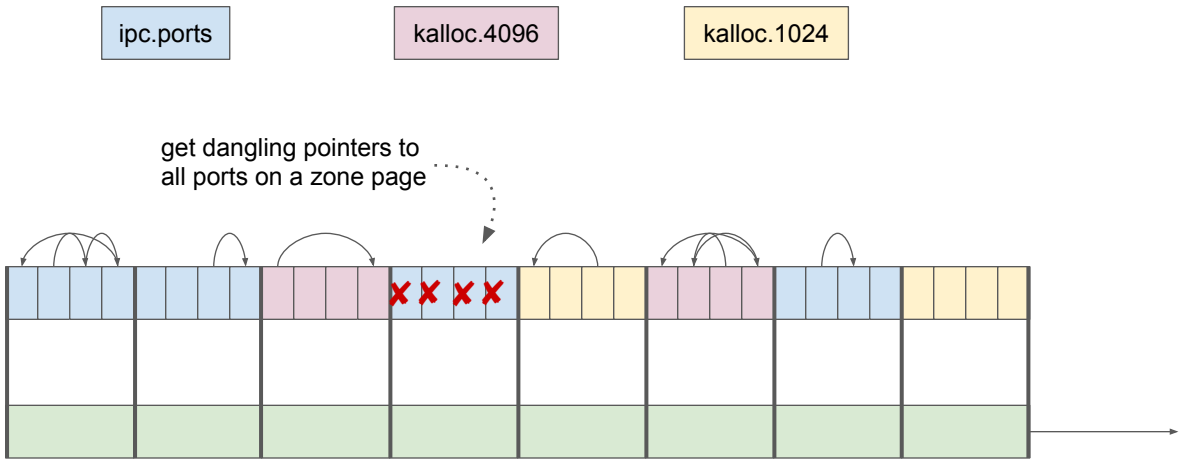
zalloc isn’t a security feature

Easy to move pages between zones

Lower level allocator easy to groom



zalloc isn’t a security feature



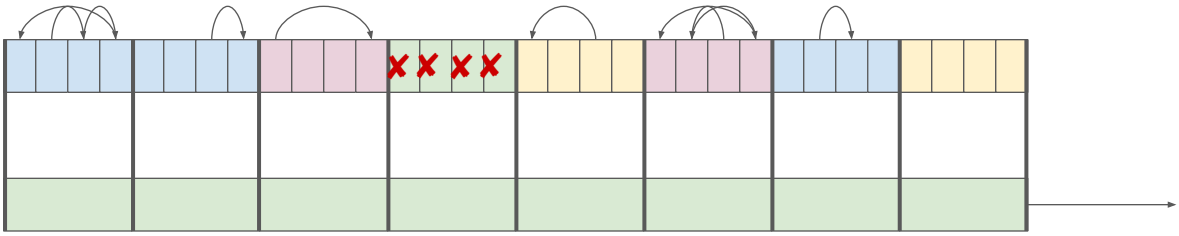
zalloc isn’t a security feature

ipc.ports

kalloc.4096

kalloc.1024

keep allocating target page size to exhaust target zone list and cause gc



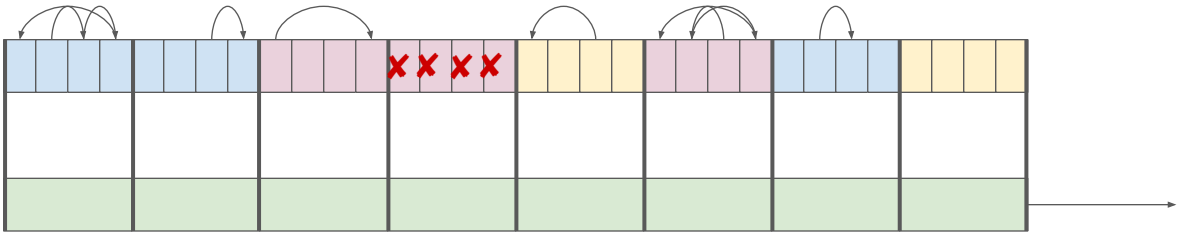
zalloc isn’t a security feature

ipc.ports

kalloc.4096

kalloc.1024

dangling pointers
now point into a
different zone



early mach port allocation

`ipc_port_alloc_kernel();` function responsible for kernel-owned port allocation

`kernel_bootstrap();` responsible for low-level kernel bootstrapping including allocation of kernel task

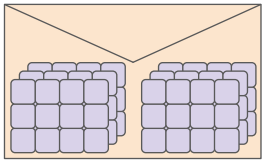
zalloc semantics mean other ports allocated shortly before or after kernel_task will be close to the kernel_task port:

```
host_port
master_device_port
system clock service
system clock control
calendar clock service
calendar clock control
host_security
host_priv
```

if we're looking for the kernel task port then the first step is to know where it might be

Even with zalloc freelist randomization we could still guess pretty well where the kernel task port would be if we could leak the address of any of these other early ports.

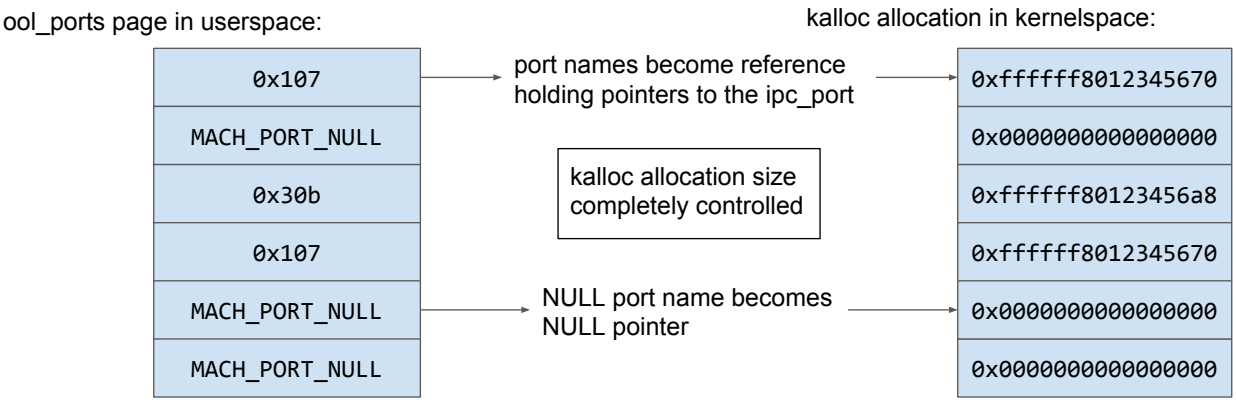
ool_port descriptors



What does this *actually* look like when it's in transit?

`ipc_kmsg_copyin_ool_ports_descriptor`

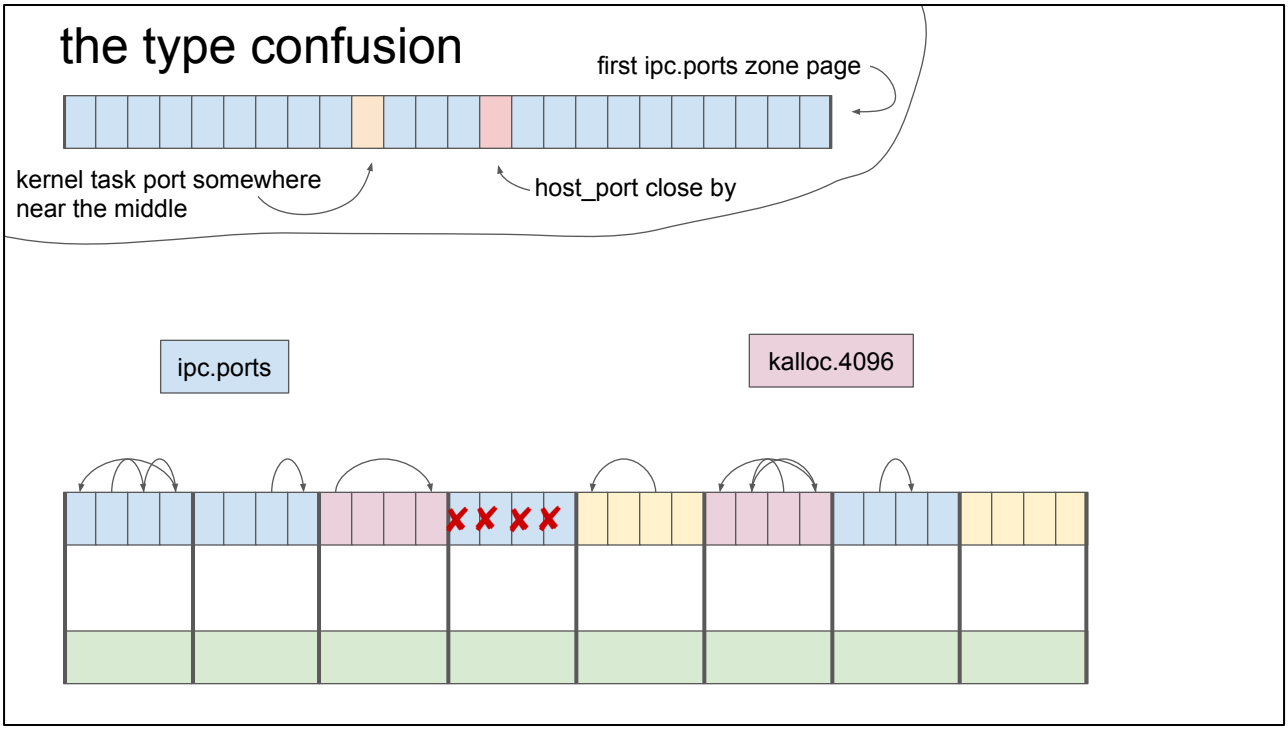
kalloc'ed array of ipc_port pointers!



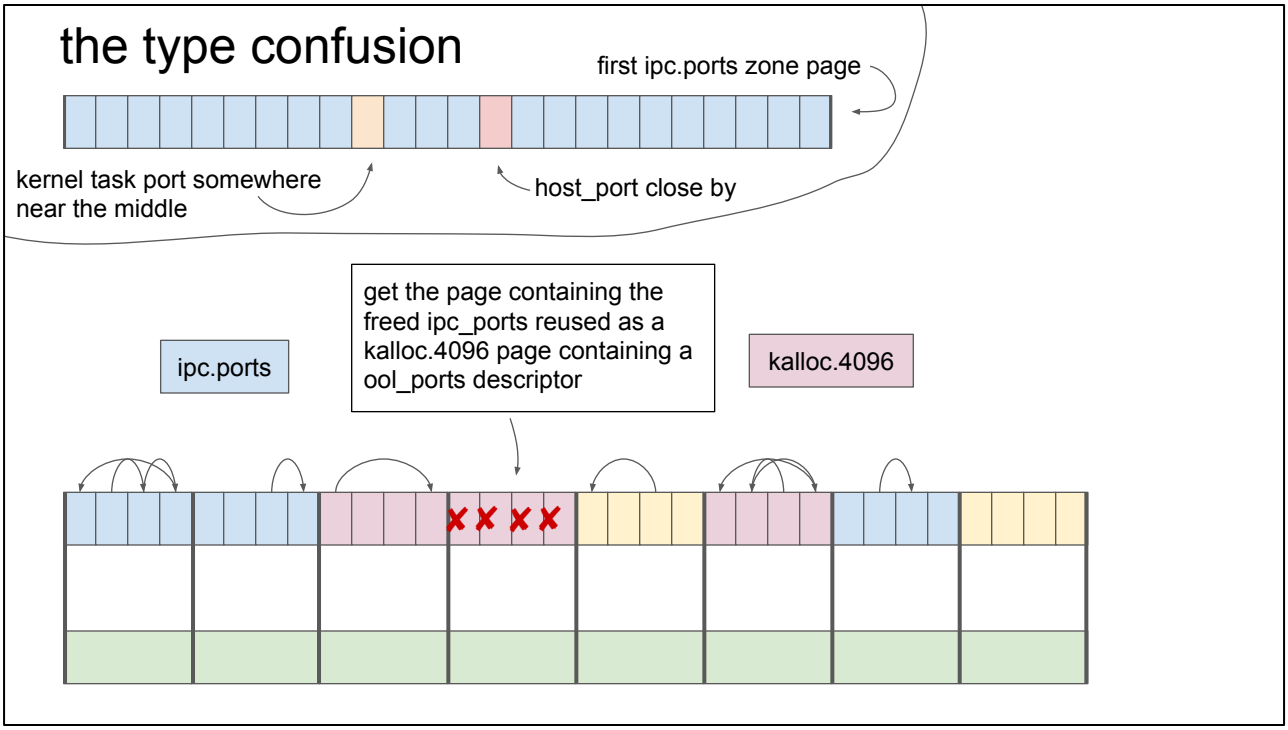
these are used to send a large number of mach ports in a mach message; rather than using mach_port_descriptor you can instead use an out-of-line port descriptor which lets you send a (now limited but still large) number of ports in a message.

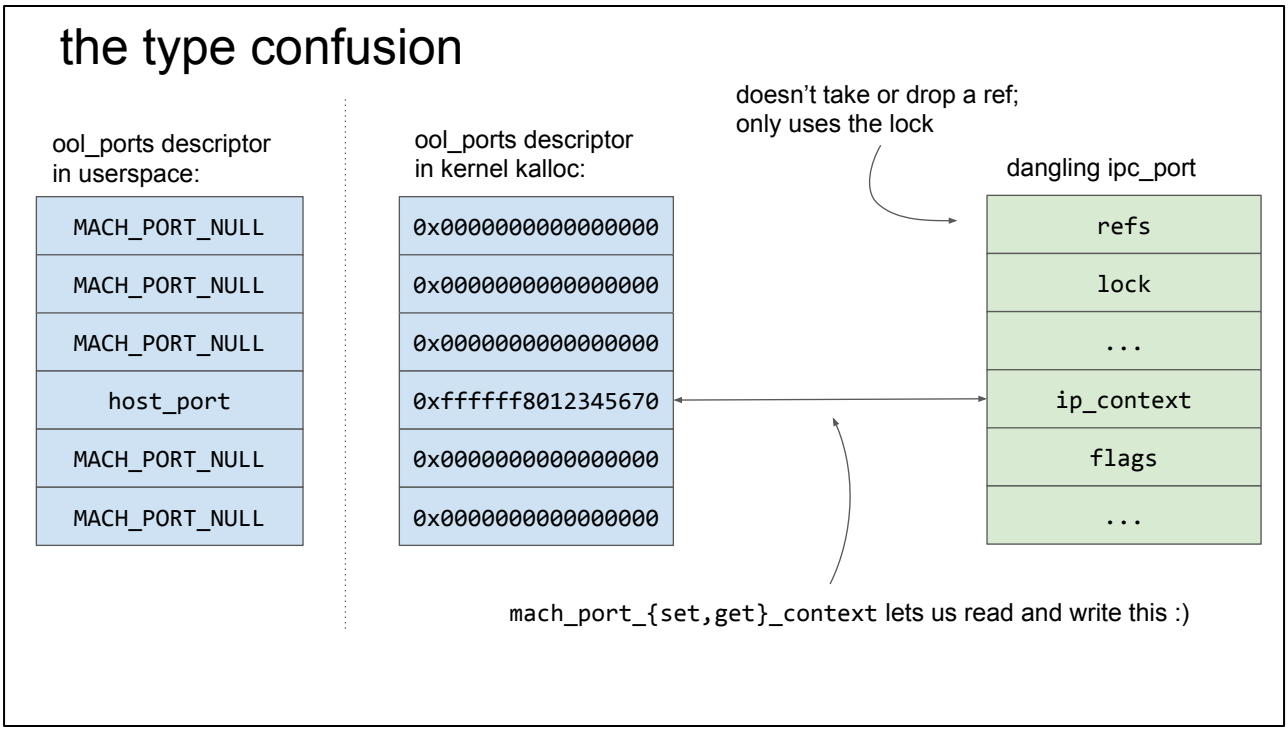
Again at the user->kernel boundary the kernel gets a reference on each of the ports and then those pointers are stored in a kalloc buffer which hangs off the message.

When the receiver dequeues the message they get the rights to the ports contained in the message.



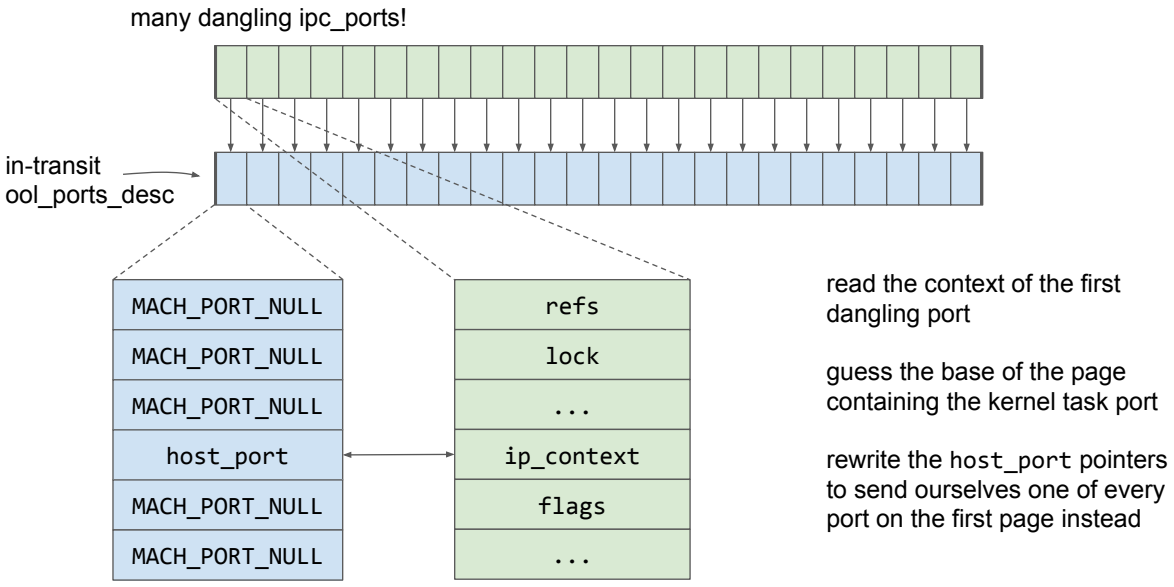
We have a send right to the host port. If we can work out where it is on the heap then we can start to guess where the kernel task port is.


the type confusion
first ipc.ports zone page
kernel task port somewhere
near the middle
host_port close by
get the page containing the
freed ipc_ports reused as a
kalloc.4096 page containing a
ool_ports descriptor
ipc.ports
kalloc.4096



The is the fundamental type-confusion primitive built from the ipc_port UaF. We've built a "fake ipc_port" object using the primitives provided by the out-of-line descriptors. The NULLs are lined up so that they overlap the lock and flags field and the valid ipc_port pointer overlaps the ip_context field which we can read and write from userspace via mach_port_get/set_context

rewrite all the ports



Since we can easily and reliably trigger the race condition we can just give ourselves a pretty large number of dangling ipc_port pointers and build a big ool descriptor. We can then just send ourselves all of the ports near the kernel task port and check whether one of them is the kernel task port when we receive the descriptor.

that’s it!

receive the message with the OOL ports descriptor

go through all the received ports; one of them is the kernel task port

mach_vm_read/mach_vm_read passing that port give you kernel memory read/write

I’ve heard there’s some hardening against this in iOS 10.3 but I haven’t had a chance to investigate yet

Links

https://bugs.chromium.org/p/project-zero/issues/detail?id=965#c2

Check out the latest edition of http://www.newosxbook.com/index.php for another writeup of these bugs with fancier diagrams plus details of @qwertyoruiopz's KPP bypass

The venerable Mac OS X internals book covers mach ports but some of the code has since been refactored