RET2 SYSTEMS
ENGINEERING BLOG

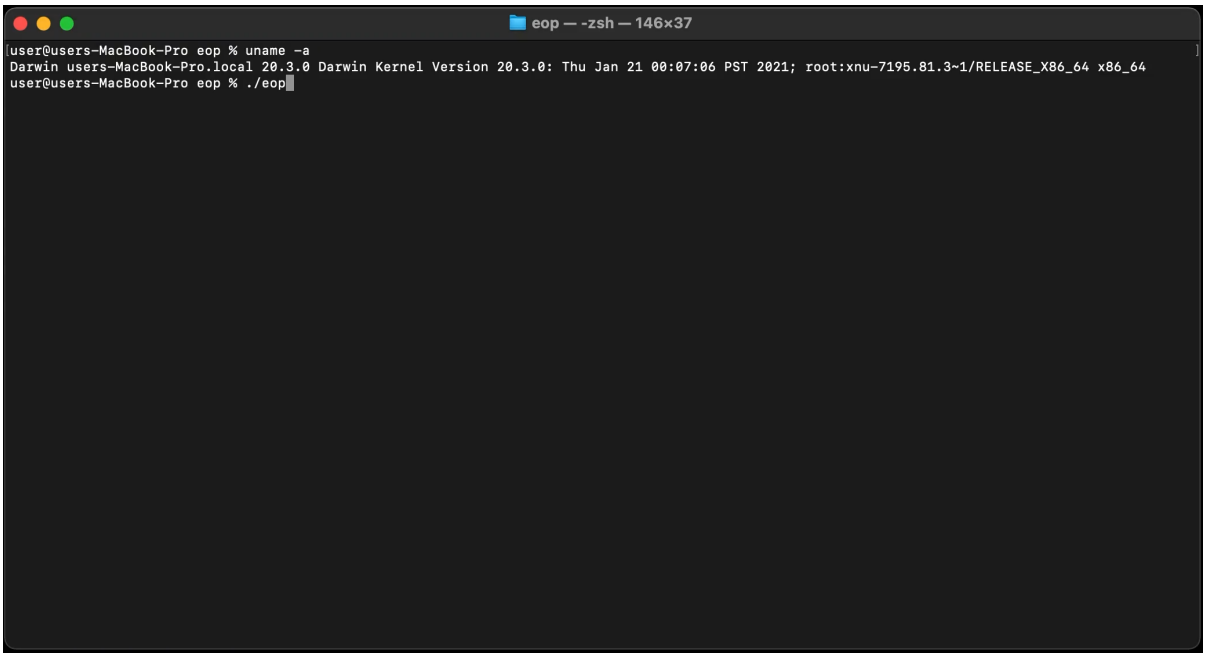
Exploiting Intel Graphics Kernel Extensions on macOS

A Pwn2Own 2021 Apple Safari Sandbox Escape

June 29, 2022 / Jack Dates

To escape the Safari sandbox for our Pwn2Own 2021 submission, we exploited a vulnerability in the Intel graphics acceleration kernel extensions (drivers) on macOS. This post will detail the bug and how we went about exploiting it to achieve reliable kernel code execution.

We delayed publishing this writeup as we discovered and reported a multitude of similar issues to Apple over the past year, the last of which was patched recently. With other researchers catching on, Intel graphics-related CVEs have become increasingly common among Apple's security update listings. Quite recently, there was even an exploit discovered in-the-wild targeting the same Intel graphics kernel extensions discussed in this post.



A demo of the exploit obtaining kernel code execution from unprivileged user context

IOKit Graphics Acceleration Overview

There are a variety of kernel drivers that implement graphics acceleration on macOS. The two relevant to the vulnerability discussed in this post are:

- `IOAcceleratorFamily2` : generic platform-agnostic code and classes, e.g. `IOAccelContext2`
- `AppleIntel*Graphics` : hardware-specific code and subclasses, e.g. `IGAccelGLContext`

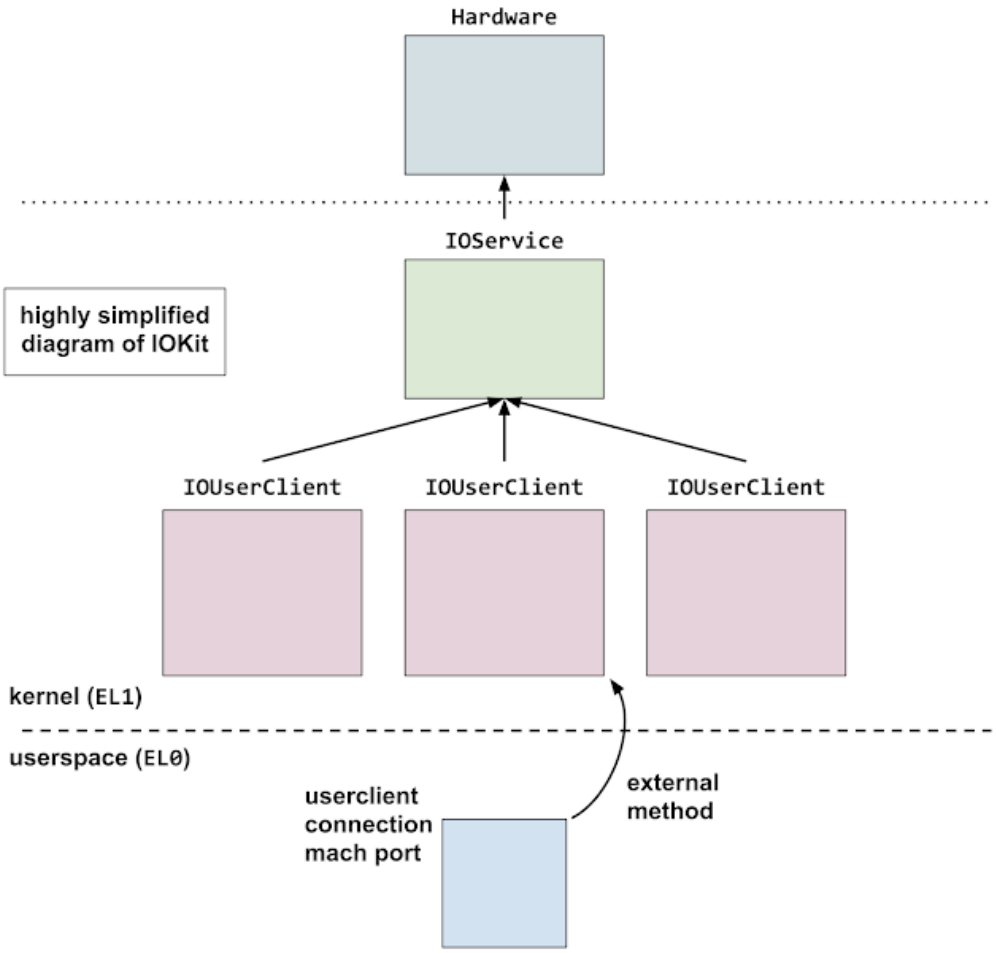
The hardware-specific kernel extension (kext) that gets loaded depends on the CPU generation:

- `AppleIntelICLGraphics` for the latest Ice Lake hardware
- `AppleIntelKBLGraphics` for Kaby Lake/Coffee Lake
- `AppleIntelSKLGraphics` for Skylake
- … and so on for older generations.

These drivers expose functionality to userspace through IOKit. In typical usage, drivers implement 3 things:

- Services
- User clients
- External methods

A userspace program that wishes to communicate with the driver first obtains a mach port for the service. It then opens a specific user client indicated by numeric ID. External methods can then be invoked on the user client, selecting the method by numeric ID, optionally passing scalar and/or binary blob arguments. External methods can be thought of as RPC calls into the kernel driver.



From a Google Project Zero blogpost which analyzed an iOS exploit chain

Another common way to interact with user clients is through shared memory (between userspace and kernel).

IntelAccelerator / IOAccelerator

On Intel-based Macs, the graphics acceleration service is named `IntelAccelerator`, or `IOAccelerator`. Opening this service is explicitly allowed by the renderer/WebContent sandbox,

although it appears there has been some effort to restrict allowed user clients and external methods. While Safari has a separate GPU process, `IOAccelerator` is still allowed by the WebContent sandbox profile (as of writing).

To get a sense of available attack surface in `IOAccelerator` we can enumerate the service's user clients with a little reverse engineering, by looking for an overridden `newUserClient` function. This function will be invoked when userspace calls `IOServiceOpen`.

`AppleIntelICLGraphics` does not implement `IntelAccelerator::newUserClient`, so we'll look to the superclass implemented in `IOAcceleratorFamily2`. There we find `IOGraphicsAccelerator2::newUserClient`, which contains a switch-case over the input ID, giving us a collection of user clients we can create (each implemented as a C++ class).

From there, we can look for external methods for each user client we are interested in. We look for each class overriding one of the following functions:

- `externalMethod`
- `getTargetAndMethodForIndex`
- `getAsyncTargetAndMethodForIndex`

Each of these will take a numeric selector identifying the external method, and will either directly invoke the implementation, or return a structure containing information about it (e.g. a function pointer and argument types/counts). The corresponding userspace API, which ends up invoking these kernel functions, is the `IOConnectCallMethod` family of functions.

Less commonly, there are also "traps," which can be invoked using one of the `IOConnectTrap` functions. The respective overridden kernel functions are `getTargetAndTrapForIndex` and `getExternalTrapForIndex`.

Some user clients also provide for mapping shared memory into userspace with `IOConnectMapMemory`. These user clients override the `clientMemoryForType` kernel function.

Inspecting a User Client

As an example, take the user client of type 6 on the `IntelAccelerator` service, `IGAccelSharedUserClient` (inheriting from `IOAccelSharedUserClient2`). Looking for external methods, we come across:

```
IOExternalMethod* IOAccelSharedUserClient2::getTargetAndMethodForIndex(..., unsigned
int index) {
  if ( index < 20 )
    return &IOAccelSharedUserClient2::sSharedMethods[index];
  return 0;
}
```

We see there is a static array of external method structures in the data section. This is a common pattern.

This specific user client (`IGAccelSharedUserClient`) is a helper of sorts, and allows creating various objects to be used with other user clients. A few of the external methods are `new_resource`, `create_shmem`, `allocate_fence_memory`, and `set_resource_purgeable`. The "namespace" of created objects can be shared with another user client by connecting the two with `IOConnectAddClient`.

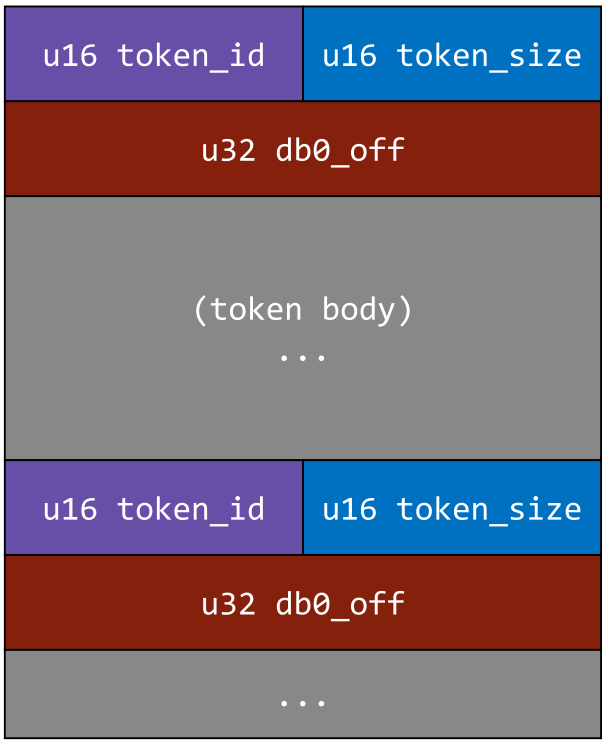
Resources (a.k.a textures), represented by the `IGAccelResource` class, will be useful later on. The `new_resource` external method creates a resource with user-specified properties and returns a

resource id back to userspace. It also maps the backing buffer of the resource (of user-specified size) into the process's address space.

Sideband Buffers

There are several user clients inheriting from `IOAccelContext2`. These "context" objects provide a few types of shared memory, type 0 being a so-called sideband buffer.

Sideband buffers can be submitted via external method 2, `IOAccelContext2::submit_data_buffers`. This eventually ends up in `IOAccelContext2::processSidebandBuffer`, which processes "tokens" in the sideband buffer according to a simple binary format:



The first two bytes indicate the token id, a sort of opcode, that will determine how the token body is to be handled. The next two bytes declare the total token size in dwords, i.e. a token with an empty body would have a size of 2.

The next slot, `db0_off`, indicates an offset into "data buffer 0." In this context, a data buffer is a special type of resource (`IGAccelResource`). A token opcode of 0 can be used to "bind" data buffer 0; the token body will contain the desired resource id to bind.

When iterating over the sideband buffer tokens, an auxiliary structure, `IOAccelCommandStreamInfo`, will contain certain fields describing the state of the command stream and current token. This provides convenient access to common fields for the various token handlers. One such field, `db0_ptr`, stores a pointer into data buffer 0, offset by the respective token's `db0_off`.

The function `processSidebandBuffer` performs some basic bounds checking to ensure `db0_off` and `token_size` are not too large, then defers handling the token to a virtual method `processSidebandToken`.

The various overridden implementations follow a similar pattern for most of the context objects. If the context is supposed to handle the opcode, it passes off to a token handler for that opcode, otherwise it

defers to the superclass `processSidebandToken` function. All of the token handlers seem to be consistently named `process_token_*`, so they are relatively easy to enumerate.

Out-Of-Bounds Write in VPHAL Handler

One of the `IOAccelContext2` subclass user clients, `IGAccelVideoContextMain`, has a handler for "VPHAL" tokens. Relevant portions of the vulnerable code are reproduced below:

```
void IGAccelVideoContextMain::process_token_VPHAL(
        IGAccelVideoContextMain *this,
        IOAccelCommandStreamInfo *info)
{
    unsigned int* cur = info->sb_cur; // current token
    IGAccelVideoContextMain::patch_vphal_command_buffer(this, info, cur+5);
}

void IGAccelVideoContextMain::patch_vphal_command_buffer(
        IGAccelVideoContextMain *this,
        IOAccelCommandStreamInfo *info,
        unsigned int *sb)
{
    // offset pointer into data buffer 0
    unsigned int* db0 = info->db0_ptr;

    // take a resource id from sb, get its "gpu address"
    unsigned long gpu_addr;
    this->bind_resource(this, info, *sb>>16, &gpu_addr, ...);

    // get index from sb
    unsigned long idx = (*sb>>3)&0x7f;

    // bounds check the index against dbuf0 base/length
    if (&db0[idx+1] <= info->dbuf0_base + info->dbuf0_size) { // <--- [1]
        // write low/high 32 bits
        db0[idx] = gpu_addr;
        db0[idx+1] = gpu_addr>>32; // <--- [2]
    }
}
```

The bounds check at `[1]` is off-by-one. It ensures that `db0[idx]` is within the bounds of data buffer 0, however the write to `db0[idx+1]` at `[2]` might be out-of-bounds. This gives us an out-of-bounds write primitive to the 4 bytes immediately after a resource buffer.

While we can only speculate, this bug may have been introduced when GPU addresses became 64-bit instead of 32-bit, and the bounds check wasn't updated to match.

Resources can be created with a parent resource and a 64-bit offset, which sets the child resource's GPU address to be the parent's plus the offset. We can discover the parent's GPU address through intended functionality of the driver, so we'll know what value to offset from. This gives us full control of the 4 bytes written out-of-bounds.

Resource Buffer Allocation

Our next task is to find a corruption target that we can place directly after the resource buffer, whose first 4 bytes would be "useful" to corrupt. It will help to understand how and where resource buffers are allocated in kernel memory so that we can reliably control what gets allocated around it.

Virtual memory from the perspective of the Mach kernel is managed through a hierarchy of `vm_map` objects. The top-level map, `kernel_map`, encompasses the entire kernel virtual memory range. Sub-

maps reserve a smaller range of virtual addresses within the parent map (which is usually `kernel_map`). Lots more info can be found here.

A resource buffer is implemented as an `IOBufferMemoryDescriptor` with, among others, the `kIOMemoryPageable` option set. The backing buffers for pageable memory descriptors come from a special set of sub-maps for pageable IOKit allocations, managed by the global structure `gIOKitPageableSpace`. This space allows for up to 8 maps of 512MB each.

The main entrypoint for allocating in pageable space is `IOIteratePageableMaps`. The algorithm is pretty simple: starting at the submap with index `hint` (map used for the last successful allocation), iterate backwards until a map is able to service the allocation. If none have enough space, create a brand new 512MB sub-map.

Ensuring Reliable Allocation Placement

Pageable space seems to primarily be used for binary-blob-like data, so we chose to target something outside pageable space, i.e. by placing the resource buffer at the very end of a sub-map, and the victim object right after.

In order to make this reliable, our strategy will look something like:

1. Force allocation of a brand new 512MB sub-map with a 512MB resource
    - it is highly unlikely a system under normal usage will have an existing 512MB hole in `kernel_map`, this effectively guarantees our resource will be placed at the end of virtual memory.
2. Place data buffer 0 at the end of the new map
    - free the 512MB resource
    - allocate a padding resource to fill the start of the map, with size `512MB - <size of dbuf0>`
    - allocate dbuf0
3. Allocate the victim object, which for reliability, should be sufficiently large enough to be placed at the end of virtual memory (i.e. there are no holes in `kernel_map` large enough for the victim object).

Now we must find a “large” victim object that we can allocate with an interesting first 4 bytes.

Identifying Viable Corruption Targets / Victim Objects

There are a few common memory allocation functions in the kernel, for example `kernel_memory_allocate` and `kalloc`. `kalloc` is the most common method of dynamically allocating memory in kernel code.

For `kalloc`, allocations of `0x4000` or less are handled by the zone allocator. `0x80000` (512KB) or less goes to `kalloc_map`, a special sub-map. Anything larger is allocated directly from `kernel_map`. If we needed to, we could completely fill `kalloc_map` such that further allocations would fall back to `kernel_map`. This means, if we are looking for candidate victim objects allocated through `kalloc`, we should search for allocations that can be at least `0x4001` bytes.

Binary Ninja analysis can be pretty helpful here, allowing us to script searching for any calls to the various allocation functions with a possibly large enough size argument. (analysis will take quite a while, and you might need to force analyze a few functions for the script to work, and/or add `__noreturn` to `panic`)

```
# if xref'ing kalloc, the internal kalloc function has no symbol name
# need to get the address manually (tailcalled from kalloc_external)
xrefs = bv.get_callers(0xffffff800028c8d0)
minsize = 0x4001
sites = []
sizes = []
```

```
for ref in xrefs:

    print(repr(ref))
    mlil = ref.function.get_low_level_il_at(ref.address).mlil
    if mlil.operation not in (MLIL_CALL, MLIL_TAILCALL):
        raise Exception("bad mlil")

    size = mlil.operands[2][1] # change second index depending on which arg is size
    print("size: "+repr(size))
    vals = size.possible_values

    if vals.type in (RegisterValueType.ConstantPointerValue, RegisterValueType.ConstantValue)\
            and vals.value < minsize:
        continue

    elif vals.type in (RegisterValueType.UnsignedRangeValue, RegisterValueType.SignedRangeValue):
        maxx = max(r.end for r in vals.ranges)
        if maxx < minsize:
            continue

    elif vals.type == RegisterValueType.InSetOfValues:
        if all(v < minsize for v in vals.values):
            continue

    sites.append(ref)
    sizes.append(vals)

print("\n\n"+"="*34+"\n")
for i in range(len(sites)):
    print("0x%016x %-48s %r"%(sites[i].address, sites[i].function.symbol.full_name, sizes[i]))
```

Corpse Footprints

The victim structure we targeted with our exploit involves corpses. In XNU, a corpse is a forked, dead version of a task typically generated during certain exceptions. It is also possible to directly generate one with a call to `task_generate_corpse`, which returns a mach port representing the corpse.

- *Note: Following Pwn2Own 2021, Apple updated the WebContent sandbox to block this interface.*

Upon creating a corpse, a sort of snapshot of its memory footprint is collected in a variably sized binary format. It is allocated in `vm_map_corpse_footprint_collect` directly from `kernel_map` with `kernel_memory_allocate`. The size is dependent on the task's virtual memory size, up to a maximum of 8MB.

The size is stored in the first field of the allocation, `cf_size`:

```
struct vm_map_corpse_footprint_header {
    vm_size_t       cf_size;        /* allocated buffer size */
    // ...
};
```

and upon destruction of the corpse in `vm_map_corpse_footprint_destroy`, `cf_size` is passed as the size to `vm_deallocate`.

By corrupting `cf_size`, we can artificially enlarge the footprint such that it overlaps adjacent victim allocations, which will then be freed along with the footprint.

This allows us to induce a UAF of the victim object, although we again should stick to sufficiently large allocations for reliability's sake. There is a caveat with this UAF technique in that the victim object must be pageable memory, as opposed to "wired" memory.

Wired Kernel Allocations

The vast majority of kernel allocations are wired. When these mappings are removed with `vm_map_remove`, a flag `VM_MAP_REMOVE_KUNWIRE` indicates that the virtual memory entry's wired count (a refcount of sorts) should be decremented. Either way, it waits for the wired count to reach zero before actually removing the entry.

The corpse footprint is one of the rare objects allocated as pageable, and so upon freeing, the unwire flag will not be used. If the overlapped victim allocation is wired, the kernel thread will hang, waiting for an unwire that will not occur. In practical terms, this means we can only UAF a pageable victim allocation.

Conveniently, we have already encountered a potentially useful pageable allocation: IOKit pageable space. The plan at this stage becomes:

1. Allocate a new pageable map with a data buffer at the end
2. Allocate the corpse footprint
3. Allocate a new-new pageable map after the footprint
4. Allocate a second data buffer in the new-new map
5. Trigger the bug, enlarging the footprint
6. Destroy the corpse, freeing both the footprint and the new-new map
7. The second data buffer now points to freed memory

Using VPHAL tokens (but this time with in-bounds offsets), we can easily write arbitrary 64-bit values to the freed data buffer. Yet again, for reliability, it is best to attempt reclaiming the freed data buffer memory with large enough objects.

Initial Kernel Information Leaks

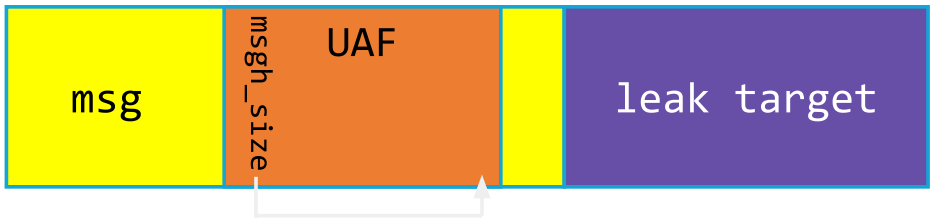
At this point, we have turned the 4 byte out-of-bounds write into the ability to arbitrarily corrupt the contents of any large allocation. To obtain a few information leaks, we'll abuse mach message bodies, which have an upper allocation size limit of roughly 64MB.

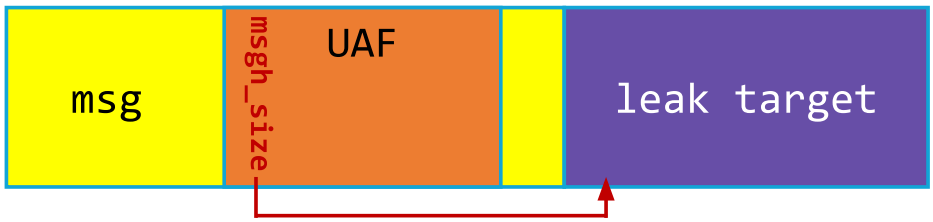
Some relevant background: mach message bodies have a header followed by the rest of the message. The message can be "simple" or "complex" as indicated by a bit in one of the header fields, `msgh_bits`. A simple message is simply raw bytes. A complex message can contain several descriptors, which can be used to send port rights and/or memory mappings to the message recipient. In either case, the header field `msgh_size` indicates the buffer size of the entire message body.

Mach messages are represented in the kernel by `ipc_kmsg` structs, which contain a field `ikm_header` pointing to the message body copied in and converted from its userspace representation. The body will be our corruption target.

`mach_port_peek` allows "peeking" at the first message on a port's receive queue. One of the peeked values is the message trailer, usually appended to the message body by the mach kernel. The trailer is normally located at `ikm_header + ikm_header->msgh_size`, but by corrupting `msgh_size`, we can move the trailer out-of-bounds to obtain an out-of-bounds read:

- *Note:* `mach_port_peek` *is also no longer allowed by the WebContent sandbox!*





We use this technique on two different leak targets:

1. The first will be a mach message body containing port descriptors.
   - When converted from their userspace representations, the numeric mach port in a port descriptor is replaced by a pointer to the corresponding `ipc_port` object. We need to leak two immediately adjacent `ipc_port` objects. The reasoning will be explained in the next section.

2. The second leak target will be the contents of an `OSArray` containing a single `OSData` object.
   - Leaking the address of the `OSData` object will give us a location from which to get a text leak later on (the first field of the `OSData` is the vtable).

Using `OSObject`'s for kernel exploitation has been done many times in the past. They can conveniently be allocated/freed by attaching the `OSObject` to property names of an `IOSurface` (something we are allowed to do from the WebContent sandbox). The objects are deserialized in the kernel from either an XML-like text format or a binary format.

The binary format (deserialized by `OSUnserializeBinary`) allows for more control over allocation sizes; e.g. we can allocate an `OSArray` with capacity for 65536 elements, then place a single `OSData` within it, leaving the rest empty.

Circumventing `zone_id_require` with a Misaligned Port

It would be relatively simple to corrupt the contents of an `OSArray` to contain a fake `OSObject` pointer. Since these are C++ classes, we'd immediately hijack control flow through a fake vtable. However, we'll need a text leak (kernel code address) at minimum to have somewhere to jump to. We also need to know

where to point the fake `OSObject` in the first place (which would then contain the fake vtable etc.); or in other words, the address of data we control.

To obtain these leaks, we'll need to get a bit creative with mach ports. We are going to craft a fake port with the goal of obtaining a semi-arbitrary read. The biggest challenge with creating a fake port is bypassing various `zone_require`/`zone_id_require` calls. This Project Zero post explains it well:

> *`zone_require` is a software mitigation introduced in iOS 13 that adds checks that certain pointers are allocated from the expected zalloc zones before using them. The most common `zone_require` checks in the iOS kernelcache are of Mach ports; for example, every time an `ipc_port` is locked, the `zone_require()` function is called to check that the allocation containing the Mach port resides in the `ipc.ports` zone (and not, for example, an `OSData` buffer allocated with `kalloc()`).*

To deal with this, we can leverage the fact that `zone_require` doesn't care about proper alignment, only that the pointer lies somewhere within the appropriate zone. We'll use the two adjacent ports we leaked earlier, and craft a fake `ipc_port` misaligned in between the two.

To introduce this fake port into our ipc space, we'll again corrupt a mach message body. Instead of corrupting the in-flight message body's `msgh_size`, we perform several writes to turn it into a complex message body containing a port descriptor with a send right to our fake port. If we can survive receiving this message, a new send right to the fake port will be added to our ipc space.

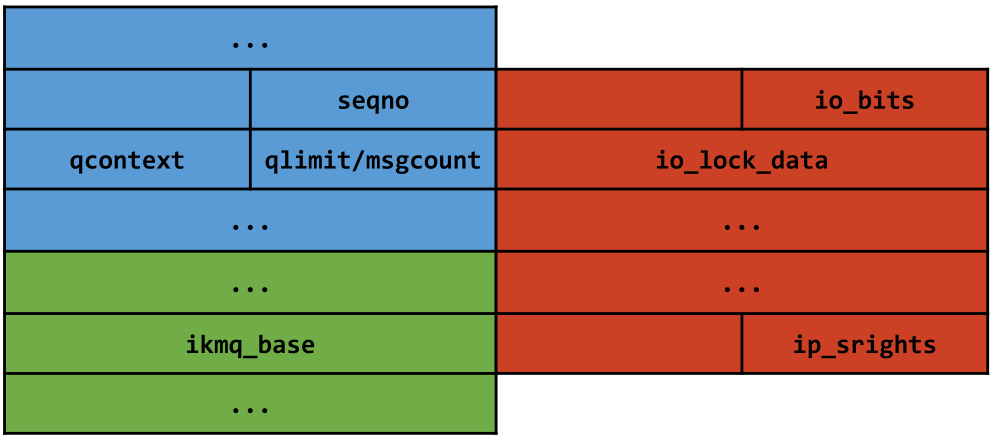
- *Another side note: we are unsure if it's possible to apply this technique to non-x86 platforms, as there is some signature validation code that is essentially nopped out on x86. After copying in a mach message, the kernel calculates a signature of the message body and all descriptors with `ikm_sign`. When copying out the message, it re-calculates the signature and checks for a match, otherwise it panics.*
- *In theory, it should be possible to race performing the message body corruption while the kernel is still copying in descriptors before the signature is calculated. If it's possible to make the race reliable by forcing the kernel to take a long time to copy in all the descriptors, or to somehow check if the race was lost and un-corrupt the message and retry, this technique might be made to work.*

When receiving a message, port descriptors are processed in `ipc_kmsg_copyout_port_descriptor`, although most of the logic is further down in `ipc_object_copyout`. Carefully tracing how the fake ipc object is processed/checked gives us some constraints on how some of the fake port's fields should look in order to survive.

- `io_bits`: int-sized bitfield at offset 0
  - high bit must be set, indicating the port is active [1]
  - remaining high 15 bits indicate the object type, must be anything other than 1 (`IOT_PORT_SET`). This value determines which zone is required by `ipc_object_validate`, which we want to be the ports zone [2]
  - bit with mask `0x400` must be 0, to indicate the kobject has no label (which would trigger some extra checks we'd rather avoid) [3]
- `io_lock_data`: 8-byte value at offset 8. must be 0 to indicate no lock is held, otherwise a deadlock/panic will occur when trying to lock the port [4]

Those are the only requirements to safely receive a send right to the fake port. Some semi-controlled bits at offset 0, and a zero at offset 8.

For the adjacent ports we leaked before, call the first one `A` and the second `B`. Our fake port will reside at `A+0x48`



`io_bits` overlaps `A->seqno`, a sequence number. This value is normally incremented every time a message is sent to the port, but it can also be manually set with `mach_port_set_seqno` (this call is no longer allowed by the sandbox). It is therefore a fully controllable 32-bit value, allowing us to satisfy the constraints on `io_bits`.

`io_lock_data` overlaps port `A`'s `qcontext`, `qlimit`, and `msgcount`. `qcontext` is normally 0. `msgcount` is the number of messages currently on the receive queue, which will be 0 if there are none. `qlimit` is an upper limit on how many messages can be placed on the receive queue. It defaults to 5, however can be set to 0 with the `MACH_PORT_LIMITS_INFO` flavor of `mach_port_set_attributes`. Therefore all 3 fields can be 0, satisfying the `io_lock_data` constraint.

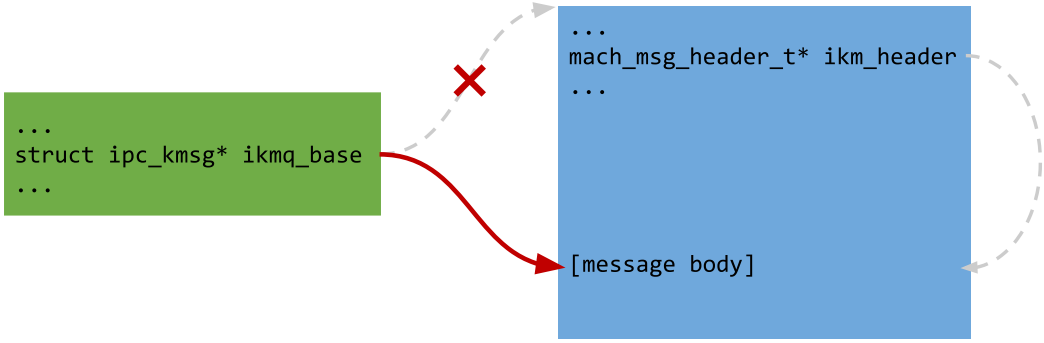
There are likely other misalignments that would work. This one leads to an interesting situation…

The `ip_srights` field keeps track of the number of send rights that exist for a port. Sending a mach message with a port descriptor containing a send right increments this field by one. Similarly, sending N port descriptors with send rights in the same message increments by N.

The `ikmq_base` field points to the `ipc_kmsg` at the head of the receive queue. This structure contains many interesting fields, the one most relevant in this case will be `ikm_header`, a pointer to the message body.

Since the fake port's `ip_srights` field overlaps `B->ikmq_base`, we can send port descriptors for the fake port to increment `B->ikmq_base`, and likewise receive those port descriptors to decrement/un-corrupt. This leads to the question of what `B->ikmq_base` can be incremented to point to, or in other words, what is normally located after an `ipc_kmsg` struct.

When sending a message, if the message body is small enough, the body is stored inline after the `ipc_kmsg` structure, otherwise it is placed in a separate allocation (this logic is in `ipc_kmsg_alloc`). If the message sent to `B` is such a message, incrementing `ikmq_base` an appropriate number of times will point it directly into the inline body, which is fully controlled.



We now have the ability to fully fake the `ipc_kmsg` at the head of `B`'s receive queue. Specifically, we will turn control of `B->ikmq_base->ikm_header` into a constrained arbitrary read.

The general idea is to repeat the following:

1. send a small message to `B` with the fake `ipc_kmsg` contents as the body
2. to a separarte helper port, send N send rights to the fake port using port descriptors, incrementing `B->ikmq_base`
3. perform some non-destructive operation on `B` to use the fake `ikm_header` for a constrained read
4. receive the port descriptors, decrementing `B->ikmq_base` back to normal
5. receive from `B` to clear the queue, repeat as needed with a new fake `ipc_kmsg`

Pseudo-arbitrary Kernel Memory Read

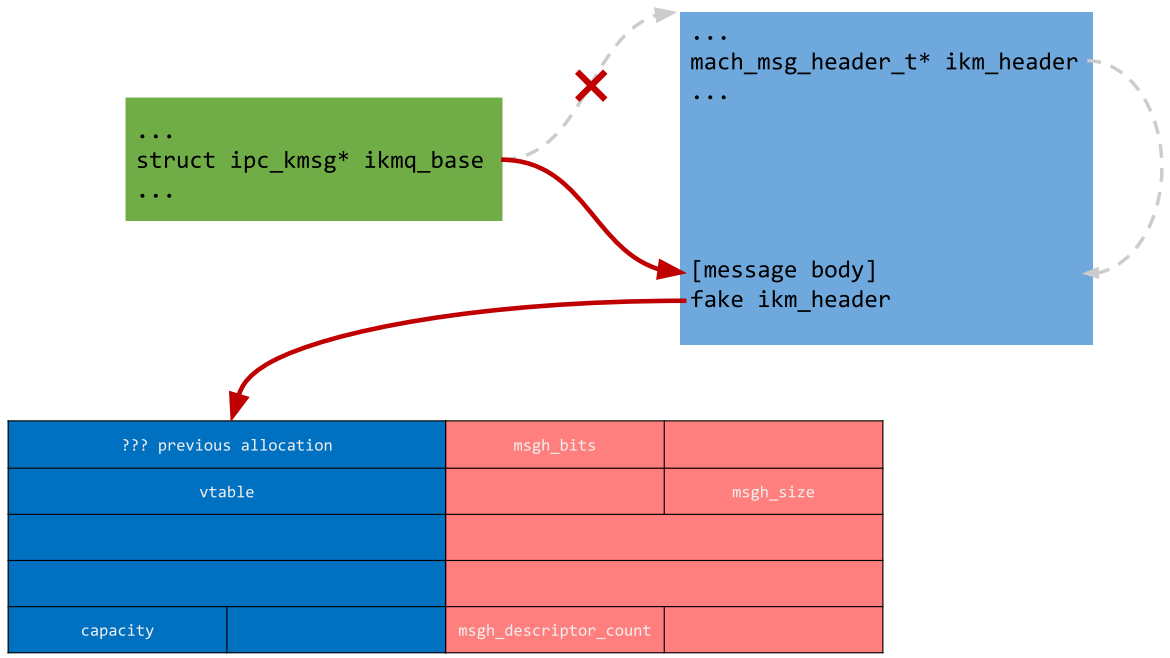
There are 2 methods we can use to leverage a fake `ikm_header` into a read primitive.

Method 1

The first uses `MACH_RCV_LARGE`, a flag that can be specified when receiving a mach message. If this flag is set and the message to be received is larger than the userspace receive buffer size, the kernel leaves the message queued, and writes out to userspace how much space is required.

The required size is calculated by `ipc_kmsg_copyout_size`. The calculation is necessary because port names (32-bit) are converted into their corresponding pointers (64-bit) when copied in from userspace, so there is a size difference from the perspective of the kernel and that of userspace. The required size is calculated as `ikm_header->msgh_size - 8`, then if the message is complex, additional subtractions occur for each port descriptor.

For a text (code address) leak, we would like to read out the vtable of the `OSData` object we leaked a data pointer to previously. We'll point `ikm_header` at `OSData - 4`, then attempt to receive the message with a very small receive buffer, and the `MACH_RCV_LARGE` option set.



`msgh_size` overlaps with the low 4 bytes of the vtable. Kernel ASLR doesn't provide enough entropy to randomize the high 32 bits of the kernel slide, so we effectively have our text leak.

One constraint here is that the memory before `OSData` must be mapped, since `ipc_kmsg_copyout_size` will check if the message is complex using `msgh_bits`. The safe thing to do would be to repeat the leak procedure until a non-page-aligned `OSData` is found, ensuring `OSData-4` will be mapped.

The second potential issue is we don't know if `msgh_bits` will have the complex bit set. If it doesn't, all is well. If it does, `msgh_descriptor_count` (the number of descriptors in the complex message) will overlap with the `OSData` capacity. By allocating the `OSData` with a size of 0, the capacity will likewise be 0. This will ensure no additional subtractions are performed on `msgh_size` for port descriptors.

Method 2

The second constrained read method uses `mach_port_peek`. Previously, we used `mach_port_peek` with a corrupted `msgh_size` to read an out-of-bounds trailer. This time, we control `ikm_header` instead. The trailer is read from `ikm_header + ikm_header->msgh_size`, so we need to ensure there is an appropriate fake `msgh_size`. The simplest thing is to find a zero in memory shortly before the value we wish to read, so the trailer will be copied out directly from the fake `ikm_header` without performing an addition.

We'll use this technique to leak the address of controlled data. We first create a helper port and leak its address with the old `mach_port_peek` technique used earlier. We then send a large mach message to this port, which will allocate the body at the end of virtual memory, at a constant offset from the resource buffer `dbuf0`. Using the pseudo-arbitrary `mach_port_peek` read, we can then traverse the `ipc_port` object to find the message body pointer. Subtracting the proper offset gives us the address of `dbuf0`. This memory is shared between kernel and userspace, so we have a kernel address containing controlled data.

Arbitrary Function Call Primitive

Armed with a text leak and the address of the shared memory resource buffer, we can proceed to hijack control flow. We allocate a large `OSArray` and corrupt an entry to point at the resource buffer.

This gives us control of a fake C++ object, i.e. vtable control.

To initiate a virtual call, we query the corrupted `OSArray` from its associated `IOSurface`. The code first copies the queried value prior to serializing it. In this case, the copy is performed by `OSArray::copyCollection`, which calls `OSArray::initWithObjects` to initialize a new array using the existing objects. This iterates over the backing store and retains each object (the first of which we've corrupted) through a virtual call to `taggedRetain`. This gives us an arbitrary function call with a controlled `this` argument.

We direct this call to `OSSerializer::serialize`:

```
OSSerializer::serialize(OSSerializer* this, OSSerialize* s) {
    return this->callback(this->target, this->ref, s);
}
```

We now have an arbitrary function call controlling the first two arguments. By directing this call back into `OSSerialize::serialize`, we can utilize the `s` argument to obtain an arbitrary function call with 3 controlled arguments. This method has been used before.

Upon returning from the hijacked `taggedRetain` call, we will be back in `OSArray::initWithObjects`, which will abort the copy if a null entry is encountered in the array. We use this behavior to bail out cleanly.

With this primitive, arbitrary read/write reduces to calling the `copyin` and `copyout` functions.

Kernel Shellcode Execution

From here, getting arbitrary code execution as kernel is relatively simple.

We allocate memory for the shellcode with `kmem_alloc_external` and copy it in with `copyin`. There didn't seem to be a convenient way to mark this memory rwx with only 3 controlled arguments, so instead we use `vm_map_store_lookup_entry` to lookup the `vm_map_entry_t` associated with the shellcode memory, which contains the mapping's permission bits. Overwriting these bits is sufficient to mark the entry rwx. Finally, we call the shellcode itself.

The shellcode overwrites the kernel version string (viewable with `uname -a`), unsandboxes the current process (Apple Safari), and gives it root credentials. If interested, see the exploit source for how this is done.

Conclusion

The most difficult part of this exploit was probably the need for info leaks, which took the exploit off on a tangent focused on bypassing the `zone_require` mitigation for ports. Changes to the sandbox since Pwn2Own 2021 have prohibited certain function calls that made this technique viable within the Safari sandbox, although it's possible to modify the technique slightly to work around these new restrictions.

The vulnerability discussed here was patched in macOS Big Sur 11.4 and assigned CVE-2021-30735. The exploit code for both the standalone EOP and full chain can be found here on GitHub.