November 2, 2017

Jailbreak for WatchOS

Testing Apple's Device Security

Jailbreak WatchOSApp

Introduction to WatchOS

Apple Watch was designed as a companion object for an iOS device. Under the hood it runs an operating system, WatchOS, which is developed by Apple. Both software and hardware combined provides many out-of-box security such as secure boot chain, application sandboxing, mandatory code signing. And most importantly secure enclave which protects sensitive information including payment information using iOS feature called Apple Pay.

From a security perspective, it is interesting to review operating system, and its underlying hardware enforced by Apple. And apply publicly known surface attack methodology to the whole Watch ecosystem to get a precise understanding of the potential risks associated with that specific platform and how potential vulnerabilities.

During this independent investigation we found two zero-day vulnerabilities (findings that are not reported previously). These vulnerabilities allows an user to dump entire WatchOS kernel and overwrite the memory buffer. By corrupting this memory buffer an adversary can break out of the sandbox (a security capability provided by Apple) and provides untethered controlled to the kernel and provide capability to point to the header of another port and read, and write it is in memory, this attack is commonly called injecting gadget or Return-Oriented Programming (ROP). The second port is than freed and reallocated a malicious user's application with read, write and execute permission. Once an adversary can read kernel memory, they can use this capability to read virtual method table, also known as vtables, a mechanism to support dynamic memory allocations to call an arbitrary function with two controlled arguments using gadgets exploited described before. This gadget technique was previously exploited using a technique called pegasus

[Exploit 1] Kernel Userspace Pointer Memory Corruption

The fundamental services and primitives of WatchOS are based on flavors of Mach. It is extensively conceived as a simple, extensible, communications microkernel. Mach provides a custom system calls called Mack traps, which are commonly used for InterProcess Communication (IPC) and they are identified using various trap-levels. The trap-level interfaces (such as `mach_msg_overwrite_trap`) and message formats are themselves abstracted in normal usage by the Mach Interface Generator (*MIG*). MIG is used to compile procedural interfaces to the message-based APIs, based on descriptions of those APIs.

Our first exploit we observed in Mach trap recipe as the application tries to file for an exception of an application crash. A filename, ***mach_voucher_extract_attr_recipe_trap,*** is another mach trap referred for all such trap-levels. We observed that userland pointer information is first read and if Mach recipe array size is less than 5120 and greater than 256 than userland pointer is allocated to kernel heap buffer. This pointer is loaded into the memory without validating the content or the size. The failure to check for the validation causes memory corruption. This may cause one of the two expected behaviors, one to cause denial of service or ROP attack.

```
kern_return_t
  mach_voucher_extract_attr_recipe_trap(struct
mach_voucher_extract_attr_recipe_args *args)
  {
    ipc_voucher_t voucher = IV_NULL;
    kern_return_t kr = KERN_SUCCESS;
    mach_msg_type_number_t sz = 0;

    if (copyin(args->recipe_size, (void *)&sz, sizeof(sz)))
<---------- (1)
      return KERN_MEMORY_ERROR;

    if (sz > MACH_VOUCHER_ATTR_MAX_RAW_RECIPE_ARRAY_SIZE)
      return MIG_ARRAY_TOO_LARGE;

    voucher = convert_port_name_to_voucher(args->voucher_name);
    if (voucher == IV_NULL)
      return MACH_SEND_INVALID_DEST;

    mach_msg_type_number_t __assert_only max_sz = sz;

    if (sz < MACH_VOUCHER_TRAP_STACK_LIMIT) {
      /* keep small recipes on the stack for speed */
      uint8_t krecipe[sz];
      if (copyin(args->recipe, (void *)krecipe, sz)) {
        kr = KERN_MEMORY_ERROR;
        goto done;
      }
      kr = mach_voucher_extract_attr_recipe(voucher, args->key,

(mach_voucher_attr_raw_recipe_t)krecipe, &sz);
      assert(sz <= max_sz);

      if (kr == KERN_SUCCESS && sz > 0)
        kr = copyout(krecipe, (void *)args->recipe, sz);
    } else {
      uint8_t *krecipe = kalloc((vm_size_t)sz);
<---------- (2)
      if (!krecipe) {
        kr = KERN_RESOURCE_SHORTAGE;
        goto done;
      }

      if (copyin(args->recipe, (void *)krecipe, args-
>recipe_size)) {         <----------- (3)
        kfree(krecipe, (vm_size_t)sz);
        kr = KERN_MEMORY_ERROR;
        goto done;
      }

      kr = mach_voucher_extract_attr_recipe(voucher, args->key,

(mach_voucher_attr_raw_recipe_t)krecipe, &sz);
      assert(sz <= max_sz);

      if (kr == KERN_SUCCESS && sz > 0)
        kr = copyout(krecipe, (void *)args->recipe, sz);
      kfree(krecipe, (vm_size_t)sz);
    }

    kr = copyout(&sz, args->recipe_size, sizeof(sz));

  done:
    ipc_voucher_release(voucher);
    return kr;
  }
```

At point (1) four bytes are read from the userspace pointer recipe_size into sz.

At point (2) if sz was less than MACH_VOUCHER_ATTR_MAX_RAW_RECIPE_ARRAY_SIZE (5120) and greater than MACH_VOUCHER_TRAP_STACK_LIMIT (256)
sz is used to allocate a kernel heap buffer.

At point (3) copying is called again to copy userspace memory into that buffer which was just allocated, but rather than passing sz (the validate size which was allocated) args->recipe_size is passed as the size. This is the userspace pointer *to* the size, not the size!

[Exploit 2] Memory corruption WatchOS

Broadcom produces Wi-Fi and Bluetooth SoCs which handles the PHY and MAC layer processing. These chips have the capability to handle many a Wi-Fi related events without even delegating it to WatchOS. WatchOS implements *AppleBCMWLANBusInterfacePCIe* driver in order to handle the PCIe interface and low-level communication protocols with the Wi-Fi SoC. Similarly, the *AppleBCMWLANCore* driver handles the high-level protocols and the Wi-Fi configuration.

WiFi SOC notifies WatchOS of an event by encoding a special packet and transmitting it to the host. These packets have an ether type of 0x886C, and do not contain actual packet data, but rather encapsulate information about events which must be handled by the driver. One of the supported event packets is the WLC_E_TRACE message, containing a trace sent from the firmware which may be logged or stored by the host. On WatchOS, these events are handled by the "handleTraceEvent" function in the "AppleBCMWLANCore" driver. Each packet of this type starts with the common event message header (which is 48 bytes long), followed by the message-trace header:

```
int64_t handleTraceEvent(void* this, uint8_t* event_packet) {

  struct msgtrace_hdr hdr;
  memmove(&hdr, event_packet + 48, sizeof(struct
msgtrace_header));

  if (hdr.version == 1) {
    ...
    if (hdr.trace_type == 0) {
      event_packet[htons(hdr.len) + 64] = 0;
      ...
    }

    ...
  }
}
```

For messages of type 0 there is no validation of the len field before using it as an index into the event packet. As a result, an attacker controlling the firmware can craft a WLC_E_TRACE event packet with a large msgtrace length field, causing an OOB NUL byte to be written at the attacker-controlled 16-bit offset.