News and updates from the Project Zero team at

Thursday, June 11, 2020

A survey of recent iOS kernel explo

Posted by Brandon Azad, Project Zero

I recently found myself wishing for a single online reference providing a brief summary of the high-level exploit flow of every public iOS kernel exploit in recent years; since no such document existed, I decided to create it here.

This post summarizes original iOS kernel exploits from local app context targeting iOS 10 through iOS 13, focusing on the high-level exploit flow from the initial primitive granted by the vulnerability to kernel read/write. At the end of this post, we will briefly look at iOS kernel exploit mitigations (in both hardware and software) and how they map onto the techniques used in the exploits.

This isn't your typical P0 blog post: There is no gripping zero-day exploitation, or novel exploitation research, or thrilling malware reverse engineering. The content has been written as a reference since I needed the information and figured that others might find it useful too. You have been forewarned.

A note on terminology

Unfortunately, there is no authoritative dictionary called "Technical Hacking Terms for Security Researchers", which makes it difficult to precisely describe some of the high-level concepts I want to convey. To that end, I have decided to ascribe the following terms specific meanings for the context of this post. If any of these definitions are at odds with your understanding of these terms, feel free to suggest improved terminology and I can update this post. :)

Exploit primitive: A capability granted during an exploit that is reasonably generic.

A few examples of common exploit primitives include: *n*-byte linear heap overflow, integer increment at a controlled address, write-what-where, arbitrary memory read/write, PC control, arbitrary function calling, etc.

A common exploit primitive specific to iOS kernel exploitation is having a send right to a fake Mach port (struct `ipc_port`) whose fields can be directly read and written from userspace.

Exploit strategy: The low-level, vulnerability-specific method used to turn the vulnerability into a useful exploit primitive.

For example, this is the exploit strategy used in Ian Beer's async_wake exploit for iOS 11.1.2:

> An information leak is used to discover the address of arbitrary Mach ports. A page of ports is allocated and a specific port from that page is selected based on its address. The `IOSurfaceRootUserClient` bug is triggered to deallocate the Mach port, yielding a *receive right to a dangling Mach port at a known (and partially controlled) address*.

The last part is the generic/vulnerability-independent primitive that I interpret to be the end of the vulnerability-specific exploit strategy.

Typically, the aim of the exploit strategy is to produce an exploit primitive which is highly reliable.

Exploit technique: A reusable and reasonably generic strategy for turning one exploit primitive into another (usually more useful) exploit primitive.

One example of an exploit technique is Return-Oriented Programming (ROP), which turns arbitrary PC control into (nearly) arbitrary code execution by reusing executable code gadgets.

An exploit technique specific to iOS kernel exploitation is using a fake Mach port to read 4 bytes of kernel memory by calling `pid_for_task()` (turning a send right to a fake Mach port into an arbitrary kernel memory read primitive).

Exploit flow: The high-level, vulnerability-agnostic chain of exploit techniques used to turn the exploit primitive granted by the vulnerability into the final end goal (in this post, kernel read/write from local app context).

Public iOS kernel exploits from app context since iOS 10

This section will give a brief overview of iOS kernel exploits from local context targeting iOS 10 through iOS 13. I'll describe the high-level exploit flow and list the exploit primitives and techniques used to achieve it. While I have tried to track down every original (i.e., developed before exploit code was published) public exploit available either as source code or as a sufficiently complete writeup/presentation, I expect that I may have missed a few. Feel free to reach out and suggest any that I have missed and I can update this post.

Search This Blog

Search

Pages

- About Project Zer
- Working at Project Zer
- 0day "In the Wild"
- Vulnerability Disclosure FA(

Archives

2020

- FF Sandbox Escape (CVE-2020-1238(Jun)
- A survey of recent iOS kernel expl(Jun)
- Fuzzing ImageIO(Apr)
- You Won't Believe what this One Li Change Did to.(Apr)
- TFW you-get-really-excited-you-patch-diff a-0day..(Apr)
- Escaping the Chrome Sandbox with R(Feb)
- Mitigations are attack surface,(Feb)
- A day^W^W Several months in the life of Proj Ze...(Feb)
- A day^W^W Several months in the life of Proj Ze...(Feb)
- Part II: Returning to Adobe Reader symbols macOS(Jan)
- Remote iPhone Exploitation Part 3: Fr Memory Cor.(Jan)
- Remote iPhone Exploitation Part 2: Bring Light ...(Jan)
- Remote iPhone Exploitation Part 1: Pok Memory v..(Jan)
- Policy and Disclosure: 2020 Edit(Jan)

2019

- Calling Local Windows RPC Servers from .NI (Dec)
- SockPuppet: A Walkthrough of a Kernel Exp for ...(Dec)
- Bad Binder: Android In-The-Wild Expl(Nov)
- KTRW: The journey to build a debuggal iPhone(Oct)
- The story of Adobe Reader symb(Oct)
- Windows Exploitation Tricks: Spoofing Nam (Sep)
- A very deep dive into iOS Exploit chains fou in ...(Aug)
- In-the-wild iOS Exploit Chain(Aug)
- In-the-wild iOS Exploit Chain(Aug)
- In-the-wild iOS Exploit Chain(Aug)
- In-the-wild iOS Exploit Chain(Aug)
- In-the-wild iOS Exploit Chain(Aug)
- Implant Teardow(Aug)
- JSC Exploits(Aug)
- The Many Possibilities of CVE-2019-86(Aug)
- Down the Rabbit-Hole(Aug)
- The Fully Remote Attack Surface of the iPh (Aug)
- Trashing the Flow of Da(May)
- Windows Exploitation Tricks: Abusing the U: Mode..(Apr)
- Virtually Unlimited Memory: Escaping tl Chrome Sa.(Apr)
- Splitting atoms in XN(Apr)
- Windows Kernel Logic Bug Class: Access Mo Mismat..(Mar)
- Android Messaging: A Few Bugs Short o Chain(Mar)
- The Curious Case of Convexity Confu(Feb)

each case I've highlighted the particular exploitation primitive granted by the vulnerability that I consider sufficiently generic.

mach_portal - iOS 10.1.1

By Ian Beer of Google Project Zero (@i41nbeer).

The vulnerability: CVE-2016-7644 is a race condition in XNU's `set_dp_control_port()` which leads to a Mach port being over-released.

Exploit strategy: Many Mach ports are allocated and references to them are dropped by racing `set_dp_control_port()` (it is possible to determine when the race has been won deterministically). The ports are freed by dropping a stashed reference, leaving the process holding **receive rights to dangling Mach ports** filling a page of memory.

Subsequent exploit flow: A zone garbage collection is forced by calling `mach_zone_force_gc()` and the page of dangling ports is reallocated with an out-of-line (OOL) ports array containing pointers to the host port. `mach_port_get_context()` is called on one of the dangling ports to disclose the address of the host port. Using this value, it is possible to guess the page on which the kernel task port lives. The context value of each of the dangling ports is set to the address of each potential `ipc_port` on the page containing the kernel task port, and the OOL ports are received back in userspace to give a send right to the **kernel task port**.

References: mach_portal exploit code.

In-the-wild iOS Exploit Chain 1 - iOS 10.1.1

Discovered in-the-wild by Clément Lecigne (@_clem1) of Google's Threat Analysis Group. Analyzed by Ian Beer and Samuel Groß (@5aelo) of Google Project Zero.

The vulnerability: The vulnerability is a linear heap out-of-bounds write of `IOAccelResource` pointers in the IOKit function `AGXAllocationList2::initWithSharedResourceList()`.

Exploit strategy: The buffer to be overflowed is placed directly before a `recv_msg_elem` struct, such that the out-of-bounds write will overwrite the `uio` pointer with an `IOAccelResource` pointer. The `IOAccelResource` pointer is freed and reallocated with a fake `uio` struct living at the start of an `OSData` data buffer managed by `IOSurface` properties. The `uio` is freed, leaving a **dangling `OSData` data buffer accessible via `IOSurface` properties**.

Subsequent exploit flow: The dangling `OSData` data buffer slot is reallocated with an `IOSurfaceRootUserClient` instance, and the data contents are read via `IOSurface` properties to give the KASLR slide, the address of the current task, and the address of the dangling data buffer/`IOSurfaceRootUserClient`. Then, the data buffer is freed and reallocated with a modified version of the `IOSurfaceRootUserClient`, such that calling an external method on the modified user client will return the address of the kernel task read from the kernel's `__DATA` segment. The data buffer is freed and reallocated again such that calling an external method will execute the `OSSerializer::serialize()` gadget, leading to an arbitrary read-then-write that stores the address of the kernel task port in the current task's list of special ports. Reading the special port from userspace gives a send right to the **kernel task port**.

References: In-the-wild iOS Exploit Chain 1 - AGXAllocationList2::initWithSharedResourceList heap overflow.

extra_recipe - iOS 10.2

By Ian Beer.

The vulnerability: CVE-2017-2370 is a linear heap buffer overflow reachable from unprivileged contexts in XNU's `mach_voucher_extract_attr_recipe_trap()` due to an attacker-controlled userspace pointer used as the length in a call to `copyin()`.

Exploit strategy: The vulnerable Mach trap is called to create a `kalloc` allocation and immediately overflow out of it with controlled data, corrupting the `ikm_size` field of a subsequent `ipc_kmsg` object. This causes the `ipc_kmsg`, which is the preallocated message for a Mach port, to believe that it has a larger capacity than it does, overlapping it with the first 240 bytes of the subsequent allocation. By registering the Mach port as the exception port for a userspace thread and then crashing the thread with controlled register state, it is possible to repeatedly and reliably overwrite the overlapping part of the subsequent allocation, and by receiving the exception message it is possible to read those bytes. This gives a **controlled 240-byte out-of-bounds read/write primitive** off the end of the corrupted `ipc_kmsg`.

Subsequent exploit flow: A second `ipc_kmsg` is placed after the corrupted one and read in order to determine the address of the allocations. Next an `AGXCommandQueue` user client is reallocated in the same slot and the virtual method table is read to determine the KASLR slide. Then the virtual method table is overwritten such that a virtual method call on the `AGXCommandQueue` invokes the `OSSerializer::serialize()` gadget, producing a 2-argument arbitrary **kernel function call primitive**. Calling the function `uuid_copy()` gives an arbitrary kernel read/write primitive.

References: Exception oriented exploitation on iOS, extra_recipe exploit code.

The vulnerability: CVE-2017-2370 (same as above).

Exploit strategy: The vulnerable Mach trap is called to create a `kalloc` allocation and immediately overflow out of it with controlled data, overwriting the contents of an OOL port array and inserting a pointer to a fake Mach port in userspace. Receiving the message containing the OOL ports yields a **send right to the fake Mach port whose contents can be controlled directly**.

Subsequent exploit flow: The fake Mach port is converted into a clock port and `clock_sleep_trap()` is used to brute force a kernel image pointer. Then the port is converted into a fake task port to read memory via `pid_for_task()`. Kernel memory is scanned backwards from the leaked kernel image pointer until the kernel text base is located, breaking KASLR. Finally, a **fake kernel task port** is constructed.

Notes: The exploit does not work with PAN enabled.

References: Yalu102 exploit code.

ziVA - iOS 10.3.1

By Adam Donenfeld (@doadam) of Zimperium.

The vulnerability: Multiple vulnerabilities in `AppleAVE2` due to external methods sharing `IOSurface` pointers with userspace and trusting `IOSurface` pointers read from userspace.

Exploit strategy: An `IOSurface` object is created and an `AppleAVE2` external method is called to leak its address. The vtable of an `IOFence` pointer in the `IOSurface` is leaked using another external method call, breaking KASLR. The `IOSurface` object is freed and reallocated with controlled data using an `IOSurface` property spray. Supplying the leaked pointer to an `AppleAVE2` external method that trusts `IOSurface` pointers supplied from userspace allows hijacking a virtual method call on the fake `IOSurface`; this is treated as a **oneshot hijacked virtual method call with a controlled target object at a known address**.

Subsequent exploit flow: The hijacked virtual method call is used with the `OSSerializer::serialize()` gadget to call `copyin()` and overwrite 2 `sysctl_oid` structs. The sysctls are overwritten such that reading the first sysctl calls `copyin()` to update the function pointer and arguments for the second sysctl and reading the second sysctl uses the `OSSerializer::serialize()` gadget to call the kernel function with 3 arguments. This 3-argument arbitrary **kernel function call primitive** is used to read and write arbitrary memory by calling `copyin()`/`copyout()`.

Notes: iOS 10.3 introduced the initial form of `task_conversion_eval()`, a weak mitigation that blocks userspace from accessing a right to the real kernel task port. Any exploit after iOS 10.3 needs to build a fake kernel task port instead.

References: Ro(o)tten Apples, ziVA exploit code.

async_wake - iOS 11.1.2

By Ian Beer.

The vulnerability: CVE-2017-13861 is a vulnerability in `IOSurfaceRootUserClient::s_set_surface_notify()` that causes an extra reference to be dropped on a Mach port. CVE-2017-13865 is a vulnerability in XNU's `proc_list_uptrs()` that leaks kernel pointers by failing to fully initialize heap memory before copying out the contents to userspace.

Exploit strategy: The information leak is used to discover the address of arbitrary Mach ports. A page of ports is allocated and a specific port from that page is selected based on its address. The port is deallocated using the `IOSurfaceRootUserClient` bug, yielding a **receive right to a dangling Mach port at a known (and partially controlled) address**.

Subsequent exploit flow: The other ports on that page are freed and a zone garbage collection is forced so that the page is reallocated with the contents of an `ipc_kmsg`, giving a fake Mach port with controlled contents at a known address. The reallocation converted the port into a fake task port through which arbitrary kernel memory can be read using `pid_for_task()`. (The address to read is updated without reallocating the fake port by using `mach_port_set_context()`.) Relevant kernel objects are located using the kernel read primitive and the fake port is reallocated again with a **fake kernel task port**.

Notes: iOS 11 removed the `mach_zone_force_gc()` function which allowed userspace to prompt the kernel to perform a zone garbage collection, reclaiming all-free virtual pages in the zone map for use by other zones. Exploits for iOS 11 and later needed to develop a technique to force a zone garbage collection. At least three independent techniques have been developed to do so, demonstrated in async_wake, v0rtex, and In-the-wild iOS exploit chain 3.

References: async_wake exploit code.

In-the-wild iOS Exploit Chain 2 - iOS 10.3.3

Discovered in-the-wild by Clément Lecigne. Analyzed by Ian Beer and Samuel Groß.

The vulnerability: CVE-2017-13861 (same as above).

Exploit strategy: Two Mach ports, port A and port B, are allocated as part of a spray. The vulnerability is triggered again with port B, leading to a **receive right to a dangling Mach port at a known address**.

Subsequent exploit flow: After another zone garbage collection, the dangling port B is reallocated with a segmented OOL memory spray such that calling `mach_port_get_context()` can identify which 4 MB segment of the spray reallocated port B. That segment is freed and port B is reallocated with pipe buffers, giving a controlled fake Mach port at a known address. The fake port is converted into a clock port and `clock_sleep_trap()` is used to brute force KASLR. The fake port is next converted into a fake task port and a 4-byte kernel read primitive is established using `pid_for_task()`. Finally, the fake port is converted into a **fake kernel task port**.

References: In-the-wild iOS Exploit Chain 2 - IOSurface.

v0rtex - iOS 10.3.3

By Siguza (@S1guza).

The vulnerability: CVE-2017-13861 (same as above).

Exploit strategy: Mach ports are sprayed and a reference on one port is dropped using the vulnerability. The other ports on the page are freed, leaving a **receive right to a dangling Mach port**.

Subsequent exploit flow: A zone garbage collection is forced using `mach_zone_force_gc()` and the page containing the dangling port is reallocated with an `OSString` buffer via an `IOSurface` property spray. The `OSString` buffer contains a pattern that initializes critical fields of the port and allows the index of the `OSString` containing the port to be determined by calling `mach_port_get_context()` on the fake port. The `OSString` containing the fake port is freed and reallocated as a normal Mach port. `mach_port_request_notification()` is called to put the address of a real Mach port in the fake port's `ip_pdrequest` field, and the `OSString`'s contents are read via `IOSurface` to get the address. `mach_port_request_notification()` is used again to get the address of the fake port itself.

The string buffer is freed and reallocated such that `mach_port_get_attributes()` can be used as a 4-byte arbitrary read primitive, with the target address to read updateable via `mach_port_set_context()`. (This is analogous to the `pid_for_task()` technique, but with slightly different constraints.) Starting at the address of the real Mach port, kernel memory is read to find relevant kernel objects. The string buffer is freed and reallocated again with a fake task port sufficient to remap the string buffer into the process's address space. The fake port is updated via the mapping to yield a 7-argument arbitrary **kernel function call primitive** using `iokit_user_client_trap()`, and kernel functions are called to generate a **fake kernel task port**.

References: v0rtex writeup, v0rtex exploit code.

Incomplete exploit for CVE-2018-4150 bpf-filter-poc - iOS 11.2.6

Vulnerability analysis and POC by Chris Wade (@cmwdotme) at Corellium. Exploit by littlelailo (@littlelailo).

The vulnerability: CVE-2018-4150 is a race condition in XNU's BPF subsystem which leads to a linear heap buffer overflow due to a buffer length being increased without reallocating the corresponding buffer.

Exploit strategy: The race is triggered to incorrectly increase the length of the buffer without reallocating the buffer itself. A packet is sent and stored in the buffer, overflowing into a subsequent OOL ports array and inserting a pointer to a fake Mach port in userspace. Receiving the message containing the OOL ports yields a **send right to the fake Mach port whose contents can be controlled directly**.

Subsequent exploit flow: The fake Mach port is converted into a clock port and `clock_sleep_trap()` is used to brute force a kernel image pointer. Then the port is converted into a fake task port to read memory via `pid_for_task()`. Kernel memory is scanned backwards from the leaked kernel image pointer until the kernel text base is located, breaking KASLR. The final part of the exploit is incomplete, but construction of a **fake kernel task port** at this stage would be straightforward and deterministic using existing code.

Notes: The exploit does not work with PAN enabled.

References: CVE-2018-4150 POC, incomplete-exploit-for-CVE-2018-4150-bpf-filter-poc exploit code.

multi_path - iOS 11.3.1

By Ian Beer.

The vulnerability: CVE-2018-4241 is an intra-object linear heap buffer overflow in XNU's `mptcp_usr_connectx()` due to incorrect bounds checking.

Exploit strategy: The kernel heap is groomed to place a 2048-byte `ipc_kmsg` struct at a 16 MB aligned address below the `mptses` structs (the object containing the overflow) associated with a few multipath TCP sockets. The vulnerability is used to overwrite the lower 3 bytes of the `mpte_itfinfo` pointer in the `mptses` struct with zeros and the socket is closed. This triggers a `kfree()` of the corrupted pointer, freeing the `ipc_kmsg` struct at the 16 MB alignment boundary. The freed `ipc_kmsg` slot is reallocated with sprayed pipe buffers. The vulnerability is triggered again to overwrite the lower 3 bytes of the `mpte_itfinfo` pointer in another `mptses` struct with zeros and the socket is closed, causing another `kfree()` of the same address. This frees the pipe buffer that was just allocated into that slot, leaving a **dangling pipe buffer**.

pipe buffer/`ipc_kmsg`. The pipe is written to change the contents of the `ipc_kmsg` struct such that receiving the message yields a send right to a fake Mach port inside the pipe buffer. The exception message is received and the pipe is rewritten to convert the fake port into a kernel read primitive using `pid_for_task()`. Relevant kernel objects are located and the fake port is converted into a **fake kernel task port**.

References: multi_path exploit code.

multipath_kfree - iOS 11.3.1

By John Åkerblom (@jaakerblom).

The vulnerability: CVE-2018-4241 (same as above).

Exploit strategy: The kernel heap is groomed to place preallocated 4096-byte `ipc_kmsg` structs near the `mptses` structs for a few multipath TCP sockets. The vulnerability is triggered twice to corrupt the lower 2 bytes of the `mpte_itfinfo` pointer in two `mptses` structs, such that closing the sockets results in `kfree()`s of the two corrupted pointers. Each pointer is corrupted to point `0x7a0` bytes into an `ipc_kmsg` allocation, creating 4096-byte holes spanning 2 messages. A Mach port containing one of the partially-freed `ipc_kmsg` structs (with the `ipc_kmsg` header intact but the message contents freed) is located by using `mach_port_peek()` to detect a corrupted `msgh_id` field. Once the port is found, the hole is reallocated by spraying preallocated `ipc_kmsg` structs and a message is placed in each. Filling the hole overlaps the original (partially freed) `ipc_kmsg`'s Mach message contents with the `ipc_kmsg` header of the replacement, such that receiving the message on the original port reads the contents of the replacement `ipc_kmsg` header. The header contains a pointer to itself, disclosing the address of the replacement `ipc_kmsg` allocation. The vulnerability is triggered a third time to free the replacement message, leaving a **partially freed preallocated `ipc_kmsg` at a known address**.

Subsequent exploit flow: The hole in the corrupted `ipc_kmsg` is reallocated by spraying `AGXCommandQueue` user clients. A message is received on the Mach port in userspace, copying out the contents of the `AGXCommandQueue` object, from which the vtable is used to determine the KASLR slide. Then the corrupted `ipc_kmsg` is freed and reallocated by spraying more preallocated `ipc_kmsg` structs with a slightly different internal layout allowing more control over the contents. A message is placed in each of the just-sprayed `ipc_kmsg` structs to modify the overlapping `AGXCommandQueue` and hijack a virtual method call; the hijacked virtual method uses the `OSSerializer::serialize()` gadget to call `copyout()`, which is used to identify which of the sprayed `AGXCommandQueue` user clients overlaps the slot from the corrupted `ipc_kmsg`. The contents of each of the just-sprayed preallocated `ipc_kmsg` structs is updated in turn to identify which port corresponds to the corrupted `ipc_kmsg`. The preallocated port and user client port are used together to build a 3-argument arbitrary **kernel function call primitive** by updating the contents of the `AGXCommandQueue` object through an exception message sent to the preallocated port.

References: multipath_kfree exploit code.

empty_list - iOS 11.3.1

By Ian Beer.

The vulnerability: CVE-2018-4243 is a partially controlled 8-byte heap out-of-bounds write in XNU's `getvolattrlist()` due to incorrect bounds checking.

Exploit strategy: Due to significant triggering constraints, the vulnerability is treated as an 8-byte heap out-of-bounds write of zeros off the end of a `kalloc.16` allocation. The kernel heap is groomed into a pattern of alternating blocks for the zones of `kalloc.16` and `ipc.ports`, and further grooming reverses the `kalloc.16` freelist. The vulnerability is repeatedly triggered after freeing various `kalloc.16` allocations until a `kalloc.16` allocation at the end of a block is overflowed, corrupting the first 8 bytes of the first `ipc_port` on the subsequent page. The corrupted port is freed by calling `mach_port_set_attributes()`, leaving the process holding a **receive right to a dangling Mach port**.

Subsequent exploit flow: A zone garbage collection is forced and the dangling port is reallocated with an OOL ports array containing a pointer to another Mach port overlapping the `ip_context` field, so that the address of the other port is retrieved by calling `mach_port_get_context()`. The dangling port is then reallocated with pipe buffers and converted into a kernel read primitive using `pid_for_task()`. Using the address of the other port as a starting point, relevant kernel objects are located. Finally, the fake port is converted into a **fake kernel task port**.

References: empty_list exploit code.

In-the-wild iOS Exploit Chain 3 - iOS 11.4

Discovered in-the-wild by Clément Lecigne. Analyzed by Ian Beer and Samuel Groß.

The vulnerability: The vulnerability is a double-free reachable from `AppleVXD393UserClient::DestroyDecoder()` (the class name varies by hardware) due to failing to clear a freed pointer.

Exploit strategy: The target 56-byte allocation is created and freed, leaving the dangling pointer intact. The slot is reallocated with an `OSData` buffer using an `IOSurface` property spray. The vulnerable method is called again to free the buffer, leaving a dangling `OSData` buffer. The slot is reallocated again with an OOL

This leaves the process with a **receive right to a dangling Mach port at a known address**.

Subsequent exploit flow: A zone garbage collection is performed and the dangling port is reallocated with a segmented OOL memory spray such that calling `mach_port_get_context()` can identify which segment of the spray reallocated the port. That segment is freed and the dangling port is reallocated with pipe buffers, giving a controlled fake Mach port at a known address. The fake port is converted into a clock port and `clock_sleep_trap()` is used to brute force KASLR. The fake port is next converted into a fake task port and a kernel read primitive is established using `pid_for_task()`. Finally, the fake port is converted into a **fake kernel task port**.

References: In-the-wild iOS Exploit Chain 3 - XPC + VXD393/D5500 repeated IOFree.

Spice - iOS 11.4.1

Vulnerability analysis and POC by Luca Moro (@JohnCool__) at Synacktiv. Exploit by Siguza, Viktor Oreshkin (@stek29), Ben Sparkes (@iBSparkes), and littlelailo.

The vulnerability: The "LightSpeed" vulnerability (possibly CVE-2018-4344) is a race condition in XNU's `lio_listio()` due to improper state management that results in a use-after-free.

Exploit strategy: The vulnerable function is called in a loop in one thread to repeatedly trigger the vulnerability by allocating a buffer from `kalloc.16` and racing to free the buffer twice. Another thread repeatedly sends a message containing an OOL ports array allocated from `kalloc.16`, immediately sprays a large number of `kalloc.16` allocations containing a pointer to a fake Mach port in userspace via `IOSurface` properties, and receives the OOL ports. When the race is won, the double-free can cause the OOL ports array to be freed, and the subsequent spray can reallocate the slot with a fake OOL ports array. Receiving the OOL ports in userspace gives a **receive right to a fake Mach port whose contents can be controlled directly**.

Subsequent exploit flow: A second Mach port is registered as a notification port on the fake port, disclosing the address of the second port in the fake port's `ip_pdrequest` field. The fake port is modified to construct a kernel read primitive using `mach_port_get_attributes()`. Starting from the disclosed port pointer, kernel memory is read to find relevant kernel objects. The fake port is converted into a fake user client port providing a 7-argument arbitrary **kernel function call primitive** using `iokit_user_client_trap()`. Finally, a **fake kernel task port** is constructed.

Notes: The exploit does not work with PAN enabled.

The analysis was performed on the implementation in the file `pwn.m`, since this seems to provide the most direct comparison to the other exploit implementations in this list.

References: LightSpeed, a race for an iOS/macOS sandbox escape, Spice exploit code.

treadm1ll - iOS 11.4.1

Vulnerability analysis and POC by Luca Moro. Exploit by Tihmstar (@tihmstar).

The vulnerability: The "LightSpeed" vulnerability (same as above).

Exploit strategy: The vulnerable function is called in a loop in one thread to repeatedly trigger the vulnerability by allocating a buffer from `kalloc.16` and racing to free the buffer twice. Another thread sends a fixed number of messages containing an OOL ports array allocated from `kalloc.16`. When the race is won, the double-free can cause the OOL ports array to be freed, leaving a dangling OOL ports array pointer in some messages. The first thread stops triggering the vulnerability and a large number of `IOSurface` objects are created. Each message is received in turn and a large number of `kalloc.16` allocations containing a pointer to a fake Mach port in userspace are sprayed using `IOSurface` properties. Each spray can reallocate a slot from a dangling OOL ports array with a fake OOL ports array. Successfully receiving the OOL ports in userspace gives a **receive right to a fake Mach port whose contents can be controlled directly**.

Subsequent exploit flow: A second Mach port is registered as a notification port on the fake port, disclosing the address of the second port in the fake port's `ip_pdrequest` field. The fake port is modified to construct a kernel read primitive using `pid_for_task()`. Starting from the disclosed port pointer, kernel memory is read to find relevant kernel objects. The fake port is converted into a fake user client port providing a 7-argument arbitrary **kernel function call primitive** using `iokit_user_client_trap()`. Finally, a **fake kernel task port** is constructed.

Notes: The exploit does not work with PAN enabled.

References: LightSpeed, a race for an iOS/macOS sandbox escape, treadm1ll exploit code.

Chaos - iOS 12.1.2

By Qixun Zhao (@S0rryMybad) of Qihoo 360 Vulcan Team.

The vulnerability: CVE-2019-6225 is a use-after-free due to XNU's `task_swap_mach_voucher()` failing to comply with MIG lifetime semantics that results in an extra reference being added or dropped on an `ipc_voucher` object.

`ith_voucher` field.

Subsequent exploit flow: The dangling voucher is reallocated by an `OSString` buffer using an `IOSurface` property spray. `thread_get_mach_voucher()` is called to obtain a send right to a newly allocated voucher port for the voucher, which causes a pointer to the voucher port to be stored in the fake voucher overlapping the `OSString` buffer; reading the `OSString` property discloses the address of the voucher port. The `OSString` overlapping the fake voucher is freed and reallocated with a large spray that both forces the allocation of controlled data containing a fake Mach port at a hardcoded address and updates the fake voucher's `iv_port` pointer to point to the fake Mach port. `thread_get_mach_voucher()` is called again to obtain a send right to the fake port and to identify which `OSString` buffer contains the fake Mach port. This leaves the process with a **send right to a fake Mach port in an `IOSurface` property buffer at a known address** (roughly equivalent to a dangling Mach port). A kernel read primitive is built by reallocating the `OSString` buffer to convert the fake port into a fake task port and calling `pid_for_task()` to read arbitrary memory. Relevant kernel objects are located and the fake port is converted into a fake map port to remap the fake port into userspace, removing the need to reallocate it. Finally the fake port is converted into a **fake kernel task port**.

Notes: The A12 introduced PAC, which limits the ability to use certain exploitation techniques involving code pointers (e.g. vtable hijacking). Also, iOS 12 introduced a mitigation in `ipc_port_finalize()` against freeing a port while it is still active (i.e. hasn't been destroyed, for example because a process still holds a right to it). This changed the common structure of past exploits whereby a port would be freed while a process still held a right to it. Possibly as a result, obtaining a right to a fake port in iOS 12+ exploits seems to occur later in the flow than in earlier exploits.

References: IPC Voucher UaF Remote Jailbreak Stage 2 (EN).

voucher_swap - iOS 12.1.2

By Brandon Azad (@_bazad) of Google Project Zero.

The vulnerability: CVE-2019-6225 (same as above).

Exploit strategy: The kernel heap is groomed to put a block of `ipc_port` allocations directly before a block of pipe buffers. A large number of `ipc_voucher` objects are sprayed and the vulnerability is triggered to decrease the reference count on a voucher and free it. The remaining vouchers on the page are freed and a zone garbage collection is forced, leaving a **dangling `ipc_voucher` pointer in the thread's `ith_voucher` field**.

Subsequent exploit flow: The dangling voucher is reallocated with an OOL ports array containing a pointer to a previously-allocated `ipc_port` overlapping the voucher's `iv_refs` field. A send right to the voucher port is retrieved by calling `thread_get_mach_voucher()` and the voucher's reference count is increased by repeatedly calling the vulnerable function, updating the overlapping `ipc_port` pointer to point into the pipe buffers. Receiving the OOL ports yields a **send right to a fake Mach port whose contents can be controlled directly**. `mach_port_request_notification()` is called to insert a pointer to an array containing a pointer to another Mach port in the fake port's `ip_requests` field. A kernel read primitive is built using `pid_for_task()`, and the address of the other Mach port is read to compute the address of the fake port. Relevant kernel objects are located and a **fake kernel task port** is constructed.

References: voucher_swap: Exploiting MIG reference counting in iOS 12, voucher_swap exploit code.

machswap2 - iOS 12.1.2

By Ben Sparkes.

The vulnerability: CVE-2019-6225 (same as above).

Exploit strategy: A large number of `ipc_voucher` objects are sprayed and the vulnerability is triggered twice to decrease the reference count on a voucher and free it. The remaining vouchers on the page are freed and a zone garbage collection is forced, leaving a **dangling `ipc_voucher` pointer in the thread's `ith_voucher` field**.

Subsequent exploit flow: The dangling voucher is reallocated by an `OSString` buffer containing a fake voucher using an `IOSurface` property spray. `thread_get_mach_voucher()` is called to obtain a send right to a newly allocated voucher port for the voucher, which causes a pointer to the voucher port to be stored in the fake voucher overlapping the `OSString` buffer; reading the `OSString` property discloses the address of the voucher port. Pipe buffers containing fake task ports are sprayed to land roughly 1 MB after the disclosed port address. The `OSString` overlapping the fake voucher is freed and reallocated to update the fake voucher's `iv_port` pointer to point to point into the pipe buffers. `thread_get_mach_voucher()` is called again to retrieve the updated voucher port, yielding a **send right to a fake Mach port at a known address whose contents can be controlled directly**. The fake port is converted into a fake task port and a kernel read primitive is established using `pid_for_task()`. Relevant kernel objects are located and a **fake kernel task port** is constructed.

Notes: The author developed two versions of this exploit: one for pre-PAN devices, and one for PAN-enabled devices. The exploit presented here is for PAN-enabled devices.

References: machswap2 exploit code, MachSwap: an iOS 12 Kernel Exploit, machswap exploit code.

The vulnerability: CVE-2019-6225 (same as above).

Exploit strategy: A large number of `ipc_voucher` objects are sprayed and the vulnerability is triggered to decrease the reference count on a voucher and free it. The remaining vouchers on the page are freed and a zone garbage collection is forced, leaving a **dangling `ipc_voucher` pointer in the thread's `ith_voucher` field**.

Subsequent exploit flow: The dangling voucher is reallocated by an OOL memory spray. A large number of Mach ports are allocated and then `thread_get_mach_voucher()` is called to obtain a send right to a newly allocated voucher port for the voucher, which causes a pointer to the voucher port to be stored in the fake voucher overlapping the OOL ports array. More ports are allocated and then the OOL memory spray is received, disclosing the address of the voucher port for the fake voucher. The dangling voucher is reallocated again with another OOL memory spray that updates the voucher's `iv_port` pointer to the subsequent page. The Mach ports are destroyed and a zone garbage collection is forced, leaving the fake voucher holding a pointer to a dangling port. The dangling port is reallocated with pipe buffers. Finally, `thread_get_mach_voucher()` is called, yielding a **send right to a fake Mach port at a known address whose contents can be controlled directly**. The fake port is converted into a fake task port and a kernel read primitive is established using `pid_for_task()`. Relevant kernel objects are located and the fake port is converted into a **fake kernel task port**.

References: In-the-wild iOS Exploit Chain 5 - task_swap_mach_voucher.

In-the-wild iOS Exploit Chain 4 - iOS 12.1.3

Discovered in-the-wild by Clément Lecigne. Analyzed by Ian Beer and Samuel Groß. Also reported by an anonymous researcher.

The vulnerability: CVE-2019-7287 is a linear heap buffer overflow in the IOKit function `ProvInfoIOKitUserClient::ucEncryptSUInfo()` due to an unchecked `memcpy()`.

Exploit strategy: The kernel heap is groomed to place holes in `kalloc.4096` before an OOL ports array and holes in `kalloc.6144` before an `OSData` buffer accessible via `IOSurface` properties. The vulnerability is triggered with the source allocated from `kalloc.4096` and the destination allocated from `kalloc.6144`, causing the address of a target Mach port to be copied into the `OSData` buffer. The `OSData` buffer is then read, disclosing the address of the target port. The heap is groomed again to place holes in `kalloc.4096` before an OOL memory buffer and in `kalloc.6144` before an OOL ports array. The vulnerability is triggered again to insert a pointer to the target port into the OOL ports array. The target port is freed and a zone garbage collection is forced, leaving a dangling port pointer in the OOL ports array. The dangling port is reallocated with pipe buffers and the OOL ports are received, giving a **receive right to a fake Mach port at a known address whose contents can be controlled directly**.

Subsequent exploit flow: The fake port is converted into a fake clock port and `clock_sleep_trap()` is used to brute force KASLR. The fake port is converted into a fake task port and a kernel read primitive is established using `pid_for_task()`. Relevant kernel objects are located and the fake port is converted into a **fake kernel task port**.

References: In-the-wild iOS Exploit Chain 4 - cfprefsd + ProvInfoIOKit, About the security content of iOS 12.1.4.

Attacking iPhone XS Max - iOS 12.1.4

By Tielei Wang (@wangtielei) and Hao Xu (@windknown).

The vulnerability: The vulnerability is a race condition in XNU's UNIX domain socket bind implementation due to the temporary unlock antipattern that results in a use-after-free.

Exploit strategy: Sockets are sprayed and the vulnerability is triggered to leave a pointer to a dangling `socket` pointer in a `vnode` struct. The sockets are closed, a zone garbage collection is forced, and the sockets are reallocated with controlled data via an `OSData` spray (possibly an `IOSurface` property spray). The fake socket is constructed to have a reference count of 0. The use after free is triggered to call `socket_unlock()` on the fake socket, which causes the fake socket/`OSData` buffer to be freed using `kfree()`. This leaves a **dangling `OSData` buffer accessible** using unspecified means.

Subsequent exploit flow: The dangling `OSData` buffer is reallocated with an OOL ports array and the `OSData` buffer is freed, leaving a dangling OOL ports array. Kernel memory is sprayed to place a fake Mach port at a hardcoded address (or an information leak is used) and the OOL ports array is reallocated with another `OSData` buffer, inserting a pointer to the fake Mach port into the OOL ports array. The OOL ports are received, yielding a send or receive right to the fake Mach port at a known address. The fake port is converted into a **fake kernel task port** by unspecified means.

Notes: The only reference for this exploit is a BlackHat presentation, hence the uncertainties in the explanations above.

The authors developed two versions of this exploit: one for non-PAC devices, and one for PAC-enabled devices. The exploit presented here is for PAC-enabled devices. The non-PAC exploit is substantially simpler (hijacking a function pointer used by `socket_lock()`).

References: Attacking iPhone XS Max.

The vulnerability: CVE-2019-8605 is a use-after-free due to XNU's `in6_pcbdetach()` failing to clear a freed pointer.

Exploit strategy: **Safe arbitrary read, arbitrary `kfree()`, and arbitrary Mach port address disclosure primitives are constructed over the vulnerability.**

The arbitrary read primitive: The vulnerability is triggered multiple times to create a number of dangling `ip6_pktopts` structs associated with sockets. The dangling `ip6_pktopts` are reallocated with an `OSData` buffer spray via `IOSurface` properties such that `ip6po_minmtu` is set to a known value and `ip6po_pktinfo` is set to the address to read. The `ip6po_minmtu` field is checked via `getsockopt()`, and if correct, `getsockopt(IPV6_PKTINFO)` is called to read 20 bytes of data from the address pointed to by `ip6po_pktinfo`.

The arbitrary `kfree()` primitive: The vulnerability is triggered multiple times to create a number of dangling `ip6_pktopts` structs associated with sockets. The dangling `ip6_pktopts` are reallocated with an `OSData` buffer spray via `IOSurface` properties such that `ip6po_minmtu` is set to a known value and `ip6po_pktinfo` is set to the address to free. The `ip6po_minmtu` field is checked via `getsockopt()`, and if correct, `setsockopt(IPV6_PKTINFO)` is called to invoke `kfree_addr()` on the `ip6po_pktinfo` pointer.

The arbitrary Mach port address disclosure primitive: The vulnerability is triggered multiple times to create a number of dangling `ip6_pktopts` structs associated with sockets. The dangling `ip6_pktopts` are reallocated with an OOL ports array spray containing pointers to the target port. The `ip6po_minmtu` and `ip6po_prefer_tempaddr` fields are read via `getsockopt()`, disclosing the value of the target port pointer. The port is checked to be of the expected type using the arbitrary read primitive.

Subsequent exploit flow: The Mach port address disclosure primitive is used to disclose the address of the current task. Two pipes are created and the addresses of the pipe buffers in the kernel are found using the kernel read primitive. Relevant kernel objects are located and a fake kernel task port is constructed in one of the pipe buffers. The arbitrary `kfree()` primitive is used to free the pipe buffer for the other pipe, and the pipe buffer is reallocated by spraying OOL ports arrays. The pipe is then written to insert a pointer to the fake kernel task port into the OOL ports array, and the OOL ports are received, yielding a **fake kernel task port**.

Notes: Unlike most other exploits on this list which are structured linearly, SockPuppet is structured hierarchically, building on the same primitives throughout. This distinct structure is likely due to the power and stability of the underlying vulnerability: the bug directly provides both an arbitrary read and an arbitrary free primitive, and in practice both primitives are 100% safe and reliable because it is possible to check that the reallocation is successful. However, this structure means that there is no clear temporal boundary in the high-level exploit flow between the vulnerability-specific and generic exploitation. Instead, that boundary occurs between conceptual layers in the exploit code.

The SockPuppet bug was fixed in iOS 12.3 but reintroduced in iOS 12.4.

References: [SockPuppet: A Walkthrough of a Kernel Exploit for iOS 12.4](), [SockPuppet exploit code]().

AppleAVE2Driver exploit - iOS 12.4.1

By 08Tc3wBB ([@08Tc3wBB]()).

The vulnerability: CVE-2019-8795 is a memory corruption in `AppleAVE2Driver` whereby improper bounds checking leads to processing of out-of-bounds data, eventually resulting in a controlled virtual method call or arbitrary `kfree()`. CVE-2019-8794 is a kernel memory disclosure in `AppleSPUProfileDriver` due to uninitialized stack data being shared with userspace.

Exploit strategy: The KASLR slide is discovered using the `AppleSPUProfileDriver` vulnerability. `OSData` buffers containing fake task ports are sprayed using `IOSurface` properties. The vulnerability is triggered to free an `OSData` buffer at a hardcoded address, leaving a **dangling `OSData` buffer accessible via `IOSurface` properties**.

Subsequent exploit flow: The dangling `OSData` buffer is reallocated with an OOL ports array and the `OSData` buffer is freed, leaving a dangling OOL ports array. The OOL ports array is reallocated with another `OSData` buffer, inserting pointers to the fake task ports sprayed earlier into the OOL ports array. The OOL ports are received, yielding send rights to the fake task ports, and `pid_for_task()` is used to read pointers to relevant kernel objects. The `OSData` buffer is freed and reallocated to convert one of the fake ports into a **fake kernel task port**.

Notes: iOS versions up to 13.1.3 were vulnerable, but the exploit presented here targeted iOS 12.4.1.

The author developed two versions of this exploit: one for non-PAC devices, and one for PAC-enabled devices. The exploit presented here is for PAC-enabled devices.

References: [ZecOps_FreeTheSandbox_iOS_PAC_TFP0_POC_BEQ_12_4_2 exploit code](), [ZecOps Task-For-Pwn 0 Bounty: TFP0 POC on PAC-Enabled iOS Devices <= 12.4.2](), [SSD Advisory – iOS Jailbreak via Sandbox Escape and Kernel R/W leading to RCE](), [SSD Advisory 4066 exploit code](), [About the security content of iOS 13.2 and iPadOS 13.2]().

oob_timestamp - iOS 13.3

checking.

Exploit strategy: The kernel map is groomed to lay out two 96 MB shared memory regions, an 8-page `ipc_kmsg`, an 8-page OOL ports array, and 80 MB of `OSData` buffers sprayed via `IOSurface` properties. The number of bytes to overflow is computed based on the current time and the overflow is triggered to corrupt the `ipc_kmsg`'s `ikm_size` field, such that the `ipc_kmsg` now has a size of between 16 pages and 80 MB. The port containing the `ipc_kmsg` is destroyed, freeing the corrupted `ipc_kmsg`, the OOL ports array, and some of the subsequent `OSData` buffers. More `OSData` buffers are sprayed via `IOSurface` to reallocate the OOL ports array containing a pointer to a fake Mach port at a hardcoded address that is likely to overlap one of the 96 MB shared memory regions. The OOL ports are received, producing a **receive right to a fake Mach port at a known address whose contents can be controlled directly**.

Subsequent exploit flow: A kernel memory read primitive is constructed using `pid_for_task()`. Relevant kernel objects are located and a **fake kernel task port** is constructed.

Notes: iOS 13 introduced zone_require, a mitigation that checks whether certain objects are allocated from the expected `zalloc` zone before they are used. An oversight in the implementation led to a trivial bypass when objects are allocated outside of the `zalloc_map`.

References: oob_timestamp exploit code.

iOS kernel exploit mitigations

Next we will look at some current iOS kernel exploit mitigations. This list is not exhaustive, but it briefly summarizes some of the mitigations that exploit developers may encounter up through iOS 13.

Kernel Stack Canaries - iOS 6

iOS 6 introduced kernel stack canaries (or stack cookies) to protect against stack buffer overflows in the kernel.

None of the exploits in this list are affected by the presence of stack canaries as they do not target stack buffer overflow vulnerabilities.

Kernel ASLR - iOS 6

Kernel Address Space Layout Randomization (Kernel ASLR or KASLR) is a mitigation that randomizes the base address of the kernelcache image in the kernel address space. Before Kernel ASLR was implemented, the addresses of kernel functions and objects in the kernelcache image were always located at a fixed address.

Bypassing or working around KASLR is a standard step of all modern iOS kernel exploits.

Kernel Heap ASLR - iOS 6

Since iOS 6 the base addresses for various kernel heap regions have been randomized. This seeks to mitigate exploits that hardcode addresses at which objects will be deterministically allocated.

Working around kernel heap randomization is a standard step of modern iOS kernel exploits. Usually this involves heap spraying, in which the kernel is induced to allocate large amounts of data to influence the shape of the heap even when exact addresses are not known. Also, many vulnerabilities can be leveraged to produce an information leak, disclosing the addresses of relevant kernel objects on the heap.

W^X / DEP - iOS 6

iOS 6 also introduced substantial kernel address space hardening by ensuring that kernel pages are mapped either as writable or as executable, but never both (often called "write xor execute" or W^X). This means that page tables no longer map kernel code pages as writable, and the kernel heap and stack are no longer mapped as executable. (Ensuring that non-code data is not mapped as executable is often called Data Execution Prevention, or DEP.)

Modern public iOS exploits do not attempt to bypass W^X (e.g. by modifying page tables and injecting shellcode); instead, exploitation is achieved by modifying kernel data structures and performing code-reuse attacks instead. This is largely due to the presence of a stronger, hardware-enforced W^X mitigation called KTRR.

PXN - iOS 7

Apple's A7 processor was the first 64-bit, ARMv8-A processor in an iPhone. Previously, iOS 6 had separated the kernel and user address space so that user code and data pages were inaccessible during normal kernel execution. With the move to 64-bit, the address spaces were no longer separated. Thus, the Privileged Execute-Never (PXN) bit was set in page table entries to ensure that the kernel could not execute shellcode residing in userspace pages.

Similarly to W^X, PXN as a protection against jumping to userspace shellcode is overshadowed by the stronger protection of KTRR.

to prevent the kernel from dereferencing attacker-supplied pointers to data structures in userspace. It is similar to the Supervisor Mode Access Prevention (SMAP) feature on some Intel processors.

While PAN has been bypassed before, modern public iOS kernel exploits usually work around PAN by spraying data into the kernel and then learning the address of the data. While the most reliable techniques involve disclosing the address of the data inserted into the kernel, techniques exist to work around PAN generically, such as spraying enough data to overwhelm the kernel map randomization and force a fixed, hardcoded address to be allocated with the controlled data. Other primitives exist for establishing shared memory mappings between userspace and the kernel, which can also be used to work around PAN.

KTRR - iOS 10

KTRR (possibly Kernel Text Readonly Region, part of Kernel Integrity Protection) is a custom hardware security mitigation introduced on the Apple A10 processor (ARMv8.1-A). It is a strong form of W^X protection enforced by the MMU and the memory controller over a single span of contiguous memory covering the read-only parts of the kernelcache image and some sensitive data structures like top-level page tables and the trust cache. It has also been referred to by Apple as Kernel Integrity Protection (KIP) v1.

While KTRR has been publicly bypassed twice before, modern public iOS kernel exploits usually work around KTRR by not manipulating KTRR-protected memory.

APRR - iOS 11

APRR (possibly standing for Access Protection Rerouting or Access Permission Restriction Register) is a custom hardware feature on Apple A11 and later CPUs that indirects virtual memory access permissions (usually specified in the page table entry for the page) through a special register, allowing access permissions for large groups of pages to be changed atomically and per-core. It works by converting the bits in the PTE that typically directly specify the access permissions into an index into a special register containing the true access permissions; changing the register value swaps protections on all pages mapped with the same access permissions index. APRR is somewhat similar to the Memory Protection Keys feature available on newer Intel processors.

APRR on its own does not provide any security boundaries, but it makes it possible to segment privilege levels inside a single address space. It is heavily used by PPL to create a security boundary within the iOS kernel.

PPL - iOS 12

PPL (Page Protection Layer) is the software layer built on APRR and dependent on KTRR that aims to put a security boundary between kernel read/write/execute and direct page table access. The primary goal of PPL is to prevent an attacker from modifying user pages that have been codesigned (e.g. using kernel read/write to overwrite a userspace process's executable code). This necessarily means that PPL must also maintain total control over the page tables and prevent an attacker from mapping sensitive physical addresses, including page tables, page table metadata, and IOMMU registers.

As of May 2020, PPL has not been publicly bypassed. That said, modern iOS kernel exploits are so far unaffected by PPL.

PAC - iOS 12

Pointer Authentication Codes (PAC) is an ARMv8.3-A security feature that mitigates pointer tampering by storing a cryptographic signature of the pointer value in the upper bits of the pointer. Apple introduced PAC with the A12 and significantly hardened the implementation (compared to the ARM standard) in order to defend against attackers with kernel read/write, although for most purposes it is functionally indistinguishable. Apple's kernel uses PAC for control flow integrity (CFI), placing a security boundary between kernel read/write and kernel code execution.

Despite numerous public bypasses of the iOS kernel's PAC-based CFI, PAC in the kernel is still an effective exploit mitigation: it has severely restricted exploitability of many bugs and killed some exploit techniques. For example, exploits in the past have used a kernel execute primitive in order to build a kernel read/write primitive (see e.g. ziVA); that is no longer possible on A12 without bypassing PAC first. Furthermore, extensive use of PAC-protected pointers in IOKit has made it significantly harder to turn many bugs into useful primitives. Given the long history of serious security issues in IOKit, this is a substantial win.

zone_require - iOS 13

zone_require is a software mitigation introduced in iOS 13 that adds checks that certain pointers are allocated from the expected `zalloc` zones before using them. The most common zone_require checks in the iOS kernelcache are of Mach ports; for example, every time an `ipc_port` is locked, the `zone_require()` function is called to check that the allocation containing the Mach port resides in the `ipc.ports` zone (and not, for example, an `OSData` buffer allocated with `kalloc()`).

Since fake Mach ports are an integral part of modern techniques, zone_require has a substantial impact on exploitation. Vulnerabilities like CVE-2017-13861 (async_wake) that drop a reference on an `ipc_port` no longer offer a direct path to creating a fake port. While zone_require has been publicly bypassed once, the technique relied on an oversight in the implementation that is easy to correct.

	An entry was added for AppleAVE2Driver exploit - iOS 12.4.1. The description for PAN was updated to clarify that it was introduced with the A10 processor, not iOS 10. The description for PPL was updated to clarify that it primarily protects userspace processes, as the kernel's code is protected by KTRR.
2020/06/11	Original post published.

Posted by Tim at 12:42 PM

No comments:

Post a Comment

Enter your comment...

Comment as: Google Account

Publish **Preview**

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)