Jailbreaking iOS in the Post-Apocalyptic Age

coolstar & tihmstar

CStar_OW

coolstar

tihmstar

tihmstar

Ages of Jailbreaking

Siguza
@s1guza

Folge ich

Antwort an @s1guza @coolstarorg

Ages of jailbreaking (IMO):

iOS 1-4: Golden Age (BootROM)
iOS 5-9: Industrial Age (rise of userland)
iOS 10-*: Post-Apocalyptic (KTRR)

12:58 - 13. Okt. 2017

Jailbreak in a nutshell

- Exploit kernel -> (get unstable kernel write)
- Get stable kernel read/write
  - Make it available to other processes
- Privilege escalation (get ability to spawn process as root)
  - Escape sandbox
  - Become root
- Bypass codesign enforcement
- System-wide code injection
- Optional: read/write root filesystem

Jailbreak in a nutshell

- ~~Exploit kernel -> (get unstable kernel write)~~
- Get stable kernel read/write
    - Make it available to other processes
- Privilege escalation (get ability to spawn process as root)
    - Escape sandbox
    - Become root
- Bypass codesign enforcement
- System-wide code injection
- Optional: read/write root filesystem

Get stable kernel read/write

And make it persistent

task_for_pid

Explanation

- Mach syscall
- Grants task port for an arbitrary process
- If one owns the task port, they own the process
  - read/write memory, control threads...
- pid 0 = kernel_task

KRW: ≤iOS 8 and ≤iPhone6

- patch kernel to allow task_for_pid(0)
- call api from any process
- use task port for kernel read/write

Bypass

```
872  kern_return_t
873  task_for_pid(
874          struct task_for_pid_args *args)
875  {
876          mach_port_name_t        target_tport = args->target_tport;
877          int                     pid = args->pid;
878          user_addr_t             task_addr = args->t;
879          proc_t                  p = PROC_NULL;
880          task_t                  t1 = TASK_NULL;
881          task_t                  task = TASK_NULL;
882          mach_port_name_t        tret = MACH_PORT_NULL;
883          ipc_port_t              tfpport = MACH_PORT_NULL;
884          void                    * sright = NULL;
885          int                     error = 0;
886          boolean_t               is_current_proc = FALSE;
887          struct proc_ident       pident = {0};
888
889          AUDIT_MACH_SYSCALL_ENTER(AUE_TASKFORPID);
             AUDIT_ARG(pid, pid);
             AUDIT_ARG(mach_port1, target_tport);

893          /* Always check if pid == 0 */
894          if (pid == 0) {
895                  (void) copyout((char *)&tret, task_addr, sizeof(mach_port_name_t));
896                  AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
897                  return KERN_FAILURE;
898          }
899
900          t1 = port_name_to_task(target_tport);
901          if (t1 == TASK_NULL) {
902                  (void) copyout((char *)&tret, task_addr, sizeof(mach_port_name_t));
903                  AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
904                  return KERN_FAILURE;
905          }
906
```

Patch

iOS 9 & iPhone5s: KPP

Explanation

- **K**ernel **P**atch **P**rotection prevents kernel from being patched (kernel text & const data segment)
- KPP runs in EL3
    - Kernel can no longer be patched
- KPP bypass possible
    - Kernel can be patched again

More info *Jailbreaking iOS from Past to Present*: https://www.youtube.com/watch?v=t01tbbjJHbs

KRW: ≤iOS 10.2.1 and ≤iPhone6s

Bypass

```
872  kern_return_t
873  task_for_pid(
874          struct task_for_pid_args *args)
875  {
876          mach_port_name_t        target_tport = args->target_tport;
877          int                     pid = args->pid;
878          user_addr_t             task_addr = args->t;
879          proc_t                  p = PROC_NULL;
880          task_t                  t1 = TASK_NULL;
881          task_t                  task = TASK_NULL;
882          mach_port_name_t        tret = MACH_PORT_NULL;
883          ipc_port_t              tfpport = MACH_PORT_NULL;
884          void                    * sright = NULL;
885          int                     error = 0;
886          boolean_t               is_current_proc = FALSE;
887          struct proc_ident       pident = {0};
888
889          AUDIT_MACH_SYSCALL_ENTER(AUE_TASKFORPID);
             AUDIT_ARG(pid, pid);
             AUDIT_ARG(mach_port1, target_tport);

893          /* Always check if pid == 0 */
894          if (pid == 0) {
895                  (void) copyout((char *)&tret, task_addr, sizeof(mach_port_name_t));
896                  AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
897                  return KERN_FAILURE;
898          }
899
900          t1 = port_name_to_task(target_tport);
901          if (t1 == TASK_NULL) {
902                  (void) copyout((char *)&tret, task_addr, sizeof(mach_port_name_t));
903                  AUDIT_MACH_SYSCALL_EXIT(KERN_FAILURE);
904                  return KERN_FAILURE;
905          }
906
```

Patch

iPhone 7: KTRR

Explanation

```
518  #if defined(KERNEL_INTEGRITY_KTRR) || defined(KERNEL_INTEGRITY_CTRR)
519          /* KTRR
520           *
521           * Lock physical KTRR region. KTRR region is read-only. Memory outside
522           * the region is not executable at EL1.
523           */
524
525          rorgn_lockdown();
526  #endif /* defined(KERNEL_INTEGRITY_KTRR) || defined(KERNEL_INTEGRITY_CTRR) */
527
528
529  #endif /* CONFIG_KERNEL_INTEGRITY */
```

- Hardware mitigation in iPhone 7
- Replaces old style KPP
- Memory controller locks down kernel pages
- Kernel text and const data marked as read-only region
- Kernel can't execute code outside read-only region

More info *Jailbreaking iOS from Past to Present*: https://www.youtube.com/watch?v=t01tbbjJHbs

host_special_ports

- XNU provides some special ports for userland
- Userland allows setting additional ports
- Allows communication with kernel and system daemons over mach ports for special purposes

Explanation

```
74  /*
75   * Always provided by kernel (cannot be set
76   */
77  #define HOST_PORT                          1
78  #define HOST_PRIV_PORT                     2
79  #define HOST_IO_MASTER_PORT                3
80  #define HOST_MAX_SPECIAL_KERNEL_PORT       7 /* room to grow */
81
82  #define HOST_LAST_SPECIAL_KERNEL_PORT      HOST_IO_MASTER_PORT
83
84  /*
85   * Not provided by kernel
86   */
87  #define HOST_DYNAMIC_PAGER_PORT           (1 + HOST_MAX_SPECIAL_KERNEL_PORT)
88  #define HOST_AUDIT_CONTROL_PORT           (2 + HOST_MAX_SPECIAL_KERNEL_PORT)
89  #define HOST_USER_NOTIFICATION_PORT       (3 + HOST_MAX_SPECIAL_KERNEL_PORT)
90  #define HOST_AUTOMOUNTD_PORT              (4 + HOST_MAX_SPECIAL_KERNEL_PORT)
91  #define HOST_LOCKD_PORT                   (5 + HOST_MAX_SPECIAL_KERNEL_PORT)
92  #define HOST_KTRACE_BACKGROUND_PORT       (6 + HOST_MAX_SPECIAL_KERNEL_PORT)
93  #define HOST_SEATBELT_PORT                (7 + HOST_MAX_SPECIAL_KERNEL_PORT)
94  #define HOST_KEXTD_PORT                   (8 + HOST_MAX_SPECIAL_KERNEL_PORT)
95  #define HOST_LAUNCHCTL_PORT               (9 + HOST_MAX_SPECIAL_KERNEL_PORT)
96  #define HOST_UNFREED_PORT                 (10 + HOST_MAX_SPECIAL_KERNEL_PORT)
97  #define HOST_AMFID_PORT                   (11 + HOST_MAX_SPECIAL_KERNEL_PORT)
98  #define HOST_GSSD_PORT                    (12 + HOST_MAX_SPECIAL_KERNEL_PORT)
99  #define HOST_TELEMETRY_PORT               (13 + HOST_MAX_SPECIAL_KERNEL_PORT)
100 #define HOST_ATM_NOTIFICATION_PORT        (14 + HOST_MAX_SPECIAL_KERNEL_PORT)
101 #define HOST_COALITION_PORT               (15 + HOST_MAX_SPECIAL_KERNEL_PORT)
102 #define HOST_SYSDIAGNOSE_PORT             (16 + HOST_MAX_SPECIAL_KERNEL_PORT)
103 #define HOST_XPC_EXCEPTION_PORT           (17 + HOST_MAX_SPECIAL_KERNEL_PORT)
104 #define HOST_CONTAINERD_PORT              (18 + HOST_MAX_SPECIAL_KERNEL_PORT)
105 #define HOST_NODE_PORT                    (19 + HOST_MAX_SPECIAL_KERNEL_PORT)
106 #define HOST_RESOURCE_NOTIFY_PORT         (20 + HOST_MAX_SPECIAL_KERNEL_PORT)
107 #define HOST_CLOSURED_PORT                (21 + HOST_MAX_SPECIAL_KERNEL_PORT)
108 #define HOST_SYSPOLICYD_PORT              (22 + HOST_MAX_SPECIAL_KERNEL_PORT)
109 #define HOST_FILECOORDINATIOND_PORT       (23 + HOST_MAX_SPECIAL_KERNEL_PORT)
110 #define HOST_FAIRPLAYD_PORT               (24 + HOST_MAX_SPECIAL_KERNEL_PORT)
111 #define HOST_IOCOMPRESSIONSTATS_PORT      (25 + HOST_MAX_SPECIAL_KERNEL_PORT)
112
113 #define HOST_MAX_SPECIAL_PORT             HOST_IOCOMPRESSIONSTATS_PORT
```

KRW: ≤iOS 10.2.1 and ≤iPhone7

Bypass

- Get kernel task through exploit
- Write kernel task to host special port 4

```
const int offsetof_host_special = 0x10;
uint64_t host_priv_kaddr = find_port(mach_host_self());
uint64_t realhost_kaddr = rk64(host_priv_kaddr + offsetof_ip_kobject);
wk64(realhost_kaddr + offsetof_host_special + 4 * sizeof(void*), port_kaddr);
```

- Userland code now calls host_get_special_port(4)
- Equivalent to task_for_pid(0)

```
task_t kernel_task = MACH_PORT_NULL;
ret = host_get_special_port(realhost, HOST_LOCAL_NODE, 4, &kernel_task);
LOG("kernel_task: %x, %s", kernel_task, mach_error_string(ret));
```

iOS 10.3: pointer check

- mach_vm_* APIs added pointer check
- Deny using kernel task from userland
- Kernel task is useless now :(
  - or is it?

Explanation

ipc_tt.c (xnu-3789.41.3 vs. xnu-3789.51.2)

ipc_tt.c - /Users/molt/Documents/dev/ios/src/kernel/xnu/xnu-3789.41.3/osfmk/kern

convert_port_to_task_with_exec_token()

```
}

/*
 * Routine:    convert_port_to_task_with_exec_token
 * Purpose:
 *      Convert from a port to a task and return
 *      the exec token stored in the task.
 *      Doesn't consume the port ref; produces a task ref,
 *      which may be null.
 * Conditions:
 *      Nothing locked.
 */
task_t
convert_port_to_task_with_exec_token(
    ipc_port_t       port,
    uint32_t         *exec_token)
{
    task_t      task = TASK_NULL;

    if (IP_VALID(port)) {
        ip_lock(port);

        if (    ip_active(port)                 &&
                ip_kotype(port) == IKOT_TASK        ) {
            task = (task_t)port->ip_kobject;
            assert(task != TASK_NULL);

            if (exec_token) {
                *exec_token = task->exec_token;
            }
            task_reference_internal(task);
        }

        ip_unlock(port);
    }

    return (task);
}

/*
 * Routine:    convert_port_to_task_name
 * Purpose:
 *      Convert from a port to a task name.
```

ipc_tt.c - /Users/molt/Documents/dev/ios/sr

convert_port_to_task_with_exec_token()

```
}

/*
 * Routine:    convert_port_to_task_with_exec_token
 * Purpose:
 *      Convert from a port to a task and return
 *      the exec token stored in the task.
 *      Doesn't consume the port ref; produces a task ref,
 *      which may be null.
 * Conditions:
 *      Nothing locked.
 */
task_t
convert_port_to_task_with_exec_token(
    ipc_port_t       port,
    uint32_t         *exec_token)
{
    task_t      task = TASK_NULL;

    if (IP_VALID(port)) {
        ip_lock(port);

        if (    ip_active(port)                 &&
                ip_kotype(port) == IKOT_TASK        ) {
            task = (task_t)port->ip_kobject;
            assert(task != TASK_NULL);

            if (task == kernel_task && current_task() != kernel_task) {
                ip_unlock(port);
                return TASK_NULL;
            }

            if (exec_token) {
                *exec_token = task->exec_token;
            }
            task_reference_internal(task);
        }

        ip_unlock(port);
    }

    return (task);
}
```

status: 13 differences

KRW: ≤iOS 12.5.5

Bypass

- Remapping task structure in kernel memory bypasses check
- Write remapped kernel task to host special port 4
- use mach_vm_* APIs for kernel read/write

```
ret = mach_vm_remap(km_fake_task_port,
                    &remapped_task_addr,
                    sizeof_task,
                    0,
                    VM_FLAGS_ANYWHERE | VM_FLAGS_RETURN_DATA_ADDR,
                    zm_fake_task_port,
                    kernel_task_kaddr,
                    0,
                    &cur, &max,
                    VM_INHERIT_NONE);


if (ret != KERN_SUCCESS) {
    printf("[remap_kernel_task] remap failed: 0x%x (%s)\n", ret, mach_error_string(ret));
    return 1;
}

if (kernel_task_kaddr == remapped_task_addr) {
    printf("[remap_kernel_task] remap failure: addr is the same after remap\n");
    return 1;
}

DEBUGLOG("[remap_kernel_task] remapped successfully to 0x%llx\n", remapped_task_addr);
```

iOS 13: zone_require

Explanation

- Kernel allocations are split in zones
- Different allocation types go to their dedicated zone
- Task structures need to be in a certain allocation zone
    - Different from what mach_vm_* APIs allocate to
- Access to task in wrong zone causes kernel panic

KRW: ≤iOS 13.7

```
private func make_fake_task(vm_map: UInt64) -> UInt64 {
    var corpse_task = mach_port_t()
    task_generate_corpse(mach_task_self_, &corpse_task)

    let corpse_task_port = electra.findPort(port: corpse_task)
    let fake_task = rk64(corpse_task_port + offsets.ipc_port.ip_kobject)
    wk32(fake_task + offsets.task.ref_count, 99) //leak refrerences
    mach_port_destroy(mach_task_self_, corpse_task)

    wk64(fake_task + offsets.task.vm_map, vm_map)
    wk32(fake_task + offsets.task.message_app_suspended, 1)
    wk32(fake_task + offsets.task.active, 1)

    return fake_task
}
```

- Bypass 1: ≤iOS 13.5
  - Alloc kernel memory
  - Copy kernel task
  - Modify zone type
- Bypass 2:
  - Create corpse task (barebones task struct in kernel)
  - Assign kernel map to corpse
  - Mark corpse task as active

Bypass

- Write fake task to host special port 4
- use mach_vm_* APIs for kernel read/write

Pointer **A**uthentication **C**odes

Explanation

- ARMv8.3 hardware extension
- Message-Authentication-Codes for pointers
- Protects **data-in-memory** in relation to **context** with a **secret-key**
  - Return value, stack pointer, function pointers, vtables, data pointers
  - Structure contents (by *hashing* values and signing the hash)
  - Context also contains structure address & type info
    - Prevents reuse and type confusion

iOS 14 & iPhone Xs: PAC (and more)

Explanation

- PAC protects task, host, port structures
- PAC prevents calling remap functions in kernel
- **P**age **P**rotection **L**ayer protects writing to kernel map
- Pointer checks against kernel map
- zone_require extended to pmap

KernelRW iOS 14-15.1.1-?

Bypass

- Init needs:
    - Early kernel read & one 8-byte kernel write
    - Address of current task structure
- Allocates
    - Two mach ports (read_port, write_port)
    - IOSurface object with a surface
- Retrieves address of ports and surface location in surface clients array
- Use write to replace surface in array with address of read_port.ip_context

KernelRW iOS 14-15.1.1-?

Bypass

- kread32:
  - Set context of read_port to readaddr
  - Call IOConnectCallMethod 8 on surface to read 4 bytes

```
uint32_t KernelRW::kread32(uint64_t where){
    kern_return_t kr = KERN_SUCCESS;
    uint64_t i_scalar[1] = {
        _IOSurface_id_write //fixed, first valid client obj
    };
    uint64_t o_scalar[1];
    uint32_t i_count = 1;
    uint32_t o_count = 1;

    std::unique_lock<std::mutex> ul(_rw_lock);
    retassure(!(kr = mach_port_set_context(mach_task_self(), _context_read_port, where-off_read_deref)),
    kr = IOConnectCallMethod(
                             _IOSurfaceRootUserClient,
                             8, // s_get_ycbcrmatrix
                             i_scalar, i_count,
                             NULL, 0,
                             o_scalar, &o_count,
                             NULL, NULL);
    retassure(!kr, "kread32 failed with error=0x%08x", kr);
    return (uint32_t)o_scalar[0];
}
```

KernelRW iOS 14-15.1.1-?

Bypass

- kwrite64:
  - Set context of read_port to address of write_port
  - Set context of write_port to writeaddr
  - Call IOConnectCallMethod 33 on surface to write 8 bytes

```
void KernelRW::kwrite64(uint64_t where, uint64_t what){
    kern_return_t kr = KERN_SUCCESS;
    uint64_t i_scalar[3] = {
        _IOSurface_id_write, // fixed, first valid client obj
        0, // index
        what, // value
    };
    uint32_t i_count = 3;
    std::unique_lock<std::mutex> ul(_rw_lock);
    retassure(!(kr = mach_port_set_context(mach_task_self(), _context_read_port, _context_write_context_addr-off_write_deref)),
    retassure(!(kr = mach_port_set_context(mach_task_self(), _context_write_port, where)), "Failed to set context with error=0x
    kr = IOConnectCallMethod(
            _IOSurfaceRootUserClient,
            33, // s_set_indexed_timestamp
            i_scalar, i_count,
            NULL, 0,
            NULL, NULL,
            NULL, NULL);
    retassure(!kr, "kwrite64 failed with error=0x%08x", kr);
}
```

KernelRW iOS 14-15.1.1-?

Bypass

- Cleanup before process exit:
  - Restore surface address in clients array (single 8-byte write)
- Handoff:
  - Receive read/write_ports and surface from other process via mach ports
  - Retrieve kernel address and perform initial setup write
  - Transfer original surface address back to other process (for cleanup)

KernelRW iOS 14-15.1.1-?

Bypass

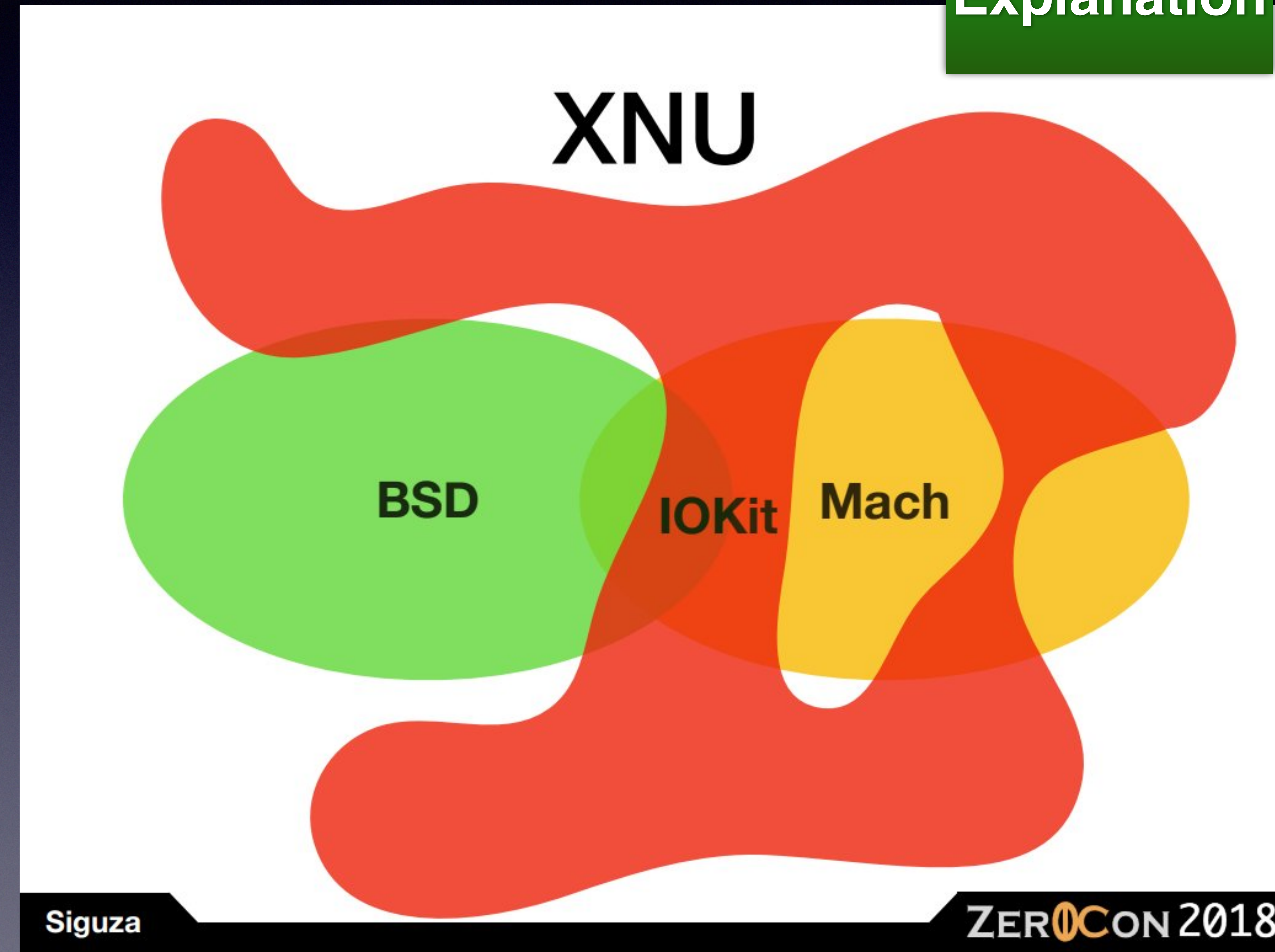
- Primitive needs to be *passed around*, not *persistent* on its own (dies on process exit)
- Jailbreak eventually passes KernelRW to launchd
- launchd holds onto the raw primitives
- Other processes can talk to launchd for kernel read/write (via libKernRW)

Jailbreak in a nutshell

- ~~Exploit kernel -> (get unstable kernel write)~~
- ~~Get stable kernel read/write~~
    - ~~Make it available to other processes~~
- Privilege escalation (get ability to spawn process as root)
    - Escape sandbox
    - Become root
- Bypass codesign enforcement
- System-wide code injection
- Optional: read/write root filesystem

XNU

- Kernel consists of
  - BSD part
  - Mach part
  - IOKit part
    which glues things together



BSD task proc

Explanation

- Each task has a BSD proc structure in kernel
- Proc structure manages resources and permissions of a process
- Each BSD proc structure has a ucred structure

BSD/posix ucred

Explanation

- Process credentials
- Manage user accounts for processes
- Contains:
  - uid (user id), gid (group id)
  - MACF label structure
    (for AMFI & sandbox)

Mandatory Access Control Framework

Explanation

- Introduced in FreeBSD
- Hooks across the kernel allow restricting permissions through policy modules despite being root
- Enforced by the Kernel
- AMFI (Apple Mobile File Integrity) and sandbox register MACF policy hooks

Privilege escalation

Priv: ≤iOS 10 (Sandbox & Root)

Bypass

- Set our proc's ucred pointer to the kernel's ucred
- Sandbox has a hardcoded check to not enforce on kernel ucred
- Having kernel ucred already grants root permissions
- Technically works up to iOS 13, but crippled in iOS 11

```
r = KCALL(OFF(copyin), &kern_ucred, self_proc + off->proc_ucred, sizeof(kern_ucred), 0, 0, 0, 0);
LOG("copyin: %s", errstr(r));
if(r != 0 || !self_ucred)
{
    goto out;
}
// Note: decreasing the refcount on the old cred causes a panic with "cred reference underflow",
LOG("stole the kernel's credentials");
setuid(0); // update host port

int newuid = getuid();
LOG("uid: %u", newuid);
```

iOS 11: "shenanigans"

Explanation

- Stealing kernel ucred without being the kernel may cause Sandbox to panic with "Shenanigans"

```
panic(cpu 0 caller 0xfffffff00a18b574): "shenanigans!"@/BuildRoot/Library/Caches/com.apple.xbs/Sources/Sandbox_executables/
Debugger message: panic
Memory ID: 0x6
OS version: 15B202
Kernel version: Darwin Kernel Version 17.2.0: Fri Sep 29 18:14:51 PDT 2017; root:xnu-4570.20.62~4/RELEASE_ARM64_T7000
```

Priv: ≤iOS 13 (Sandbox)

Bypass

- We don't necessarily need kernel ucred
- Don't copy kernel ucred, but ucred.cr_label to escape sandbox
- We're now unsandboxed but still mobile user

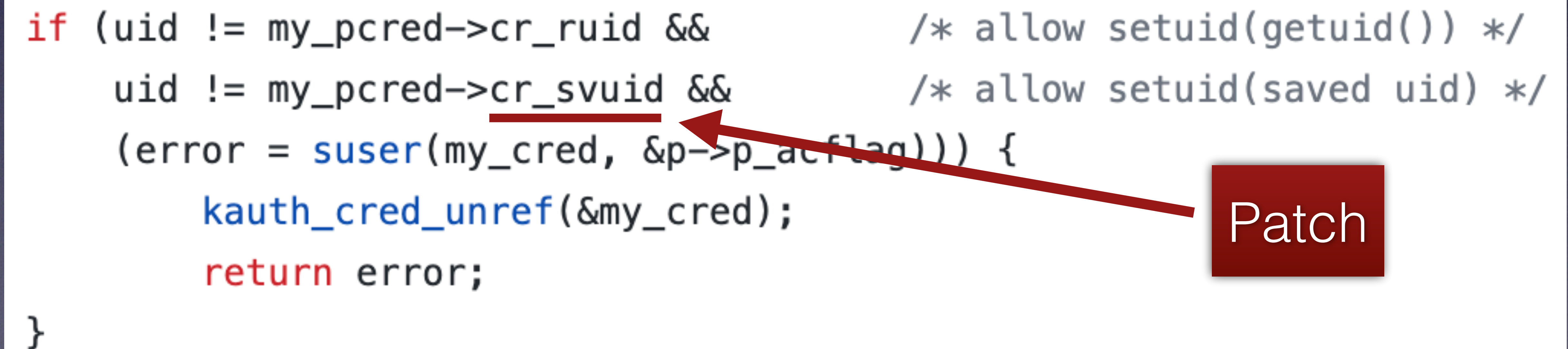
Copy from kernel ucred

```
/*
 * In-kernel credential structure.
 *
 * Note that this structure should not be used outside the kernel, nor shoulc
 * it or copies of it be exported outside.
 */
struct ucred {
        TAILQ_ENTRY(ucred)      cr_link; /* never modify this without KAUTH_C
        u_long  cr_ref;                 /* reference count */

struct posix_cred {
        /*
         * The credential hash depends on everything from this point on
         * (see kauth_cred_get_hashkey)
         */
        uid_t   cr_uid;                 /* effective user id */
        uid_t   cr_ruid;                /* real user id */
        uid_t   cr_svuid;               /* saved user id */
        short   cr_ngroups;             /* number of groups in advisory list
        gid_t   cr_groups[NGROUPS];     /* advisory group list */
        gid_t   cr_rgid;                /* real group id */
        gid_t   cr_svgid;               /* saved group id */
        uid_t   cr_gmuid;               /* UID for group membership purposes
        int     cr_flags;               /* flags on credential */
} cr_posix;
        struct label    *cr_label;      /* MAC label */
        /*
         * NOTE: If anything else (besides the flags)
         * added after the label, you must change
         * kauth_cred_find().
         */
        struct au_session cr_audit;             /* user auditing data */
};
```

Priv: ≤iOS 15.1.1 (Get Root)

Bypass

- Once we're out of sandbox we can call setuid(0)
- setuid has a check to allow if ruid, uid, or svuid match
- Patching ruid causes problems as ucreds may be reused and may elevate random processes
- We can patch svuid to 0
- Call setuid(0) twice (update uid and euid on a new ucred)



iOS 14: Data PAC

- Data pointers & struct members are now protected by PAC

Explanation

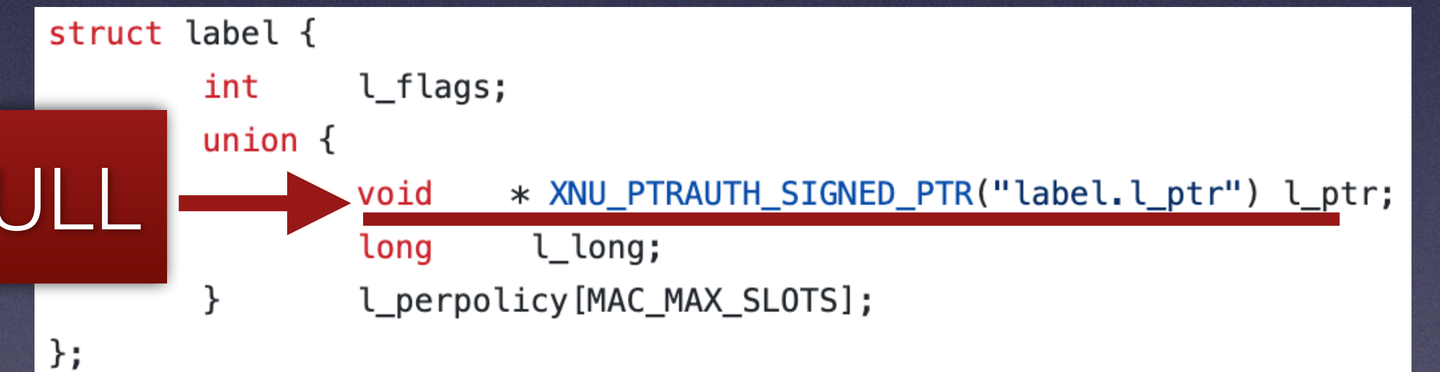
```
/*
 * In-kernel credential structure.
 *
 * Note that this structure should not be used outside the kernel, nor
 * it or copies of it be exported outside.
 */
struct ucred {
	LIST_ENTRY(ucred)	cr_link; /* never modify this without
#if defined(__STDC_VERSION__) && __STDC_VERSION__ >= 201112L && !defined(__STDC_NO_ATOMICS__)
	_Atomic u_long		cr_ref;  /* reference count */
#elif defined(__cplusplus) && __cplusplus >= 201103L
	_Atomic u_long		cr_ref;  /* reference count */
#else
	volatile u_long		cr_ref;  /* reference count */
#endif

	struct posix_cred {
		/*
		 * The credential hash depends on everything from this point on
		 * (see kauth_cred_get_hashkey)
		 */
		uid_t	cr_uid;		/* effective user id */
		uid_t	cr_ruid;	/* real user id */
		uid_t	cr_svuid;	/* saved user id */
		u_short	cr_ngroups;	/* number of groups in advisory list */
#if XNU_KERNEL_PRIVATE
		u_short	__cr_padding;
#endif
		gid_t	cr_groups[NGROUPS];/* advisory group list */
		gid_t	cr_rgid;	/* real group id */
		gid_t	cr_svgid;	/* saved group id */
		uid_t	cr_gmuid;	/* UID for group membership purposes */
		int	cr_flags;	/* flags on credential */
	} cr_posix;
	struct label	* OS_PTRAUTH_SIGNED_PTR_AUTH_NULL("ucred.cr_label") cr_label;	/* MAC label */

	/*
	 * NOTE: If anything else (besides the flags)
	 * added after the label, you must change
	 * kauth_cred_find().
	 */
	struct au_session cr_audit;		/* user auditing data */
};
```

Protected

Priv: ≤iOS 14 (Sandbox)

Bypass

- ucred.cr_label protected by PAC
- label contains l_ptr array with pointers to MACF policies
    - AMFI is policy 0, sandbox is policy 1
- iOS 14 allows setting PAC'd pointers to NULL
- Setting sandbox pointer to NULL escapes sandbox, similar to iOS 13



iOS 15: More data PAC

Explanation

- They protected *just* this pointer against NULL-ing*
- Upper bits on a NULL pointer now require a PAC signature

```
struct label {
        int     l_flags;
        union {
                void    * XNU_PTRAUTH_SIGNED_PTR_AUTH_NULL("label.l_ptr") l_ptr;
                long      l_long;
        }       l_perpolicy[MAC_MAX_SLOTS];
};
```

Protected against NULL

* ok they also did protect the label itself, but that couldn't be NULL'd anyways

eta 2023

iOS 15 sandbox bypass ?

Jailbreak in a nutshell

- ~~Exploit kernel -> (get unstable kernel write)~~
- ~~Get stable kernel read/write~~
    - ~~Make it available to other processes~~
- ~~Privilege escalation (get ability to spawn process as root)~~
    - ~~Escape sandbox~~
    - ~~Become root~~
- Bypass codesign enforcement
- System-wide code injection
- Optional: read/write root filesystem

Bypass codesign enforcement

AMFI: ≤iPhone6s

- Patch kernel (AMFI.kext) to treat every binary as being in trustcache
  - Minor differences in patching between ≤iOS 11 and ≥iOS 12, but same idea
- KTRR on ≥iPhone7 prevents kernel from being patched :(

AMFI: ≤iPhone X (or with PAC bypass)

Explanation

- AMFI contains Trustcaches (static & dynamic)
- Static trustcaches for binaries built-in to iOS
- Dynamic trustcaches for binaries shipped with Xcode
- Calling kernel function allows loading new trustcache to mark set of binaries as trusted
- Used on Electra & Chimera jailbreaks (for iOS 11 & 12) for jailbreak-bundled binaries
- Requires PAC bypass on iPhone XS or newer
- Only usable for a limited number of trustcaches (will run out of Kernel Memory)

AMFI: ≤iPhone X (theory)

Bypass

- Load large dynamic trustcache with placeholder hashes
- jailbreakd computes hashes before each binary runs
  - If hash not already in kernel memory, write to trustcache placeholder slots

AMFI: ≥iPhone Xs with PAC bypass (theory)

Bypass

- ≥iPhone Xs has kernel functions for load/**unload** trustcache
- Load first dynamic trustcache with jailbreak base binaries
- jailbreakd computes hashes before each binary runs
  - jailbreakd loads trustcache for the binary
  - jailbreakd unloads trustcache after the binary runs
  - Codesignature is cached for future runs in vnode

AMFI: ≤iOS 11 and ≤iPhone X

Bypass

- AMFI calls amfid in userspace for binaries not in in trustcache as long as they contain any signature
- amfid calls **MISValidateCodeSignatureAndCopyInfo**
- We can load a dylib into amfid to patch the function to return 0 and compute a CDHash
  - We put dylib in trustcache (write to kernel memory)
- amfid returns that the binary is trusted
- Binary runs
- Deployed in mach_portal, triple_fetch, liberiOS, and Electra jailbreaks

iOS 12: CoreTrust

Explanation

- CoreTrust.kext added to kernel
- AMFI calls CoreTrust for binaries not in trustcache before going to amfid
- CoreTrust requires valid signature that chains back to Apple
  - amfid doesn't get called if CoreTrust verification fails
- amfid verifies certificate expiry and provisioning profiles

iPhone XS: **P**age **P**rotection **L**ayer

Explanation

- Hardware mitigation introduced with iPhone XS
- Protects certain data segments & **p**age **map**s in the kernel
- Only __PPLTEXT section of the kernel can write to protected regions
- Must call trampoline functions to change CPU state and enter PPL
  - Like microkernel with syscalls in kernel (both in EL1)
  - PAC prevents attacker calling said functions

iPhone XS: pmap_cs

Explanation

- Is part of PPL
- Holds the trustcache & validated code signature blobs
- Distinguished binary trust levels
  - TL1 - App Store / Sideloaded (anything allowed by amfid)
  - TL2 & 3 - Trustcaches (Xcode Developer Image & iOS built-in binaries)
- TL1 libraries can't be loaded to higher trust binaries
  - Prevents loading 3rd-party dylib into amfid

Codesign vnode cache

Explanation

- AMFI gets called when **ubc_cs_blob_add** in the kernel tries loading a code signature for a binary
- Kernel maintains cache of csblobs on each vnode
  - vnode is the kernel representation of a file
  - csblob is the kernel representation of a code signature
- AMFI doesn't get called if vnode already has a code signature attached

AMFI: ≤iOS 12-? (with kexec)

Bypass

- jailbreakd gets called before a binary runs
- If signature wasn't loaded, write to its vnode cache
  (replicate ubc_cs_blob_add by calling kernel functions)
  - Patch code signature to add arbitrary entitlements before running binary
  - Call PPL trampoline to register code signature on ≥iPhone Xs
- Bypasses all of AMFI (including CoreTrust)
- Changing code signature of system binaries changes its hash
  - Demotes its trustlevel to allow injecting into it
- Deployed in Chimera jailbreak

AMFI: ≤iOS 14 (no kexec)

Bypass

- jailbreakd gets called before a binary runs
- jailbreakd signs it with a free (expired) developer certificate
  - Can add arbitrary entitlements
- Call fcntl syscall F_ADDSIGS
  - Loads code signature into kernel
  - Code signature passes CoreTrust verification (valid signature by allowed entity)
  - *If amfid checks pass, signature is attached to vnode*
- AMFI does not get called again when the binary runs
- Deployed in Odyssey & Taurine jailbreaks

AMFI: ≤ iOS 13 - Patching amfid

Bypass

- Don't load dylib into amfid
- Get task port for amfid (allows reading/writing to its memory)
- Register exception port to amfid (we are the debugger now)
  - Corrupt GOT pointer of MISValidateCodeSignatureAndCopyInfo
  - amfid crashes next time it's called
  - Catch exception message and read binary file name from cpu registers
  - Manually write CDHash to memory
  - Continue program flow as if validation passed

AMFI: ≤ iOS 13 getting amfid task port

Bypass

- Use task_for_pid() to get amfid task port
- AMFI requires either:
  - Local process **task_for_pid-allow** entitlement
  - Target process **get-task-allow** entitlement
- ucred.cr_label contains AMFI slot with entitlements
- Steal label of arbitrary process with correct entitlement (e.g. /bin/ps)

```
/*
 * In-kernel credential structure.
 *
 * Note that this structure should not be used outside the kernel, nor should
 * it or copies of it be exported outside.
 */
struct ucred {
	TAILQ_ENTRY(ucred)	cr_link; /* never modify this without KAUTH_C
	u_long	cr_ref;			/* reference count */

struct posix_cred {
	/*
	 * The credential hash depends on everything from this point on
	 * (see kauth_cred_get_hashkey)
	 */
	uid_t	cr_uid;			/* effective user id */
	uid_t	cr_ruid;		/* real user id */
	uid_t	cr_svuid;		/* saved user id */
	short	cr_ngroups;		/* number of groups in advisory list
	gid_t	cr_groups[NGROUPS];	/* advisory group list */
	gid_t	cr_rgid;		/* real group id */
	gid_t	cr_svgid;		/* saved group id */
	uid_t	cr_gmuid;		/* UID for group membership purposes
	int	cr_flags;		/* flags on credential */
} cr_posix;
	struct label	*cr_label;	/* MAC label */
	/*
	 * NOTE: If anything else (besides the flags)
	 * added after the label, you must change
	 * kauth_cred_find().
	 */
	struct au_session cr_audit;		/* user auditing data */
};
```

Copy from /bin/ps

AMFI: ≤iOS 14: getting amfid task port

- ucred.cr_label.l_ptr is OSDictionary from XML
  - Protected by PAC
  - Contains keys and values
    - Not protected by PAC
- Replace entitlements with the ones we need from other processes
  - Grab **task_for_pid-allow** from /bin/ps

Bypass

Protected

```
struct label {
        int       l_flags;
        union {
                void     * XNU_PTRAUTH_SIGNED_PTR("label.l_ptr") l_ptr;
                long       l_long;
        }       l_perpolicy[MAC_MAX_SLOTS];
};
```

Copy from /bin/ps

```
119  class OSDictionary : public OSCollection
120  {
121          friend class OSSerialize;
122
123          OSDeclareDefaultStructors(OSDictionary);
124
125  #if APPLE_KEXT_ALIGN_CONTAINERS
     ted:
             unsigned int    count;
             unsigned int    capacity;
130          nsigned int    capacityIncrement;
131          st ct dictEntry {
132                  OSTaggedPtr<const OSSymbol>        key;
133                  OSTaggedPtr<const OSMetaClassBase> value;
134  #if XNU_KERNEL_PRIVATE
135                  static int compare(const void *, const void *);
136  #endif
137          };
138          dictEntry    * OS_PTRAUTH_SIGNED_PTR("OSDictionary.dictionary") diction
139
140  #else /* APPLE_KEXT_ALIGN_CONTAINERS */
141
```

iOS 14: Userland PAC

Explanation

- Processes share PAC keys depending on origin
  - Platform binaries (except WebKit / iMessageBlastdoor)
  - Team ID
- Jailbreak app is not a platform binary
  - PAC keys don't match amfid
  - Can't sign pointers to manipulate process state (exception handling)

iOS 14: Userland PAC

Explanation

- Apple accidentally documented this for us

Managing PAC register state

xnu generally tries to avoid reprogramming the CPU's PAC-related registers on kernel entry and exit, since this could add significant overhead to a hot codepath. Instead, xnu uses the following strategies to manage the PAC register state.

A keys

Userspace processes' A keys (`AP{IA,DA,GA}Key`) are derived from the field `jop_pid` inside `struct task` . For implementation reasons, an exact duplicate of this field is cached in the corresponding `struct machine_thread` .

A keys are randomly generated at shared region initialization time (see "Signed pointers in shared regions" below) and copied into `jop_pid` during process activation. This shared region, and hence associated A keys, may be shared among arm64e processes under specific circumstances:

1. "System processes" (i.e., processes launched from first-party signed binaries on the iOS system image) generally use a common shared region with a default `jop_pid` value, separate from non-system processes.

   If a system process wishes to isolate its A keys even from other system processes, it may opt into a custom shared region using an entitlement in the form `com.apple.pac.shared_region_id=[...]` . That is, two processes with the entitlement `com.apple.pac.shared_region_id=foo` would share A keys and shared regions with each other, but not with other system processes.

2. Other arm64e processes automatically use the same shared region/A keys if their respective binaries are signed with the same team-identifier strings.

iOS 14: Userland PAC

Bypass

- Change **A key** a thread by overwriting it in the kernel

Copy from amfid

Managing PAC register state

xnu generally tries to avoid reprogramming the CPU's PAC-related registers on kernel entry and exit, since this could add significant overhead to a hot codepath. Instead, xnu uses the following strategies to manage the PAC register state.

A keys

Userspace processes' A keys (`AP{IA,DA,GA}Key`) are derived from the field `jop_pid` inside `struct task`. For implementation reasons, an exact duplicate of this field is cached in the corresponding `struct machine_thread`.

A keys are randomly generated at shared region initialization time (see "Signed pointers in shared regions" below) and copied into `jop_pid` during process activation. This shared region, and hence associated A keys, may be shared among arm64e processes under specific circumstances:

processes" (i.e., processes launched from first-party signed binaries on the iOS system image) generally use a shared region with a default `jop_pid` value, separate from non-system processes.

m process wishes to isolate its A keys even from other system processes, it may opt into a custom shared region using an entitlement in the form `com.apple.pac.shared_region_id=[...]`. That is, two processes with the entitlement `com.apple.pac.shared_region_id=foo` would share A keys and shared regions with each other, but not with other system processes.

2. Other arm64e processes automatically use the same shared region/A keys if their respective binaries are signed with the same team-identifier strings.

AMFI: iOS 14: Userland PAC Bypass

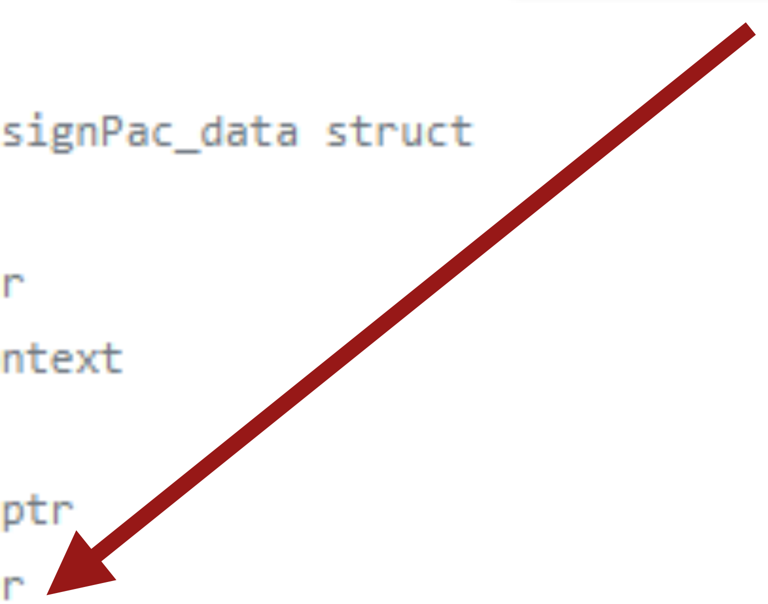
- Changing PAC keys of running thread will crash it if it calls C functions
- GOT pointers are signed with **A key**
- We control what runs on our thread
- Craft signing oracle in assembly without relying on libc

```
__attribute__((naked))
static void signPac_signingFunction(unsigned int *gadgetState,
                                    struct signPac_data **signPac_signPtrs,
                                unsigned int *signPac_signPtrCount,
                                    void *alwaysNull3,
                                    void *alwaysNull4,
                                    void *alwaysNull5,
                                    void *alwaysNull6,
                                    void *alwaysNull7){
    //can use x0 -> x7 for arguments
    //can use x9 -> x15 for temporary registers
    //copy x0 -> x2 to x9 -> x11

    //can use up to x15 safely without messing with the stack

    __asm__ volatile (
                 "mov x9, x0\n" //x0 - x2 get clobbered for sleep
                 "mov x10, x1\n"
                 "mov x11, x2\n"
```

AMFI: iOS 14: Userland PAC **Bypass**

- Changing PAC keys of running thread will crash it if it calls C functions
- GOT pointers are signed with **A key**
- We control what runs on our thread
- Craft signing oracle in assembly without relying on libc

```
//state 2 sign
"ldr x1, [x10, #0]\n" //x1 = signPac_data *data
"ldr w2, [x11]\n" //w2 = count

"adr x13, #0\n" //x13 is now a repeat pointer for signing
"add x13, x13, #12\n"

"mov x14, #0\n" //start sign loop

"cmp w14, w2\n" //check if we finished signing -- signing repeat pointer po
"b.ne #16\n"

"mov x0, #1\n" //set to state 1
"str w0, [x9, #0]\n"
"br x12\n" //we're done here

"lsl x0, x14, #4\n"
"add x0, x0, x1\n" //grab the signPac_data struct

"ldr x3, [x0, #0]\n" //grab ptr
"ldr x4, [x0, #8]\n" //grab context

"xpaci x3\n" //strip PAC from ptr
"pacia x3, x4\n" //sign pointer
"str x3, [x0, #0]\n"

"add w14, w14, #1\n"
"br x13"
```



AMFI: iOS 14: Userland PAC

Bypass

- Changing PAC keys of running thread will crash it if it calls C functions
- GOT pointers are signed with **A key**
- We control what runs on our thread
- Craft signing oracle in assembly without relying on libc
- Turns out there is a mach API for this o.O

```
//state 2 sign
"ldr x1, [x10, #0]\n" //x1 = signPac_data *data
"ldr w2, [x11]\n" //w2 = count

"adr x13, #0\n" //x13 is now a repeat pointer for signing
"add x13, x13, #12\n"

"mov x14, #0\n" //start sign loop

"cmp w14, w2\n" //check if we finished signing -- signing repeat pointer po
"b.ne #16\n"

"mov x0, #1\n" //set to state 1
"str w0, [x9, #0]\n"
"br x12\n" //we're done here

"lsl x0, x14, #4\n"
"add

"ldr
"ldr

"xpac
"paci
"str

"add
"br x
```

Signing oracle

YOUR SCIENTISTS WERE SO PREOCCUPIED WITH WHETHER OR NOT THEY COULD...

THEY DIDN'T STOP TO THINK IF THEY SHOULD.

AMFI: ≤iOS ~~13~~ 14 - Patching amfid

Bypass

- Don't load dylib into amfid
- Get task port for amfid (allows reading/writing to its memory)
- Register exception port to amfid (we are the debugger now)
  - Corrupt GOT pointer of MISValidateCodeSignatureAndCopyInfo
  - amfid crashes next time it's called
  - Catch exception message and read binary file name from cpu registers
  - Manually write CDHash to memory
  - Continue program flow as if validation passed

iOS 15: OSEntitlements

Explanation

- Switches from XML to DER entitlements
- Is backed by new OSEntitlements object in kernel
- OSEntitlements is closed source in AMFI.kext
- Protected by PAC

Entire blob is protected

```
 1 void *__fastcall OSEntitlements::queryEntitlementsFor(__int64 a1, const void *a2, __int64 a3, __int64 a4)
 2 {
 3   void *v8; // x23
 4
     OSEntitlements::adjustContext((OSEntitlements *)a1);
     v8 = &unk_461C8;
     IORWLockRead(*(IORWLock **)(a1 + 0x90));
     if ( *(_BYTE *)(a1 + 120) )
     {
       if ( a2 && memcmp((const void *)(a1 + 0x10), a2, 0x14uLL) )
11       OSEntitlements::queryEntitlementsFor();
12     ptrauth_utils_auth_blob_generic(a1 + 0x10, 0x60LL, 0xBD9DLL, 1LL, *(_QWORD *)(a1 + 0x70));
13     v8 = (void *)CEContextQuery(a1 + 0x28, a3, a4);
14   }
15   IORWLockUnlock(*(IORWLock **)(a1 + 0x90));
16   return v8;
17 }
```

Overview

Before you distribute an app, you apply a code signature to it. The signature certifies that you are the app's creator and enables the system to detect unwanted modifications — whether accidental or malicious — that happen after you sign your app. As a security measure, iOS refuses to launch an app that has a missing or invalid signature.

Starting in iOS 15, iPadOS 15, tvOS 15, and watchOS 8, the system checks for a new, more secure signature format that uses Distinguished Encoding Rules, or DER, to embed entitlements into your app's signature. Apps signed with a previous signature format will not launch.

iOS 15 AMFI bypass ?

eta 2023

12   ssh root@192.168.0.170 -p 2222

```
iPad:~ root# uname -a
Darwin iPad 21.0.0 Darwin Kernel Version 21.0.0: Sun Aug 15 20:55:56 PDT 2021; root:xnu-8019.12.5~1/RELEASE_ARM64_T8030 iPad12,1 arm Darwin
iPad:~ root# sw_vers
ProductName:    iPhone OS
ProductVersion: 15.0.1
BuildVersion:   19A348
iPad:~ root# apt --version
apt 2.5.0 (iphoneos-arm64)
iPad:~ root# dpkg -v
dpkg: error: unknown option -v

Type dpkg --help for help about installing and deinstalling packages [*];
Use 'apt' or 'aptitude' for user-friendly package management;
Type dpkg -Dhelp for a list of dpkg debug flag values;
Type dpkg --force-help for a list of forcing options;
Type dpkg-deb --help for help about manipulating *.deb files;

Options marked [*] produce a lot of output - pipe it through 'less' or 'more' !
iPad:~ root# ls /var/jb/
Applications/  Library/  System/  User@  bin/  boot/  cheyote/  dev/  etc/  lib/  mnt/  sbin/  tmp/  usr/  var/
iPad:~ root#
```

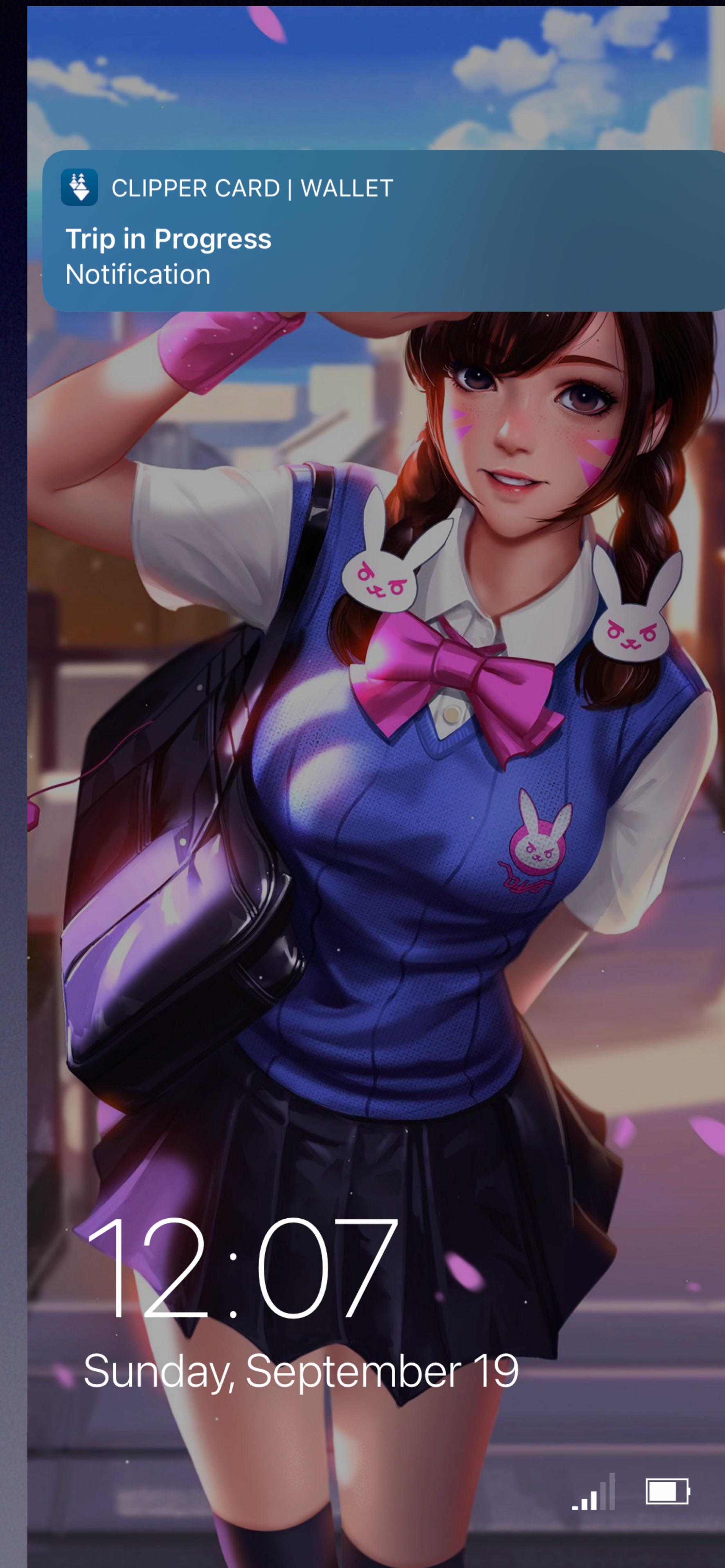
Jailbreak in a nutshell

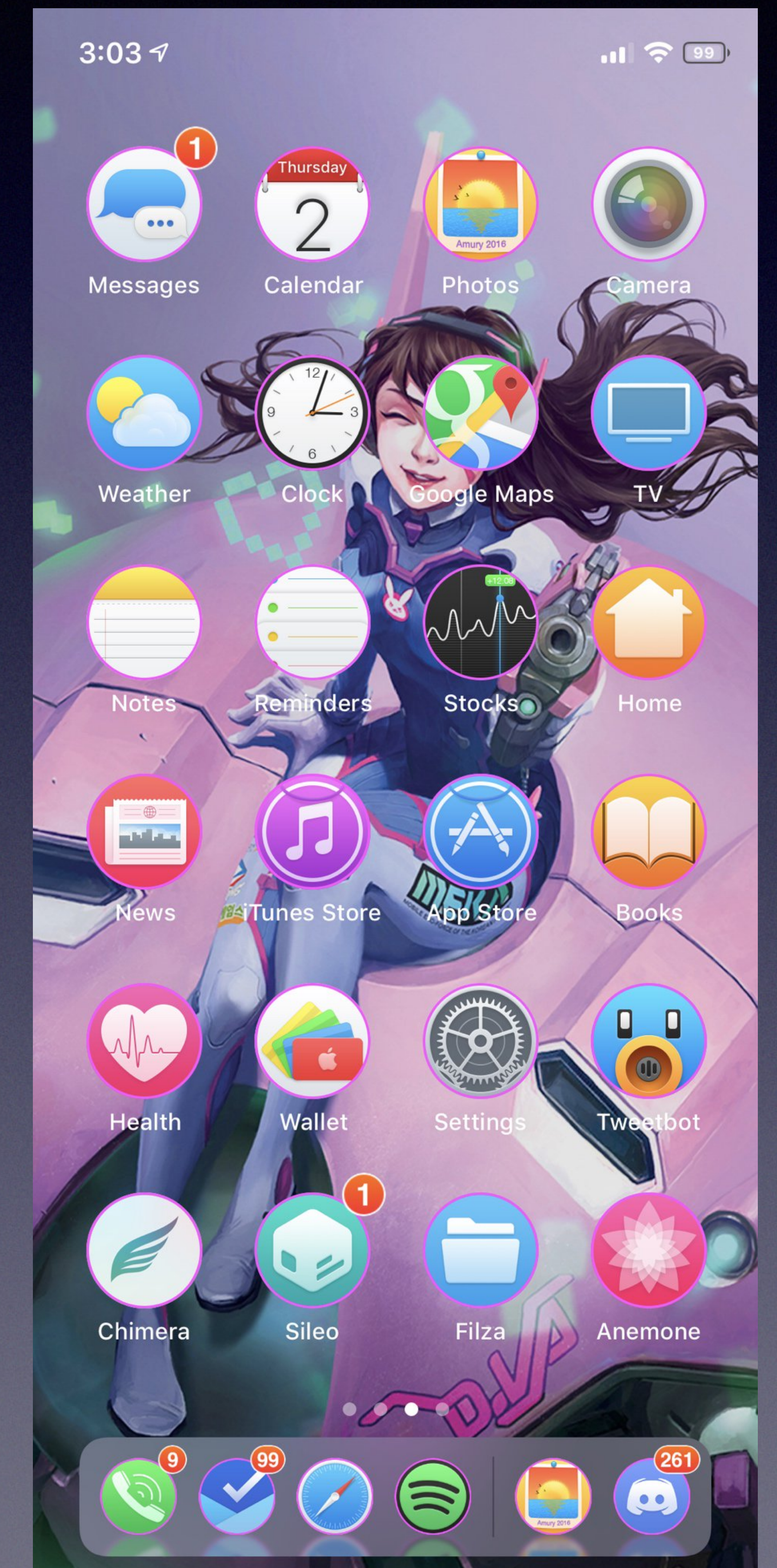
- ~~Exploit kernel -> (get unstable kernel write)~~
- ~~Get stable kernel read/write~~
  - ~~Make it available to other processes~~
- ~~Privilege escalation (get ability to spawn process as root)~~
  - ~~Escape sandbox~~
  - ~~Become root~~
- ~~Bypass codesign enforcement~~
- System-wide code injection
- Optional: read/write root filesystem

System-wide code injection

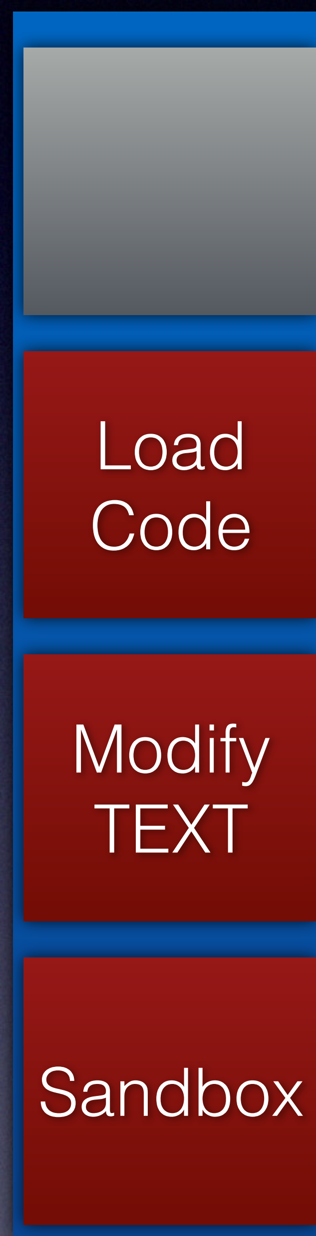
Why system-wide code injection?

- Allows users to install modifications (*tweaks*) to the system
- Endless customizations
- e.g. custom icons, custom home screens, more dock icons, etc.





Requirements for system-wide code injection



- Load custom code in process on loading
  - Must be able to modify TEXT segment
- Loosen sandbox restrictions to load assets and tweak preferences

≤iPhone 6S

Patching the kernel - ≤iPhone 6S

iPhone 6s

Load Code

Modify TEXT

Sandbox

- Patch sandbox MACF hooks
- Patch check in **vm_fault_***

Bypass

```
printf("Found sbops 0x%llx\n",sbops);

WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_access)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_chroot)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_create)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_deleteextattr)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_exchangedata)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_exec)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_getattrlist)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_getextattr)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_ioctl)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_link)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_listextattr)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_open)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_readlink)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_setattrlist)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_setextattr)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_setflags)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_setmode)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_setowner)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_setutimes)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_setutimes)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_stat)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_truncate)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_unlink)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_notify_create)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_fsgetpath)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_vnode_check_getattr)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_mount_check_stat)), 0);

WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_proc_check_fork)), 0);
WriteAnywhere64(NewPointer(sbops+offsetof(struct mac_policy_ops, mpo_iokit_check_get_property)), 0);
```

```
if (cs_bypass) {
        /* code-signing is bypassed */
        *cs_violation = FALSE;
} else if (VMP_CS_TAINTED(m, fault_page_size, fault_phys_offset)) {
```

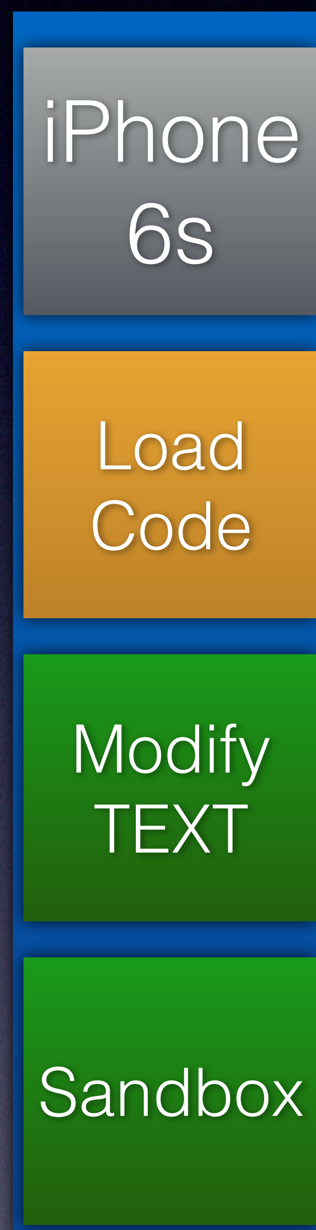
How do daemons get spawned on iOS

Explanation

- **launchd** (PID 1) is equivalent to **initd** or **systemd** on Unix systems
  - Calls posix_spawn to execute daemons
- Users tap apps on SpringBoard, but launchd executes them (similar to daemons)

Loading custom code into a process ≤iPhone 6S

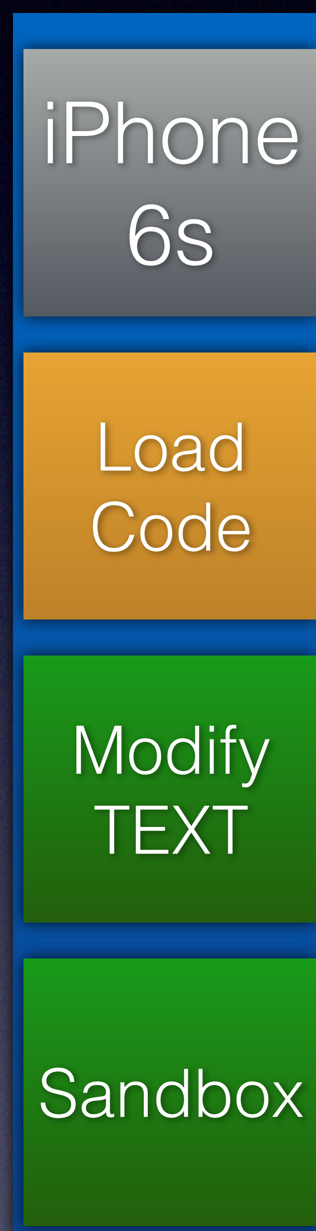
Explanation



- dyld is the dynamic linker in iOS, macOS, etc.
- dyld loads libraries specified by **DYLD_INSERT_LIBRARIES** env var on process launch
  - Requires **get-task-allow** entitlement ← via kernel patch

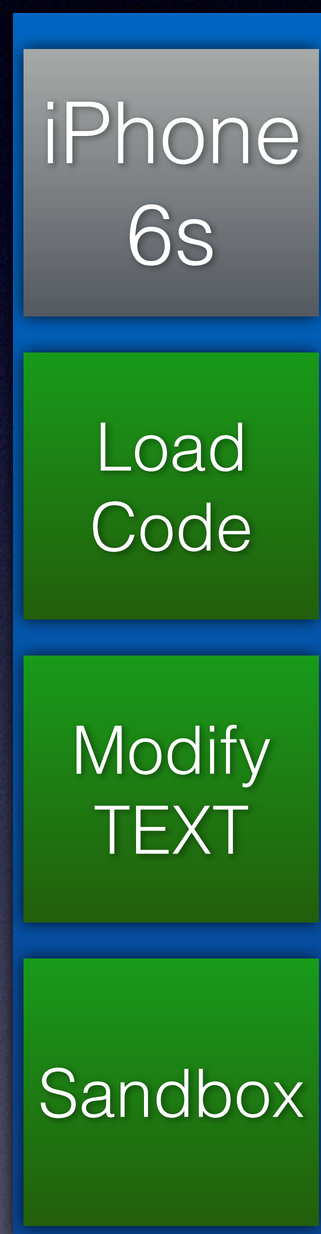
Setting environment variables in daemons on iOS ≤iPhone 6S

Bypass



- Load a dylib into launchd
    - Use task port
- Dylib hooks posix_spawn() and adds dyld env var
- dyld in the new process loads requested library

System-wide code injection ≤iPhone 6S



- Completed

≥iPhone 7 (kexec)

Setting environment variables in daemons on iOS (with kexec)

Bypass

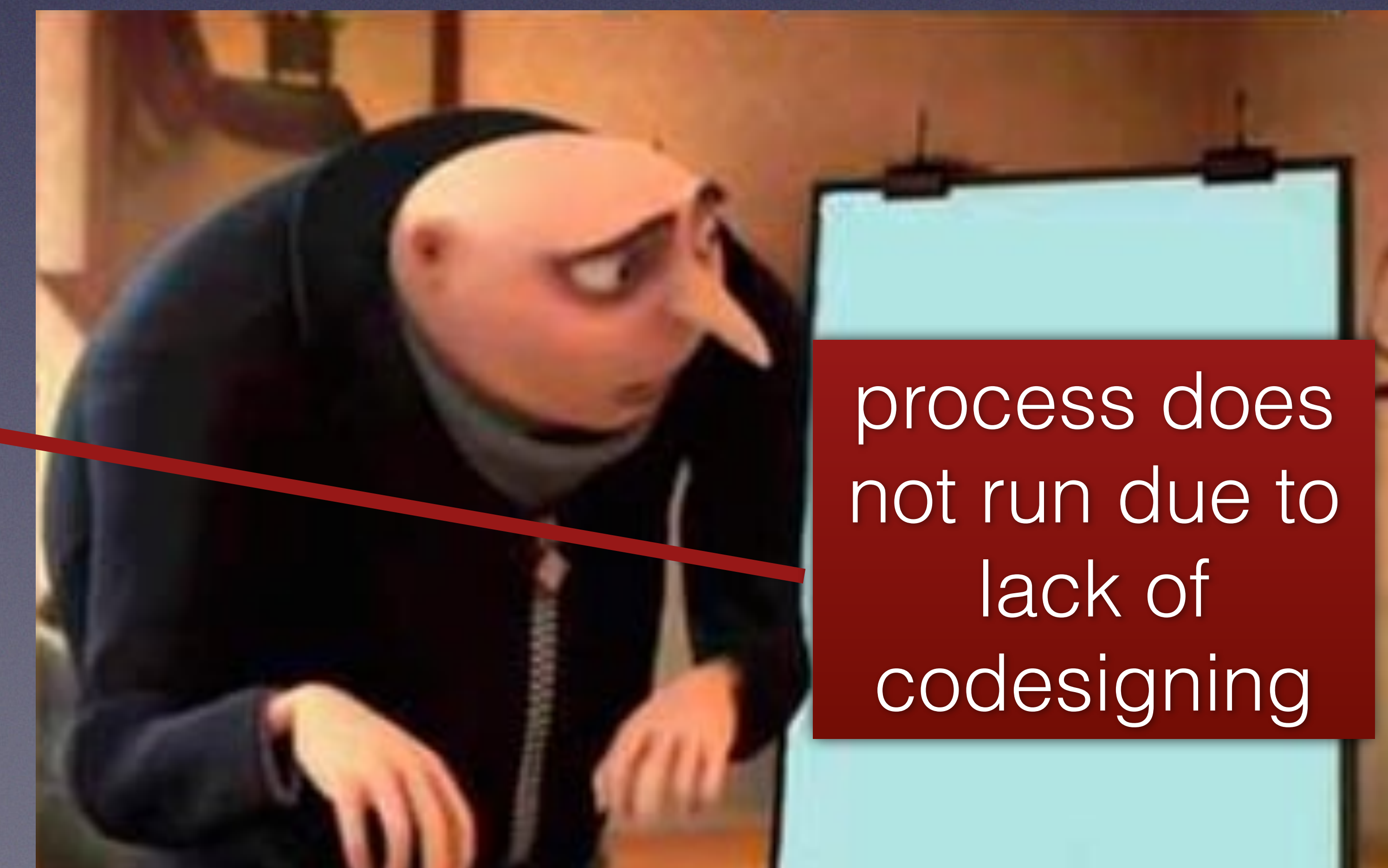
with kexec

Load Code

Modify TEXT

Sandbox

- ~~Load a dylib into launchd~~ ← load dylib into trustcache
    - Use task port
- Dylib hooks posix_spawn() to add dyld env var
- ~~dyld in the new process loads requested library~~



Problems ≥iPhone 7 (kexec)

Explanation

with kexec

Load Code

Modify TEXT

Sandbox

- CoreTrust prevents loading binaries without Apple authenticated code signature
  - Can't patch kernel (KTRR)
  - Bypass requires calling jailbreakd before spawning processes

Setting environment variables in daemons on iOS (with kexec)

with kexec

Load Code

Modify TEXT

Sandbox

Bypass

- Load a dylib into launchd ← load dylib into trustcache
  - Use task port
- Dylib hooks posix_spawn() to add dyld env var
  - Calls jailbreakd to load modified code signature before the binary runs
- dyld in the new process loads the requested library

What about other processes?

with kexec

Load Code

Modify TEXT

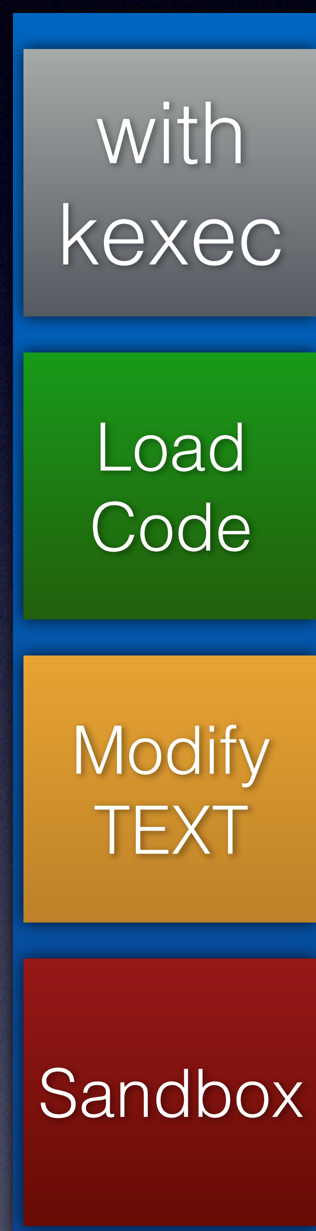
Sandbox

Bypass

- Other processes can call posix_spawn, fork+exec, system...
- Most functions wrap around either posix_spawn or exec
- exec can be hooked and redirected to posix_spawn with **POSIX_SPAWN_SETEXEC** attribute
- posix_spawn can be hooked similar to launchd by injected dylib
- Injection is now system-wide
(every process has the dylib injected)

Modifying TEXT in arbitrary processes ≥iPhone 7

Explanation



- Codesigning is still enforced despite loading custom code signatures
- Can't modify arbitrary process's TEXT
  - ..... or can we?

Modifying TEXT in arbitrary processes ≥iPhone 7

Explanation

with kexec

Load Code

Modify TEXT

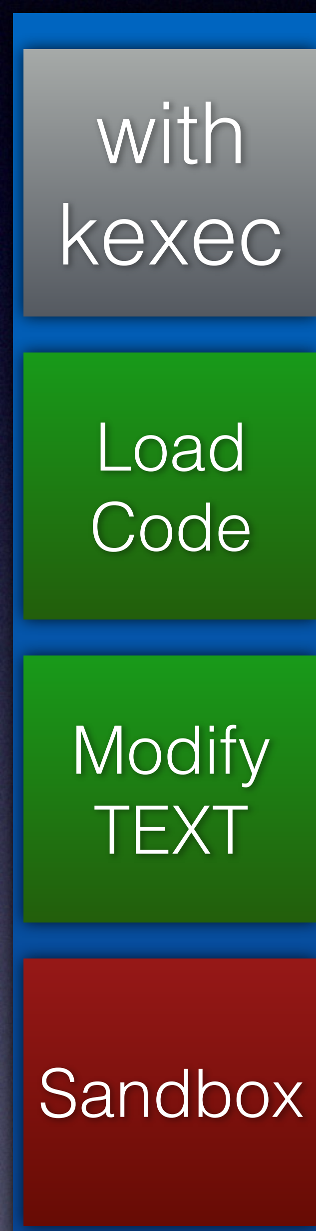
Sandbox

- Debugging through Xcode requires modifying TEXT for breakpoints from lldb
- Guarded by get-task-allow
  - But we have the entitlement, why does the process still crash?
- Process must be marked as debugged

```
#define CS_DEBUGGED            0x10000000  /* process is currently or has previously been debugged and allowed to run with invalid pages */
```

Modifying TEXT in arbitrary processes ≥iPhone 7

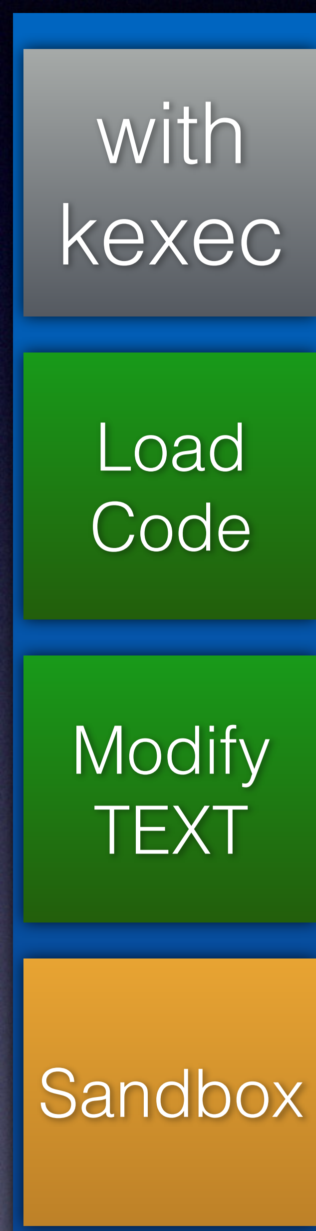
Bypass



- dylib loaded into all processes calls jailbreakd on launch
- jailbreakd adds **CS_DEBUGGED** flag in kernel
- Process can now patch TEXT

Loosening sandbox restrictions

Explanation



- Tweaks need to be able to read certain directories from app sandbox
- Sandbox supports adding extensions via a syscall if provided appropriate token
- Token can be generated outside of sandbox by calling **sandbox_extension_issue_file**
  - Can be passed via environment variable to our injected dylib from launchd
- Dylib calls **sandbox_extension_consume** with token
  - Access to additional directories granted
- **Supported API in iOS / macOS. Not a security vulnerability**

Setting environment variables in daemons on iOS (with kexec)

Bypass

with kexec

Load Code

Modify TEXT

Sandbox

- Load a dylib into launchd ← load dylib into trustcache
  - Use task port
- Dylib calls sandbox_extension_issue_file to get sandbox tokens
- Dylib hooks posix_spawn() to add dyld and sandbox env vars
  - Calls jailbreakd to load modified code signature before the binary runs
- dyld in the new process loads the requested library

≥iPhone XS (no kexec)

Setting environment variables in daemons on iOS (no kexec)

no kexec

Load Code

Modify TEXT

Sandbox

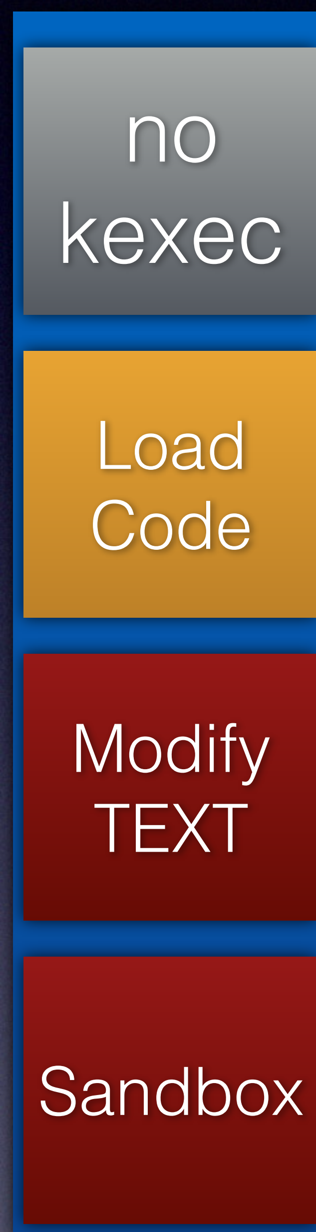
- Load a dylib into launchd
    - Use task port

dylib is not in trustcache

- Dylib calls sandbox_extension_issue_file to get sandbox tokens
- Dylib hooks posix_spawn() to add dyld and sandbox env vars
    - Calls jailbreakd to load modified code signature before the binary runs
- dyld in the new process loads the requested library

Problems ≥iPhone XS (no kexec)

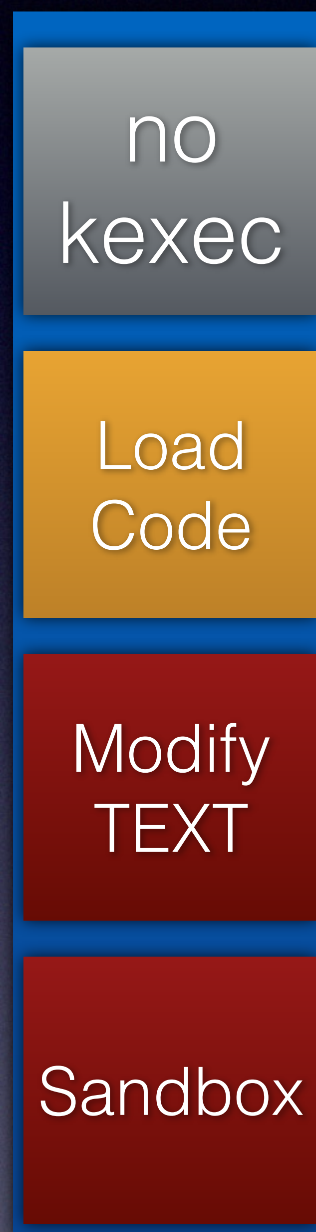
Explanation



- PAC prevents calling kernel functions
- PPL (pmap_cs) prevents TL1 binaries from injecting into TL 2 or 3 (launchd is TL3)

Recap: ≤iOS 14 - Patching amfid

Bypass



- Don't load dylib into amfid
- Get task port for amfid (allows reading/writing to its memory)
- Register exception port to amfid (we are the debugger now)
    - Corrupt GOT pointer of MISValidateCodeSignatureAndCopyInfo
    - amfid crashes next time it's called
    - Catch exception message and read binary file name from cpu registers
    - Manually write CDHash to memory
    - Continue program flow as if validation passed

Patching ~~amfid~~ launchd (Approach 1, no kexec)

Bypass

no kexec

Load Code

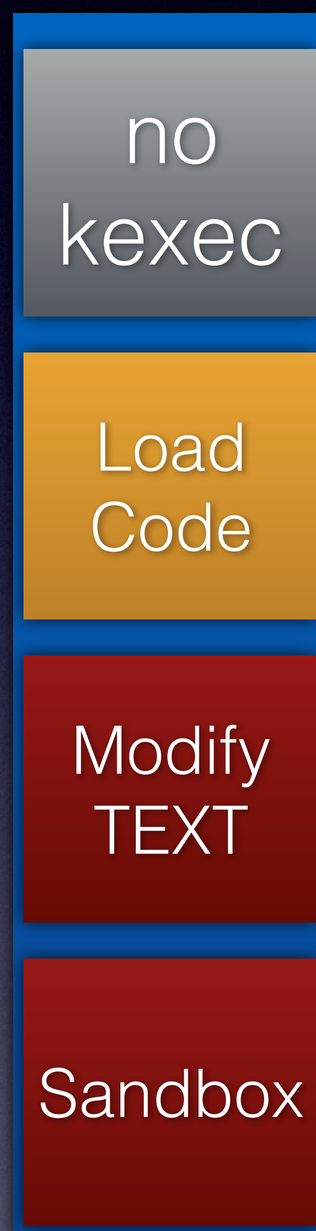
Modify TEXT

Sandbox

- Don't load dylib into ~~amfid~~ launchd
- Get task port for launchd (allows reading/writing to its memory)
- Register exception port to launchd we are the debugger now)
  - Corrupt GOT pointer of posix_spawn
  - launchd crashes next time it's called
  - Catch exception message and read binary file name from cpu registers
  - load code signatures and modify env vars
  - Continue program flow ~~as if validation passed~~

Pitfalls to Approach 1 (no kexec)

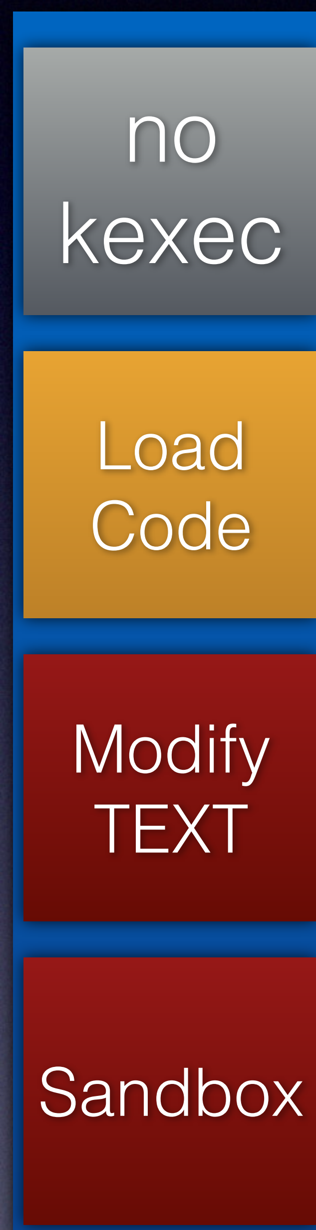
Explanation



- jailbreakd must remain alive to *debug* launchd
- jailbreakd crashing means launchd crashing
- launchd crashing means kernel panic
- iOS has a horrible habit of killing random non-launchd processes if the device is low on memory
- Not as stable as loading dylib into launchd
    - ... but can we?

Loading a dylib into launchd (no kexec)

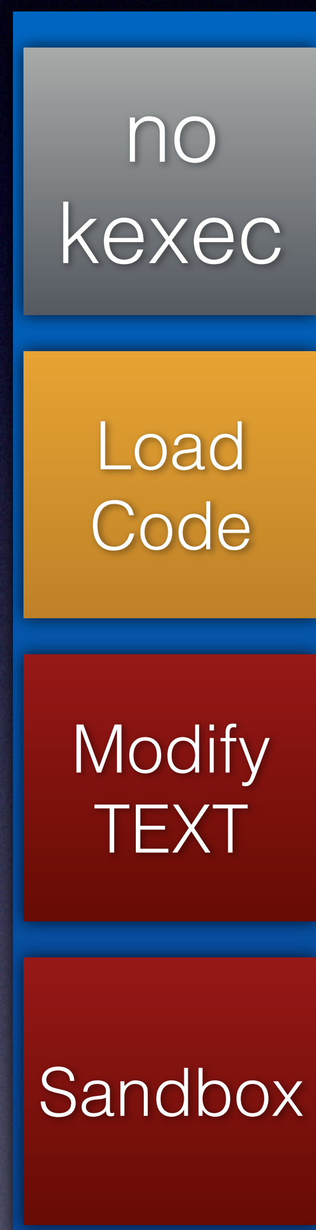
Explanation



- Need to somehow demote launchd's TL
- launchd TL is behind PPL
- jailbreakd can demote TL of newly spawned binaries by modifying its signature
- Need to somehow respawn launchd

Userspace reboots

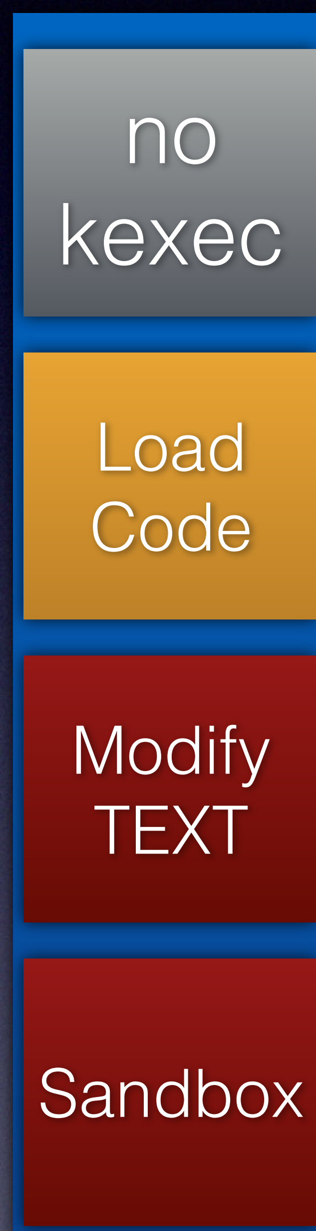
Explanation



- Introduced to launchd in iOS 9 / macOS 10.11
- "launchctl reboot userspace"
- Can run automatically overnight if iOS device is low on RAM
- Stops all daemons, launchd exec's itself, new launchd starts daemons up again

Hooking a userspace reboot (no kexec)

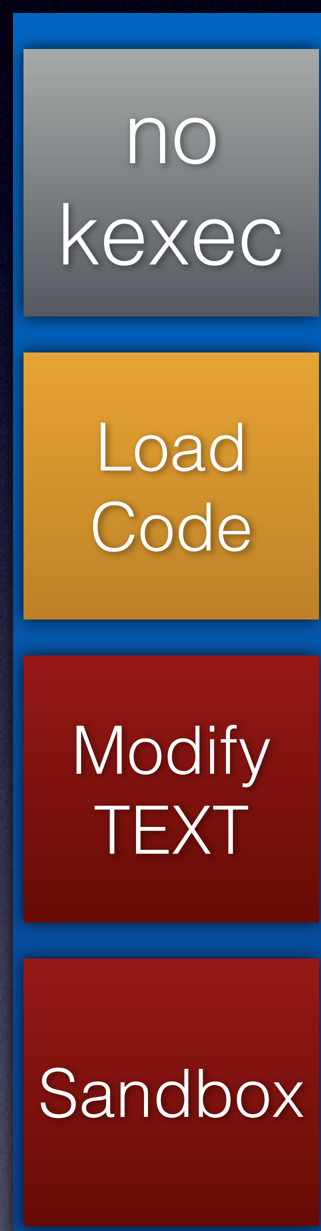
Explanation



- Can't inject dylib into launchd as its TL isn't demoted until userspace reboot
- Can't *debug* launchd as all daemons are dead during a userspace reboot
  - ....or can we?

Recap: Codesign vnode cache

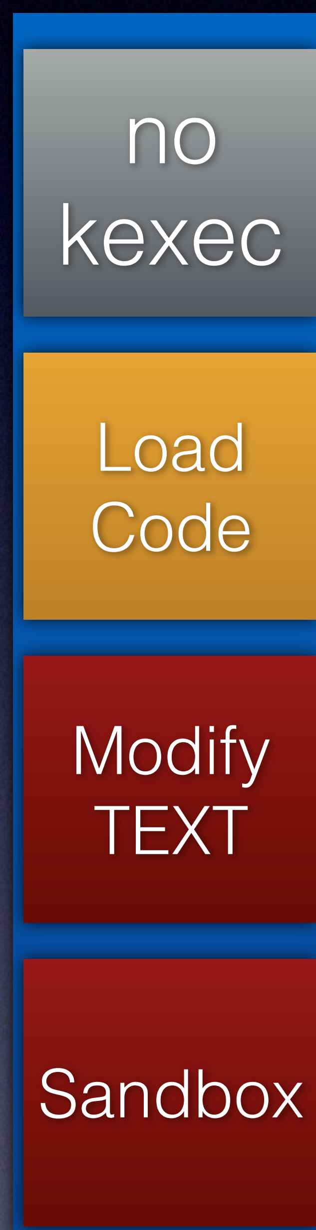
Explanation



- AMFI gets called when **ubc_cs_blob_add** in the kernel tries loading a code signature for a binary
- Kernel maintains cache of csblobs on each vnode
  - vnode is the kernel representation of a file
  - csblob is the kernel representation of a code signature
- AMFI doesn't get called if vnode already has a code signature attached

Hooking a userspace reboot (no kexec)

Bypass



- Cache code signature of jb binaries and launchd in vnode by calling jailbreakd
- Double-fork + exec to spawn 2nd jailbreakd instance
  - "detached" from launchd / initd -> not a daemon
  - Temporarily *debug* launchd to kickstart injection
- Ask launchd politely to userspace reboot (kills amfid & 1st jailbreakd instance)
- launchd exec's itself, detached jailbreakd injects dyld env var
- New launchd is demoted and loaded our dylib
- Detached jailbreakd (2nd instance) can now exit

Setting environment variables in daemons on iOS (no kexec, Approach 2)

Bypass

no kexec

Load Code

Modify TEXT

Sandbox

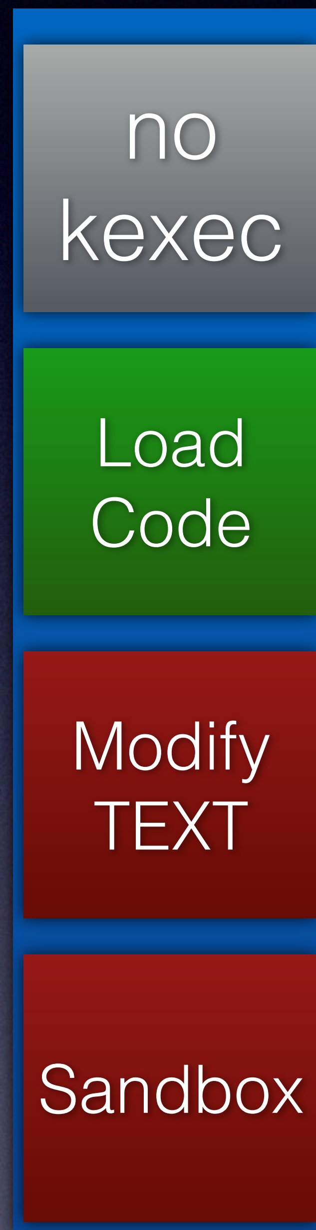
- Load a dylib into launchd
  - Demoted via userspace reboot

- Dylib calls sandbox_extension_issue_file to get sandbox tokens
- Dylib hooks posix_spawn() to add dyld and sandbox env vars
  - Calls jailbreakd to load modified code signature before the binary runs
- dyld in the new process loads the requested library

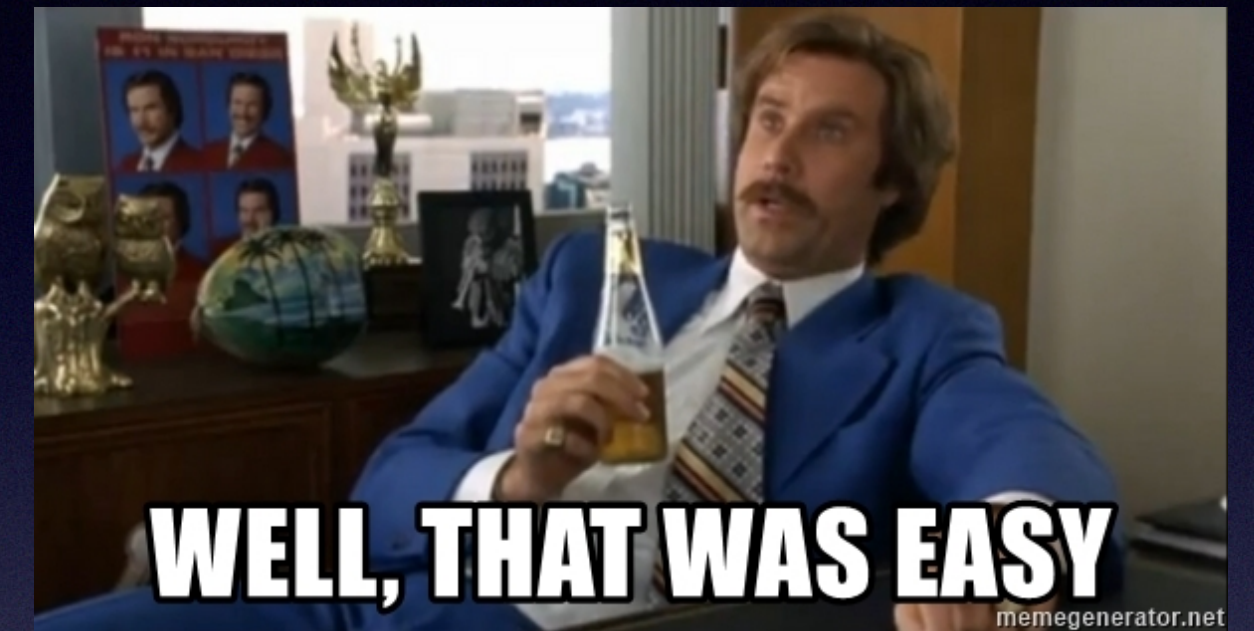
jailbreakd is not running yet

Setting environment variables in daemons on iOS (no kexec, Approach 2)

Bypass

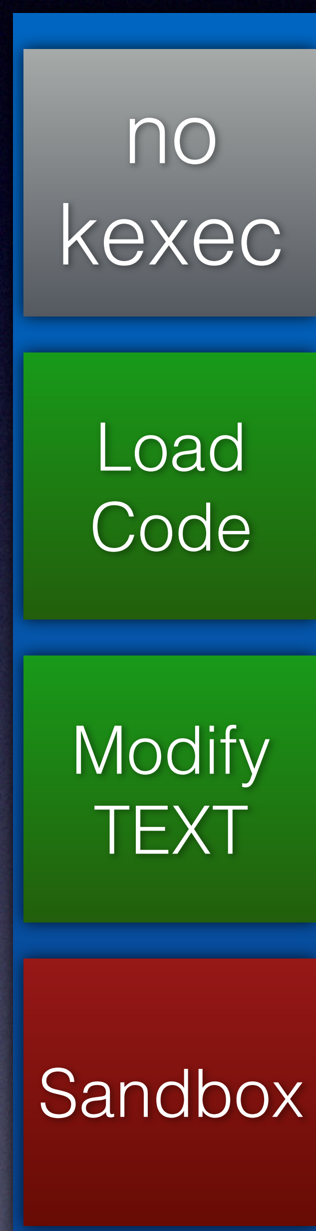


- Load a dylib into launchd
  - Demoted via userspace reboot
- Dylib calls sandbox_extension_issue_file to get sandbox tokens
- Dylib tells launchd to restart amfid, amfidebilitate and jailbreakd (kill codesigning again)
- Dylib hooks posix_spawn() to add dyld and sandbox env vars
  - Calls jailbreakd to load modified code signature before the binary runs
- dyld in the new process loads the requested library



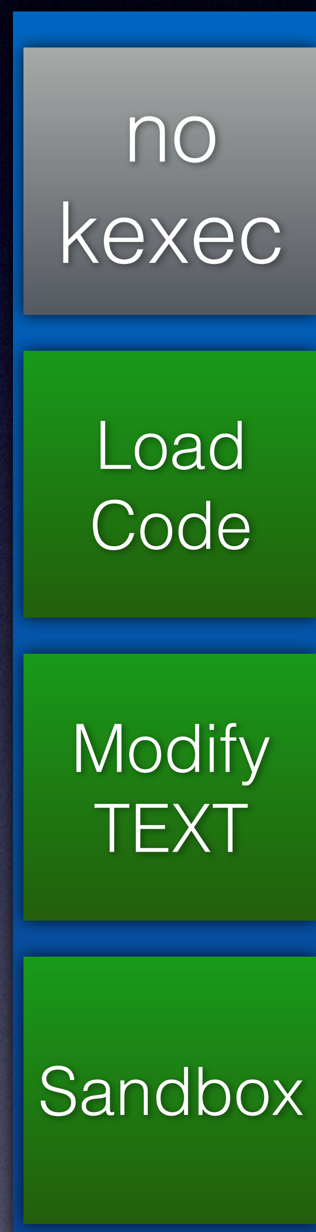
Modifying TEXT in arbitrary processes ≥iPhone XS (no kexec)

Bypass



- Dylib loaded into all processes calls jailbreakd on launch
- jailbreakd adds **CS_DEBUGGED** flag in kernel
- Process can now patch TEXT

Loosening sandbox restrictions

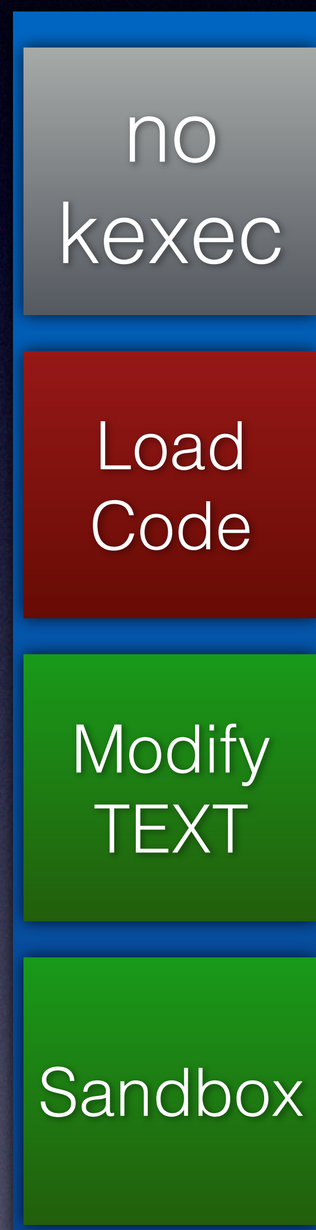


Explanation

- Tweaks need to be able to read certain directories from app sandbox
- Sandbox supports adding extensions via a syscall if provided appropriate token
- Token can be generated outside of sandbox by calling **sandbox_extension_issue_file**
  - Can be passed via environment variable to our injected dylib
- Dylib calls **sandbox_extension_consume** with token
  - Access to additional directories granted
- **Supported API in iOS / macOS. Not a security vulnerability**

KernelRW iOS 14-15.1.1-?

Explanation



- Primitive needs to be *passed around*, not *persistent* on its own (dies on process exit)

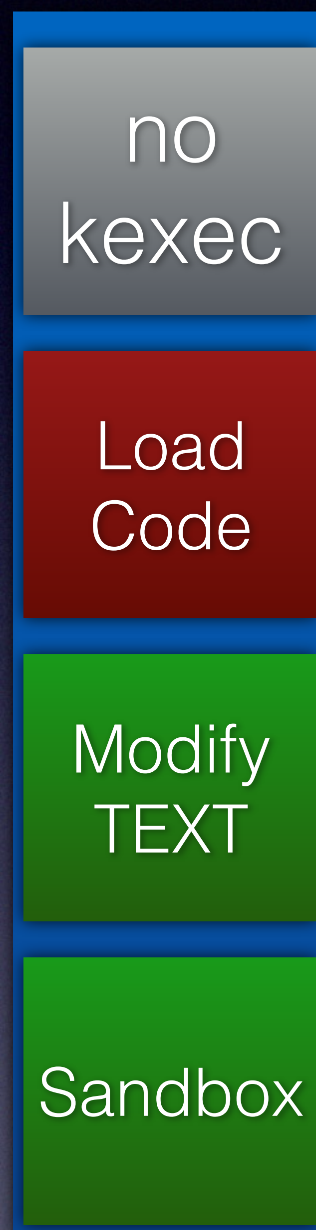
oops

- Jailbreak eventually passes KernelRW to launchd

still need to handle this. Lost during the userspace reboot

- launchd holds onto the raw primitives
- Other processes can talk to launchd for kernel read/write (via libKernRW)

Hooking a userspace reboot (no kexec)

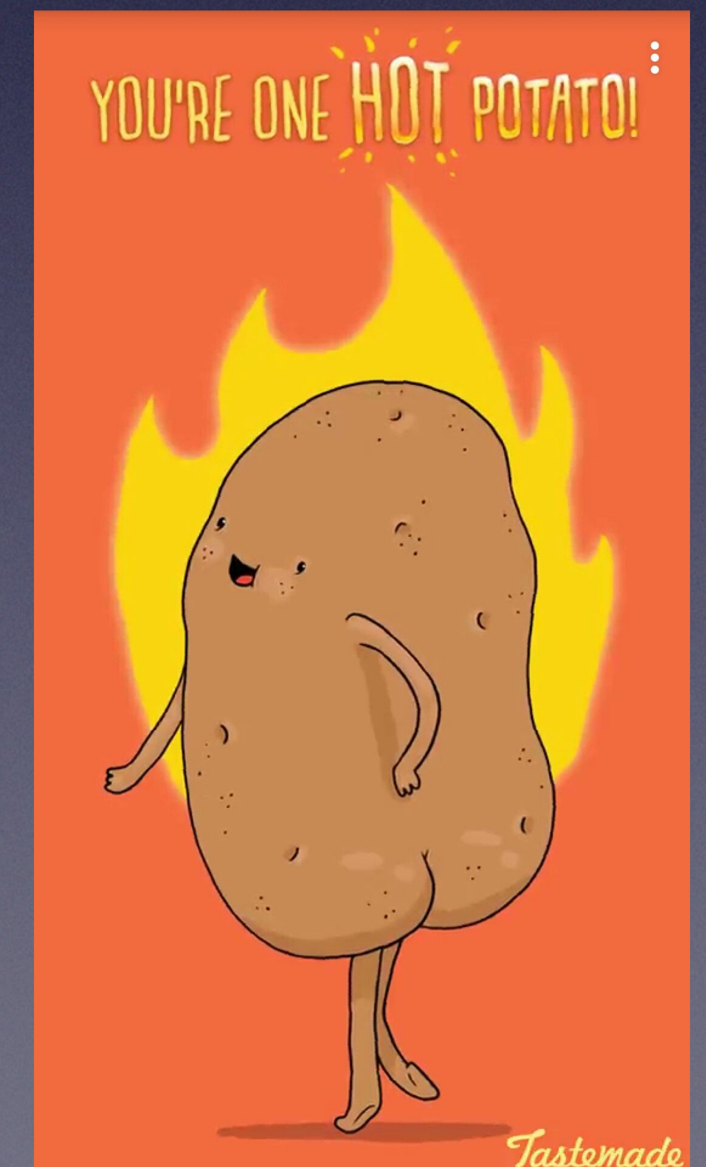


Bypass

- Cache code signature of jb binaries and launchd in vnode by calling jailbreakd
- Double-fork + exec to spawn 2nd jailbreakd instance
  - "detached" from launchd / initd -> not a daemon
  - Temporarily *debug* launchd to kickstart injection
- Ask launchd politely to userspace reboot (kills amfid & 1st jailbreakd instance)
- launchd exec's itself, detached jailbreakd injects dyld env var
- New launchd is demoted and loaded our dylib
- Detached jailbreakd (2nd instance) can now exit

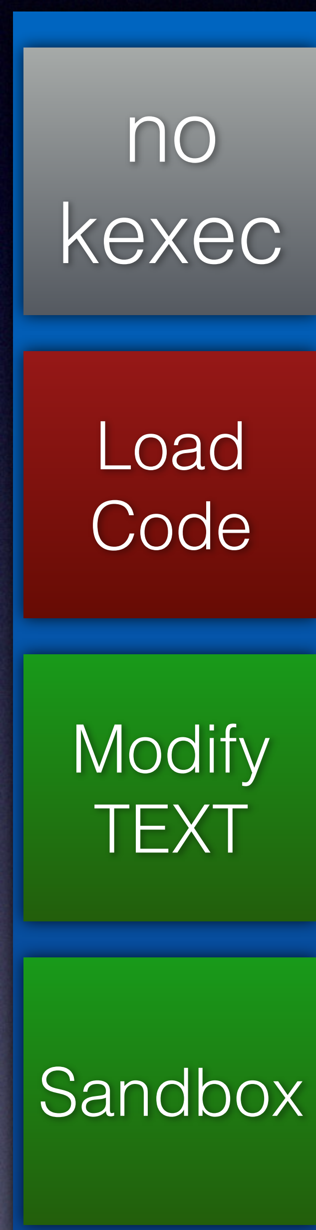
jailbreak passes KernelRW to detached jailbreakd

detached jailbreakd passes KernelRW to launchd



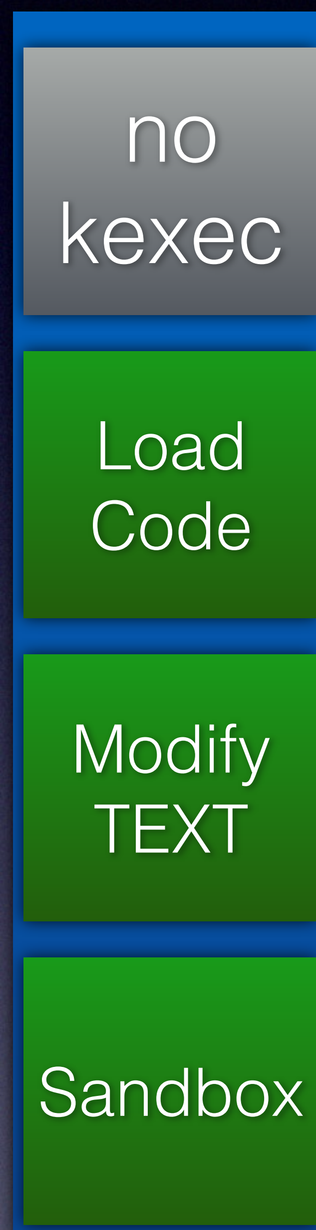
Persisting KernelRW

Explanation



- Jailbreak plays hot potato with KernelRW until it is passed to demoted launchd
- Demoted launchd can still userspace reboot afterwards
- jailbreakd can respawn
- How to persist KernelRW?

Persisting KernelRW



Bypass

- launchd holds onto KernelRW after the first userspace reboot
- amfidebilitate and jailbreakd (in daemon form) can talk to launchd to do kernel read/write
- launchd calls posix_spawn on next userspace reboot (with exec attribute)
- Pass KernelRW to temporary detached jailbreakd before launchd execs itself
  - KernelRW gets passed back to launchd once it relaunched

Jailbreak in a nutshell

- ~~Exploit kernel -> (get unstable kernel write)~~
- ~~Get stable kernel read/write~~
  - ~~Make it available to other processes~~
- ~~Privilege escalation (get ability to spawn process as root)~~
  - ~~Escape sandbox~~
  - ~~Become root~~
- ~~Bypass codesign enforcement~~
- ~~System-wide code injection~~
- Optional: read/write root filesystem

Questions?

Bonus: Read/Write rootFS ≤iOS 14

Explanation

- iOS ships with a read-only root filesystem
  - Could simply remount it as read/write prior to iOS 7
  - macOS 10.15 and higher also use a read-only root filesystem
- Jailbreak provides* a read/write root filesystem for users (and tweaks) to place files on

*until now? (as of iOS 14)

mnt: ≤iOS 11.2, ≤iPhone 6S

Bypass

- Patch kernel to allow mounting as read/write
- Call mount() to remount the root filesystem as read/write

mnt: ≤iOS 11.2 (all devices)

patch

Bypass

- mount has a check to prevent remounting ROOTFS
  - Temporarily unset MNT_ROOTFS flag in kernel
  - Call mount() to remount the root filesystem as read/write
  - Reset MNT_ROOTFS flag in kernel

```
        if ((vp->v_flag & VROOT) &&
            (vp->v_mount->mnt_flag & MNT_ROOTFS)) {
                if (!(flags & MNT_UNION)) {
                        flags |= MNT_UPDATE;
                } else {
                        /*
                         * For a union mount on '/', treat it as fresh
                         * mount instead of update.
                         * Otherwise, union mouting on '/' used to panic the
                         * system before, since mnt_vnodecovered was found to
                         * be NULL for '/' which is required for unionlookup
                         * after it gets ENOENT on union mount.
                         */
                        flags = (flags & ~(MNT_UPDATE));
                }

#if SECURE_KERNEL
                if ((flags & MNT_RDONLY) == 0) {
                        /* Release kernels are not allowed to mount "/" as rw *
                        error = EPERM;
                        goto out;
                }
#endif
```

iOS 11.3: APFS Snapshot

Explanation

- Root filesystem is now mounted from a read-only APFS snapshot
- Snapshots are used under the hood of time machine backups
- Snapshots are inherently unmodifyable
- We need to mount the live fs, not snapshot

mnt: iOS 11.3-11.4.1 ≤iPhone 7

Bypass

- Find the vnode of /dev/disk0s1s1
  - Follow pointers in: rootfs vnode -> mount -> devvp
  - Unset the flag that specifies it's in use/mounted already
- Live fs can be temporarily mounted to another directory
- Rename root fs snapshot
- Reboot
- Live fs gets mounted as read-only on subsequent boots

mnt: ≤iOS 11.~~2~~ 4 (all devices)

patch

Bypass

- mount has a check to prevent remounting ROOTFS
  - temporarily unset MNT_ROOTFS flag in kernel
  - call mount() to remount the root filesystem as read/write
  - reset MNT_ROOTFS flag in kernel

```
	if ((vp->v_flag & VROOT) &&
	    (vp->v_mount->mnt_flag & MNT_ROOTFS)) {
		if (!(flags & MNT_UNION)) {
			flags |= MNT_UPDATE;
		} else {
			/*
			 * For a union mount on '/', treat it as fresh
			 * mount instead of update.
			 * Otherwise, union mouting on '/' used to panic the
			 * system before, since mnt_vnodecovered was found to
			 * be NULL for '/' which is required for unionlookup
			 * after it gets ENOENT on union mount.
			 */
			flags = (flags & ~(MNT_UPDATE));
		}

#if SECURE_KERNEL
		if ((flags & MNT_RDONLY) == 0) {
			/* Release kernels are not allowed to mount "/" as rw *
			error = EPERM;
			goto out;
		}
#endif
```

iOS 12: Snapshot flag

Explanation

- iOS 12 has a flag set on the snapshot in kernel memory
  - Can't just rename it anymore
    - ...or can we?

mnt: ≤iOS 14 Unsetting Snapshot flag

Bypass

- Live fs temporarily mounted to another directory
  - Get vnode of temporary directory
  - Snapshot vnode is on the cached vnodelist off the new mount
  - Flag lives on the vnode's v_data (filesystem specific data)
  - Can simply unset the flag
- Rename root fs
- Reboot
- Live fs gets mounted as read-only on subsequent boots

mnt: ≤iOS 1~~1~~.4 14 (all devices)

- mount has a check to prevent remounting ROOTFS
  - temporarily unset MNT_ROOTFS flag in kernel
  - call mount() to remount the root filesystem as read/write
  - reset MNT_ROOTFS flag in kernel

patch

Bypass

```
if ((vp->v_flag & VROOT) &&
    (vp->v_mount->mnt_flag & MNT_ROOTFS)) {
        if (!(flags & MNT_UNION)) {
                flags |= MNT_UPDATE;
        } else {
                /*
                 * For a union mount on '/', treat it as fresh
                 * mount instead of update.
                 * Otherwise, union mouting on '/' used to panic the
                 * system before, since mnt_vnodecovered was found to
                 * be NULL for '/' which is required for unionlookup
                 * after it gets ENOENT on union mount.
                 */
                flags = (flags & ~(MNT_UPDATE));
        }

#if SECURE_KERNEL
        if ((flags & MNT_RDONLY) == 0) {
                /* Release kernels are not allowed to mount "/" as rw *
                error = EPERM;
                goto out;
        }
#endif
```

iOS 15: Sealed Snapshot

Explanation

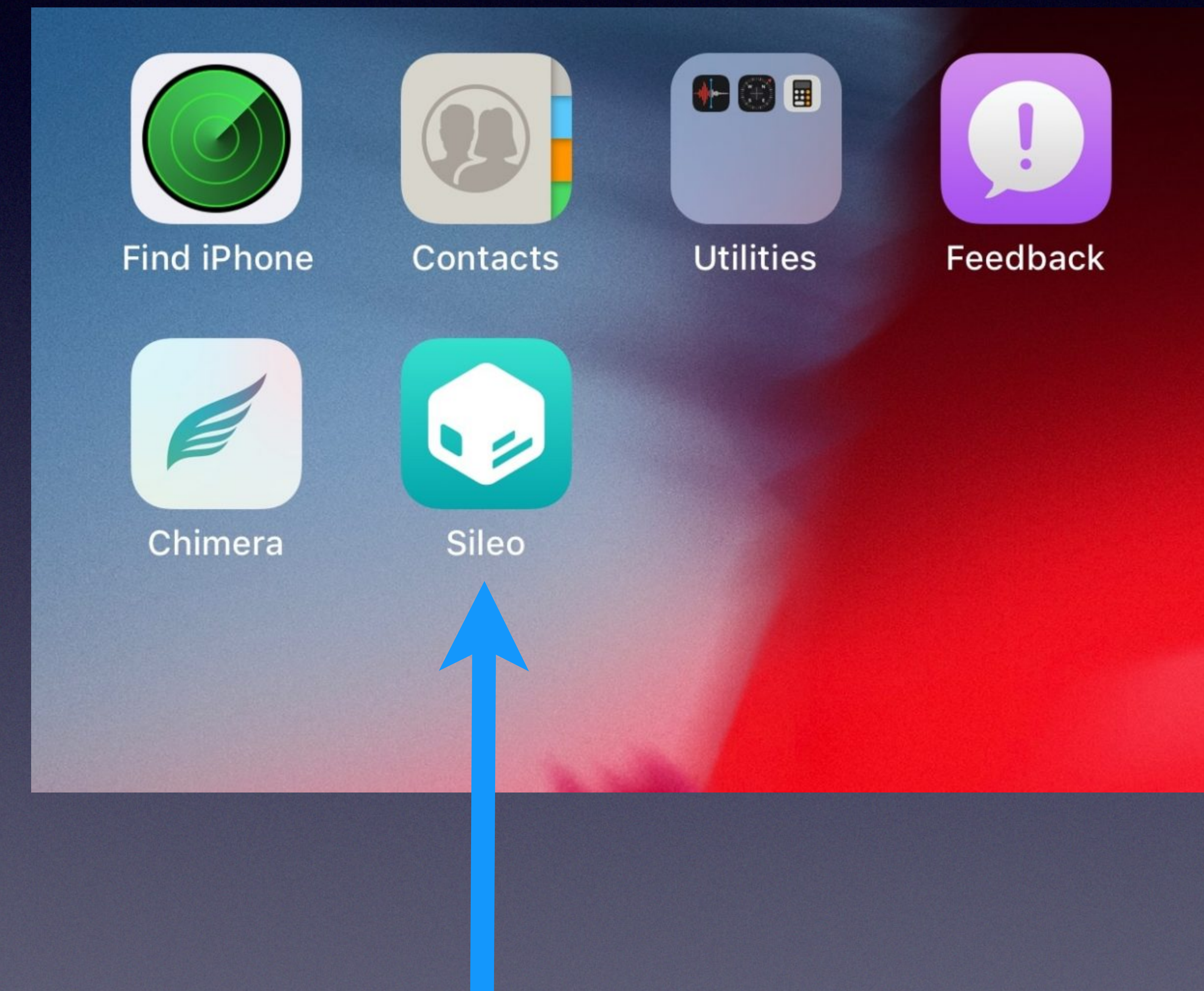
- APFS snapshot sealed in iOS 15
  - Also sealed in macOS Big Sur
- iOS 15 ensures that live fs is never mounted
- Can rename snapshot, but device will bootloop
- Jailbreak app runs too late
  - Will have to live without read/write root fs :(
    - Can simply place our files elsewhere (like on the data volume)

Jailbreak in a nutshell

- ~~Exploit kernel -> (get unstable kernel write)~~
- ~~Get stable kernel read/write~~
  - ~~Make it available to other processes~~
- ~~Privilege escalation (get ability to spawn process as root)~~
  - ~~Escape sandbox~~
  - ~~Become root~~
- ~~Bypass codesign enforcement~~
- ~~System-wide code injection~~
- ~~Optional: read/write root filesystem~~

Congrats you're jailbroken!

- untar bootstrap with useful binaries (e.g. shell commands and Sileo)
  - On iOS 14 and lower this can go to the root filesystem!
  - On iOS 15 this will need to be somewhere else
- Call LaunchServices to register a new app and/or start an SSH server
- Your device is now jailbroken



Current state of affairs

Explanation

- Full jailbreak with read/write root filesystem up to iOS 14 on all devices
- Apple's mitigations have resulted in increasing complexity of the jailbreak
  - Jailbreak tool, codesign bypass, userland code injection, system-wide code injection now require tight integration since iOS 12
  - Integration has gotten even tighter with even persistent Kernel read/write depending on the userland injection library since iOS 14
  - Jailbreak and libhooker depend on each other, are no longer separate (unlike like prior jailbreaks and Substrate)
  - Newer jailbreaks replace kernel patch functionality by doing system-wide userland code modification

iOS 15: eta 2023???

Questions?