iOS jailbreak internals (1): Remount root filesystem after iOS 11.3 – A new mitigation bypass

Xiaolong Bai and Min (Spark) Zheng @ Alibaba Security Lab

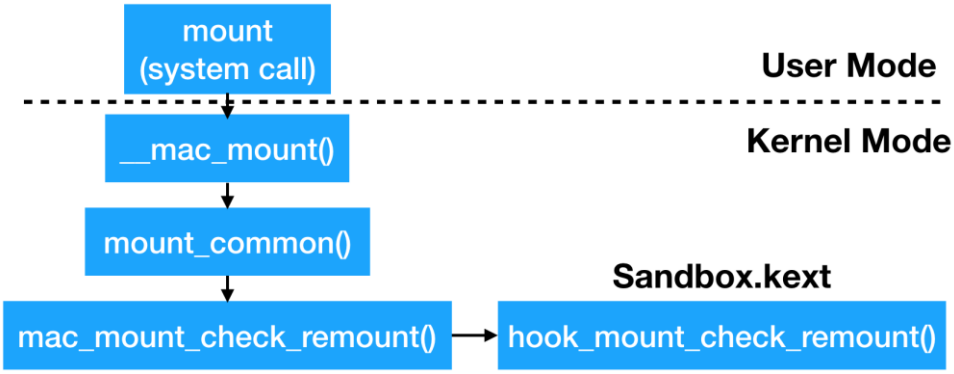
0x00 Introduction

A RW root partition is important for jailbreaks due to a need of installing unsandboxed programs and modifying system settings. Basically, root filesystem on iOS is always read-only. Therefore, remount root filesystem is a critical step in jailbreaking modern iOS. But, Apple will not allow the root filesystem to be remounted easily.

In this blog, we will introduce Apple’s new mitigation incurred in iOS 11.3 that prevents root filesystem from being remounted as RW, and also propose a brand-new mitigation bypass technique. According to our research, our new mitigation bypass will work with Ian Beer’s incoming tfp0 on iOS 11.3.1. That means, you can get a jailbreak on iOS 11.3.1!

0x01 Remounting before iOS 11.3

When we remount a filesystem by mount() system call, the kernel function __mac_mount() is called, which eventually calls mount_common(). In mount_common(), there is a MACF check mac_mount_check_remount() to check whether the remount is allowed. This MACF check is handled by sandbox kext in hook_mount_check_remount(), which checks whether the filesystem to be remounted is a root filesystem (ROOTFS). If true, it will stop the filesystem from being remounted.



Before iOS 11.3, the most common method to remount root filesystem aims at bypassing this sandbox check, which is proposed by Xerub. This method removes ROOTFS, RDONLY, NOSUID flags in root filesystem's mount flag and then remounts, which is thoroughly described in Jonathan Levin's HITB 18 AMS talk.

0x02 Basics of iOS filesystem

But, when there is a bypass, there is always a new mitigation. After iOS 11.3, if we remount the root filesystem in the old way. The kernel will panic as follows when we make changes (write or create) to files.


```
  "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:
```


This illustrates a new mitigation in iOS' s filesystem. We studied iOS filesystem and propose new methods to bypass the latest mitigation. In this blog, we will explain one of our new methods. Before showing the details of our new method, let' s first introduce some basics of iOS filesystem' s structures and explain why the panic happens.

Apple supports many filesystems including APFS, HFS+, FAT, and so on. When a filesystem is mounted, a general "mount" structure (as shown below) is generated, which represents how the filesystem is mounted.

```
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 */
```

In this structure, mnt_flag have options like RDONLY, ROOTFS, NOSUID, representing the basic mount status. mnt_ops is a list of function pointers implemented by filesystems for operations like mount, unmount, ioctl, etc. mnt_data is a pointer pointing to the private "mount" structure of the specific mounted filesystem. mnt_data describes in detail how the specific filesystem (e.g., APFS) is organized and contains critical data for the filesystem to operate on. Internally, a specific filesystem's mount/remount/unmount operations check and manipulate this private "mount" structure to take effect.

0x03 Root Cause (iOS 11.3 new mitigation)

In the panic log, the path "com.apple.xbs/Sources/apfs/apfs-748.52.14" illustrates that the panic happens in a APFS filesystem. That means the root filesystem on iOS is mounted as an APFS. APFS (short for Apple File System) is a new filesystem proposed by Apple, which is currently deployed on all Apple devices. It is designed with features like clones, encryption, snapshot, etc.

So, what really happens to APFS that causes the panic? To answer this question, we first need to know, how the root filesystem is mounted here. By executing "mount" command on iOS, we can get the information of currently mounted filesystems as follows.

```
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)
```

There is a strange prefix "com.apple.os.update-CA59XXXX@" in the device path mounted on root filesystem. What is this strange prefix? To answer this question, let's do some experiments on macOS.

```
$ tmutil localsnapshot /
Created local snapshot with date: 2018-05-30-154704
$ tmutil listlocalsnapshots /
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
/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)
```

Here, "tmutil localsnapshot /" creates a new snapshot of the root partition named "com.apple.TimeMachine.2018-05-30-154704" and "mount -t apfs -o -s=com.apple.TimeMachine.2018-05-30-154704 / /tmp" mounts the snapshot to /tmp. After this mount, we can see that there is also a prefix "com.apple.TimeMachine.2018-05-30-154704@". Then, we can know that the "com.apple.os.update-CA59XXXX@" on the iOS root partition also represents a snapshot.

Apple explains snapshot as "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." Apparently, iOS mounts root filesystem as a read-

only snapshot. This, root filesystem mounted as snapshot, is the latest mitigation proposed in iOS 11.3, which causes remount to fail.

So, why the previous remount method fails? In abstract, though we modify the mount flag of the root filesystem to be read-write, the root filesystem still represents a snapshot. Especially, the filesystem's private "mount" structure (mnt_data) is not modified and still represents a read-only snapshot. That means, the previous method remounted a "writable" read-only snapshot, which apparently causes conflicts. As a result, a panic is inevitable.

To explain the technical details of the root cause, let's look back into the panic log. The panic log says that we are lack of an extent covering a size. In APFS filesystem, the term "extent" is an internal data structure representing a file's location and size. By reverse engineering apfs kernel extension, we confirm that file extents are organized as btrees and stored in APFS's private "mount" structure (i.e. mnt_data in mount structure, see 0x02). But, a snapshot mount does not have valid file extent structures in its mnt_data. When we try to do file changes, the APFS filesystem looks up for a valid extent in the mounted filesystem's mnt_data. As a result, in the previous remount method, when we attend to make file changes to the remounted snapshot root filesystem, though the modified mnt_flag allows APFS to look up for extents, it cannot find a valid extent in the snapshot mount's mnt_data. This failure incurs the panic.

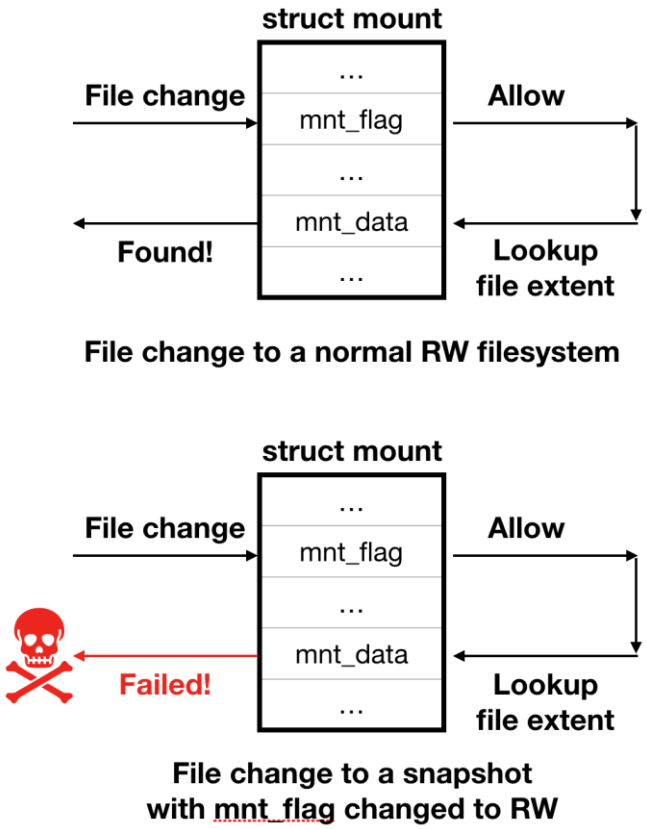


Fig.1 The root cause of why previous remount method fails

0x04 Our New Bypass

With these findings, a straightforward solution comes up soon: we need to make file extent lookups in a snapshot' s mnt_data succeed. To achieve this goal, we need to create a mnt_data that has valid file extents organized as in a normally RW-mounted root filesystem. However, creating a valid mnt_data manually is a difficult and complicated task. Can we ask APFS to create a mnt_data as the same as the one in a RW root? We believe the answer is YES if we can make a new RW mount of the root filesystem and get mnt_data from the new mount.

Then, the basic idea of our new bypass comes into mind, which consists of these steps:

1. Mount the device (/dev/disk0s1s1) of root partition to a path (e.g. /var/mobile/tmp) as read-write
2. Get mnt_data from the new mount

3. Change root filesystem’ s mnt_flag and remount (like the previous remount method)
4. Replace root mount’ s mnt_data with the new mount’ s mnt_data

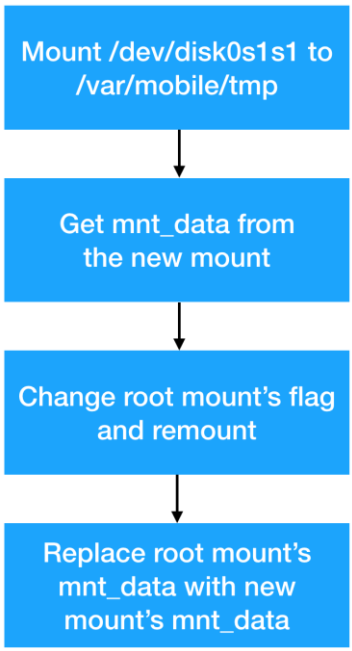


Fig.2 Steps of our new bypass

But, here is a new issue, that is, iOS does not allow the root partition device (/dev/disk0s1s1) to be mounted for twice. In mount_common(), there is a check that disallows multiple mounts of the same device.

```
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(), it gets v_specflags from the device vnode, and checks it with SI_MOUNTEDON and SI_ALIASED flags.

```
/*
 * 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;
    }
    if (vp->v_specflags & SI_ALIASED) {
        for (vq = *vp->v_hashchain; vq; vq = vq->v_specnext) {
            if (vq->v_rdev != vp->v_rdev ||
                vq->v_type != vp->v_type)
                continue;
            if (vq->v_specflags & SI_MOUNTEDON) {
                error = EBUSY;
                break;
            }
        }
    }
out:
    SPECHASH_UNLOCK();
    return (error);
}
```

In our solution, we clear the device vnode’ s v_specflags to bypass this vfs_moutedon() check.

Finally, the pseudocode of our new bypass to remount the root filesystem as RW is as follows

```
void remountRootAsRW(){
    char *devpath = strdup( “/dev/disk0s1s1” );
    uint64_t devVnode = getVnodeAtPath(devpath);
    writeKern(devVnode + off_v_specflags, 0); // clear dev vnode’ s v_specflags

    /* 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);
```

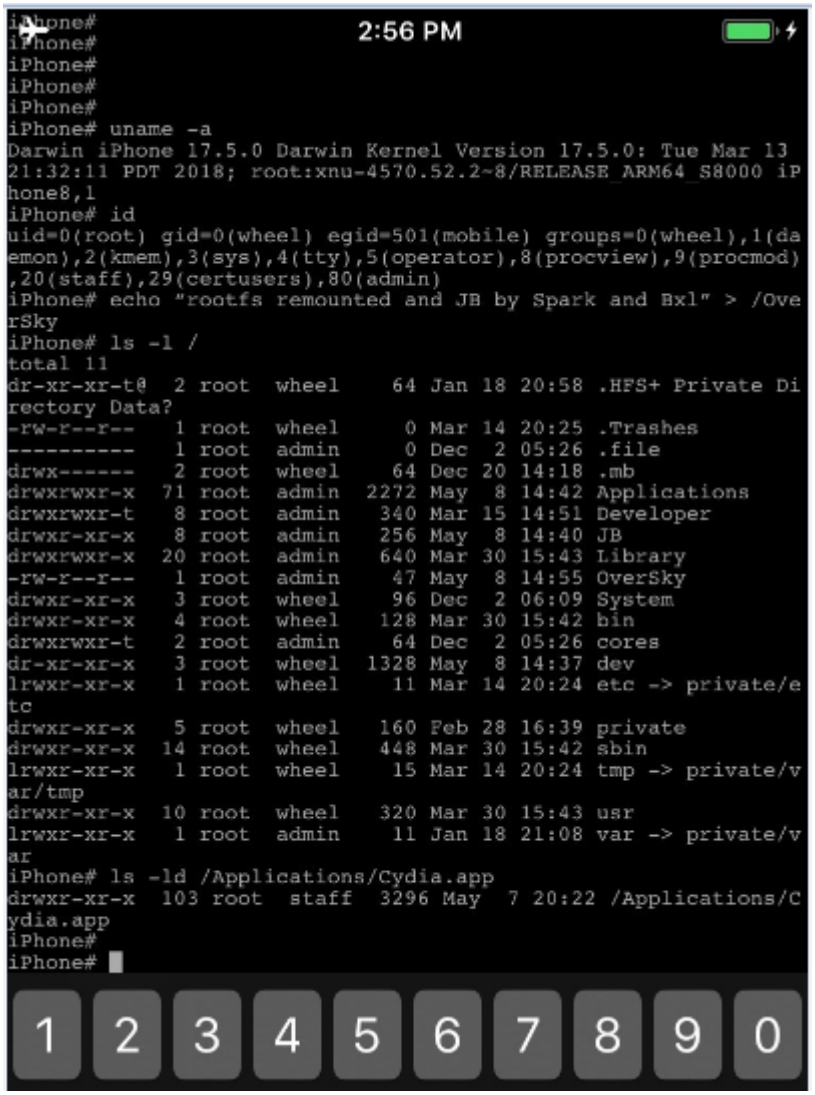
```
    uint64_t newMPMount = readKern(newMPVnode + off_v_mount);
    uint64_t newMPMountData = readKern(newMPMount + off_mnt_data);

    /* 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);
}
```

In this code, readKern() reads value from a kernel address and writeKern() writes value to a kernel address, which can be found in jailbreaks like Xerub, Electra, V0rtex, mach_portal, or Qilin toolkit. getVnodeAtPath() is our new gadget to get the vnode address of a path, which employs Ian Beer’ s technique to execute code in kernel.

With our new bypass, you now have a RW root filesystem. You can make changes to system files, install binaries in unsandboxed paths, etc. The following picture shows a successful file remount status and JB on iOS 11.3.1:



0x05 Conclusion

One thing you should know about this new bypass is that modifications you make to the root filesystem will be discarded after you reboot. So, this is an untethered remount solution (we may discuss persistent file system remount solutions in a near future). But, apparently, this is enough for most current jailbreaks.

Last but not least, we will talk more about iOS jailbreak and macOS vulnerability detection topics at DEFCON (August 9-12, 2018 @ Las Vegas). And, welcome to follow us on twitter: @bxl1989 and @SparkZheng. :-)

Nikita Caine Kronenberg	Congrats! DEF CON presentation accepted! One bite and all your dreams
Nikita Caine Kronenberg	Congrats! DEF CON presentation accepted! - Fasten your seatbelts: We a