Writeup: Arbitrary code execution in U-Boot when loading a crafted ext4 partition

SUMMARY

A buffer overflow in the ext4fs drivers of u-boot results in **arbitrary code execution** when a crafted ext4 partition is accessed, probed, or otherwise calls **`ext4fs_mount (fs/ext4/ext4_common.c)`**. Successful exploitation of this issue allows attackers to run arbitrary code in the context of the loaded U-boot shell. All versions of Das U-Boot are vulnerable before versions *2019.07-rc1*, which has yet to be released. Hence, all systems in the competition that use the inbuilt ext4fs drivers are vulnerable. This vulnerability was reported to Das U-Boot and was assigned CVE-2019-11059 by MITRE.

DESCRIPTION OF ATTACK

When **`ext4fs_mount`** is called, it reads in the ext4 superblock corresponding to the partition being accessed. It then determines the size of the block group descriptor using the following code:

```
if (le32_to_cpu(data->sblock.revision_level) == 0) {
    ...
} else {
    ...
    fs->inodesz = le16_to_cpu(data->sblock.inode_size);
    fs->gdsize = le32_to_cpu(data->sblock.feature_incompat) &
        EXT4_FEATURE_INCOMPAT_64BIT ?
        le16_to_cpu(data->sblock.revision_level) : 32;
}
```

Figure 1: `u-boot/fs/ext4_common.c:2347-2360`

However, there seems to be some confusion with regards to the ext4 specifications. In particular, older versions of ext4 disk layout documentation incorrectly stated that the block group descriptor only expands to 64 bytes[1], but the newest version of the documentation fixes this to state that it expands to ***at least*** 64 bytes[2]. The result is that the underlying code has varying assumptions at different points, and we exploit this this to our advantage.

`ext4fs_read_inode` attempts to read in the block group descriptor into a **64-byte stack-allocated `ext2_block_group`** struct by calling **`ext4fs_blockgroup`**. However, despite being a constant size struct, it also dynamically calculates the block number of the offset of the group descriptor table and reads it into the struct:

```
int log2blksz = get_fs()->dev_desc->log2blksz;
int desc_size = get_fs()->gdsize;
desc_per_blk = EXT2_BLOCK_SIZE(data) / desc_size;
blkno = le32_to_cpu(data->sblock.first_data_block) + 1 +
       roup / desc_per_blk;
blkoff = (group % desc_per_blk) * desc_size;
...
return ext4fs_devread((lbaint_t)blkno << (LOG2_BLOCK_SIZE(data) - log2blksz),
       blkoff, desc_size, (char *)blkgrp);
```

Figure 2: `u-boot/fs/ext4_common.c:1563-1583`

[1]Ext4 Disk Layout, retrieved at 10/21/2013 10:43 PM, archived at https://djwong.org/docs/ext4_disk_layout.pdf : "*On an ext4 filesystem with the 64bit feature enabled, the block group descriptor **expands to the full 64 bytes** described below.*"

[2] Ext4 Disk Layout, retrieved at 04/15/2019 04:57AM, https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout "*On an ext4 filesystem with the 64bit feature enabled, the block group descriptor **expands to at least the 64 bytes described** below ; the size is stored in the superblock.*"

Therefore, by setting the **revision level (`data->sblock.revision_level`)** of the file system to `1`, setting the `EXT4_FEATURE_INCOMPAT_64BIT` flag in the `feature_incompat` field, and the **block group descriptor size (`get_fs()->dev_desc->log2blksz`)** to greater than `0x40` **bytes**, we can cause an easily controllable buffer overflow by appending bytes to the end of the block group descriptor, and we overflow the 64 byte `ext2_block_group` struct allocated on the stack.

This allows us to exploit systems built on u-boot that access ext4 partitions but otherwise provide the user with a limited interface. For example, it is possible to bypass the MES shell in this way.

This bug is still present as of the latest revision of u-boot (v2019.04-rc1). There are likely multiple bugs of a similar nature in u-boot.

PROOF OF CONCEPT

The POC was written against the MESH shell running on qemu. The script takes a raw SD card image generated as input and modifies the appropriate bytes. Starting up MES in qemu causes **MESH** to try to access the second (tampered) ext4 partition on the SD card:

```
$ ./start-qemu.sh
PetaLinux environment set to '/opt/pkg/petalinux'
…
Performing first time setup...
Done!
qemu-system-aarch64: Trying to execute code outside RAM or ROM at 0x0000000041414140
```

From here, we have arbitrary code execution, allowing us full control of the target system in its current execution context, which is sufficient to takeover the system. For instance, we can do direct control flow to a ROP chain that prints secrets present on the MES shell. We can also form a ROP chain that overwrites the boot arguments passed and boot, so we boot into a shell in Petalinux instead.

REMEDIATION

Following the updated specifications, we dynamically allocate a chunk of memory of the correct size.

```
-       struct ext2_block_group blkgrp;
+       struct ext2_block_group *blkgrp;
…
+       /* Allocate blkgrp based on gdsize (for 64-bit support). */
+       blkgrp = zalloc(get_fs()->gdsize);
+       if (!blkgrp)
+               return 0;
…
        inodes_per_block = EXT2_BLOCK_SIZE(data) / fs->inodesz;
-       blkno = ext4fs_bg_get_inode_table_id(&blkgrp, fs) +
+       blkno = ext4fs_bg_get_inode_table_id(blkgrp, fs) +
…
+
+       /* Free blkgrp as it is no longer required. */
+       free(blkgrp);
+
```

Figure 3: blobdiff of patch, which can be viewed at http://git.denx.de/?p=u-boot.git;a=commit;h=febbc583319b567fe3d83e521cc2ace9be8d1501

The above patch was pushed and merged into the master branch of U-Boot as of April 10th, 2019. The next public release version of U-Boot containing the patch will be 2019.07-rc1, which is to be released on April 28th, 2019.