**Affected products**
This vulnerability affects certain bootloader-locked Qualcomm based Samsung Galaxy products containing Samsung eMMC. Tested and confirmed to unlock the bootloader of the Verizon Samsung Galaxy S5. While possibly every device containing a Samsung eMMC controller is vulnerable to the attack described below, it’s usefulness as a bootloader unlock is likely limited to select Samsung Galaxy devices.

**Samsung’s unlock mechanism**
Manufacturers employ a wide variety of methods and mechanisms to determine and control a device’s bootloader lock status. Motorola uses TrustZone protected fuses, HTC uses data in a write-protected region of eMMC, and some LG devices use a signed blob in the boot (kernel) partition. There are likely many other methods, but the concept is the same: only allow the unlock status to be changed via a controlled and deliberate method. Additionally, when unlocking the bootloader of most devices, the user’s data partition is wiped to ensure that the unlock wasn’t performed in attempt to maliciously gain access to a user’s data.

In some cases, like most devices built for use on the Verizon network, the ability for a user to unlock the bootloader is blocked entirely. One such device was the Galaxy S5 used for this research. The Galaxy S5 (and probably some/many other Galaxy devices) uses a unique mechanism for determining a device’s unlock/dev status. As Researched and discovered by @ryang (http://forum.xda-developers.com/member.php?u=766721):

> A blob in the aboot partition image is read and decrypted. Then, the device’s eMMC CID is hashed and compared to the value of the decrypted blob. If they match, the device is considered unlocked.

Basically, at some point in the manufacturing process, when a device is configured to be a “dev-edition” device, Samsung hashes the device’s eMMC CID, pads it, signs (or encrypts?) it with their private key, and places that signed data in the device’s aboot partition. At every boot, aboot is able to verify that the device is actually a dev-edition device.

What is an eMMC CID? According to eMMC documentation:

> “The Device IDentification (CID) register is 128 bits wide. It contains the Device identification information used during the Device identification phase (e•MMC protocol). Every individual flash or I/O Device shall have an unique identification number. Every type of
SamDunk

**eMMC Device shall have a unique identification number.**"

Great! We can simply change the eMMC CID to match one from a factory dev-edition device, and then flash the aboot partition (containing the hashed and signed CID blob) from the same dev-edition device. Except, elsewhere in the eMMC documentation:

“Programming of the Device identification register. This command shall be issued only once. The Device contains hardware to prevent this operation after the first programming. Normally this command is reserved for the manufacturer.”

Effectively, the CID is a serial number programmed at the factory and, according to the eMMC standard, is only programmable once. What “hardware” prevents the CID from being reprogrammed? Is it really stored in some OTP region? The fact that an eMMC contains mass amounts of non-volatile reprogrammable NAND memory should make any researcher skeptical that it is truly “write-once”.

**Enter vendor commands**

Generally speaking, vendor commands are commands/codes/instructions that don’t appear in any official standards and allow a host/user/factory/software to interact with a device in a non-standard way. Specifically for Samsung eMMC, a special (mostly) undocumented CMD62 in conjunction with some “secret” arguments are used. Some of Samsung’s vendor commands were publicly released by Samsung due to a software defect in their eMMC controller firmware. Samsung needed the linux/android kernel to be able to patch the eMMC firmware in realtime, and thus had to release and document some of their proprietary eMMC vendor commands:

https://android.googlesource.com/kernel/omap/+/3c57a612f12b069bb863ec0c74a26850d76a9e8%5E1

Samsung provided a critical function for enabling further research: mmc_mov_read_cmd. With a simple loop of this function, we are able to dump the entire running firmware and ram of the eMMC controller.
Finding something useful
The first step was to find what function in Samsung’s firmware handled a host’s PROGRAM_CID request. A quick (and quite lucky, as most eMMC commands use a function pointer table) search for the immediate value 26 (eMMC command 26 is PROGRAM_CID) reveals this:
Some basic setup is done, then the opcode is checked to decide whether or not the host is trying to program the CSD register or the CID register. Since we are trying to reprogram the CID register, let’s look at the program_cid function called when R0 = 0x1A (26):
The function read_security_register is important and very simple:

It simply reads a value from ram, stores it in the caller’s pointer, and returns. Back to the program_cid function:
The security register is read and bit 0 is tested. If bit 0 is clear, a branch is taken to loc_4AC84, and ultimately the CID programming fails. If bit 0 == 1, the function continues:

1.) Bit 0 is cleared, and the security register is updated with the new value
2.) The CID is programmed

Surely this “security_register” can’t be the “hardware to prevent this operation after the first programming” as described in the eMMC specifications; it is simply a dword in the controller’s RAM. How can we get bit 0 set in the security_register? A quick refs search to write_security_register answers this question quickly:

vendor_security_functions() is a real mess:
We need to find a call to `write_security_register` with a value of 1, to allow us to reprogram the CID. Zooming in:

Bingo:

```
; CODE XREF: vendor_security_functions:jump_to_cid_backdoor
MOV R0, SP
read_security_register
LDR R0, [R0, #0x30+var_30]
orr.w R0, R0, #1
write_security_register
```

You can see very clearly, the security register is read, bit 0 is set, and the security register is updated, enabling programming of the CID.

Working backwards through the ~100 argument calculations in the vendor command handler, I’ll simply state that the arg value that needs to be passed with the CMD62 is 0xEF50. **Issuing the vendor command with argument 0xEFAC62EC followed by the vendor command with argument 0xEF50 is Samsung’s backdoor to allow reprogramming of the eMMC CID.**
**SamDunk**

**Doing the deed**
I’ve pushed code to github to change the CID in Samsung eMMC using the backdoor described above: [https://github.com/beaups/SamsungCID](https://github.com/beaups/SamsungCID)

**Caveats (important):**

1.) It is generally a bad idea to do much with Samsung vendor codes from userspace, as you can not take a lock on the eMMC. The program_cid backdoor appears to be quite safe, but do NOT try to update the code to do operations such as reading/writing controller memory. You’ll want to do that with a kernel module. You’ve been warned.

2.) While the code will change the device’s CID, you will need to check (after reboot) to see that the CID changed “perfectly”. In some instances, a few bits are kept from the original CID. If that happens, you will need to write a kernel module, dump the controller memory, find the current CID in the controller’s memory, patch it to 0, and program the CID again.

3.) I have not researched what else Samsung/Android/bootloaders/apps/etc. might use the device’s CID for. Change your CID at your own risk.

4.) If your goal is to turn your device into a dev-edition device, YOU will need to find the dev-edition aboot image and the corresponding CID. Further, you should only attempt to flash any aboot through ODIN. If ODIN rejects the flash, you’ve done something wrong.

5.) Only UID 0 has write access to mmcblk0, which is needed to issue the necessary IOCTLs.

6.) Do not PM, email, call, or otherwise stalk me looking for support, supporting files, or anything else.