Digital Investigation 24 (2018) S60–S67

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2018 Europe — Proceedings of the Fifth Annual DFRWS Europe

Forensics acquisition — Analysis and circumvention of samsung secure boot enforced common criteria mode

Gunnar Alendal [a, *], Geir Olav Dyrkolbotn [a, c], Stefan Axelsson [a, b]

[a] Department of Information Security and Communication Technology, NTNU, Gjøvik, Norway
[b] Halmstad University, Sweden
[c] Norwegian Defence Cyber Academy (NDCA), Jørstadmoen, Norway

Keywords:
Common criteria
CC mode
Mobile security
Mobile device management
Forensic acquisition
Smart phone
Samsung secure boot

A B S T R A C T

The acquisition of data from mobile phones have been a mainstay of criminal digital forensics for a number of years now. However, this forensic acquisition is getting more and more difficult with the increasing security level and complexity of mobile phones (and other embedded devices). In addition, it is often difficult or impossible to get access to design specifications, documentation and source code. As a result, the forensic acquisition methods are also increasing in complexity, requiring an ever deeper understanding of the underlying technology and its security mechanisms. Forensic acquisition techniques are turning to more offensive solutions to bypass security mechanisms, through security vulnerabilities.

Common Criteria mode is a security feature that increases the security level of Samsung devices, and thus make forensic acquisition more difficult for law enforcement.

With no access to design documents or source code, we have reverse engineered how the Common Criteria mode is actually implemented and protected by Samsung's secure bootloader. We present how this security mode is enforced, security vulnerabilities therein, and how the discovered security vulnerabilities can be used to circumvent Common Criteria mode for further forensic acquisition.

Introduction

Digital forensics is the recovery and investigation of data found in digital devices (Carrier, 2002). Garfinkel (2010) discusses the difficulties that awaits digital forensics, what challenges exist in today's tools, research and knowledge and how digital forensic research should move forward to keep digital forensics a valid method for the years to come. The prediction is that both the recovery, forensic acquisition, and investigation will become increasingly harder as complexity and security mechanisms, like encryption, grow in use. Faced with this ever increasing security of Commercial of-the-shelf (COTS) products, law enforcement faces an increasing challenge when it comes to the ability to do forensic acquisition. Where before law enforcement could bypass security mechanisms by e.g. accessing data at a lower level, like forensic desoldering (chip-off), to read content off data storage directly, today's, often mandatory, encryption of user data on mobile devices invalidates such methods. The ability to read stored data on the device's storage is simply not enough. Reading encrypted data has little value without the corresponding encryption key(s). The addition of security features like device-tied encryption keys, supported by hardware and a TrustZone, gaining access to such encryption keys is made even harder. This might then require law enforcement to power on the device, in order to try to extract keys or decrypted data through interaction with the security mechanisms protecting the user data. This type of interaction often means installing or modifying code on the device. Even though law enforcement have legitimate cause for their “hacking”, this is activity that in other contexts would be regarded malicious and illegal, also known as an attack. Therefore, to protect against such attacks, most mobile device vendors protect code running on the devices, from the first code executed at power on and all the way through to a full operating system, like Android, is up and running. This is often referred to as a *Secure Boot*, and refers to the trust in code executed on the device. This code should only be certified and official code, made by the vendor, and properly signed to prove authenticity.

* Corresponding author.
E-mail addresses: gunnaale@stud.ntnu.no (G. Alendal), geir.dyrkolbotn@ntnu.no (G.O. Dyrkolbotn), stefan.axelsson@ntnu.no (S. Axelsson).

https://doi.org/10.1016/j.diin.2018.01.008

Law enforcement always strives to acquire as much data as possible to support any ongoing investigation. So bypassing such complex security schemes, if possible, forces law enforcement to invest in deeper knowledge and costly equipment to perform advanced forensic acquisition, utilising such attacks.[1] Law enforcement is then investing in the discovery and use of security vulnerabilities, to bypass security mechanisms to acquire digital evidence.

On the other hand, seen from a user and enterprise perspective, with the growing use of these devices, both end users and enterprises are demanding more secure devices to help protect sensitive data. The need to secure mobile devices, especially in an enterprise context is important, as devices moving in and out of the enterprises network, unchallenged, introduces attractive attack vectors for cyber criminals and cyber espionage.

Mobile Device Management (MDM) solutions can enable the centralised control of devices that are used in the enterprise. Enterprises can then monitor, control and administrate devices in a systematic manner, across device vendors and service providers. Samsung supports such solutions by offering a.o. a feature they refer to as *Common Criteria mode* or simply *CC mode* (Samsung, 2017a). CC mode is a security feature designed to increase the device's protection against unauthorised access and can therefore pose an additional challenge to law enforcement trying to acquire data from devices with CC mode enabled. A major challenge is that CC mode denies access to the device firmware update mechanism, a common method used by law enforcement to gain access to data.

This paper presents the reverse engineering results of CC mode and how discovered security vulnerabilities can be used to circumvent CC mode for further forensic acquisition.

The rest of the paper is organised as follows: Section "Related work and contributions" discusses related work and how our contribution relates. Section "Samsung secure boot model" introduces the Samsung secure boot model. Section "Samsung CC mode and SBOOT" describes the CC mode related parts of the Samsung secure boot and how this relates to the secure execution environment, TrustZone. Section "Unauthorised disabling of CC model" discusses attacks on the CC mode. In section "Conclusion" we discuss the implications of our findings and offer our conclusions.

Related work and contributions

Recovering data from mobile devices can be achieved by reading data from storage or from volatile memory (RAM). The two sources of data differs in both how data is stored and how data can be retrieved. Data in long term storage is often stored well structured in file systems, as it has to be able to be read by different operating systems, and other tools. Data structures in RAM are often less well documented, and the formats more volatile, as it needs only survive to the next restart of the device. RAM is repopulated each time the device is restarted.

Nathan Scrivens et al. (Scrivens and Lin, 2017) summarised many of the current options for forensic acquisition of storage on Android mobile devices. According to Scrivens et al., the main options are chip-off, de-soldering storage for off-device reading, JTAG (Joint Test Action Group) interface for in-circuit reading of storage, rooting and exploitation solutions for recovering data by breaking the security of the device, Android Debug Bridge (ADB) by utilising device debug capabilities for forensic acquisition, and finally backup solutions retrieving data through normal or rooted user access. These different methods have different requirements and weaknesses. Chip-off requires physical access to underlying storage media, and can not deal with the increasing use of encryption on storage devices. JTAG is a interface often used during development and testing of a device, and can be used to communicate directly with the underlying storage media. However, the JTAG test pins can be hard to find and access on different devices, and can also be secured against unauthorised access, and also disabled by the vendor before shipping. ADB is a powerful debug interface supported by Android, but it is not enabled by default on most Android devices, nor does it give root access. Finally, backup applications are rarely accessible to unauthenticated users and are often of limited use for forensics.

Seung Jei Yang et al. (2015) demonstrated a different approach: doing forensic acquisition of storage media through the misuse of the device firmware update protocols. This will give access to the underlying storage and the ability to dump its content. Unfortunately this method will also be insufficient if the data stored is encrypted.

Seung Jei Yang et al. (2017) recently demonstrated a different use for the device firmware update protocols. Instead of acquiring storage they have demonstrated how to acquire RAM through this update protocol. This can again be used to acquire encryption keys used to encrypt storage, in addition to save user data that resides in RAM at the time of RAM acquisition.

Guido et al. (2016) demonstrated *hawkeye*, an agent to do rapid acquisition of Android devices. Although their goal is to reduce the amount of data needed to be transferred during the acquisition process, this is an example of a forensic agent that needs to be injected into the device to function as expected. This is done by installing a custom boot image on the device to facilitate hawkeye injection. Installing this custom image is done through the device firmware update protocol and access to firmware update mechanism is a requirement.

As we can see, access to a device's firmware update protocol can be vital for successful forensic acquisition. Any functionality denying this access is therefore limiting the possibilities for law enforcement to acquire data from a given device. CC mode is preventing law enforcement access to the firmware update mode on Samsung devices. Our contribution is to analyse and circumvent CC mode to gain access to the firmware update mode. For completeness, we have also included the discussion of a MDM setting, also affecting access to the firmware update mode.

Our reverse engineering of CC mode reveals security vulnerabilities in the design and implementation of these security mechanisms, and demonstrates how such security vulnerabilities can be discovered and used in digital forensic acquisitions.

Our contribution shows that law enforcement trying to acquire data from a device can disable CC mode and get access to firmware update mode, thus removing the extra layer of security enforced by CC mode. Disabling CC mode can then enable existing methods but also increases the attack surface in general, increasing the possibility to discover new vulnerabilities and methods.

CC mode and methodology

CC mode is built on top of the phone's Android security model and hardware, to increase enterprise security. Samsung has made available several guidance documents for Common Criteria evaluation for many of their different phone models (Samsung, 2017c).

Samsung provides a wide range of management APIs to control a Samsung device (Samsung, 2017b). These APIs can be used in 3rd party MDM solutions. To further promote the use of CC mode in MDM solutions, Samsung has made available a Common Criteria mode APK (Samsung, 2017a). This Android application package

[1] With the word "attacks", in the context of this article, we mean: exploiting vulnerabilities for forensic data acquisition purposes by law enforcement agencies.

(APK) is installed on the evaluated device and sets a number of default policies and security settings. This APK is intended for evaluators and IT adminis, to test the features of Samsung's CC mode. Samsung provides a long list of compatible phones, e.g. the Samsung Galaxy S6, with model name `SM-G920F` and the Samsung Galaxy S7 Edge, with model name `SM-G935F`. It is unknown to the authors what requirements are needed for a particular model to be compatible, but for blocking the access to the firmware update mode, the bootloader of compatible models must have code to handle this blocking. It is the bootloader that implements the firmware update mode.

The policy and settings set by the CC mode APK is the basis for the testing done in this paper. When we refer to CC mode, we refer to settings set by the CC mode APK. Our main test device was a Samsung Galaxy S7 Edge (SM-G935F) running firmware version `G935FXXS1DQGA_G935FNEE1DQF3_NEE`.[2]

One crucial feature of CC mode is the ability to only allow Firmware Over the Air (FOTA) firmware updates. This is to protect against an attacker with physical access to the device, trying to install unauthorised firmware on the device. Updating firmware in this way on Samsung devices is done through what is called *ODIN* mode. CC mode will both block ODIN mode and any attempt to boot an unofficial boot image already stored on the device.

Other features of Samsung CC mode and Common Criteria in general will not be discussed further as these features does not influence the blocking of ODIN mode.

Samsung devices come in different hardware configurations, where system-on-a-chip (SoC) implementations from Qualcomm (e.g. Snapdragon) and Samsung (Exynos) are the most common. Although the phone models share the same name, like *Samsung Galaxy S7*, they are very different in e.g. hardware components and bootloader code. In this paper we only focus on Samsung devices based on the Exynos SoC variants. Examples of devices with Exynos SoCs are Samsung Galaxy S6 (models `SM-G920F/SM-G925F`) and Samsung Galaxy S7 (models `SM-G930F/SM-G935F`).

Access to ODIN mode is enforced by the Samsung bootloader. The bootloader is part of the secure start-up of the device and is native code responsible for starting the device. On the studied models with the Exynos SoC, the bootloader responsible for ODIN mode is often referred to as *SBOOT*. SBOOT is built from Samsung proprietary code, and documentation and source code are not publicly available. We have analysed how the secure bootloader knows that CC mode is enabled and how this is used to limit access to certain features. We have also analysed the security of the storage of this CC mode configuration, as well as how SBOOT can change the configuration or simply disable CC mode. This leaves SBOOT not only responsible for enforcing the configuration, but also changing the setting.

Both design and implementation details of many security features are generally not available, and hence many such features may be left unexplored by the research community. To be able to analyse the enforcement of CC mode and how ODIN mode is blocked, we reverse engineered SBOOT with both static and dynamic analysis techniques. With access to the firmware for our test device, we reverse engineered the binary SBOOT code. Most of the static reverse engineering effort was done using the tool *IDA Pro* from Hex-Rays (Ilfak Guilfanov, 2017). We also developed our own exploit based on the SBOOT vulnerability disclosed in a security blog by Nitay Artenstein (2017). We used this exploit as a tool to perform more dynamic analysis of how CC mode is enforced and used to protect against unauthorised firmware updates. Our exploit is fully developed using the Python scripting language, with the aid of the Keystone assembler framework (Keystone team, 2017) for creating binary ARM code to be executed as part of the exploitation. Our goal was to be able to evade or disable CC mode, to get access to ODIN mode.

Samsung secure boot model

The code that is implementing ODIN mode, and thereby flashing firmware on Samsung devices, is located in the bootloader of Samsung devices. Therefore the bootloader must be able to turn off access to ODIN mode when CC mode is enabled on the device, in order for CC mode to disable firmware updates, e.g. designed to sidestep centralised control. To better understand how this mechanism works, some background on the Samsung secure boot model, applicable for devices using the Exynos SoC, follows.

To maintain security and trust in code running on devices, Samsung utilises a secure boot model (Samsung, 2017d), where all code running from power on until a complete Android system is running, is signed. This includes the integrity of the TrustZone and the baseband processor, that handles most of the radio functions. The security of bootloaders is therefore crucial for the integrity of the device. Nilo Redini et al. (2017) explored vulnerabilities in both design and implementation of bootloaders for a range of devices, and emphasised on the importance of a secure boot by demonstrating several attacks. A simplified boot model used by Samsung Exynos devices is show in Fig. 1. This shows how execution is started at the BootROM and carries on through the boot process through to the Android kernel.

Samsung provides a generic description of their platform security (Samsung, 2017d). This describes that the signature chain is rooted in the *Samsung Secure Boot Key*, SSBK, used to sign Samsung approved executable boot components. The public part of this key is stored in the phone's hardware at manufacture and will not change during the device lifetime. This is used by the BootROM when the device powers on. As seen in Fig. 1, the BootROM makes sure all executable code fetched from storage during boot is signed by Samsung. Booting a device with this model starts with the primary bootloader, loaded from Read-Only Memory (ROM). This primary bootloader loads the next bootloaders, Boot loader 1 (BL1) and Boot Loader 2 (BL2) from storage, e.g. flash, to RAM, checks the signature and advances execution to BL1. BL1 will carry out its tasks, often related to hardware initialisation, and advance

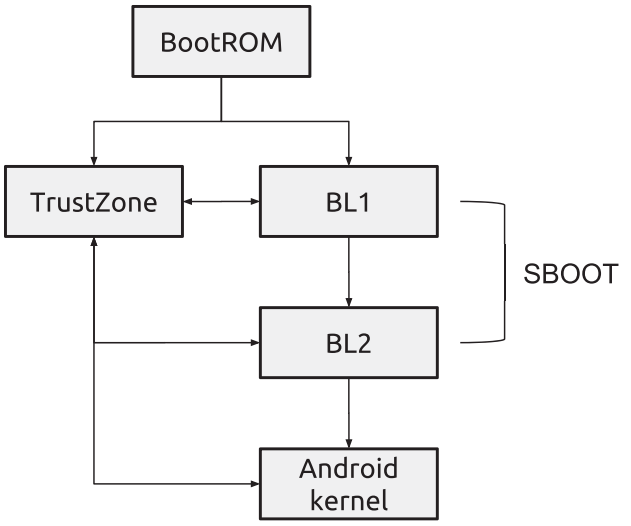

Fig. 1. Overview of the Samsung Secure Boot model from BootROM to an Android kernel.

[2] `G935FXXS1DQGA_G935FNEE1DQF3_NEE.zip SHA-1:67CA63BCAF53C9D48-A5D5DF43A8F5E56544081AC.`

execution to BL2. BL2 is a more complex bootloader, with larger code base, which in turn loads and checks the signature of the Android kernel before advancing execution to it. As a final stage, the Android kernel boots and loads the full operating system which enables all the device's features.

If all these bootloader stages maintain the signed integrity from the SSBK key, Samsung refers to this as *secure boot*. Note that Samsung distinguishes this from *trusted boot*, which also includes *Rollback Protection (RP)*, preventing the "downgrading" of any executable code to official, but vulnerable, older versions of the bootloaders or Android OS.

Note that Samsung does not forbid installing unofficial Android kernels, not signed by Samsung. This is however considered *tampering* with the device, and consequently a one-time programmable tamper fuse (eFuse) will be set. This fuse cannot be unset and the device is from here on marked as "Warranty void". This fuse is often referred to as the *KNOX Warranty Bit* (Kanonov and Wool, 2016) and is a Samsung proprietary way of marking the device as having been tampered with. Samsung can prevent the installation of such unofficial Android kernels if *Factory Reset Protection (FRP)* is set on the device. FRP is a setting that is enabled e.g. if the user adds a Google account to the system. FRP will prevent the installation of unofficial firmware updates, but will not deny access to ODIN mode.

In this paper we will refer to BL1 and BL2 as *SBOOT*. SBOOT is the first code loaded from writable storage and can therefore be upgraded as part of what is often referred to as *firmware upgrades*. Firmware upgrades for Samsung devices are often big archives that can upgrade different parts of the Samsung code environment, like the Android OS, the baseband processor or the SBOOT bootloaders. Any upgrade to the SBOOT will be included in a file called *sboot.bin*, so in order to analyse executable code belonging to SBOOT, we need to analyse the *sboot.bin* file. As seen from Fig. 1, SBOOT is a crucial part of the Samsung Secure Boot model.

Kanonov et al. (Kanonov and Wool, 2016) have analysed and found weaknesses in the security of the Samsung KNOX secure containers, and also described the Secure Boot process, related to the security of KNOX containers. They also discuss other important security features, like runtime protections, named TIMA, and e.g. it's use in attestation of a device. Device attestation is to test the authenticity and integrity of the security measures and policies.

SBOOT is responsible for a range of tasks before it loads and executes the Android kernel. These tasks are part of the Secure/Trusted boot and includes loading TrustZone dedicated applications, also known as *trustlets*. A trustlet is a small and dedicated application created to solve a specific, often sensitive task, like digital rights management (DRM). Trustlets run in the TrustZone.

The TrustZone is a separate execution environment, supported by the hardware, that divides each processor core into two separate "worlds". Often we refer to these different execution environments as the *normal* world and the *secure* world; the TrustZone. SBOOT is not part of the TrustZone and is running in *normal* world. When SBOOT is done executing its needed boot routines, it will load and execute the Android kernel. The Android kernel also runs in the same *normal* world. The TrustZone does not influence the enforcing of CC mode during boot and are therefore left out of further discussions in this paper.

Looking into the different steps performed by SBOOT, we can analyse the interaction with the CC mode configuration and how this is enforced. This will be explained in the next section.

Samsung CC mode and SBOOT

The Samsung Common Criteria Administrator Guidance, section 4.3.2.2 (Samsung, 2017), states that to place the device in the evaluated configuration, *CC mode* must be enabled on the phone. This mode will, once enabled, enforce FIPS-validated crypto, disable USB connectivity in recovery mode and only allow Firmware Over the Air (FOTA) updates to the system. The Samsung Common Criteria mode APK (Samsung, 2017a) will enable CC mode on any supported model, for testing purposes. We will only focus on the parts of CC mode that affects SBOOT. The CC mode setting that affects SBOOT is either *on* or *off*.

After installing the APK and enabling CC mode, we can start to investigate how this affects SBOOT and how SBOOT enforces the blocking of ODIN mode. There are two ways of updating the firmware on our test devices; over-the-air through FOTA or with physical access through ODIN. Blocking ODIN mode is a crucial part of CC mode, since we'll see later that if we are given access to ODIN mode we can simply disable CC mode altogether. It is expected to be more difficult for an attacker to install unauthorised firmware updates through FOTA, as this is an online feature with secure communication to Samsung firmware servers.

With a combination of static reverse engineering of SBOOT and dynamic reverse engineering using an SBOOT exploit, we have analysed how SBOOT is affected by enabling CC mode, how this setting is stored, and how to attack it to disable CC mode. Our analysis shows that the CC mode setting is stored in flash, on a data partition called *PARAM*.

The PARAM partition

An Android device's storage is divided into several logical partitions, where *system* and *userdata* are the most important ones. The first contains the Android operating system files (OS) and the latter contains most of the user data. Another partition, the *PARAM* partition, is a rather small logical partition that contains a few JPG pictures used by SBOOT, e.g. the Samsung Galaxy boot logo displayed when the device is powered on. In addition, there is a file, *adv-env.img* (See Table 1), that a.o. contains parameters submitted to the Android kernel when SBOOT passes the execution to the Android kernel after SBOOT has loaded and checked the signature of the Android kernel. The PARAM partition is upgraded through firmware updates, with updates in the file *param.bin*, which is part of the firmware archives.

PARAM is however also storage for some other important settings. Our analysis show that in the last few 512-byte blocks of the PARAM partition, Samsung stores important settings like CC mode,

Table 1
SBOOT environment variables, stored in *adv-env.img*.

Index	Name	Example setting
0	REBOOT_MODE	0
1	SWITCH_SEL	3
2	DEBUG_LEVEL	18505
3	SUD_MODE	0
4	DN_ERROR	0
5	CHECKSUM	3
6	ODIN_DOWNLOAD	1
7	SALES_CODE	0
8	SECURITY_MODE	1526595585
9	NORMAL_BOOT	0
10	CP_DEBUG_LEVEL	22015
11	USERBOOT_MODE	0
12	DIAG_MODE	0
13	CHARGING_MODE	48
14	INT_RSVD14	0
15	LCD_RES	1
16	CMDLINE	console = ram loglevel = 4
17	BARCODE_INFO	(null)
18	KEEP_LOG	(null)

MDM settings, the current system status ("Samsung Official" or "Custom") and flags named AFW and UCS. The way SBOOT address these settings, is to count backwards in number of 512-blocks from the end of the PARAM partition. See Fig. 2.

To access the CC mode setting on our primary test device, SBOOT references the block `PARTITIONSIZEinBLOCKs (PARAM)_-_4` in the PARAM partition. This is a static offset in the SBOOT code, but can change with different versions of the SBOOT binary. The CC mode configuration is 64 bytes of encrypted data stored at the start of the referenced block. These 64 bytes are read and decrypted with a function we call`S_CC_decrypt`.[3] As the function `S_CC_decrypt` only takes two arguments; the input buffer with the 64 encrypted CC mode bytes, and a zero-initialised output buffer to receive the decrypted content, we assume the key material needed for the decryption is contained in, or retrieved by, the function itself. Looking closer at the `S_CC_decrypt`, we can see it uses a whitebox AES Cipher, where the AES key is not exposed during encryption or decryption (Chow et al., 2003). This means that the function `S_CC_decrypt` can decrypt the CC mode data without exposing the key in static code or dynamic runtime analysis. The function is a decryption oracle. SBOOT also contains the corresponding `S_CC_encrypt`, though our analysis does not find this function to be called by the SBOOT binary. As it turns out, a native Android library, `/system/lib64/libSecurityManagerNative.so` matches the two WAES encrypt/decrypt oracles, discovered also by André Moulu (2016). Since SBOOT does not seem to call `S_CC_encrypt`, this leads us to think that the CC mode configuration is only written by the Android environment and not by the SBOOT bootloader. SBOOT simply queries the configuration. Given these WAES oracles, we can freely read and write the CC config from both the SBOOT and the Android environment, if we control execution. This will also be the case if we have other means of write access to the PARAM partition.

The decrypted CC mode data contains the magic characters `timg` in bytes `0–3` and the characters `NOCC` (`CCON` read little-endian) or `FFOC` (`COFF` read little-endian) in bytes `4–7`, signalling CC mode on or off respectively.

The CC mode setting is read early in SBOOT's execution and a global flag variable is set to signal the CC mode configuration. This flag can be queried through a function we call `S_CC_MODE_isSet`, which returns true if CC mode is enabled.

Another setting assumed to be important for MDM managed devices, is the `MDM setting`. This setting is not set by the Samsung CC mode APK. However, we assume that MDM solutions use this setting actively, and we therefore include this setting and its effect on CC mode in our analysis. The MDM setting is stored in block `PARTITIONSIZEinBLOCKs (PARAM)_-_3` in the PARAM partition. It is an unencrypted setting in the first 32 bytes in the corresponding block. These bytes are read in during boot, as with CC mode, and sets a global variable corresponding to the MDM setting in PARAM. The MDM global variable can have three different values, where `1` and `2` seems to mean that MDM is in use.

The following pseudo code describes the different byte values in the MDM block and the corresponding MDM setting:

The MDM setting is set to `1` if `block[30] == 2 && block[31] == 6 && block[3] == 8 && block[7] == 8`.

The MDM setting is set to `2` if `block[30] == 2 && block[31] == 6 && block[3] != 8 && block[7] == 8`.

The MDM setting is set to `3` if `block[30] == 2 && block[31] == 6 && block[3] == 8 && block[7] != 8`.

[3] All functions named by the authors are prefixed with `S_`. Function names comes from educated guesses made from error message strings referenced inside functions.

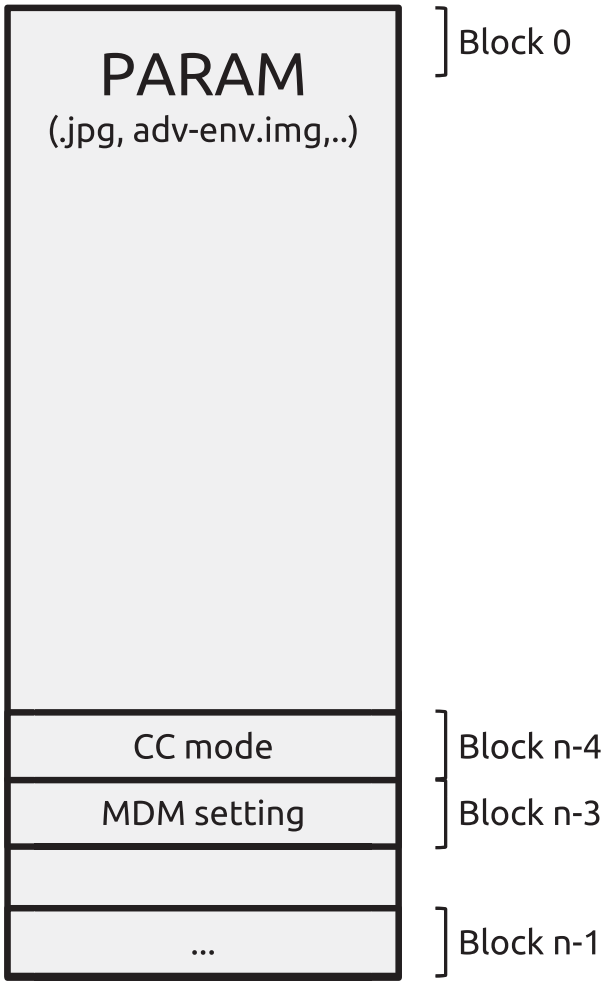

Fig. 2. PARAM partition.

So if the global MDM variable in SBOOT is set to `1` or `2`, `MDM mode` is considered enabled and this will also affect how SBOOT permits access to ODIN mode.

SBOOT enforcing CC mode

`S_CC_MODE_isSet` is called at three different locations in SBOOT; when the phone is trying to enter ODIN mode, when the phone tries to boot a kernel with no or invalid signature, and when the SBOOT sends status variables to the TrustZone. It's the first of these three that is crucial for denying access to ODIN mode.

The decision of *when* during the boot process to check for CC mode is crucial for the success of denying access to ODIN mode. As the bootloader is responsible for initialising the device as well as firmware updates to the system, it must also check for potential errors. These checks for errors are intermingled with security related checks, such as checking for CC mode. This intermingling makes the security checks vulnerable to changes in the execution path caused by errors, as specific errors can make the execution path change, such that certain security checks never takes place. Our analysis shows that this situation can arise for CC mode. Fig. 3 shows the pseudo code for the function responsible for enabling ODIN mode, `S_boot_enter_download_mode`. The function `S_boot_enter_download_mode` is called from various locations in the SBOOT boot process. It will check if it should go to ODIN mode. If so, it will call a function, `S_USB_mode_enter`, which will

```
1  S_boot_enter_download_mode(int reason){
2    if (reason == 1) {
3      S_draw_image("warning_L.jpg");
4        if (S_user_cancel()){
5          S_reboot_device();
6        }
7        else {
8          S_draw_image("download_L.jpg");
9        }
10   }
11   else {
12     if (reason == 6) {
13       S_draw_image("download_error_L.jpg");
14       S_USB_mode_enter(0);
15     }
16     S_draw_image("download_L.jpg");
17     if (reason == 3) {
18       // SUD mode ..
19     }
20     if (reason != 4) {
21       goto error();
22     }
23   }
24   //...
25   if (S_CC_MODE_isSet()) {
26     S_screen_print(
27         "DOWNLOAD_IS_BLOCKED_BY_CC_MODE");
28     S_sleep(1000);
29     S_power_off_device();
30   }
31   S_USB_mode_enter(0);
32 }
```

Fig. 3. Simplified pseudo code of `S_boot_enter_download_mode`.

apply the configuration to the device and switch to ODIN mode. `S_USB_mode_enter` can be called with one parameter, where `0` means ODIN/download mode.

`S_boot_enter_download_mode` itself receives one parameter, reason (Fig. 3, line 1), which indicates the reason for entering ODIN mode. The function acts accordingly based on the value of this parameter. The pseudo code shows that `S_USB_mode_enter` is called from two different locations. The call in line `31` is only reached if the call to the function `S_CC_MODE_isSet` in line `25` returns `false`. If `S_CC_MODE_isSet` returns `true`, SBOOT will print "DOWNLOAD IS BLOCKED BY CC MODE" to the device screen and eventually power off the device. The other call to `S_USB_mode_enter` is found at line `14`, and this call is not preceded by a call to `S_CC_MODE_isSet`, meaning this call seems to ignore if any CC mode is set. So if `S_boot_enter_download_mode` is called with parameter 6, the device will enter ODIN mode, even if CC mode is set. Backtracking callers to `S_boot_enter_download_mode` identifies the situation in where this happens.

SBOOT has a table of environment variables, stored in the *adv-env.img* file of the PARAM partition. These values are listed in Table 1. These values are ways of influencing the execution of SBOOT and since they are stored in PARAM, they survive a device reboot. As already mentioned, some of these are also passed on as parameters to the Android kernel, but the discussion of these are outside the scope of this article.

One example is the `REBOOT_MODE`, signalling SBOOT which boot mode to use, where normal boot (`0`), ODIN/download mode (`1`), upload mode (`2`) and recovery mode (`4`) are example settings. See Table 2 for a full listing of values for `REBOOT_MODE`.

Many of these environment variables are checked during boot, and one of them is interesting with respect to CC mode; `DN_ERROR`.

Table 2
REBOOT_MODE variable values.

Name	Value
REBOOT_MODE_NONE	0
REBOOT_MODE_DOWNLOAD	1
REBOOT_MODE_UPLOAD	2
REBOOT_MODE_CHARGING	3
REBOOT_MODE_RECOVERY	4
REBOOT_MODE_FOTA	5
REBOOT_MODE_FOTA_BL	6
REBOOT_MODE_SECURE	7
REBOOT_MODE_FWUP	9
REBOOT_MODE_EM_FUSE	10

During normal boot, SBOOT calls a rather complex function, we call `S_boot_set_boot_mode`, that decides which boot mode to choose for the device. This is based on numerous checks on hardware, battery state, environment variables and so on. During these tests there's a call to a function, `S_s5p_check_download`, which will return a value different from 0 if it should go to ODIN/download mode and returns 0 if it should not go to download mode. Taking the path in which `S_boot_set_boot_mode` returns `0`, no ODIN/download mode, takes us to a call to a function we call `S_env_get`, called with the integer value of `DN_ERROR` from Table 1. This call returns the integer value of the environment variable `DN_ERROR`. If this is set, there is a call to the already discussed function `S_boot_enter_download_mode`, with the parameter value of `6`. As we have already analysed `S_boot_enter_download_mode` and located a bypass of the CC mode check if the input parameter is equal to `6`, we now have a way to bypass the CC mode check. We can simply set the environment value `DN_ERROR` to a non-zero value and reboot the device. The device will enter ODIN mode, even if CC mode is enabled. This seems to us like an emergency ODIN mode, where the bootloader is requiring a firmware update as the result of a failed firmware update. Fig. 4 shows the screen shown on a Galaxy S7 Edge (`SM-G935F`) when in this mode.

MDM mode

We include a discussion on the MDM mode for completeness as this setting is expected to be used by some MDM solutions that supports Samsung Exynos devices. MDM mode also affects how SBOOT prevents access to ODIN mode. SBOOT access this setting through a function we call `S_MDM_MODE_isSet`. This function will return true (`1`) if the global MDM variable is either `1` or `2`. It does not seem to make a distinction between the two. SBOOT calls this function from three different functions; when in ODIN mode, when

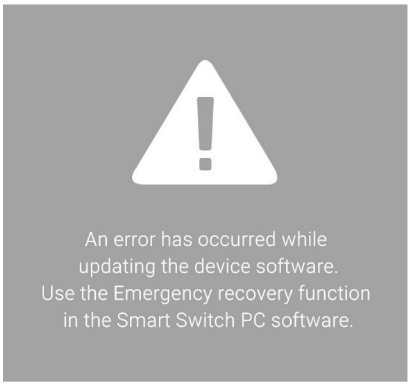

Fig. 4. *Emergency* ODIN/download mode.

booting, and for providing the MDM setting to the TrustZone. In ODIN mode SBOOT is checking for MDM mode and will prevent firmware upgrade to any partition other than the SBOOT ("BOOTLOADER") itself, if MDM mode is set. So we can see that while CC mode prevents access to ODIN mode itself, MDM mode will be checked in ODIN mode as well.

SBOOT also checks for MDM mode during boot. These checks are done *before* the CC mode check done in `S_boot_enter_download_mode`, already discussed in Section "Samsung CC mode and SBOOT" Checks are done inside the function called `s5p_check_download`, where the SBOOT decides if it should call `S_boot_enter_download_mode`. Note that the call to `s5p_check_download` is done before the check for the environment variable `DN_ERROR` is done, which is only checked if `s5p_check_download` returns a non-zero value, meaning no ODIN mode. We have already seen in Section "Samsung CC mode and SBOOT" that setting the environment variable `DN_ERROR` to a non-zero value will put the device in an emergency ODIN mode. This will then be the same situation even if MDM mode is set. One important difference is that ODIN mode with MDM mode set will only allow firmware updates of the SBOOT partition.

As MDM mode is an unencrypted setting stored in the PARAM partition, any erasing or overwriting of the MDM setting block will reset and disable the MDM setting on the device, and therefore not prevent access to ODIN mode anymore.

Unauthorised disabling of CC mode

Based on our analysis of SBOOT, and the way the bootloader enforces CC mode to prevent ODIN/download mode, we have found three different attacks to disable CC mode.[4] All attacks have been verified through successful tests on our test device. We will also discuss the effect of an optional MDM setting, although this was not enabled by the Samsung CC mode APK.

Modifying the PARAM partition

As discussed in section "The PARAM partition", we can modify the CC mode setting if we have *write* access to the PARAM partition. This can be either by physical access to the underlying flash storage, or through ODIN/download mode. Given PARAM write access, we can simply change the CC mode setting and encrypt the setting with the WAES encryption oracles from either SBOOT, or `/system/lib64/libSecurityManagerNative.so`. We can also simply overwrite the CC mode data bytes with zero bytes, with the same effect. So simply flashing a stock `param.bin` from a corresponding firmware upgrade archive will disable CC mode.

If MDM mode is also present, flashing a stock PARAM through ODIN mode is denied. This is not the case with physical access to the underlying flash storage, as overwriting the MDM setting block will disable MDM mode.

SBOOT exploitation

Since the bootloader is responsible for reading and enforcing the CC mode setting in the PARAM partition, any attack on the execution flow of SBOOT will have the potential to bypass CC mode and enable ODIN/download mode. Based on a vulnerability discovered by Nitay Artenstein (2017), we have developed a fully functional exploit to make SBOOT ignore the CC mode settings. One way of doing this is to patch the code flow of SBOOT to call `S_boot_enter_download_mode` with the parameter 6 and then overwrite the PARAM partition in the same way as detailed in section "Modifying the PARAM partition" Another way could be to patch the `S_CC_MODE_isSet` function to always return false by either patching the return code to `0` or by changing the global variable it references to `0`. This way we can bypass the blocking of booting unofficial and unsigned kernels, and the CC mode enabled setting is not reported to TrustZone before booting the kernel, in effect disabling CC mode on the booted Android system.

If MDM mode is also activated, this can also be bypassed simply by setting the global SBOOT MDM mode setting to 0, resulting in the function `S_MDM_MODE_isSet` always returning false. This is expected, as any arbitrary changes to the SBOOT code and execution flow will leave all security checks done by SBOOT ineffective.

Setting `DN_ERROR`

We have seen the effect of setting the `DN_ERROR` environment variable to a non-zero value in section "SBOOT.enforcing CC mode" This has been tested through a console interface provided by SBOOT. This console can be reached with the aid of a custom USB connector and a simple RS232-to-USB serial converter (Nitay Artenstein, 2017). After entering the SBOOT console, one can simply type `set env DN_ERROR 2` followed by a `save env` and `reset`. This will try to reboot the device with a normal boot, but the non-zero `DN_ERROR` environment variable will force the device into an emergency ODIN/download mode. From here we can modify the PARAM partition like in section "Modifying the PARAM partition".

If MDM mode is also active, the device will still enter ODIN mode. However, since the MDM mode is also checked by ODIN mode when flashing firmware, only changes to the bootloader, SBOOT, are allowed. This will prevent this attack.

Conclusion

In this paper we have successfully demonstrated how to disable Common Criteria (CC) mode on selected Samsung devices. The effect of disabling the CC mode increases the device's attack surface and can further be used in forensic acquisition. This will open up the device for misuse of the firmware update protocol for direct storage or RAM acquisition, in addition to both signed and unsigned firmware updates through ODIN/download mode, depending on the *Factory Reset Protection* and *Rollback Protection* settings on the device. If one uses the SBOOT exploit attack from section "SBOOT exploitation" we can easily avoid both of these defences as well. This is because these security settings are also enforced by the SBOOT bootloader and can therefore easily be changed/disabled.

We have found and tested the effect of several weaknesses in the enforcing of CC mode on our tested device. Using exploits to attack SBOOT will break the chain-of-trust anchored in the boot process. This will break the trust in all code running in the same *normal* world on the application processor on this device. With such a powerful attack we can replace or adapt to most of the security features of SBOOT. This is not unexpected, but emphasises the need for a secure bootloader and chain-of-trust.

As future work we suggest testing these attacks on actual MDM solutions utilising the Samsung CC mode feature and/or the MDM setting. The effect on a MDM solution after disabling CC mode by changing or erasing the CC mode setting and/or the MDM setting in the PARAM partition has not been tested and could lead to other attack scenarios of forensic interest.

[4] We have not considered if or how any commercial forensic tools support bypassing CC mode.

Acknowledgements

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the R&D project Ars Forensica grant agreement 248094/O70.

References

André Moulu, 2016. How to Lock the Samsung Download Mode Using an Undocumented Feature of Aboot. https://ge0n0sis.github.io/posts/2016/05/how-to-lock-the-samsung//download-mode-using-an-undocumented-feature-of-aboot/. (Accessed 30 July 2017).

Carrier, B., 2002. Defining digital forensic examination and analysis tools. Int. J. Digit. Evid. 1, 2003.

Chow, S., Eisen, P.A., Johnson, H., Oorschot, P.C.V., 2003. White-box cryptography and an aes implementation. In: Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography. Springer-Verlag, London, UK, UK, pp. 250–270. http://dl.acm.org/citation.cfm?id=646558.694920.

Garfinkel, S.L., 2010. Digital forensics research: the next 10 years. Digit. Investig. 7, S64–S73. https://doi.org/10.1016/j.diin.2010.05.009. https://doi.org/10.1016/j.diin.2010.05.009.

Guido, M., Buttner, J., Grover, J., 2016. Rapid differential forensic imaging of mobile devices. Digit. Invest. 18, S46–S54. https://doi.org/10.1016/j.diin.2016.04.012. http://www.sciencedirect.com/science/article/pii/S1742287616300457.

Ilfak Guilfanov, 2017. Ida-Disassembler & Decompiler. https://hex-rays.com/. (Accessed 27 July 2017).

Kanonov, U., Wool, A., 2016. Secure containers in android: the samsung knox case study. In: Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices. ACM, New York, NY, USA, pp. 3–12. https://doi.org/10.1145/2994459.2994470. http://doi.acm.org/10.1145/2994459.2994470.

Keystone team, 2017. Keystone-the Ultimate Assembler. http://www.keystone-engine.org/. (Accessed 28 September 2017).

Nitay Artenstein, 2017. Exploiting Android S-boot: Getting Arbitrary Code Exec in the Samsung Bootloader (1/2). http://hexdetective.blogspot.no/2017/02/exploiting-android-s-boot-getting.html. (Accessed 27 July 2017).

Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2017. Bootstomp: on the security of bootloaders in mobile devices. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC, pp. 781–798. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini.

Samsung, 2017. Samsung Android 7 on Galaxy Devices Guidance Documentation. https://docs.samsungknox.com/CCMode/Adminguidev3_0.pdf. (Accessed 7 September 2017).

Samsung, 2017a. Common Criteria Mode Apk. https://www.samsungknox.com/en/article/common-criteria-mode. (Accessed 28 July 2017).

Samsung, 2017b. Developer Tools Overview. https://seap.samsung.com/sdk. (Accessed 27 July 2017).

Samsung, 2017c. Guidance Documents for Common Criteria Evaluation. https://support.samsungknox.com/hc/en-us/articles/115015114407. (Accessed 5 January 2018).

Samsung, 2017d. Platform Security. http://developer.samsung.com/tech-insights/knox/platform-security. (Accessed 28 July 2017).

Scrivens, N., Lin, X., 2017. Android digital forensics: data, extraction and analysis. In: Proceedings of the ACM Turing 50th Celebration Conference-China. ACM, New York, NY, USA, p. 26. https://doi.org/10.1145/3063955.3063981, 1–26:10. http://doi.acm.org/10.1145/3063955.3063981.

Yang, S.J., Choi, J.H., Kim, K.B., Chang, T., 2015. New acquisition method based on firmware update protocols for android smartphones. Digit. Invest. 14, S68–S76. https://doi.org/10.1016/j.diin.2015.05.008. http://www.sciencedirect.com/science/article/pii/S1742287615000535. the Proceedings of the Fifteenth Annual DFRWS Conference.

Yang, S.J., Choi, J.H., Kim, K.B., Bhatia, R., Saltaformaggio, B., Xu, D., 2017. Live acquisition of main memory data from android smartphones and smartwatches. Digit. Invest. 23, 50–62. https://doi.org/10.1016/j.diin.2017.09.003.