Secondary Stage Boot Loader

# Introduction

The Second Stage Boot Loader (SSBL) is a piece of firmware that can be installed in Talaria TWO to enhance the flexibility of booting the applications on the device. The SSBL enables the following features on Talaria TWO:

1. Selective boot of one of the applications loaded into flash.
2. Over the Air (OTA) update of applications (requires additional OTA application).

SSBL can be built with secureboot which provides a secure way of loading encrypted and signed applications. This prevents loading of unauthorized applications and performing a flash readout of application contents.

# Description of Operation

The Second Stage Boot Loader (SSBL) is a special application that is written to Talaria TWO’s flash. On boot up, the primary bootloader loads & starts SSBL. SSBL reads the image index from the boot.json file, parses the part.json file and picks the image info from the array index read from boot.json file. The SSBL then loads the image from the sector mentioned in part.json into the RAM. Applications supported by the SSBL are stripped ELF files written to flash memory.

In case of secureboot mode, the configuration files are encrypted.

The memory layout mentioned in section 5.1 is for the RAM, where the SSBL and the application triggered by the SSBL is loaded into memory for execution. Section 5.2 explains the flash layout where SSBL and multiple applications can be stored in the flash.

## Memory Layout

Figure 1 shows the memory layout when using SSBL.

0x40000

SSBL

Bootargs

Reserved

App Area

.text

.data

.bss

0xbfffc

0x42000

0x90000

Figure 1: Memory layout on loading the SSBL application

1. There is a total of 512KB RAM in Talaria TWO
   1. The RAM starts at 0x40000 and ends at 0xc0000.
2. User Application area
   1. Starts at 0x42000 to 0x90000.
   2. Contains application .text, .data and.bss sections.
3. Bootargs
   1. The memory location for bootargs is at 0xbfffc and it grows backwards.
4. SSBL area
   1. Starts at 0x90000.

App Area

.text

.data

.bss

heap

0x42000

0xbfffc

0x40000

Reserved

Bootargs

Figure 2: Memory layout after loading the application

Figure 3 shows the signed and encrypted ELF memory layout when using SSBL in secureboot mode.

A picture containing diagram

Description automatically generated

Figure : Signed and encrypted ELF memory layout

1. In this case, only the .text and .data sections of the application ELF are encrypted.
2. The .virt segment cannot be encrypted. Ensure no sensitive code is placed in this section of the memory layout.
3. Code sections can be forced into .text by either specifying in the linker script or by adding \_\_ramcode in the function declaration.

|  |
| --- |
| int \_\_ramcode  main(void)  {  ... |

## Flash Layout

Figure 4 shows the layout of flash memory when using the SSBL. To use the SSBL, flash must contain at minimum the SSBL, the filesystem, and one application.

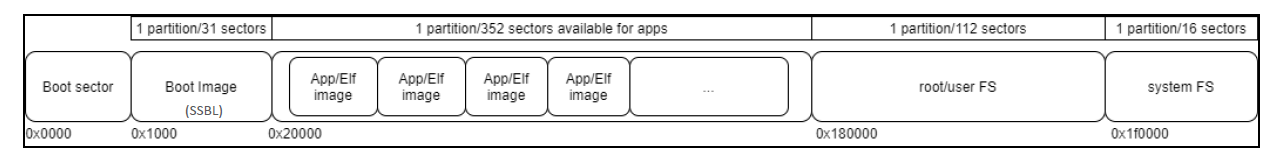


Figure 4: Flash layout when using the SSBL

Figure 5 shows the layout of flash memory when using secure SSBL.

Text

Description automatically generated with medium confidence

Figure : Flash layout for SSBL with secureboot

## SSBL Operation Flow

### Non-Secure SSBL

Boot ROM

Load SSBL at 0x1000

Mount Filesystem

Read boot.json file and get “boot Index”

Read Part.json file and get image info at “boot index”

Read ELF (from image info), Parse ELF to retrieve segments (.bss, .txt etc.)

Get VM sector address for .virt segment

Set VM sector address as Boot params

Load the app

Run loaded app

Copy .txt, .data,. bss to RAM

Figure : Non-Secure SSBL Flow Diagram

### Secureboot SSBL

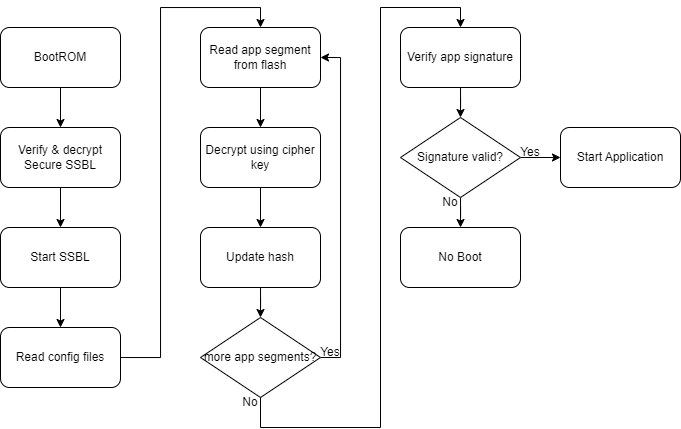


Figure : Secureboot SSBL Flow Diagram

## SSBL Configuration

The SSBL is configured with JSON files present in the flash-based filesystem. Table 1 provides a description of the relevant files and their purpose. The contents of these files can be updated at installation time or by a running application to modify the behavior of the SSBL.

|  |  |
| --- | --- |
| **File** | **Purpose** |
| part.json | 1. Image table which is a json array of applications image information. Each element in the image array gives information like image name starting sector of the elf, boot arguments etc. 2. Application boot arguments 3. Additional SSBL options |
| boot.json | This is a json file stored in root/user FS. It contains the image index. This is the index in the image information array present in part.json file. SSBL gets the index of the image to be loaded from this file. |

Table 1: SSBL Configuration Files

**Note**: For SSBL in secureboot mode, the configuration files are encrypted.

**part.json**

|  |
| --- |
| {  "image" : [  {  "name" : "iperf\_vm",  "version" : "1.0",  "start\_sector" : 32,  "bootargs\_start": 1,  "ssid" : "innotest",  "passphrase" : "123467890",  "bootargs\_end" : 1  },  {  "name" : "hello\_world",  "version" : "1.0",  "start\_sector" : 232,  "bootargs\_start": 1,  "ssid" : "innotest",  "passphrase" : "123467890",  "bootargs\_end" : 1  }  ],  "baudrate" : 2560000,  "timeout" : 0,  "verbose" : 1  } |

1. General parameters:
   1. baud – baud rate used by SSBL when using hio
   2. timeout – timeout used by SSBL when using hio
   3. verbose – verbosity mode
   4. image []: image information
2. Image information:
   1. name: name of application
   2. version: version number of applications
   3. sector: start sector of image in flash
   4. bootargs\_start: The following objects will be boot params
   5. bootargs\_end: end of boot arguments

**boot.json**

|  |
| --- |
| boot.json  { image : 0 } |

where,

image – The image to boot from part.json

## SSBL Boot Arguments

SSBL can pass bootargs to an application by utilizing the filesystem. SSBL reads the bootargs from the part.json file and stores the bootargs at memory location 0xbfffc where it grows backwards. The size occupied by the bootargs is dependent on the length and count of the bootargs read from the filesystem. Figure 8 shows how they are stored in memory.

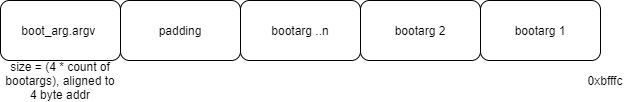


Figure 8: SSBL Bootargs stored in memory

# Building Components

This section describes building the required components for the SSBL.

## Create File System (root.img) file

The root folder at root\_fs contains the files that will be put into the filesystem image to be flashed to Talaria TWO. Before building the filesystem image for the first time, the configuration files need to be updated based on the applications you will load and your particular use of the SSBL (refer section 5.3.2).

Once the SSBL configuration files are updated, run the following command from within the root folder to build the filesystem image:

**Non-secure SSBL**:

For non-secure SSBL, filesystem files come from root\_fs and the fs directory of the application.

|  |
| --- |
| ~/sdk\_x.y/script$ python3 ./build\_rootfs\_generic.py --folder\_path apps/ssbl |

**Note**: x and y in sdk\_x.y refer to the SDK release version. For example: *sdk\_2.4/*.

**Secureboot SSBL**:

For secureboot SSBL, filesystem files come from root\_fs and the fs\_secure directory of the application.

|  |
| --- |
| python3 ./script/build\_rootfs\_generic.py --folder\_path examples/secure\_files/ --secure True --keyfile ./apps/ssbl/enroll.json |

# Flashing Components

After the SSBL, filesystem, and applications have been built, follow the instructions in this section to flash the components to Talaria TWO.

**Note**: If Talaria TWO has been flashed before, connect GPIO17 to ground on the peripheral header of the EVK, then press and release reset before following the instructions here. This will inhibit flash boot and allow the flash helper to be loaded, provided fuses have not already been blown.

## Non-Secure SSBL

### Flashing

The following commands will write the SSBL and other components to flash. Run the commands from the sdk\_x.y directory:

**Load flash helper**

|  |
| --- |
| ./script/boot.py --device /dev/ttyUSB2 --reset=evk42\_bl ./apps/gordon/bin/gordon.elf |

**Invalidate the boot Image**

|  |  |
| --- | --- |
| |  | | --- | | dd if=/dev/zero of=./empty.img bs=1K count=1 | |

|  |
| --- |
| ./script/flash.py --device /dev/ttyUSB2 write 0x1000 ./empty.img |

**Write partition**

|  |
| --- |
| ./script/flash.py --device /dev/ttyUSB2 from\_json ./tools/partition\_files/ssbl\_part\_table.json |

**Flash SSBL**

|  |
| --- |
| ./script/flash.py --device /dev/ttyUSB2 write 0x1000 ./apps/ssbl/fast\_ssbl.img |

**Flash filesystem**

|  |
| --- |
| ./script/flash.py --device /dev/ttyUSB2 write 0x180000 ./apps/ssbl/root.img |

**Flash apps**

iPerf3 should be flashed into 0x2000 (which is start\_sector 32 as mentioned in part.json), while helloworld.elf should be flashed into 0xE8000 ( which is start\_sector sector 232).

|  |
| --- |
| ./script/flash.py --device /dev/ttyUSB2 write 0x20000 ./apps/iperf3/bin/iperf3.elf  ./script/flash.py --device /dev/ttyUSB2 write 0xE8000 ./apps/helloworld/bin/helloworld.elf |

Open a miniterm at baud rate of 2457600 and reset the EVB.



Figure : Miniterm console output

Reset the board either by executing the following command or by pressing the reset button on the EVB.

|  |
| --- |
| ./script/boot.py --device /dev/ttyUSB2 --reset=evk42 |

### Expected Output

|  |
| --- |
| Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PWAEWWWWAE Build $Id: git-a042e9a42 $  Build $Id: git-a042e9a42 $  vm.flash\_location=0x00034c00 sys.reset\_reason=1 passphrase=1234567890 ssid=innotest  addr f8:e9:43:d2:00:e7  network profile created for ssid: innotest  [1.535,586] CONNECT:60:32:b1:33:b5:7b Channel:11 rssi:-37 dBm  [4.370,448] MYIP 192.168.0.107  [4.370,495] IPv6 [fe80::fae9:43ff:fed2:e7]-link  IPerf3 server @ 192.168.0.107  ----------------------------------------  Iperf3 TCP/UDP server listening on 5201  ---------------------------------------- |

Run iPerf3 client for this application.

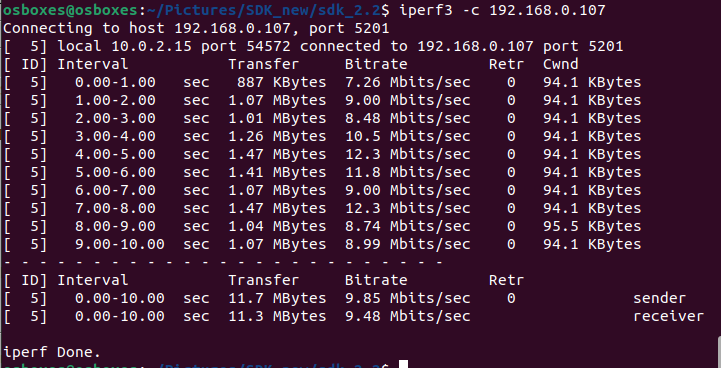


Figure : iPerf3 Client

To run the helloworld application, make changes in sdk\_x.y/root\_fs/root/boot.json and execute the following to generate the root image:

|  |
| --- |
| sdk\_x.y/root\_fs/root$ python3 ../../script/build\_rootfs\_generic.py --folder\_path apps/ssbl |

Repeat the steps in section 7.1 and reset the board. On reboot, the new application will be loaded.

## Secure SSBL

### Building

1. Build secure SSBL

|  |
| --- |
| #for development  cd <sdk>/apps/ssbl$  make clean  make KEY=enroll.json SECUREBOOT=1 DEBUGSECURE=1  #for production  cd <sdk>/apps/ssbl$  make clean  make KEY=enroll.json SECUREBOOT=1 |

1. Build data filesystem

|  |
| --- |
| cd <sdk>  python3 ./script/build\_rootfs\_generic.py --folder\_path examples/secure\_files/ --secure True --keyfile ./apps/ssbl/enroll.json |

### Flashing and Testing

1. Generate combined "First" application and SSBL
   1. For emulating/testing SecureSSBL

|  |
| --- |
| cd <sdk>/apps/ssbl$  make KEY=enroll.json SECUREBOOT=1 DEBUGSECURE=1 |

* 1. For production

|  |
| --- |
| cd <sdk>/apps/ssbl$  make KEY=enroll.json SECUREBOOT=1 |

1. Enroll keys
   1. For emulating SecureSSBL without burning the fuse

|  |
| --- |
| cd <sdk>$  ./script/boot.py --reset=evk42\_bl apps/gordon/bin/gordon.elf  cd <sdk>/apps/ssbl$  ../../script/flash.py enroll --keyfile=enroll.json --secureboot puf --fuse-location emulated |

* 1. For production SecureSSBL and burning the fuse

|  |
| --- |
| cd <sdk>$  ./script/boot.py --reset=evk42\_bl apps/gordon/bin/gordon.elf  cd <sdk>/apps/ssbl$  ../../script/flash.py enroll --keyfile=enroll.json --secureboot puf --fuse-location one-time-programmable-fuses |

1. Flash SSBL partition table

|  |
| --- |
| cd <sdk>$  ./script/flash.py from\_json tools/partition\_files/ssbl\_part\_table.json |

1. Flash both.img at 0x1000
   1. For emulating SecureSSBL

|  |
| --- |
| cd <sdk>/apps/ssbl$  ../../script/flash.py write 0x1000 out/both.img |

* 1. For production SecureSSBL

|  |
| --- |
| cd <sdk>/apps/ssbl$  ../../script/flash.py write 0x1000 out/ssbl\_secure.img |

1. Build filesystem

|  |
| --- |
| cd <sdk>  python ./script/build\_rootfs\_generic.py --folder\_path examples/secure\_files/ --secure True --keyfile ./apps/ssbl/enroll.json |

1. Create signed application

**Note**: When building the application, build it as <app>.elf.sign and pass it as the KEY parameter

|  |
| --- |
| cd <sdk>/apps/ssbl$  make KEY=enroll.json <app>.elf.sign |

1. Encrypt application

**Note**: Move <app>.elf.sign into SSBL directory

|  |
| --- |
| cd <sdk>/apps/ssbl$  make KEY=enroll.json <app>.elf.enc |

1. Flash application at 0x20000

|  |
| --- |
| cd <sdk>/apps/ssbl$  ../../script/flash.py write 0x20000 app.elf.enc |

1. Flash filesystem

|  |
| --- |
| cd <sdk>/apps/ssbl$  ../../script/flash.py write 0x180000 root\_secure.img |

### Expected Output

When DEBUGSECURE=1

|  |
| --- |
| Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWAE  FIRST:SWWWWAHESi Build $Id: git-a042e9a42 $  \*\*\*Warning! Make sure to remove this code section once in production\*\*\*  secureboot\_secret:  8b5678a045ba66b7ea956d3292aae8dc29ded8de9010efd40980a091734b786b11000000  \*\*\*Warning! Make sure to remove this code section once in production\*\*\*  cipher key: 4e3b0b9792183c53ecc78a38c64a45c071b97bc40b0baba308ed76db8a46cef1  public key: 20b003d2f297be2c5e2c83a7e9f9a5b9eff49111acf4fddbcc0301480e359de6dc809c49652aeb6d63329abf5a52155c766345c28fed3024741c8ed01589d28b  Build $Id: git-a042e9a42 $  vm.flash\_location=0x0002d000 sys.reset\_reason=1  Application Information:  ------------------------  Name       : Secure files demo application  Version    : 1.0  Build Date : Apr 20 2023  Build Time : 07:10:32  Heap Available: 248 KB (254360 Bytes)  Original message: Hello! This is a plain text file.  Writing message to encrypted file  Reading file as ciphertext  Cipher text message: 1~␒M}rQo앺{AÛ␒\*\_/rY0  Reading and decrypting file  Plain text message: Hello! This is a plain text file. |