Security Target for Secure Thingz Secure Boot Manager

Secure Thingz Ltd.

Version	Change date	Author	Notes
1	2019-09-12	Amyas Phillips	Complete first draft
2	2019-09-13	Amyas Phillips	Internal STz review
3	2019-09-27	Amyas Phillips	Address feedback from evaluator
4	2019-11-15	Amyas Phillips	Address feedback from evaluator
5	2019-12-03	Amyas Phillips	Revised vulnerability analysis
6	2019-12-11	Amyas Phillips	Minor corrections
7	2019-12-17	Amyas Phillips	Minor corrections
8	2023-02-14	Tanay Kore	3rd Party Component Audit
9	2023-06-09	Tanay Kore	Minor Corrections and addressing of action list comments
10	2023-10-13	Richard Turner	Address review comments, add references to new SBM Authentication document, correct errors.
11	2023-11-10	Richard Turner	Revise to suit SESIP 1.2.
12	2023-12-04	Tom Pearson	Clean up final table.
13	2024-02-06	Richard Turner	Address review comment from action item list 5.0.
14	2024-05-29	Richard Turner, Andrew Bott	Address feedback from certifier.
15	2024-06-17	Ed Beckett	Address division of functions between Tomcrypt and micro-ecc.
16	2024-07-12	Ed Beckett	Clarifies that micro_ECC is used to generate the shared secret.
17	2024-09-30	Ed Beckett	Address feedback from certifier.
18	2024-11-15	Christo Smallwood	Address feedback from certifier.
19	2024-11-29	Christo Smallwood	Address feedback from certifier.

20	2024-12-10	Christo Smallwood	Formatting text and links for consistency
21	2024-12-16	Ed Beckett	Formatting

Table of Contents

1 Introduction

The Security Target describes the Platform (in this chapter) and the exact security properties of the Platform that are evaluated under [SESIP] (in chapter “Security requirements and implementation”) and that a potential consumer can rely upon the product upholding if they fulfil the objectives for the environment (in chapter “Security Objectives for the operational environment”).

SESIP v1.2 is used in this document.

1.1 ST reference

See title page.

1.2 Platform reference

TOE name	Secure Thingz Secure Boot Manager
TOE version	3.00
TOE identification	3.00
TOE Type	Secure bootloader for microcontrollers used in IoT applications

1.3 Included guidance documents

The following documents are included with the platform:

Reference	**Name**	**Version**
[Manual]	Embedded Trust User Guide	14
[SBM Authentication]	SBM Authentication and Integrity	1.0

1.4 Platform functional overview and description

The Target of Evaluation (TOE) consists of a bootloader for microcontrollers (MCUs) called the Secure Boot Manager (SBM).

The TOE is intended to be used by an embedded C developer who integrates it into an Internet of Things (IoT) Product, along with an IoT Application and other parts of an IoT Platform.

In SESIP terms, the TOE is a component of the IoT Platform. The TOE scope is depicted in Figure **1**. The blue block is within the evaluated scope and the grey blocks are outside of the evaluated scope. Out of scope are the IoT Application, the other components of the IoT Platform including the RTOS and the microcontroller, the integrated development

environment (IDE) used to configure and build the TOE, and the programming system used to provision the TOE onto the microcontroller.

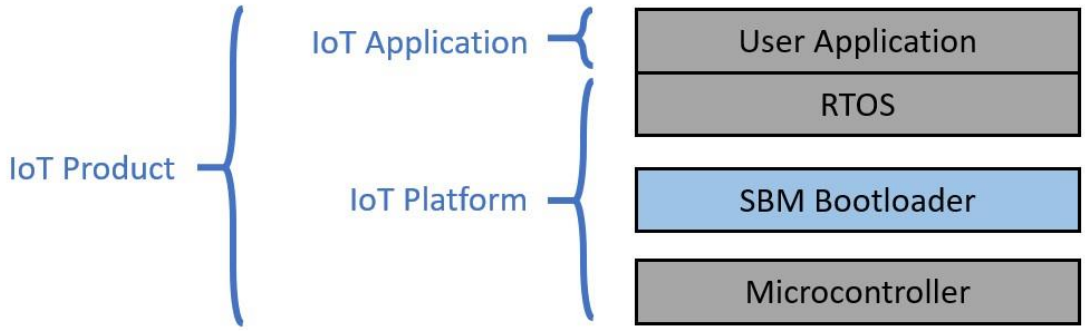


Figure 1 **TOE scope (in blue) in terms of SESIP**

The TOE is a specialised application that is run when the microcontroller starts, before the user application, to which it provides several security benefits. The main security features of the TOE are as follows:

- **Configuration of hardware security features**. The TOE ensures that hardware security features of each supported microcontroller are used effectively, for example to write-protect the TOE, set debug lock features, store unique device keys, and set lifecycle states.
- **Provisionable key store**. The TOE accesses a data structure in Flash memory called the Provisioned Data Block (PDB). This data structure includes device private keys, and trust anchor certificates, as specified by the composite developer. The PDB is created, personalised and programmed onto each device by a compatible provisioning system in the factory, providing each device with a unique cryptographic identity. Where suitable hardware features are available the TOE sets access permissions so that data in the PDB cannot be modified by the user application, before passing control to that application. Where hardware features for secure key generation and storage such as a PUF or silicon root of trust (ROT) are provided, device secret keys are generated and stored there instead of in the PDB.
- **Anti-cloning.** Multiple features protect firmware from being cloned onto other devices. Except during development, the TOE permanently activates debug lock features of the host microcontroller to prevent external readout or modification of the internal firmware. The device hardware ID is used to seed a hash of the provisioned data that is included in the TOE at provisioning time, allowing the TOE to verify that it is not running on a cloned device. The device hardware ID can also be incorporated into device identity certificates included in the provisioned data, allowing the user application to verify that it is not running on a cloned device.
- **Secure boot**. The TOE receives the program pointer immediately on device reset. The TOE checks that a signature over the main application image is present and

validates it against a trust anchor stored in the PDB, proving the authenticity of the main application image. Finally, the TOE passes the program pointer to the user application. Using microcontroller-specific features, the TOE write-protects itself to ensure that a compromised user application cannot persist itself by also compromising the trusted bootloader.

- **Firmware update**. The TOE includes an installer, which determines whether a more recent valid user application image is available, and if so, installs that before executing it. New images must be linked for a predetermined location in memory known as the active slot, packaged with version metadata, signed by a party trusted by the TOE, distributed to target devices, and stored in a predetermined location in memory known as the update slot by another process, usually the user application. The active slot is always located in internal Flash memory in execute-in-place (XIP) devices, aligned on erase sectors. The update slot can be located in internal Flash memory, also aligned on erase sectors, or it can be in external serial Flash. By having the installer copy new images into a fixed location before executing them, reliance on position-independent code (PIC), runtime linking, multiple versions of images linked for different locations, or knowledge of slot state outside of the IoT device is avoided. Rollbacks can be accomplished only by repackaging an older image with a newer version number and distributing that for installation.

The TOE is located at the reset vector of the microcontroller (see Figure **2**) so that on start it can perform a sequence of operations before passing control to the user application.

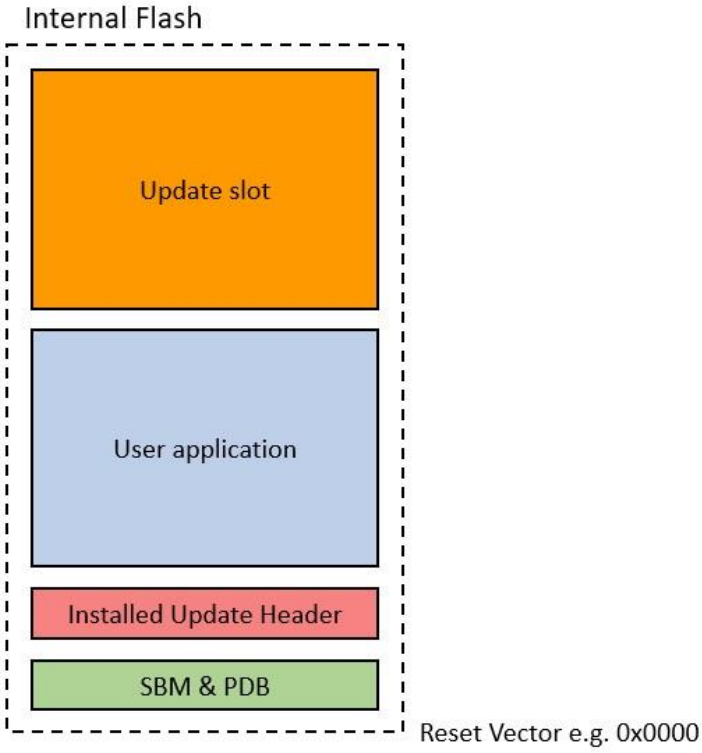


Figure 2 **Example flash memory layout of MCU devices using the TOE (the SBM bootloader).**

While the TOE must be located at the reset vector, the location of other slots is configurable, subject to hardware constraints and the SBM features enabled by the developer:

- The update slot may be located in external Flash.
- The user application is always located in internal Flash in the active slot and must be linked for that location.
- The Installed Update Header slot is used by the TOE to store metadata received with the user application image, including version information and a signature.
- All slots in internal Flash are aligned on Flash erase sectors so that they and/or their neighbouring slots can be independently updated.

Knowledge of the memory map is built into the TOE. The TOE makes use of a default memory map and other hardware-specific features for each supported microcontroller. Supported microcontrollers are listed in Table 1.

Vendor	**MCU family**	**Released MCU variant** (excluding package variants)	**Write Protect**	**Hardware Lockdown**	**Hardware Security Support**
Microchip	SAML11	SAML11E16A	yes	✓	Trustzone
Nordic	nRF52	nRF52840	no	✓	-
NXP	LPC5500	LPC55S06	yes	✓	Trustzone, PUF
NXP	LPC5500	LPC55S16	yes	✓	Trustzone, PUF
NXP	LPC5500	LPC55S28	no	✓	PUF
NXP	LPC5500	LPC55S69	yes	✓	Trustzone, PUF
NXP	i.MX1064	MIMxRT1064xxx6A	no	✓	-
NXP	K22	MK22FN512xxx12	yes	✓	-
NXP	K24	MK24FN256xxx12	yes	✓	-
NXP	KV54	MKV58F1M0xxx24	yes	✓	-
NXP	K64	MK64FN2M0xxx18	yes	✓	-
NXP	K65	MK65FN2M0xxx18	yes	✓	-
NXP	K66	MK66FN2M0xxx18	yes	✓	-
Renesas	RA2L	R7FA2L1AB	yes	✓	-
Renesas	RA4W	R7FA4W1AD	yes	✓	-
Renesas	RA6M	R7FA6M3AH	yes	✓	-
Renesas	RX65N	R5F565NE	no	✓	TSIP Full
Renesas	RX65T	R5F566TA	no	✓	TSIP Lite
Renesas	RX65T	R5F566TE	no	✓	TSIP Lite
Renesas	RX72M	R5F572MN	no	✓	TSIP Full
Renesas	RX72N	R5F572NN	no	✓	TSIP Full
Renesas	RX72T	R5F572TK	no	✓	TSIP Lite
ST	STM32F4	STM32F405xG	yes	✓	-
ST	STM32F4	STM32F405xE	yes	✓	-
ST	STM32F4	STM32F407xG	yes	✓	-

ST	STM32F4	STM32F412ZG	yes	✓	-
ST	STM32F4	STM32F429xI	yes	✓	-
ST	STM32F4	STM32F469xI	yes	✓	-
ST	STM32F7	STM32F777xI	yes	✓	-
ST	STM32F7	STM32F725xE	yes	✓	-
ST	STM32F7	STM32F725xG	yes	✓	-
ST	STM32F7	STM32F735xG	yes	✓	-
ST	STM32H7	STM32H725xE	yes	✓	-
ST	STM32H7	STM32H725xG	yes	✓	-
ST	STM32H7	STM32H735xG	yes	✓	-
ST	STM32H7	STM32H743xI	yes	✓	-
ST	STM32H7	STM32H745xI_M7	yes	✓	-
ST	STM32H7	STM32H747xI_M7	yes	✓	-
ST	STM32H7	STM32H753xI	yes	✓	-
ST	STM32H7	STM32H755xI_M7	yes	✓	-
ST	STM32H7	STM32H7B3xI	yes	✓	-
ST	STM32L4	STM32L452xE	yes	✓	-
ST	STM32L4	STM32L475xG	yes	✓	-
ST	STM32L4	STM32L4P5xG	yes	✓	-
ST	STM32L4	STM32L4Q5xG	yes	✓	-
ST	STM32L4	STM32L4R5xI	yes	✓	-
ST	STM32L4	STM32L4S5xI	yes	✓	-
ST	STM32L5	STM32L552xE	yes	✓	Trustzone
ST	STM32U5	STM32U5G9xJ	yes	✓	Trustzone
ST	STM32U5	STM32U5G7xJ	yes	✓	Trustzone
ST	STM32U5	STM32U5F9xJ	yes	✓	Trustzone
ST	STM32U5	STM32U5F7xJ	yes	✓	Trustzone
ST	STM32U5	STM32U5A9xJ	yes	✓	Trustzone
ST	STM32U5	STM32U5A5xJ	yes	✓	Trustzone
ST	STM32U5	STM32U599xJ	yes	✓	Trustzone
ST	STM32U5	STM32U595xJ	yes	✓	Trustzone
ST	STM32U5	STM32U585xI	yes	✓	Trustzone
ST	STM32U5	STM32U575xI	yes	✓	Trustzone
ST	STM32WB55	STM32WB55xG	yes	✓	-
Silicon Labs	FG23	EFR32FG23B010-F512IM48	yes	✓	Trustzone
Silicon Labs	PG22	EFM32PG22C200-F512IM40	yes	✓	Trustzone

Table 1 Supported microcontrollers

The TOE is provided to embedded developers as C source code in an IAR Embedded Workbench (a popular IDE) project that is generated by an IDE plugin called Embedded Trust. The project includes all source code plus configuration files for the selected microcontroller. This allows the TOE source code to be inspected as part of any security audit.

IAR Embedded Workbench and the Embedded Trust plugin are distributed in Windows installers downloadable from https://iar.com. Once installed, the Embedded Trust release notes and user guide and example projects are accessible directly via the Help menu. Example projects are accessible via Help / Information Centre / Example Projects / Embedded Trust / Getting Started.

When the IDE builds the TOE project, the Embedded Trust plugin also creates a set of instructions for a factory provisioning system provided as another component of the Embedded Trust product, Secure Deploy, describing how the PDB is to be constructed and programmed onto each device along with the built TOE image. The PDB in each device contains a unique secret identity key, identity certificate chain, and firmware trust anchors. Developers can configure the identity certificate chain and other details of the provisioning process in the graphical user interface (GUI) when creating a new bootloader project. The provisioning system can be configured to automate the provisioning of a batch of devices of set size in a single run. This is achieved using the Secure Deploy family of tools supplied by Secure Thingz which is outside the scope of this document.

The workflow for preparing and provisioning the TOE onto microcontrollers in IoT devices is summarized in Figure 3.

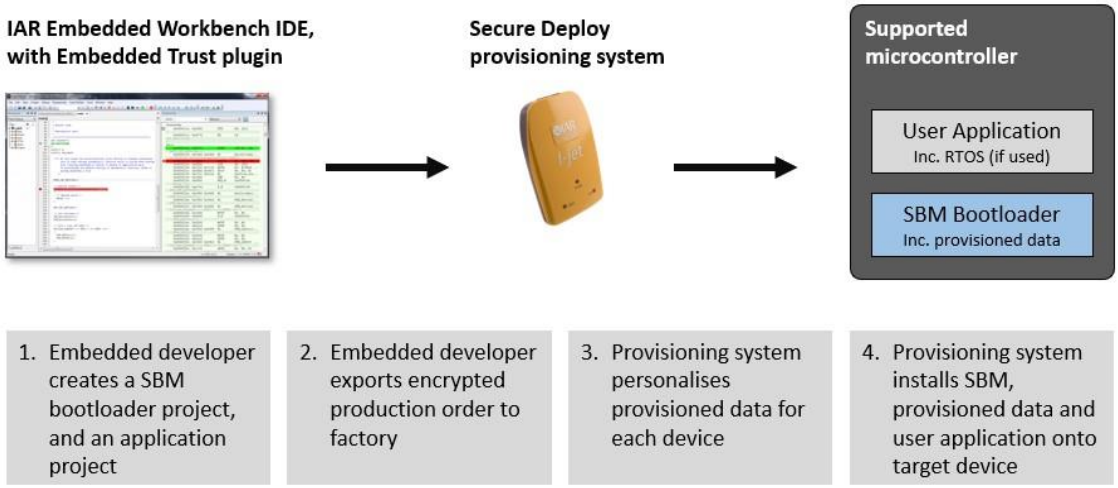


Figure 3: Overview of how the TOE is prepared, and provisioned onto microcontrollers

Several optional developer-specific or board-specific functions in the TOE are provided in the source code, where they are called at specific points in the start-up sequence. These functions are called “OEM API functions” and must be implemented by the developer. Developers are instructed not to modify bootloader code or behaviour outside of these functions because doing so will lead to unsupported behaviour.

The OEM API functions provide log outputs that can be routed to a serial line or a log file, a handler for boot failures, and a tamper-detection handler. By default, they are disabled. They are enabled by setting conditional compilation flags in the TOE source code – flags which are in turn set by options selected in the IDE by the developer when creating the bootloader project. These selections also set a number of other conditional compilation flags that can be used to disable certain features of the TOE for convenience during development.

Developers create a separate project for the user application, where they have a free hand. The TOE exposes a set of functions called the “Application API” to the user application as wrapper functions in a library provided for inclusion in that application. The wrapper functions map arguments into and results out of buffers whose addresses and sizes are passed into a function in the TOE called the SBM Access Method, a pointer to which is located at a fixed memory location in the TOE, set using #defines in both the TOE (the bootloader) and the user application C code. This Application API exposes functions for:

- Signing, validation and key derivation
- Checking the contents of the update slot and initiating an update
- Writing a new update image to an update slot
- Checking the metadata of the installed application
- Checking the status of PDB slots
- Retrieving certificates from the PDB
- Checking the version of the TOE

When the Embedded Workbench IDE builds the user application the Embedded Trust plugin packages the resulting binary image in a structure called a software update (SWUP) file. The SWUP includes metadata about the package’s image including its version number, a signature created using the developer’s secret firmware signing key, and information allowing the target devices to decrypt the firmware image.

The TOE installer will install only valid SWUPs found in the update slot. SWUPs may be delivered into the update slot over the air to deployed IoT devices via a running previous version of the user application or programmed directly into the update slot on the production line in the factory at the same time as the TOE is provisioned.

2 Security Objectives for the operational environment

In order for the platform to fulfil its security requirements, the operational environment (technical or procedural) must fulfil the following objectives.

- Only authorized and trustworthy personnel shall have access to the TOE development environment (as described in [Manual] section “Mastering”).
- Only supported microcontrollers shall be used (as described in Table 1).
- Users must ensure they have an authentic copy of IAR Embedded Workbench with Embedded Trust plugin (see section 1.3 “included guidance documents” for SBM Authentication)

3 Security requirements and implementation

3.1 Security Assurance Requirements

The claimed assurance requirements package is: **SESIP1** as defined in [SESIP].

3.1.1 Flaw Reporting Procedure (ALC_FLR.2)

In accordance with the requirement for a flaw reporting procedure (ALC_FLR.2), including a process to generate any needed update and distribute it, the developer has defined the following procedure:

Flaws can be reported by email at security-alert@iar.com, the Vulnerability Disclosure Policy and procedure is outlined at www.iar.com/vulnerability-disclosure-policy. These channels are monitored by the IAR Systems customer service team. Bug reports including security flaws are passed to the embedded software development team at IAR Embedded Security. A fix is developed and merged into the current release candidate version of the TOE, where it forms part of the next release of the Embedded Trust product, of which the TOE is a part. Embedded Trust releases are typically several months apart. For severe issues an updated version of the current release is made available as soon as the fix is available.

Customers are notified by email when new releases of Embedded Trust are made. Known issues and new fixes are described in the release notes of each release. The reporter of the issue is updated on progress via IAR customer support.

When new releases of the TOE are made customers can update their factory production lines to provision the newer release onto newly manufactured IoT devices. They cannot update the TOE on devices that have already been provisioned with a previous version, for two reasons:

1. As a bootloader, a function of the TOE is to verify and install updated user application images. It cannot update itself, because in a XIP microcontroller that would require replacing the code it is currently executing – something that cannot be accomplished reliably.
2. As a keystore, a function of the TOE is to provide the host device with an immutable identity, which, by definition, must not change. On microcontrollers where the root identity secrets are stored as part of the TOE, the TOE cannot be updated without issuing a new identity to the device.

To minimize the risk of serious security flaws in the TOE leading to compromise of the IoT device, its codebase is kept as small and stable as possible.

3.1.2 Vulnerability Survey (AVA_VAN.1)

AVA_VAN.1 requires that the TOE demonstrate an attack potential rating of at least 16. We do so in two steps. First, we review known vulnerabilities in the TOE and its components.

Second, we systematically review potential attacks on the TOE. At each step, we show that no attacks with attack potential less than 16 can be identified.

3.1.2.1 Survey of known vulnerabilities

Table 2 lists software components used in the TOE, including third-party libraries. Details of each are provided, including how vulnerability notices are monitored.

Name of software component	Version	Supports specific microcontrollers ?	Maintainer	Where is it obtained?	How are announcements and new releases monitored?
Secure Boot Manager	3.00	See Table 1	Secure Thingz	https://www.iar.com/products/security/iar-embedded-trust/	We invite vulnerability reports directly, per section 3.1.1 Flaw Reporting Procedure (ALC_FLR.2), and monitor public channels including https://cve.mitre.o rg/cve/
microecc	commit b335ee8 **Please Note:** All commits between this and the product release date don't have any effect on the security of this component and only change the gitignore file and some syntax changes to asm_arm.inc.	n/a	Ken MacKay	https://github.com/kmackay/micro-ecc	We are notified of new releases by GitHub and monitor them for vulnerability announcements
SHA256	RFC4634	n/a	IETF	https://datatracker.ietf.org/doc/rfc4634/	We monitor IETF Announce mailing list for vulnerability announcements about drafteastlake-sha2 or draft-eastlakesha2b

libtomcrypt	1.17	n/a	libtom team	https://github.com/libtom/libtomcrypt	We are notified of new releases by GitHub and monitor them for vulnerability announcements. If these vulnerabilities are not resolved by the authors within a reasonable time frame (within 30 days). Then the library would be forked internally and the vulnerability addressed, until a formal fix has been released.
Atmel SAML11 Series Device Support	1.0.109	SAML11E16A	Microchip	http://packs.download.atmel.com/	We subscribe to Microchip PCNs for SAML11 at https://www.microchip.com/pcn
Microchip MPLAB Harmony PIC32CMLS00 Dev Pack	1.0.141	PIC32CM5164LS0 0100	Microchip	https://github.com/Microchip-MPLAB-Harmony/dev_packs/tree/master/Microchip/PIC32CM-LS_DFP/1.1.162/PIC32CM-LS00	We subscribe to Microchip PCNs for PIC32CMLS00 at https://www.microchip.com/pcn
MCUXpresso SDK for LPCXpresso55S06	2.8.2	LPC55S06	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.

Name of software component	Version	Supports specific microcontrollers ?	Maintainer	Where is it obtained?	How are announcements and new releases monitored?
					nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements
MCUXpresso SDK for LPCXpresso55S16	2.9.1	LPC551X	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements

MCUXpresso SDK for LPCXpresso55S28	2.9.1	LPC552X	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements
MCUXpresso SDK for LPCXpresso55S69	2.9.1	LPC55S69	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements
MCUXpresso SDK for Kinetis Freedom K22F	2.9.0	MK22FN512VLH12	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements
MCUXpresso SDK for Kinetis TWR-K24F	2.3.1	MK24FN256VDC12, MK24FN1M0VLL12	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements

Name of software component	Version	Supports specific microcontrollers ?	Maintainer	Where is it obtained?	How are announcements and new releases monitored?
MCUXpresso SDK for Kinetis Freedom K64F	2.9.0	MK64FN1M0VLL12	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements

MCUXpresso SDK for Kinetis Freedom K66F	2.9.0	MK65FN2M0VMI18, MK66FN2M0VMD18	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements
MCUXpresso SDK for i.MX RT1064	2.9.2	MIMXRT1064	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements
MCUXpresso SDK for Kinetis TWR-KV58M	2.8.0	MKV58F220M	NXP	https://mcuxpresso.nxp.com/en	We subscribe to product announcements from NXP and monitor NXP user forums at https://community.nxp.com/community/mcuxpresso/mcuxpresso-sdk for vulnerability announcements
Silicon Labs Gecko SDK	4.1.0	EFM32PG22C200F512IM40, EFR32FG23B010F512IM48	Silicon Labs	https://www.silabs.com/developers/gecko-softwaredevelopment-kit	We are notified of new releases by GitHub and vis the Silicon Labs Gecko SDK Developer page and monitor them for vulnerability announcements
STM32CubeF7	1.15.0	STM32F777ZIT6	ST Micro	https://www.st.com/en/embeddedsoftware/stm32cubef7.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for

Name of software component	Version	Supports specific microcontrollers ?	Maintainer	Where is it obtained?	How are announcements and new releases monitored?
					vulnerability announcements.

STM32CubeF4	1.24.1	STM32F405VGT, STM32F407VG, STM32F412ZGT6, STM32F429II, STM32F469NI	ST Micro	https://www.st.com/en/embedded-software/stm32cubef4.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeL4	1.14.0	STM32L452RE, STM32L475xG, STM32L4R5ZI, STM32L4S5ZIT6	ST Micro	https://www.st.com/en/embeddedsoftware/stm32cubel4.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeL5	1.1.0	STM32L552	ST Micro	https://www.st.com/en/embedded-software/stm32cubel5.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeH7	1.8.0	STM32H753xI, STM32H725ZG, STM32H735xG STM32H753ZI, STM32H7B3xI	ST Micro	https://www.st.com/en/embedded-software/stm32cubeh7.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeWB	1.12.1	STM32WB55RG	ST Micro	https://www.st.com/en/embedded-software/stm32cubewb.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for vulnerability announcements.
STM32CubeU5	1.0.2	STM32U585	ST Micro	https://www.st.com/en/embedded-software/stm32cubeu5.html	We subscribe to product announcements from ST and monitor ST user forums at https://community.st.com/ for

Name of software component	Version	Supports specific microcontrollers ?	Maintainer	Where is it obtained?	How are announcements and new releases monitored?
					vulnerability announcements.
Trusted Secure IP Driver	1.11	R5F565NEHDFC R5F566TA, R5F566TE, R5F572MN, R5F572NN, R5F572TK	Renesas	https://www.renesas.com/eu/en/products/software-tools/software-os-middleware-driver/security-crypto/trusted-secure-ip-driver.html	Renesas account managers contact us with any vulnerability announcements.
RX Family RX Driver Package	1.13	R5F565NEHDFC, R5F566TA, R5F566TE, R5F572MNHBDB, R5F572NN, R5F572TK	Renesas	https://www.renesas.com/eu/en/products/software-tools/software-os-middleware-driver/software-package/rx-driver-package.html	We subscribe to product announcements through Renesas MyPages and https://www.renesas.com/us/en/support/pcnsearch.html and monitor them for security vulnerabilities.
RA Family Flexible Software Package (FSP)	3.7.0	R7FA2L1AB, R7FA4M2AD, R7FA4W1AD, R7FA6M3AH	Renesas	https://www.renesas.com/eu/en/software-tool/flexible-software-package-fsp	We subscribe to product announcements through Renesas MyPages and https://www.renesas.com/us/en/support/pcnsearch.html and monitor them for security vulnerabilities.

Table 2 First- and third-party software used in the TOE, and associated channels which are monitored for security vulnerability announcements.

A search of the public Common Vulnerabilities and Exposures[1] (CVE) database for known vulnerabilities in these components shows that no applicable vulnerabilities have been reported.

Two CVEs do exist against libtomcrypt 1.17 but are not applicable to the TOE:

i) CVE-2019-17362 reports a vulnerability in the der_decode_utf8_string function. This function is not present in the TOE.

[1] https://cve.mitre.org/cve/search_cve_list.html

ii) CVE-2018-12437 reports that under certain conditions a vulnerability exists in the ECDSA signature function. This function is not present in the TOE; the micro-ecc library is used for all ECDSA signature operations.

3.1.2.2 Survey of potential vulnerabilities

This survey uses the method of attack trees[1] to develop a tree of possible attack chains on the TOE, using information about its design and implementation. Well-known methods of attacking secure embedded systems have been considered at each node in the attack tree. Where appropriate they have been identified as potential attacks. The document [AM] provides an enumeration of well-known methods of attack.

Note that the survey considers only attacks on the TOE. Attacks on a composite system may be able to proceed to the attackers' goals by other means.

Attack trees are constructed by working backwards from the attackers' goals:

Goal A: **Extract software IP**

A1. Modify the TOE so that it does not disable debug interfaces on first boot
- A.1.1. Alter TOE in supply chain or on-chip before first boot
- A.1.2. Inject faults to re-enable debug access

A2. Extract firmware decryption group private key
- A.2.1. Probe or image internal Flash memory to obtain firmware decryption key
- A.2.2. Use simple power analysis (SPA) / differential power analysis (DPA) during firmware decryption operations
- A.2.3. Use higher-order DPA during firmware decryption operations
- A.2.4. Use electromagnetic emissions analysis (EMA) during firmware decryption operations
- A.2.5. Use fault analysis (FA) during firmware decryption operations

A3. Exploit bugs or undocumented features in the TOE to externally influence it during boot to expose firmware decryption group private key or proprietary firmware A4. Exploit factory test modes to access provisioned data

Goal B: **Take over a device (execute attacker's software)**

B1. Have TOE boot malware placed in the active slot
- B.1.1. [AND] Re-enable debug access
  - B.1.1.1. Inject faults to re-enable debug access
  - B.1.1.2. Exploit bugs or undocumented features in the TOE to externally influence it during boot to re-enable debug access
- B.1.2. [AND] Disable TOE validation of the active application during boot
  - B.1.2.1. Inject faults to disable validation of the active application during boot

[1] https://www.schneier.com/academic/archives/1999/12/attack_trees.html

B.1.2.2. Modify the running user application to write changes to the installed TOE
B.1.2.2.1. Provoke a buffer overflow in the user application to execute specially crafted code to write changes to the TOE
B.1.2.2.2. Compromise unvalidated software loaded outside the validated boot chain to write changes to the TOE
B.1.2.3. Modify the running TOE to write changes to its own code
B.1.2.3.1. Send a malformed SWUP to cause undocumented behaviour in the TOE such that the attacker can execute code to write changes to the TOE
B.1.2.4. Probe the TOE in internal Flash to write changes to its code
B.1.2.5. See [Alter TOE in supply chain or on-chip before first boot]
B.1.2.6. Exploit bugs or undocumented features in the TOE to circumvent validation of the user application at boot

B2. Have TOE install invalid SWUP containing malware
B.2.1. [AND] Inject faults during authentication of invalid SWUP
B.2.2. [AND] Deliver invalid SWUP to target device

B3. Have TOE install stale but validly signed SWUPs containing old firmware with known vulnerabilities

Goal C: **Masquerade as an authentic device**

C1. Exfiltrate device's private identity key using attacker's software
C.1.1. See [Take over the device (execute attacker's software)]
C.1.2. Provoke a buffer overflow in the user application to execute specially crafted code to exfiltrate the device identity key

C2. Use simple SPA/DPA during elliptic curve digital signal algorithm (ECDSA) signature operations to obtain device identity key

C3. Use higher-order DPA during ECDSA signature operations to obtain device identity key

C4. Use EMA during ECDSA signature operations to obtain device identity key

C5. Modify random number generator (RNG) behaviour during ECDSA operations to obtain device identity key

C6. Probe or image the device private key in internal Flash memory

C7. Exploit bugs or undocumented features in the TOE to cause it to expose device's private key externally during boot

C8. See [Exploit factory test modes to access provisioned data]

C9. Use FA during ECDSA signature operations to obtain device identity key

Goal D: **Manufacture counterfeit devices**

D1. Install firmware image on unauthorised devices
D.1.1. [AND] See [Extract software IP]
D.1.2. [AND] Run software on unauthorized devices

Goal E: **Remotely disable a deployed device**

E1. Send a malformed SWUP such that the TOE installs a non-functional user application
E.1.1. See [Take over the device (execute attacker's software)]

E2. Modify the TOE to always invalidate the active and update slots

E.2.1. See [Modify the running user application to write changes to the installed TOE] and [Modify the running TOE to write changes to its own code]

Leaf nodes in the attack tree describe attacks on the TOE. Next, we discuss the feasibility of these attacks.

Attack	Feasibility
A.1.1. Alter TOE in supply chain or on-chip before first boot	Supply-chain security is out of scope of the TOE. Users are advised to ensure they have an authentic copy of IAR Embedded Workbench with Embedded Trust plugin, and to employ a secure provisioning system to deliver their project from development onto devices. Users are advised to ensure the security of provisioned information on-chip before first boot by operating a secure production environment, by utilizing a secure firmware programming interface (where available), and/or by disabling the debug interface using the programmer immediately after initial programming.
A.2.1. Probe or image internal Flash memory to obtain firmware decryption key **B.1.2.4. Probe the TOE in internal Flash to write changes to its code** **C6. Probe or image the device private key in internal Flash memory**	Resistance to probing attacks is in a feature of the microcontroller platform and out of scope of the TOE. On devices, boards or microcontrollers fitted with latching tamper detection circuits, the composite designer may implement or specify a tamper detection mesh to use this feature.
A.2.2. Use simple SPA/DPA during firmware decryption operations **A.2.3. Use higher-order DPA during firmware decryption operations**	An attacker with the ability to have the TOE decrypt SWUPs while monitoring power or emissions side channels may attempt to extract the firmware decryption group private key, hence the ephemeral symmetric firmware encryption key, and thus the software IP. The TOE employs a crypto library (microecc) that implements anti-SPA/DPA measures to frustrate this. These measures are detailed in [TJERAND]. Further, new SWUPs are checked for a valid signature and higher software version number before decrypting them. This ensures that only code in genuine OEM-signed SWUPs is run at all.

A.2.4. Use EMA during firmware decryption operations **C4. Use EMA during ECDSA signature operations to obtain device identity key**	Electromagnetic emissions may leak information. The TOE is not designed with a design goal of preventing information leakage through near-field or far-field electromagnetic emissions, and other than by minimizing the use of private keys provides no protection against attacks utilizing such side-channels. The TOE effectively limits the number of usages of the firmware decryption group private key by checking that new SWUPs are validly signed and declare a higher software version number, before decrypting them. The device identity key is used to sign a random hash challenge during each transport layer security (TLS) protocol handshake. This key cannot be access directly by the application. The application must request the signing of data by the key via the Secure API provided by the TOE. An attacker with physical access can attempt an EMA side-channel analysis on it by stimulating multiple TLS connections. The TOE does not limit the rate of TLS connections, but the composite designer may be able to do so at system design level. To frustrate EMA attacks more completely users must rely on other parts of the platform, in particular the MCU hardware,

	and on device-level design features such as cans, potting and tamper meshes. EMA attacks, especially at die level, require expertise and equipment that at attack rating 22 puts them beyond the reach of a basic attack potential:

	Identification phase		**Exploitation phase**	
Elapsed time	<1 week	2	<1 day	3
Expertise	Expert	5	Expert	4
Knowledge of the TOE	Public	0	Public	0
Access to the TOE	<10 samples	0	<10 samples	0
Equipment	Specialised	3	Specialised	4
Open samples	Public	0	Public	0
PHASE TOTALS		10		11
ATTACK RATING	**21**			

<table>
<tr><td>A.2.5. Use FA during firmware decryption operations

C9. Use FA during ECDSA signature operations to obtain device identity key</td><td>A specialization of fault injection attacks, FA attacks seek to inject bit errors into cryptographic operations in tamper-proof microcontrollers, in such a way as to leak data about secret keys.

This kind of attack requires expertise and equipment beyond the reach of a basic attack potential:

<table>
<tr><th></th><th colspan="2">Identification phase</th><th colspan="2">Exploitation phase</th></tr>
<tr><td>Elapsed time
Expertise
Knowledge of the TOE</td><td><1 week
Expert</td><td>2
5</td><td><1 day
Expert</td><td>3
4</td></tr>
<tr><td>Access to the TOE Equipment</td><td>Public
<10 samples
Specialised</td><td>0
0
3</td><td>Public
<10 samples
Specialised</td><td>0
0
4</td></tr>
<tr><td>Open samples
PHASE TOTALS
ATTACK RATING</td><td>Public</td><td>0
10</td><td>Public</td><td>0
11</td></tr>
<tr><td colspan="4">21</td></tr>
</table>
</td></tr>
<tr><td>A3. Exploit bugs or undocumented features in the TOE to externally influence it during boot to expose firmware decryption group private key or proprietary firmware</td><td>The only external input the TOE shall accept during boot is a SWUP stored in external memory. That SWUP is authenticated before use. Correct and robust behaviour in processing that SWUP is ensured using good development practices.</td></tr>
<tr><td>A4. Exploit factory test modes to access provisioned data</td><td>No factory test modes are implemented in the TOE.</td></tr>
<tr><td></td><td>Implementors of such modes must take care to permanently disable them before deployment, or else ensure they do not expose secret material.</td></tr>
</table>

A.1.2 Inject faults to reenable debug access

B.1.1.1. Inject faults to re-enable debug access

B.1.1.2. Exploit bugs or undocumented features in the TOE to externally influence it during boot to re-enable debug access

The TOE irreversibly disables debug and programming interfaces, preventing attackers with physical access to a target MCU's debug ports from obtaining provisioned data, user data, or software IP. Irreversible debug lockdown is a required feature on all microcontrollers supported by the TOE. Once set, no software attack can reverse it.

The resistance of the MCU's debug lock feature to fault injection attacks is out of scope of the TOE.

B.1.2.1. Inject faults to disable validation of the active application during boot

B.2.1. [AND] Inject faults during authentication of invalid SWUP

An attacker may inject faults during TOE execution by means of clock or voltage glitches. By causing execution to skip instructions, this technique can be used to bypass validations to, for example, install untrusted code, or to change lifecycle state. It can also be used to weaken cryptographic functions, potentially allowing the extraction of private keys.

This kind of attack requires expertise and equipment that at attack rating 22 puts it beyond the reach of a basic attack potential:

	Identification phase		**Exploitation phase**	
Elapsed time	<1 week	2	<1 day	3
Expertise Knowledge of the TOE	Expert	5	Expert	4
Access to the TOE Equipment	Public	0	Public	0
Open samples	<10 samples	0	<10 samples	0
PHASE TOTALS	Specialised	3	Specialised	4
	Public	0	Public	0
		10		11
ATTACK RATING	**21**			

B.1.2.2.2. Provoke a buffer overflow in the user application to execute specially crafted code to write changes to the TOE **C.1.2. Provoke a buffer overflow in the user application to execute specially crafted code to exfiltrate the device identity key**	A corrupted user application may try to enable malware to become permanently resident on the device by disabling boot time validation of the user application by the TOE. Use is made of write-protection features provided by the MCU where available. Additionally, the TOE has been designed in such a way that the composite developer isolates user applications on microcontrollers that support TrustZone-M trusted execution environment (TEE) technology. MCUs are available with a range of isolation mechanisms, from none, through OSes exploiting privileged modes and memory protection units, to integrated secure elements and TEE partitioning with trusted supervisory firmware. It is up to the composite developer to configure their chosen isolation mechanism to prevent writes to the TOE Flash and direct reads of provisioned data by user applications. If none of these options is available, composite developers rely on attackers being unable to find ways to compromise running software on target devices. Executing buffer overflow attacks on Harvard architecture execute-from-flash MCUs is a difficult task that as well as a suitable buffer overflow vulnerability requires knowledge of at least the firmware binary image and ideally the source code, where a Flash write function must be available. Confidence in this approach can be increased using firewalls and robust API testing.

B.1.2.2.1. Compromise unvalidated software loaded outside the validated boot chain to write changes to the TOE	Although the TOE validates the next application in the boot sequence, nothing in the TOE prevents that application from loading further applications, potentially including untrusted applications. Composite developers must take care to write protect the TOE, employ an immutable ROM bootloader, or apply appropriate application isolation techniques in such situations.
B.1.2.3.1. Send a malformed SWUP to cause undocumented behaviour in the TOE such that the attacker can execute code to write changes to the TOE	An attacker may try to compromise the TOE directly by sending a malformed SWUP. The SBM validates each SWUP's signature before doing any further processing, so an attacker would have to compromise the composite developers' firmware repository, firmware signing key or firmware signing procedures to attempt this. This is outside the scope of the TOE.

B.1.2.6. Exploit bugs or undocumented features in the TOE to circumvent validation of the user application at boot	The source code is provided to the customer. The TOE is written to validate the next application in the boot sequence, i.e. the user application. It does this by verifying a signature calculated over the contents of the user application using a public key as part of a PKI chain of x509 certificates. An attacker must be in possession of the signing key which is held securely by the customer to create a valid user application signature. For an unmodified TOE, no vulnerabilities are known as a result of combination of code transparency, device manufacturer claims and continuous testing/QA process (both automated and manual). The only way to circumvent validation of the user application at boot would be to modify the TOE source code directly in order to skip the signature verification step.

B.2.2. [AND] Deliver invalid SWUP to target device	The TOE can process new SWUPs placed in internal or external Flash memory. SWUPs are expected to be delivered via external processes. Because the TOE validates the integrity, freshness and authenticity of each SWUP the delivery processes are not required to be secure. For instance, new SWUPs could be placed on a removable SD card. Nevertheless, use of secure channels to deliver SWUPs does no harm and further reduces opportunities for attack. Additionally, the SWUPs are encrypted.
B3. Have TOE install stale but validly signed SWUPs containing old firmware with known vulnerabilities	Downgrade attacks where a validly signed but old SWUP with known vulnerabilities is delivered to target devices to enable further attacks are prevented by having the TOE check that new SWUPs declare higher application version number than the currently-installed SWUP before installing them.
C2. Use simple SPA/DPA during ECDSA signature operations to obtain device identity key **C3. Use higher-order DPA during ECDSA signature operations to obtain device identity key**	A device's identity key is used to signa random hash challenge during each TLS handshake, so an attacker with physical access can attempt a side-channel analysis on it by stimulating one or more TLS connections. The TOE employs a crypto library (micro-ecc) that implements anti-SPA/DPA measures to frustrate this. These measures are detailed in [TJERAND].

C5. Modify RNG behaviour during ECDSA operations to obtain device identity key	High quality random numbers are essential to many cryptographic operations. The only such operation implemented in the TOE is ECDSA signature. An attacker in possession of an ECDSA signature and the random number used in its creation can recover the device's secret identity key. Thus, the randomness of the RNG is critical. As part of the physical MCU platform the RNG is outside the scope of the TOE, but all MCUs supported by the TOE are equipped with high-quality RNGs, whose behaviour is not trivial to alter while retaining other MCU functionality.
C7. Exploit bugs or undocumented features in the TOE to cause it to expose device's private key externally during boot	The TOE contains no functions that expose devices' private keys, internally or externally. Its correct and robust behaviour is ensured using good development practices.
D.1.2. [AND] Run software on unauthorized devices	The TOE performs a check at boot time, that the provisioned data includes a hash seeded with the device hardware ID. A firmware image that has been extracted from an authentic device, for example by sniffing it from a serial programming line in the factory, will fail this check on any other device and
	the TOE will terminate. A counterfeiter would have to reverse engineer and modify the firmware image to disable this check. If the counterfeit devices are to connect to a genuine web service, a further check can be implemented at the genuine web service that the device certificates are valid via the TLS protocol. The TOE provides Secure API functions that facilitate TLS on the device side (extract of the device certificate from the PDB, and signing of data with the private key).

The following security practices are employed in development of the TOE:

1. Traceability. Code commits are linked back to requirements management. This helps ensure only documented features enter the codebase.
2. Code review. All code commits undergo review for quality and function before merging. This helps prevent bugs or undocumented features from entering the codebase.
3. Unit testing. All functions are subject to unit tests that run automatically on each merge request. Unit tests are themselves subject to code review, including by dedicated testers, for thoroughness. Unit tests include valid and invalid inputs. This helps ensure robust and stable behaviour.

4. API testing. All application and OEM APIs are subject to API tests including malformed requests that run automatically on each merge request. API tests are themselves subject to code review, including by dedicated testers, for thoroughness. This helps ensure robust and stable API behaviour.

3.1.3 Operational User Guidance (AGD_OPE.1)

Objective	How Criteria is met	Relevant Section of User Guidance
AGD_OPE.1.1C: The operational user guidance shall describe, for each user role, the user-accessible functions and privileges that should be controlled in a secure processing environment, including appropriate warnings.	The [Manual] describes the main functionality and additional features of the 3 types of user licences and highlights which ones include use of the TOE. The [Manual] also describes the functionality of the TOE, access to functionality of the TOE isn't controlled by specific user roles and privileges. Warnings relating to failed licence checks which limit the user from accessing configuration options of the TOE are also highlighted.	[Manual] Page 18 – Types of Embedded Trust Licenses [Manual] Page 55 – Security Context creation problems
AGD_OPE.1.2C: The operational user guidance shall describe, for each user role, how to use the available interfaces provided by the TOE in a secure manner.	The [Manual] outlines ways to use the TOE in our trusted environment as well as the correct ways to use the TOE for secure development and secure production.	[Manual] Page 24 – Security and Embedded Trust [Manual] Page 66 – Security Menu
AGD_OPE.1.3C: The operational user guidance shall describe, for each user role, the available functions, and interfaces, in particular all security parameters under the control of the user, indicating secure values as appropriate.	The [Manual] describes security parameters such as secure memory map configurations, debug port locking, secure installation, and secure memory partitioning as well as how to control or change them.	[Manual] Page 102 - Descriptions of the Application API functions [Manual] Page 119 - Descriptions of the OEM API functions
AGD_OPE.1.4C: The operational user guidance shall, for each user role, clearly present each type of security-relevant event relative to the user-accessible functions that need to be performed, including changing the security characteristics of entities under the control of the TSF.	The [Manual] goes into detail about various secure values such as number of certificates and keys and various other secure values.	[Manual] Page 67 – Security Context wizard

AGD_OPE.1.5C: The operational user guidance shall identify all possible modes of operation of the TOE (including operation following failure or operational error), their consequences and implications for maintaining secure operation.	The [Manual] identifies the modes of operation of the TOE and it also provides guidance on troubleshooting and error handling in order to maintain secure operation.	[Manual] Page 34 – Security Boot Manager APIs [Manual] Page 54 – Troubleshooting [Manual] Page 61 – Reference Information on the GUI [Manual] Page 100 – Error Handling [Manual] Page 102 - Descriptions of the Application API functions
AGD_OPE.1.6C: The operational user guidance shall, for each user role, describe the security measures to be followed in order to fulfil the security objectives for the operational environment as described in the ST.	The [Manual] provides an overview of the TOE's security objectives.	[Manual] Page 17 – Introduction [Manual] Page 97 – Keys on the SBM/Application Interface [SBM Authentication] – Securing an SBM (Section 4)
AGD_OPE.1.6C: The operational user guidance shall be clear and reasonable.		[Manual] [SBM Authentication]

3.1.4 Preparative Procedures (AGD_PRE.1)

Objective	**How Criteria is met**	**Relevant Section of User Guidance**
AGD_PRE.1.1D: The developer shall provide the TOE including its preparative procedures.	The [SBM Authentication] describes the preparatory procedures such as how the user downloads the TOE.	[SBM Authentication] – MyPages Access (Section 2)
AGD_PRE.1.1C: The preparative procedures shall describe all the steps necessary for secure acceptance of the delivered TOE in accordance with the developer's delivery procedures.	The [SBM Authentication] describes the method of delivery and how secure delivery is ensured.	[SBM Authentication] – Section 2 Download [SBM Authentication] – Section 3 Validation
AGD_PRE.1.2C: The preparative procedures shall describe all the steps necessary for secure installation of the TOE and for the secure preparation of the operational environment in accordance with the security objectives for the operational environment as described in the ST.	The [SBM Authentication] describes the method of secure installation of the TOE. The [Manual] describes the way the optimal secure environment is prepared and how the TOE meets security objectives.	[Manual] Page 42 – Working with projects in the Embedded Workbench IDE [Manual] Page 52 – Using Embedded Trust without the IAR Embedded Workbench IDE

3.2 Security Functional Requirements

The platform fulfils the following security functional requirements:

3.2.1 Verification of Platform Identity

The platform provides a unique identification of the platform type, including all its parts and their versions.

Self-assessment:

1. The developer can identify the version of the TOE they are working with in the IAR Embedded Workbench IDE, where the Embedded Trust plugin adds an "Embedded Trust Release Notes" option to the "Help" menu. Selecting the option shows the release notes in the system web browser. The release notes identify the installed version of the Embedded Trust plugin. Embedded Trust uses semantic versioning. The TOE is versioned along with the Embedded Trust plugin.

   This is tested by selecting the "Embedded Trust Release Notes" option in the "Help" menu in IAR Embedded Workbench and verifying that the system web browser loads release notes including the correct Embedded Trust version number.

2. The user application can identify the version of the TOE installed using the Application API function *STZ_getSBMInformation*. This function returns the version number of the TOE. This version number is optionally defined by the developer by setting the conditional compilation flag *SBM_REPORT_SBM_VERSION* and defining the string *SBM_VERSION_ID*. The developer is given control of this to allow them to increment the version number for their bootloader project when they change their implementation of functions called by the OEM APIs. To keep track of which version of the TOE is installed on which IoT devices, the developer must keep production records showing what version of the TOE was installed onto each device. Alternatively, they can have connected devices report *SBM_VERSION_ID* to a central service, and maintain a separate record of which version of the TOE is associated with each *SBM_VERSION_ID*.

   This is tested by enabling *SBM_REPORT_SBM_VERSION* and defining *SBM_VERSION_ID* and verifying that this produces a TOE build that reports the correct version number to the user application via the *STZ_getSBMInformation* application API function.

3.2.2 Section removed

Heading retained to preserve paragraph numbering.

3.2.3 Verification of Platform Instance Identity

The platform provides a unique identification of that specific instantiation of the platform, including all its parts.

Self-assessment:

As a provisionable key store the TOE is provisioned with a unique secret identity key (256 bit ECC key using curve NIST P-256) and a corresponding certificate (PEM format X.509 containing the corresponding 256 bit ECC public key, with a SHA-256 hash). The certificate is issued by a CA installed on the factory provisioning system and is installed onto the device with the complete chain of CA certificates back to the root. This certificate provides each individual device with a unique cryptographic identity, verifiable by challenging the device to prove possession of the corresponding private key by signing a piece of data provided by the challenging party.

This is tested by provisioning the TOE onto a target microcontroller using the Embedded Trust provisioning system and having a test user application report the installed identity certificates and prove possession of the corresponding private key.

3.2.4 Attestation of Platform Genuineness

The platform provides an attestation of the "Verification of Platform Identity" and "Verification of Platform Instance Identity", in a way that ensures that the platform cannot be cloned or changed without detection.

Self-assessment:

Composite developers can guarantee that end users receive only genuine devices, not clones or counterfeits, by having remote Internet services or the user application or both check the TOE for a validly signed device identity certificate. Such certificates are only issued by the factory provisioning system provided as a component of the Embedded Trust product. This system signs device identity certificates using a CA key generated by the composite developer and installed via secure channels into a secure provisioning system located on the authorised production line. Only Embedded Trust provisioning systems specifically authorised by the composite developer receive this CA key. Without it, no counterfeiter can issue a valid device identity certificate.
Validly certified devices cannot be cloned either, because their private identity keys are never exposed either before or after being provisioned onto each device.

This is tested by provisioning the TOE onto a target microcontroller using the Embedded Trust provisioning system and having a test user application report the installed identity certificate chain and verifying that the composite developer's production CA certificate is present in that chain.

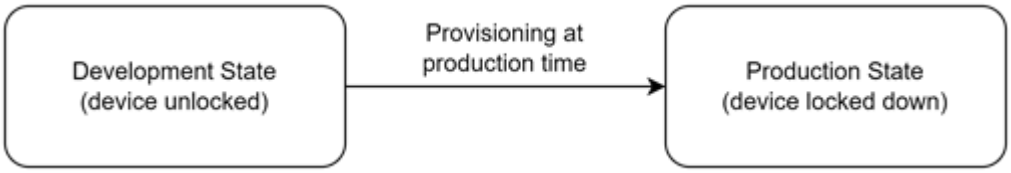
3.2.5 Section removed

Heading retained to preserve paragraph numbering. 3.2.6 Factory Reset of Platform

The platform can be reset to the state in which it exists when the product embedding the platform is delivered to the user, before any personal user data, user credentials, or user configuration is present on the platform.

Self-assessment:

There are two states that the TOE can exist in. A development state, during which the data can be freely manipulated, and a production state, in which the data remains fixed an immutable. After production-time provisioning of the TOE and user application, the IoT device is ready for operation. The TOE always remains in the state it was originally after production. The user application can only interact with the TOE via an API which does not facilitate the storage of any data originating in the user application. Therefore, the user application has no means by which to change the state of the SBM following production.



3.2.7 Secure Install of Application

The application must be installed in the field such that the integrity and authenticity of the application is maintained.

Self-assessment:

The composite developer can install a first SWUP file to the update slot on devices in the field by loading a SWUP into an external memory device such as an SD card. The TOE will then validate the SWUP and install the user application into the active slot before passing execution to it.
Validation consists of a check that the SWUP is signed by the composite developer. The signature is generated by the Embedded Trust plugin in the IAR Embedded Workbench IDE when the composite developer exports the SWUP. The signature is generated using ECDSA with NIST curve P-256. The signature is verified by the TOE using the same algorithm, the necessary trust anchor certificate being part of its provisioned data. This check verifies both authenticity and integrity of the SWUP.

Application firmware binaries packaged in SWUPs are encrypted using a 128-bit AES key in GCM mode. This key is derived by both parties using ECIES key agreement as described in section 3.2.9 part 2. The key pair used in the ECIES procedure by the Embedded Trust plugin is ephemeral. A new key pair is generated for each SWUP export operation. The key pair used in the ECIES procedure by the TOE is part of its provisioned data and is known as an Update Group Key pair because it may be provisioned onto a group of devices of the same underlying hardware platform, which will receive the same SWUP file.

This is tested by enabling debug access in a test build of the TOE and writing both valid and invalid SWUP files into the test device's update slot, verifying that on reset only valid SWUPs are installed.

3.2.8 Secure Update of Application

The application can be updated to a newer version in the field such that the integrity and authenticity of the application is maintained.

Self-assessment:

The composite developer can load an updated version of the application into the update slots of selected devices in the field by implementing an over-the-air SWUP distribution mechanism in all over-the-air updatable versions of the application. Once a SWUP has been downloaded into the update slot it will be validated and decrypted into the active slot by the TOE on next reset. Reset can be triggered by the running version of the application.

Validation steps include a check that the SWUP is signed by the composite developer as detailed in section 3.2.7, and that its version number is higher than that of the currently installed application. Version numbers are set by the composite developer at project build time and consist of one to three integers, each from 0 to 255, separated by periods, for example: 9.4.4. Decryption also proceeds as detailed in section 3.2.7.

This is tested by enabling debug access in a test build of the TOE and writing both valid and invalid SWUP files into the test device's update slot, verifying that on reset only valid SWUPs are installed.

3.2.9 Cryptographic Operation

The platform provides signing, signature verification and key agreement functionality with elliptic-curve cryptography as specified in NIST FIPS 186-5 and SP 800-186 for key lengths 256-bit and modes elliptic curve integrated encryption scheme. The micro-ecc library is used for all ECDSA signature operations.

Self-assessment:

Operations	Algorithm	Specification	Key Lengths	Modes	Number of Keys
Signing, Signature Verification	ECC ECDSA with P-256	NIST FIPS 186-5 NIST SP 800-186	256 Bit	ECDSA with SHA256	1
Key Agreement	ECC ECDH with P-256	NIST SP 800 56Ar3	256 Bit	ECIES using an ephemeral key	1
Encryption, decryption	AES-128GCM	NIST FIPS 197 NIST SP 800-38d	128 Bit	AES-128 for confidentiality and GCM for integrity	1
Hashing	SHA256	NIST PUB 180-4	256 Bit (hash)	n/a	n/a

The platform provides:

1. **ECDSA digital signature generation and verification** functions per NIST FIPS 186-5 Digital Signature Standard (DSS) section 6.4, using curve P-256 (i.e. a key length of 256 bits) per section 3.2.1.3 of SP 800-186. These functions are made available in the Application API as *STZ_signUsingKey* and *STZ_verifyUsingKey*.

Signature generation is tested by verifying that If the Application API *STZ_signUsingKey* function is called with a 256b (SHA-256) hash and an index to a 256b ECC private key as arguments, a signature based on that hash is returned, and the has is recoverable using the public part of that key and the NIST P-256 curve.

Signature verification is tested by verifying that if the Application API *STZ_verifyUsingKey* function is called with a signature based on a 256b hash and a 256b ECC public key as arguments, the hash is returned.

2. **ECIES key agreement** per NIST SP800 56Ar3 Recommendation for Pair-Wise KeyEstablishment Schemes Using Discrete Logarithm Cryptography, sections 6.2.2.2 and 5.7.1.2, excepting that the keying material is the shared secret returned directly from the ECDH process and not put through a key derivation function (note this uses an ephemeral key for one of the two keys). Key agreement also uses curve P-256 (i.e. 256 bit private and public keys), per section 3.2.1.3 of SP 800-186. This function is made available in the Application API as *STZ_generateSharedSecret*.

   This is tested by verifying that if the Application API *STZ_generateSharedSecret* function is called with an ephemeral ECC NIST P-256 private key and an index to an ECC NIST P-256 public key as arguments, an ephemeral 256b secret is returned, which is the keying material used for the AES-128 key and the IV.

Note **ECDH key agreement** is performed using micro-ECC to generate shared secrets.

4 Mapping and sufficiency rationales

4.1 SESIP1 sufficiency

Assurance Class	Assurance Families	Covered by	Rationale
ASE: Security Target evaluation	ASE_INT.1 ST Introduction	Section "Introduction" and "Title"	The ST reference is in the Title, the TOE reference in the "Platform reference", the TOE overview and description in "Platform functional overview and description".
	ASE_OBJ.1 Security requirements for the operational environment	Section "Security Objectives for the operational environment"	The objectives for the operational environment in "Security Objectives for the operational environment" refers to the guidance documents.
	ASE_REQ.3 Listed Security requirements	Section "Security Functional Requirements"	All SFRs in this ST are taken from [SESIP]. "Verification of Platform Identity" is included. Exclusion of "Section removed" is addressed in section" Flaw Reporting Procedure (ALC_FLR.2)".
	ASE_TSS.1 TOE Summary Specification	Section "Security Functional Requirements"	All SFRs are listed per definition, and for each SFR the implementation and verification is defined in Vulnerability Survey (AVA_VAN.1)
ALC: Life-cycle support	ALC_FLR.2 Flaw reporting procedures	Section "Flaw Reporting Procedure (ALC_FLR.2)"	The flaw reporting and remediation procedure is described.
AVA_VAN.1	AVA_VAN.1 Vulnerability survey	Section "Vulnerability Survey (AVA_VAN.1)"	The vulnerability survey and associated test results are described.

AGD_OPE.1	AGD_OPE.1 Operation user guidance	Section “Operational User Guidance (AGD_OPE.1)”	The Objectives and how they are met are described in the table in section 3.1.3.
AGD_PRE.1	AGD_PRE.1 Preparative procedures	Section “Preparative Procedures (AGD_PRE.1)”	The Objectives and how they are met are described in the table in section 3.1.4.

5 References

[SESIP] GlobalPlatform Technology, Security Evaluation Scheme for IoT Platforms, version 1.2

[AM] Application of Attack Potential to Smartcards and Similar Devices, version 3.0

[TJERAND] Comparative Study of ECC Libraries for Embedded Devices, Tjerand S, COSADE 2019