Security Gateway Overview

Requirements analysis

Document number	2201
Version	3
Issue date	2022-03-22

1 Rationale

This document describes the Canis Security Gateway from the perspective of requirements analysis. It describes the architecture and design of the gateway then compares this to requirements as part of a gap analysis. There are also recommendations for new requirements.

Obviously there are limitations to the security provided by a gateway:

- An attacker might have direct physical access to the trusted CAN bus
- An attack might use legitimate traffic flows across the gateway to subvert a device on the trusted bus and then use that to mount attacks

A security gateway therefore forms only part of the defenses of a CAN bus.

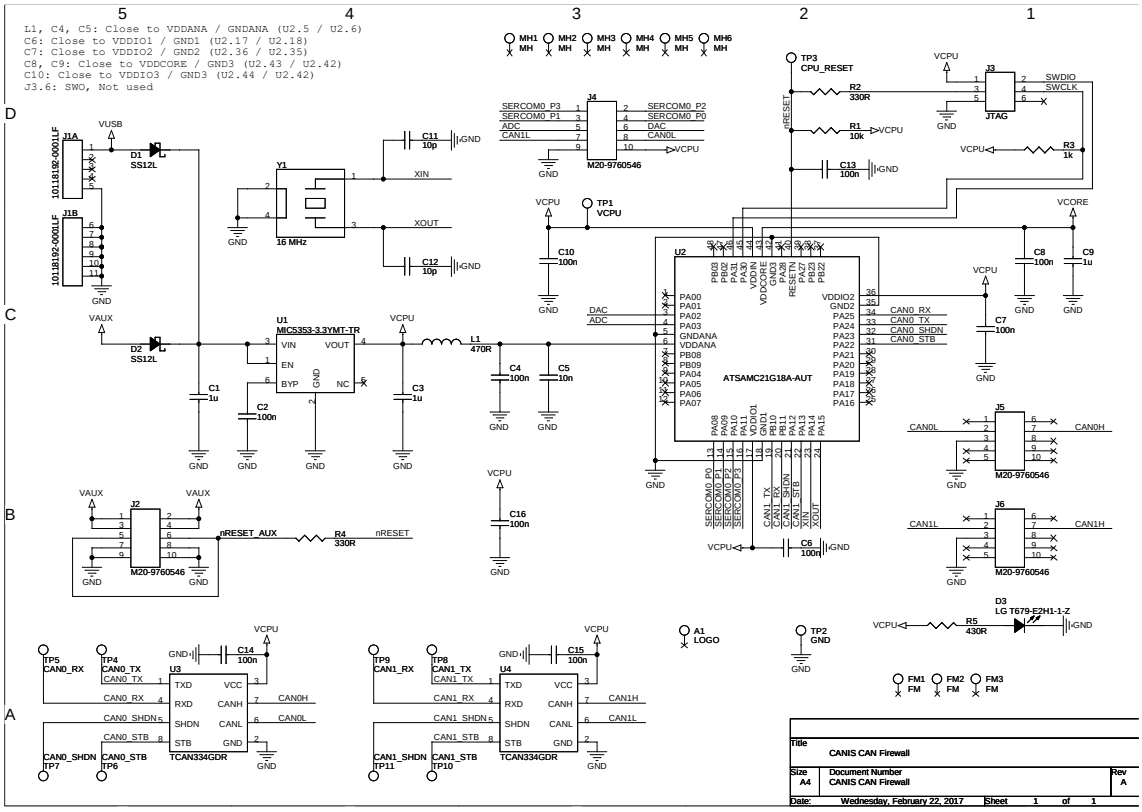
The document begins with a description of the gateway, and then compares to a formal set of requirements with an analysis of gaps.

2 Architecture

2.1 Hardware

The diagram below shows the hardware architecture. An off-the-shelf microcontroller (MCU) with two CAN controller interfaces is used, each using standard CAN transceivers. The Canis hardware uses the SAMC21 (a Cortex M0 based MCU) and the Microchip MCP2562FD CAN transceivers. The MCU was chosen specifically for its handling of CAN frame buffering (see later).

The hardware also includes an external I/O connector, which provides six connections to the general purpose I/O (GPIO) pins of the MCU (which can be configured to be a serial port, analog in, analog out or digital I/O).



The two CAN interfaces are:

- Inside. This is the trusted CAN bus where normal traffic operates.
- Outside. This is the untrusted side and the side an attacker is assumed to access.

2.2 Software

The software consists of:

- A bootloader for programming the main flash memory
- A simple flash ‘file system’ to store configuration information, firewall rules, event logs, and encryption keys
- CAN drivers for the dual CAN controllers

- AES and CMAC encryption software
- Rule execution functions
- Event logging system
- Management server

The software is written in MISRA C to run on pure 'bare metal' system. No RTOS or other external software (e.g. libraries) is used.

2.3 Configuration

The system is configured off-line by defining the properties and firewall rules. This is done via a configuration file that describes these, processed by a tool that then produces a binary configuration for download into the flash file system in the gateway. A gateway is turned from a generic to a specific device by this configuration process.

The process is described below:

The configuration file is in JSON format with Jinja2 pre-processor commands (allowing comments and file inclusion, for example). The configuration tool parses the file and produces a binary file for programming into the gateway.

Access to the gateway is via a diagnostic tool that connects over CAN securely. The interface that the gateway communicates on can be configured to include the Outside bus because in some system there is no physical connector giving access to the Inside bus. The access is secured by an end-to-end cryptographic scheme called CryptoCAN and operates a command-response scheme. The commands include:

- Programming the firmware of the gateway
- Programming configuration information
- Rotating encryption keys
- Programming firewall rules
- Extracting event logs

The diagnostic tool interfaces to an SQL key database where the keys for each gateway are stored. The provisioning process for a new gateway and for key rotation will stores in the SQL database the newly created keys for a given device (each device has a unique serial number programmed into the flash file system). This programming is done as a multi-phase transaction so that if the power were to fail on the gateway while being programmed then it would not be bricked.

3 Configuration

3.1 Configuration file

3.1.1 Simple example

A very simple example of the configuration file is given below:

```
{
    "untrusted->trusted": {
        "frames": [
            {
                "can_id": "S0x500",
                "name": "frame1"
            }
        ]
    }
}
```

This is about the simplest example of a configuration: the CAN bit rate settings for both controllers go to defaults and a single forwarding rule is defined from the untrusted bus: a standard ID frame with ID 0x500 is forwarded.

3.1.2 Fields

A frame can be defined to contain *fields*: these are signals packed within the payload of a given CAN frame:

```
    "trusted->untrusted": {
        "frames": [
            {
                "length": 8,
                "clear_unused": true,
                "can_id": "S0x501",
                "name": "frame2",
                "fields": [
                    {
                        "length": 16,
                        "position": 0,
                        "name": "vehicle_speed"
                    }
                ]
            }
        ]
    }
```

Here a single 8-byte frame is coming out from the trusted bus to the untrusted bus. A single field within the frame is defined, of size 16 bits positioned at the last two bytes of the CAN frame (in big endian format, the default). The `clear_unused` setting is a rule that all other fields are to be cleared to zero so that only defined information is passed out of the trusted bus.

3.1.3 Rewrite rules

A frame can include re-write rules that set specified fields to specific values. In the example below the fields of a frame are given fixed values .

```
"rewrite_payload": {
    "CoolFanActResponse_ACM": 0,
    "CoolFanActResponse_ACM_update_bit": 0,
    "SetTimer2Time": 0,
    "LanguageSet": 0,
    "AmbientNoiseRate": 0,
    "SetTimer2Time_update_bit": 0
},
"tag": 13,
"not_forwarded": true,
```

This can be useful in cases where fields are used by off-the-shelf devices, but the policy is not to allow them to see varying sensor values (when from untrusted to trusted this is typically to ensure compatibility but limiting the attack surface, and when from trusted to untrusted this is for confidentiality or backward compatibility).

The example also shows how frames can be given user-defined *tags* that are used in event logging (typically these tags are be used as keys in a user application to identify the frame).

The `not_forwarded` keyword is used to create a drop rule. This is typically used to override an existing configuration file (perhaps one that had been provided by a central authority), normally done by using the pre-processor to include it from an override file (the configuration file format is designed to allow the declaration of a frame across multiple files). It can also be used when defining a frame that the gateway itself uses. For example, the following declares a frame used for commands to control the gateway:

```
{
    "name": "NSP_CMD",
    "length": 8,
    "nsp_command": true,
    "tag": 60,
    "not_forwarded": true,
    "can_id": "E0x1fffe000"
},
```

The `nsp_command` marker indicates that this is a gateway command frame (typically from a diagnostic tool).

As well as payloads, CAN IDs can be rewritten:

```
{
    "name": "EDGE_DIAG_RESP",
    "rewrite_id": "E11111111111111110000000100000",
    "can_id": "S0x7fe"
}
```

This can be used for fitting off-the-shelf devices into a new network.

3.1.4 Real-time rules

The real-time behavior of a frame can be specified:

```
    "untrusted->trusted": {
        "frames": [
            {
                "realtime": {
                    "J/ms": 5.0,
                    "T/ms": 100.0,
                    "frame_budget": 1,
                    "W/ms": 10.0
                },
                "can_id": "S0x500",
                "name": "frame1",
            }
        ],
        "baud": {
            "bits_per_second": 500000
        }
    }
```

The `realtime` rule specifies a frame's period (in this case 100ms) and its frame *jitter* (i.e. the variability in frame arrival time, caused by variable handling time in the sender device and variable latency across the untrusted bus). Frames are allowed to *burst*: a segmented message means a certain number may arrive back-to-back. The frame *budget* indicates the maximum number in the burst, and the frame *window* (specified by `W`) indicates the window in which the burst is confined. The jitter and period apply to the burst of frames. In this example, the burst is of size 1 (i.e. a singleton frame).

The reason for this specific model is twofold:

- It captures segmented messaging without requiring the transmitter to eke out the frames within a segmented message
- It is amenable to timing analysis that can guarantee the latencies of frames on the bus provided they are constrained to this model

The second point is important: a burst can be allowed through the firewall if it fits within the timing model, and its behavior within the constraints model can be analysed on the trusted side to prove that all other frames on the trusted bus will still meet their latency requirements.

The timing analysis for CAN bus is out of scope for this document but it has been used in many projects and a security gateway must be able to constrain traffic from an untrusted side so that the results of the analysis can be relied upon on the trusted bus.

The bus bit rate can also be configured (there are defaults for the bit rate, as well as the other CAN settings such as SJW and sample point).

3.1.5 Group control

Groups of frames can be defined, with groups enabled and disabled by external commands.

```
"group_names": {
    "DIAG": 1,
    "INTERLOCKED_SRS": 2
},
```

Here two groups are being defined. Frames can be allocated to specific groups:

```
{
    "name": "SRS_OFF",
    "tag": 63,
    "enabled_group": "INTERLOCKED_SRS",
    "can_id": "S0x701"
}
```

This indicates that the frame can only be forwarded if the group is enabled. The group can be enabled by default and disabled by remote command (using a gateway command frame).

Groups can be allowed to be enabled only if the interlock switch is set (so that remote commands are prevented from enabling the group if the switch is not set):

```
"interlock_enabled_groups": [
    "INTERLOCKED_SRS"
]
```

Typical uses for groups include:

- Critical frames that should only be allowed if some physical mode is set (e.g. a trailer is connected)
- Frames that should only come from certain sources at certain times (e.g. a diagnostic tool is connected)

3.1.6 Encryption

The gateway supports frames encrypted using CryptoCAN[1].

```
{
   "name": "EDGE_DIAG_CMD",
   "cryptocan": {
        "decrypt_keypair": "EPHEM_DIAG",
        "mode": "decrypt",
        "pdu_id": 3405643778,
        "receive_encryption_bit": 0
    },
    "can_id": "E0x1fffe010"
},
```

Here an encrypted frame called EDGE_DIAG_CMD is defined. It is encrypted with a named keypair and it to be decrypted by the gateway before being forwarded. The keypairs (one for encryption, one for authentication) are stored in the gateway and can be rotated by a programming tool. It is intended that each gateway has unique set of keys so that physically breaking into one specific gateway device does not open up all other gateways.

The secure frame has a defined Protocol Data Unit (PDU) to globally identify the frame (this can map on to a CAN ID but it doesn't have to) since CryptoCAN is an end-to-end encryption scheme: the frame may have originated remotely from a cloud-hosted tool and put on to the untrusted bus by a telematics device, for example. The configuration allows the authentication code to map the CAN ID to the PDU ID to perform the authentication.

[1] CryptoCAN is a scheme for secrecy and authentication that uses two 8-byte CAN frames to carry an authenticated payload

The `receive_encryption_bit` keyword specifies which bit in the CAN ID is used to for encrypted frame sequencing (this is how CryptoCAN ties pairs of CAN frames together).

3.2 Configuration tool

The configuration tool is an executable command line tool that processes the configuration file (or, more specifically, joining a collection of file fragments and parsing those). It builds an abstract definition of the system described by the files and then from this it produces a binary image containing:

- Groups (including enabled on boot and enabled by interlock)
- Key pairs
- Device part number and other configuration information
- Firewall rules (forward/drop rules, real-time rules, encrypt/decrypt rules, etc.)
- Global CAN ID masks (e.g. a J1939 bus would typically have the priority and addressing fields masked out prior to identification)
- Frame identification table (a 'perfect' hash table that the firmware uses to identify the CAN frame from its ID after masks are applied)

This binary image is programmed into a gateway remotely over CAN to provision it as a specific gateway.

3.3 Programming

The gateway supports remote commands for programming, with commands contained in CryptoCAN frames. Each command fits within a single CryptoCAN frame and has optional 4 byte parameter. The table below gives the command set:

PING	This sends a value and expects it sent back.
PING_NO_RESPONSE	This is a ping command but the gateway should not send a response (used because CryptoCAN always drops the first encrypted frame after a reset)
GET_SERIAL_[A|B|C|D]	This is a set of four commands that obtain the serial number of a device. This command-response sequence does not used CryptoCAN because it is part of the bootstrap process to talk to a gateway.
CONNECT	This initiates a secure connection to the gateway.
DISCONNECT	
CONNECTED_PING	This is part of the challenge-response protocol for tool and gateway to authenticate to each other.

SET_KEYPAIR_AES_[A|B|C|D]	This group of four commands sets a specific encryption key to a new value.
SET_KEYPAIR_MAC_[A|B|C|D]	This sets a specific authentication key
WRITEBACK_KEYPAIR	This causes the key to be written to stable storage (this is part of a multipart handshake to ensure that if the power fails during a key update then the gateway is not bricked).
MODIFY_KEYPAIR	
SET_CSPRNG_KEY	This sets the key for the random number generator (the SAMC21 does not have a true random number generator so one is synthesised from a stored secret key).
MODIFY_TOOL_KEYPAIR	These commands set the keypair for the tool-gateway communication.
APPLY_TOOL_KEYPAIR	
CONFIRM_TOOL_KEYPAIR	
SET_SWITCHES	These commands set certain configuration values and are done via a multi-phase transaction to be secure against a power fail or crash during update.
FLUSH_ALL_SWITCHES	
RELOAD_ALL_SWITCHES	
BEGIN_CONFIG_WRITE	These are the commands for writing a new configuration (the firewall rules, etc.) to the gateway. The configuration is not stored in stable storage[2] so there is a process for checking the storage and re-attempting a failed write.
WRITE_CONFIG_DATA	
END_CONFIG_WRITE	
CRC_CONFIG	This command requests the CRC of the configuration. If the CRC doesn't match then the
RESTART_FIREWALL	This remotely reboots the gateway and starts it operating normally.
SET_SERIAL_[A|B|C|D]	These four commands sets the serial number of the gateway.
SET_TSEG1	These commands set the CAN bus properties for the bus that the tool will connect on (the other bus is set from the configuration file written by the tool).
SET_TSEG2	
SET_PRESCALE	
SET_SJW	
SET_FROM_TOOL_SECURE_ID	These change the CAN ID used for communication between tool and gateway.
SET_TO_TOOL_SECURE_ID	
SET_FROM_TOOL_INSECURE_ID	
SET_TO_TOOL_INSECURE_ID	

[2] Stable storage is a non-volatile file structure where the file is always valid and is updated atomically: it cannot be left in an intermediate state if there is (e.g.) a power failure during update. It normally requires storing at least two copies of the data, with a rollback mechanism on boot.

SET_DEVICE_ID	The device ID is used to identify the device on the bus: in general there may be more than one security gateway on a trusted bus and the tool will typically need to communicate with each.
APPLY_FIREWALL_SETTINGS	These commands are used to test a gateways settings are valid before writing them to non-volatile storage to prevent a gateway from being bricked.
CONFIRM_FIREWALL_SETTINGS	
GET_CONFIG_STATUS	
CONFIRM_CONFIG	
RESET_TO_FACTORY	This resets the gateway to factory settings.

As can be seen from the command set, avoiding bricking a gateway is critical: there is a multi-phase update sequence for critical information (e.g. keys, serial number, baud rates) so that if the tool fails to re-connect with the new settings then they will be rolled back. The use of stable storage for these ensures that the last phase consisting of writeback will either succeed or fail to a valid state to allow the tool to re-attempt setting changes.

4 Management

4.1 Introduction

The security gateway provides management functions for a remote device to access. The rationale is that a trusted device on a CAN bus can communicate securely with the gateway to manage it. This might be to enable certain groups (e.g. a secure user interface device might be used to enable diagnostics tool frames being passed through) or it might be used for intrusion detection (e.g. a telematics device might route external commands to the security gateway to upload security events to a central monitoring system).

4.2 Management commands

The gateway accepts management commands through the same interface as the programming tool described earlier, except that a different keypair is used to secure the connection. Depending on the architecture of the management system, those keys may be stored locally (e.g. in a user interface device) and must therefore be stored securely.

The table below shows the management commands:

SNAPSHOT_EVENT	Requests an event is transferred to a snapshot buffer.
GET_NUM_EVENTS	Requests the number of events.
GET_SNAPSHOT_PARAMETER	Uploads an event-specific parameter from the specified snapshot buffer.
ENABLE_FRAME_GROUPS	Enables a group of frames (e.g. over-the-air programming frames)
DISABLE_FRAME_GROUPS	Disables a group of frames
SET_FLAGS	Controls global settings of the gateway
GET_TIME	This is used to sync the local time of the security gateway with the management device's time (e.g. in order to translate event timestamps into real time).
SET_TIME	Sets the gateway's local time.
REQUEST_PART_NUMBER_[A|B|C|D]	Requests a user-specific part number of the gateway. This can be used for configuration management.

4.3 Event logging

Events are logged by the gateway to provide some information about things that have happened. It is not intended to be a full intrusion detection system (IDS) but does provide some functions.

There are six event logs, and they store the following:

- Non-matching Frame (NMF). A frame is seen on the bus that is not identified by the ID matching rules.
- Group. A frame is dropped because the group is not enabled.
- Realtime. A frame is dropped for violating the real-time rules.
- Content. A frame is dropped because a field within the frame violated the range check rule.
- CryptoCAN. A frame was dropped because its authentication code failed.
- System. A system event occurred.

The logs operate in a linear/ring system: a list stores the first n events, and a ring stores the most recent m events. Thus many events will leave a trace of when they first occurred, and of what happened most recently.

Each event logged is assigned a millisecond-accurate timestamp with a range of 136 years. In addition, there are event-specific parameters stored. For example, NMF events store the CAN ID and payload of the non-matching frame. The system event log stores timestamps of management events, including when the time was changed.

The events are logged into RAM and are lost if the gateway is powered off: the assumption is that events are periodically extracted by a management device which either uploads them or stores them locally in non-volatile storage.

5 CAN frame handling

5.1 CAN drivers

The CAN drivers are interrupt driven and run fast enough to remove incoming frames from the CAN controller before the hardware FIFO overflows. The outgoing queueing is done in priority order using the SAMC21 hardware priority system (which queues up to 32 frames and enters them into arbitration in CAN ID order).

The drivers therefore avoid *priority inversion* (that can cause arbitrary delays in a CAN frame buffer).

5.2 Transmit FIFO queueing

FIFO queues for transmission suffer from priority inversion are normally to be avoided. But in one specific case they are necessary: for CAN frames of the same ID that form part of a segmented message. The SAMC21 CAN hardware will pick an arbitrary frame to send when there are several with the same ID (in effect re-ordering frames). This cannot work for segmented messaging, so the gateway allows a FIFO queue to be allocated to a frame, and FIFO queues feed the main priority queue.

If a frame with FIFO allocated is received then it is put into the priority queue unless a prior frame with that ID is already in the queue, in which case it will go into its FIFO queue. When prior frame is transmitted from the priority queue then the head of its FIFO will be transferred to the priority queue.

The policy for a FIFO overflow is set in the configuration file: if `drop_oldest` is set then new frames added to the FIFO cause old frames at the head of the FIFO to be discarded (the default behavior is to drop the newest frames if there is no room).

5.3 Frame jitter

Frames are processed immediately and queued for transmission (if not dropped) immediately, meaning they inherit an outgoing jitter equal to the incoming jitter plus the variability in the processing time (frames with real-time rules attached will of course be dropped if the incoming jitter is too large).

6 NMFTA Requirements Analysis

6.1 Introduction

This section will look at how the Canis Labs security gateway maps on to formal NMFTA requirements for a security gateway.

There is some terminology shift between the Canis Labs terms and the NMFTA terms. These are outlined below.

NMFTA	Canis Labs
"UND"	"Untrusted" or "Outside"
"TND"	"Trusted" or "Inside"
"Abstract Unintended Gateways"	"Hijacked devices" (devices on the trusted bus that have been hijacked and can be used to attack the CAN bus)
"Address claim attack"	Dynamic spoofing (abusing the J1939 dynamic protocol for allocating addresses to ECUs)

6.2 Security Requirements for Abstract Unintended Gateways

These requirements are out of scope for this document.

6.3 Security Requirements for Abstract (Intended) Gateways

6.3.1 Gateway Configuration Protected (AGW-S-000)

Compliant. The configuration is programmed only by an authorized and authenticated connection to a programming tool. The configuration is protected against corruption during programming by stable storage and against memory corruption by a CRC.

6.3.2 Prevents OTA (AGW-S-001)

Compliant. OTA frames from a programming / diagnostic tool can be blocked by the Canis Labs gateway by assigning them to a mode and then having the mode enabled by some means (which could be a secure management command from a telematics gateway, or by a group assigned to the hardware interlock switch).

6.3.3 Prevents DoS (AGW-S-002)

Compliant. The real-time rules can be set in the configuration file to prevent frames flooding the bus (while allowing segmented messaging).

6.3.4 Prevents spoofing (AGW-S-003)

Compliant. Forwarding rules prevent frames originating on the untrusted bus from masquerading as frames originating on the trusted bus.

There are two other types of spoofing beyond this simple spoofing:

1. Frames from the untrusted bus masquerading as frames from *another device* on the untrusted bus.
2. Frames from the trusted bus masquerading as frames from another device on the trusted bus.

Case (1) could happen where an attacker left a wired attack device (like a Raspberry Pi) on the untrusted bus and then when it saw a diagnostic tool start a connection session (after a mode was enabled, for example) it could attack the diagnostic tool with a CAN protocol attack (pushing it error passive or bus off) and then masquerade as the diagnostic tool – an Ambush Attack. This can be addressed by use of cryptographic authentication between the diagnostic tool and the security gateway. The Canis Labs security gateway can authenticate and decrypt CryptoCAN traffic.

Case (2) is the situation where a device on the trusted bus has been attacked via a remote code execution (RCE) exploit. There are generic security requirements for hardening these devices (see Security Requirements for Abstract Unintended Gateways).

Case (2) is not prevented by the Canis Labs security gateway described in this document. Canis Labs has developed hardware IP to detect several CAN protocol attacks (like the Bus Off attack) and this could be incorporated into a security gateway to detect attacks and disable the mode.

6.3.5 Prevents exfiltration (AGW-S-004)

Compliant. The re-write rules in the configuration file describe earlier prevent unwanted exfiltration: either generically erasing all bits not defined by a field or by setting a defined field to a fixed value.

6.3.6 Prevents elevation (AGW-S-005)

Compliant. The use of modes and a management interface / interlock can prevent privileged frames from entering the trusted bus.

NB: The note in section 6.3.4 on the Ambush Attack still stands.

6.3.7 Prevents data loss (AGW-S-006)

Compliant. There are mechanisms to prevent buffer overflows. Caveat: the system configuration (including timing behavior of the trusted bus) must not exceed the limits of the security gateway resources.

Prevention of buffer overflows requires two things:

1. Rate limiting to prevent overflows
2. Analysis of the rate limits to determine *a priori* that the buffer space allocated is sufficient to not have frame drops from overflows

The latter means applying timing analysis to the defined traffic patterns to determine peak loads. All the traffic on the trusted bus should be known so that

those peak loads for egressing frames can be known (recall that the latency jitter must be known).

For the untrusted bus, the timing analysis assumptions may not hold because the traffic on that bus cannot be trusted. This means that the untrusted to trusted flow can only be guaranteed free of data loss only within certain bounds.

As discussed earlier, operation of the untrusted bus cannot be guaranteed (including the cases of CAN frame and CAN protocol attacks from one device to another on that untrusted bus). This is a fundamental weakness of the security gateway approach and there may be a requirement to have several untrusted buses (perhaps via several gateways) to confine specific threats to a specific untrusted bus.

6.3.8 Preserves high side operation (AGW-S-007)

Compliant. The Canis Labs gateway imposes per-frame rate limits and leaves the trusted bus amenable to timing analysis to prove that all timing requirements are met. No CAN protocol attacks can come through the gateway because there is no direct access to the CAN TX pin of a transceiver connected to the trusted bus.

NB: The risk of a hijacked device on the trusted bus remains and is addressed by other requirements (see "Security Requirements for Abstract Unintended Gateways").

NB: Timing analysis needs to be carried out (using appropriate tool support) to prove that the rate limits defined in the configuration file are sufficient to protect the bus.

6.3.9 Security assurance (AGW-S-008)

Compliant. This requirement refers to the security of the gateway itself: if the gateway is not secure then it could itself be hijacked and overridden. The Canis Labs gateway addresses this in several ways, including by avoiding all external libraries and being pure bare metal software.

There is further discussion of this later in this document.

6.3.10 Preserves performance (AGW-S-009)

Compliant. This requirement specifies that the performance of the trusted bus can be guaranteed. That that requires both (1) that there can be no performance degradation from traffic passed through and (2) that also traffic that is passed through is done so within timing constraints.

Case (1) is met by the real-time rules which place constraints on the traffic that can be passed through. This does require that timing analysis is performed on all the traffic on the trusted bus with formally defined latency deadlines for all frames so that this requirement can be proved.

Case (2) is met by timing analysis on the security gateway software to determine worst-case response times between the arrival of an incoming frame and its placement in the outgoing CAN controller. Note that the jitter reduction scheme

of holding back an early frame needs to be excluded when performing this analysis for the security gateway.

6.3.11 Mode switch (AGW-S-010)

Compliant. The mode switch support in the Canis Labs gateway is sufficient to achieve this.

NB: A mode can be activated by a physical mode switch (which is grounded to activate) or by an authenticated encrypted command from an authorizing source (local or remote).

6.3.12 Mode switch indicated (AGW-S-011)

Compliant. The status of the Canis Labs gateway is reported in response to a secure command from a management device. This management device can periodically request the status and report it in human readable form.

6.3.13 Security Requirements for CAN Gateways

6.3.13.1 Performant (CGW-S-001)

Compliant. All rules pertaining to a frame need to execute in less CPU time than it takes to receive the frame. But note that certain configurations may result in non-compliance if the performance of the hardware is insufficient.

Frame identification is one of the most CPU-intensive parts to handling incoming frames. Some CAN controllers help with hardware support but few can cope with the number of different frames a gateway could be required to support. The requirements ought to include a lower bound on the number of frames supported.

The real-time drop rules are expensive operations in software: they require accurate timestamps (to better than a microsecond) but a long range. The Canis Labs gateway software uses 64-bit arithmetic for this to ensure monotonicity and simpler equation evaluation, but this is CPU intensive on a low-end 32-bit MCU.

The field rules in the Canis Labs gateway are transformed to 64-bit operations on a payload, so masking out unused bits and setting fields to constant values is done in one operation (the values are pre-computed by the configuration tool). Range checking the fields however is a serial operation, and custom hardware designed for this could perform the operations in parallel.

The Canis Labs gateway performs cryptographic operations in software, and this is slow. Hardware acceleration for this is necessary to be able to process frames at full speed.

NB: Domain-specific hardware designed for gateway operations can be used to accelerate the following operations:

- Handing frame identification in hardware using an ID table
- Implementing real-time drop rules in hardware
- Handling field rules in parallel
- Encryption and authentication

6.3.13.2 Re-write / Masking (CGW-S-002)

Compliant. The Canis Labs re-write and masking rules are included in the configuration file description.

6.3.13.3 Preserved Atomic Multicast (CGW-S-005)

Compliant. The atomic multicast feature of CAN is preserved by not dropping legal frames (the atomicity property of CAN is that every device connected to a bus and operating in error active mode receives a frame at least once). Caveat: that this property cannot be guaranteed for traffic from the trusted to untrusted bus because the untrusted bus may not be performant.

NB: Legal frames are dropped only if there is insufficient buffer space to hold them. This happens if the frames queued for transmission are not sent rapidly enough, and that happens for one of two reasons:

1. The system goes through periods of *legal* transient overload where frames just happen to take close to their worst-case latencies, but this was not allowed for in the sizing of buffers for the security gateway.
2. The system goes through periods of *illegal* transient overload where frames exceeded their worst-case latencies and although the worst-case latencies were allowed for in the sizing of buffers, this is not sufficient to eliminate frame dropping.

In case (1) the security gateway was not configured according to systems analysis that determined the worst-case performance. This indicates a derived requirement that timing analysis must be performed on the trusted bus to meet this requirement. This is only valid for the trusted bus: the untrusted bus cannot be guaranteed to be performant.

In case (2) the system did not behave correctly. The untrusted bus cannot be guaranteed to be performant and in this case the buffer space may well be insufficient (for example, high priority traffic is flooding the bus). Guarantees of atomicity in this direction are therefore conditional.

6.3.13.4 Won't Drop Frames (CGW-S-005a)

Compliant. This is derived from CGW-S-005. See the discussion of that requirement.

6.3.13.5 No Priority Inversion (CGW-S-005b)

Compliant. The Canis Labs gateway drivers use the SAMC21 hardware to transmit frames in a 32-deep priority queue. Caveat: a configuration must not exceed the limits of an implementation.

NB: the priority queue may not be deep enough for all frames (see frame dropping discussion earlier), depending on the configuration. To validate this requires timing analysis performed on the destination bus.

6.3.13.6 Preserves Ordering (CGW-S-005c)

Compliant. FIFO queues that feed into the main priority queue are created if specified in the configuration file. A FIFO is for a specific frame ID only and therefore the requirement for no priority inversion (CGW-S-005b) is met.

6.3.13.7 FIFO but also Priority (CGW-S-005d)

Compliant. The size of the FIFO and the semantics are defined for dealing with overflows (drop oldest vs. drop newest). Note that overflows can be prevented by using timing analysis (see discussion for requirement CGW-S-005).

6.3.13.8 Preserves Jitter (CGW-S-005e)

Non-compliant. The Canis Labs gateway does not control jitter (except insofar as rate limiting frame arrivals). To implement this requirement a frame that arrives 'early' (in relation to its period) should be held over from being queued until this minimum time since the last arrival has expired.

For frame bursts, the burst itself can be held back (i.e. a segmented message can be made to be periodic) but within a burst frames can still be sent as fast as they arrive.

The implementation of this requirement in software is difficult: a delay in handling the timing event for the expiry of the early arrival holdback time will be a new source of jitter (many frames may have their expiries coincide, causing considerable CPU time to handle this).

Canis Labs is designing its own CAN buffering system to implement this holdback time in hardware, so the frame can be put into the queue but with an expiry time and the hardware will automatically process this time without CPU intervention.

6.3.13.9 Impervious To Address Claim Attacks (CGW-S-100)

Compliant. The address claim attack requires a low-level CAN protocol attack, and the security gateway prevents this from taking place on the trusted bus (see discussion for requirement AGW-S-007). Caveat: This applies to the trusted bus only because an address claim protocol on the untrusted bus will not be prevented.

7 Additional requirements

7.1 Introduction

The requirements in this section are derived from requirement AGW-S-008 (Security Assurance) to harden the security of the security gateway itself (there have been instances of security gateways themselves containing vulnerabilities and therefore being a vector for attacks).

7.2 Secure hardware

There is an additional set of requirements on the security gateway not met by the Canis Labs gateway: a secure hardware platform (the Canis Labs gateway predates the wide availability of secure MCU hardware).

A secure hardware platform should:

- Resist physical exfiltration of data (e.g. prevent reading out of configuration data and firmware via a JTAG connector)
- Not permit reprogramming by unauthorized firmware (i.e. there should be a secure boot process)
- Store encryption keys securely so that if software in the MCU is compromised (by an RCE, for example) then the keys cannot be read out

The above requirements can be met with an MCU that includes an SHE+ HSM. Today these are widely available devices. Hardware encryption should anyway be included for performance reasons.

7.3 Rollback resistance

The rollback to a previous version of firmware should be prevented: it is a common attack to force the existing firmware to fail on boot and for a system to boot from an older copy kept the system. The reason for keeping an old version of the firmware is normally to prevent a system from being bricked if power is lost during programming (see earlier discussion) but once the system has been securely upgraded then the previous version should be erased.

The same operation applies to cryptographic keys: any previous version of a key should be erased once the operation has completed correctly.

7.4 Secure programming language

There are well-known programming weaknesses that cause problems, such as buffer overflows. Many of these are due to inherently dangerous languages such as C. The Canis Labs gateway firmware is written in C because of availability of tools at the time. Today the SPARK Ada subset would be more appropriate to a high security application: there are formal verification tools of the type that hardware design has been using for some years, plus the open source tools now can generate code for Arm Cortex M and RISC-V based MCUs. If custom hardware is used for the bulk of security gateway features then the software will be

moderate, confined mostly to management, and it should be amenable to formal verification to prove there are zero RCE vulnerabilities.

8 Systems analysis

8.1 Introduction

Several of the requirements discussed in this document highlight the need for a systems view, specifically in the area of real-time behavior. This section discusses what must be done at a system level, how it can be done, and why it must be done.

8.2 Buffer analysis

A security gateway can allocate buffer space for frames (both a single priority queue and also FIFO queues that feed it) but the amount of space to allocate has to come from systems analysis. There are two ways to do this:

- Allocate some buffer space, see if there are overflows, allocate more buffer space and hope that fixes it
- Calculate the worst-case buffer space requirement

The first approach doesn't meet the requirements of CAN atomicity, for example, and dropping frames can introduce all kinds of functional and timing faults, triggering 'ghost' errors in a system.

The second approach works out how long a frame can be in the buffer by looking at the worst-case latencies for frames on the bus and worst-case arrival pattern from the other bus. The worst-case arrival patterns are given by timing analysis – and real-time drop rules can enforce them. Similarly, worst-case latencies are given by timing analysis.

8.3 Timing analysis on CAN

There are two steps to performing timing analysis on CAN:

1. Derive the model for CAN traffic.
2. Calculate the worst-case latencies for the traffic.

Step (1) is done by inherent knowledge of the traffic on the bus (where periodicities and frame IDs are known *a priori* from some database of frames) or by observing the behaviour of a bus for some time to measure frame periodicities. A hybrid approach can be used too: the base model derived from observation and then augmented with human knowledge of the design of a system.

Step (2) is a straightforward calculation based on the behavior of the CAN protocol. Some assumptions are necessary for this: the drivers must be free of priority inversion: when a frame is created by the application and put into the drivers it must also be entered into arbitration. Nearly all CAN hardware can be driven in a way that avoids priority inversion so this requirement is on the software drivers being written properly.

The timing analysis is straightforward and has been used in the automotive industry for at least two decades. Tool support exists, both for automating step (1) and for performing the calculations in step (2). With the right tool hardware, it is

even possible to detect cases of priority inversion in software drivers (for example, Canis Labs has implemented CAN controller hardware that captures low-level events in the protocol that can be used to infer the behavior of software in each ECU).

9 History

9.1 Issue 02 2022-02-07

- Added specific note that this document contains requirements gap analysis
- Formal requirement numbers added to headers
- Discussion on AGW-S-009 (“Preserves Performance”) added
- Discussion on CGW-S-005 (“Preserves Atomic Multicast”) added
- Added section on systems analysis
- Added history section

9.2 Issue 03 2022-03-21

- Updated requirement numbers to match Strictdoc updates
- Restructured text for automatic processing