Atonomi Embedded Device SDK

Developer Guide and API Reference

Document Version 1.0.20180515

Contents

1

Introduction to Atonomi

Atonomi provides a network solution which tackles the question of how to establish identity, trust, and reputation of IoT devices. The blockchain-based approach allows companies to securely exchange services and data to, from, and involving these devices.

The Atonomi whitepaper [1] provides a more complete description of the Atonomi network. The information that follows is a brief overview of the important concepts.

1.1 Device Identity

The concept of device identity (sometimes referred to as the identity token) is essential to securing IoT devices on the Atonomi Network. Based upon the capabilities of the hardware this identity token could be represented by a variety of things. Ideally it is a device_id embedded within a secure hardware element on the device. Device identity will be created during the device or application development process. Developers need to ensure a unique identifier is given to each device as the Atonomi Network doesn't allow duplicate registrations. Identity also incorporates an Elliptic Curve 25519 public/private key pair created during the either the development or manufacturing process, as this is required for later device registration with Atonomi. The SDK includes an EC 25519 key generation tool.

1.2 Device Registration

The Atonomi Identity Registration Service (IRS) is the primary component, that when globally distributed, becomes the central hub of the Atonomi Identity Registry Network (IRN). The IRS is a cloud-based, globally accessible, highly available, high volume service that essentially provides the first step in the Atonomi process. Devices that have been Atonomi enabled via the Atonomi Embedded SDK will contact the Identity Registration Service upon first production boot. Devices will submit to the IRS their identity token. The manufacturer or developer of the device will have pre-registered that devices identity with the IRS via the manufacturer GUI. Assuming the IRS finds a match to the devices identity in its white-list, the device will become activated (within the Atonomi network).

[1] https://uploads-ssl.webflow.com/5a9f110b6e90d20001b2307d/5aa600a2dc199e000140f98a_Atonomi-Network-White-Paper-v0.9.1.3s%20(1).pdf

Note: The Atonomi smart contract includes a device registration function that writes new device_ids to the blockchain when the manufacturer or developer adds new devices to the whitelist via the interface. The Atonomi smart contract also facilitates the payment in ATMI for the Registration of devices.

1.3 Device Activation

The Atonomi Activation Functionality comes into play after the Atonomi enabled device has been sold to an end user. The end user will receive activation instructions that send them to the Atonomi web portal where they will enter the device identifier and pay for device activation via MetaMask. The Atonomi Embedded SDK, integrated into the application code by the developer or OEM, will contact the Activation Service upon first production boot. Assuming the IRS finds a match to the devices identity in its whitelist, its signature is verified, and the activation payment cleared, the device will become activated within the Atonomi network.

Note: The instructions the device owner receives will include a URL for a device activation portal and the public device_id for the device. The portal will be a simple web page, backed by the MetaMask plugin that will take in the device_id and when the Activate 'button' is pressed invoke MetaMask to handle the activation transaction.

1.4 Device Validation & Reputation

Devices that have been Atonomi enabled via the Atonomi Embedded SDK and that have successfully registered and activated on the Atonomi network can begin to utilize the Atonomi Validation functionality. Devices will exchange their signed device identifier with other devices that they want to transact with. Each device can then call the Atonomi Validation Service, pass in its counterpart's signed device identifier and receive back an indicator of whether or not the device is an Atonomi network 'member' or not. In the event that the other device is a member of the network, the Atonomi Validation Service will also send back the current reputation for that device.

Note: For the purposes of Validation the Embedded SDK will have two additional methods for validating another devices identity. The first method will involve producing a CENTRI Protected Sessions handshake package to be sent to another device for validation. The second method will involve taking in a signed device identity (from another device via the Protected Sessions handshake) and submitting it to the Atonomi Validation Service to be validated. The latter method will return a reputation score if the other device is valid.

1.5 Atonomi Smart Contract

The Atonomi smart contract includes three basic functions: Registration, Activation, and Reputation.

The Registration Function of the Atonomi smart contract governs the writing of a new manufacturer member and their Ethereum address to the blockchain during manufacturer setup. It also governs the writing of new device_ids to the blockchain when the manufacturer adds new devices to the whitelist via the interface. It also facilitates the payment in ATMI for the Registration of devices.

The Activation Function of the Atonomi smart contract governs the writing of newly activated device_ids to the blockchain when the device owner boots the device and activates it via the interface. It also facilitates the payment in ATMI for the activation of devices.

The Reputation Function of the Atonomi smart contract governs the writing of updated device reputation scores to the blockchain after the reputation auditors have processed the reports. It also facilitates the payment in ATMI for the reputation score updates.

2

Embedded Device SDK

Atonomi's Embedded Device SDK is a software library providing mechanisms to communicate with the Atonomi network. Atonomi provides a network solution which tackles the question of how to establish identity, trust, and reputation of IoT devices. The blockchain-based approach allows companies to securely exchange services and data to, from, and involving these devices.

The Embedded Device SDK provides a small-footprint code implementation of messaging routines for communications between an IoT device and the Atonomi Identity Registration Network (IRN). Aimed at embedded systems, this library is intended to be easy to use, integrate, and deploy, and provides support for vast numbers of SoCs and operating environments, especially those which leverage ARM Cortex-M or Cortex-A IP cores.

The SDK supports three service endpoints: Activation, Validation, and Reputation.

2.1 Terms

The following terms are used throughout this document and associated source code.

- A *Protected Session (PS)* is a solution developed by CENTRI Technology which provides secure, encrypted communications between devices. Atonomi leverages this solution to secure sensitive data.
- A *message* is a plaintext representation of some form of device-identifying information. These are secured directly via CENTRI Protected Sessions.
- The term *envelope* is a container for data that is to be protected via a Protected Session. Envelopes are encrypted, fully secure, and contain no plaintext *messages*. Plaintext *messages* can be recovered, however, but only after the *envelope* is decrypted (*i.e.*, opened).
- Several bytes are prepended to the protected *envelope* to form the Atonomi *packet*. These *packets* comprise the core messaging protocol between devices and Atonomi servers.
- The term *cross-signing* is a piece of data used by or pertaining to some device that has been signed by another device using that other device's public/private keypair.

2.2 Device Requirements

- Developers are required to implement a callback to obtain random data from a Hardware Random Number Generator.
- The SDK functions use up to fourty-four hundred (4400 bytes) of stack space. Developers are responsible for ensuring this space is available in order to prevent a stack overflow or stack-heap collision.
- errno-compatible declarations must be available. Developers may either include errno.h from a platform's libc, or instead use the provided atmi_errno.h file. The SDK is careful to restrict itself to those values which are consistent across all major UNIX platforms (Linux, BSD, Solaris, AIX, IRIX, etc.), so the source of errno declarations that is most convenient should be preferred.

Note: Any modifications to the SDK code, in whole or in part, may result in a failure to successfully communicate with Atonomi servers.

2.3 Public Repositories & Project Layout

The SDK is made available via a GitHub-hosted git repository [1], where it can easily be cloned or otherwise downloaded for use.

In attempt to maximize ease of use, the SDK consists of a single-file C implementation, which lends itself to being quickly and easily integrated into any new or existing codebases. This exposes the API which developers can use to construct packetized messages for the Atonomi network.

Atonomi is using CENTRI's Protected Sessions to serve as its protocol's cryptographic backend, which is a component **not** accounted for by the C source alone. Due to licensing requirements, that component instead must be provided as a prebuilt static library and is supplied by its vendor, CENTRI Technology. However, builds for multiple architectures are available and can be found within the `lib/` subdirectory. See table 2.1 for more details about supported platforms.

ISA Family	**Core Family**	**Provided Static Library**
`x86_64`	Intel/AMD (64-bit,SSE2)	`libps-centri-x64-W.X.Y.Z.a`
`armv6m`	ARM Cortex-M0/M0+/M1 only	`libps-centri-armv6m-W.X.Y.Z.a`
`armv7m`	ARM Cortex-M3-compatible	`libps-centri-armv7m-W.X.Y.Z.a`
`armv7a`	ARM Cortex-A (32-bit)	`libps-centri-armv7a-W.X.Y.Z.a`

Table 2.1. *Machine architectures currently supported by CENTRI Protected Sessions. Note `W`, `X`, `Y`, and `Z` indicate arbitrary numeric values which collectively denote the library's release version.*

Taken together, the C source and an appropriate CENTRI PS library are all that are required to get up and running.

[1] `https://github.com/atonomi/device-sdk`

2.4 Endpoints & HTTP

The SDK allows one to pack and unpack messages into a secure bytestream for transmission to and from Atonomi servers. Due to not every device necessarily having a means of connecting directly to the Internet, no means of performing this transfer are included in the SDK, and the developer is responsible for providing the means to do this.

Communications are performed via a simple HTTP-based transaction mechanism (HTTP version 1.1, specifically). A secure, packed message is provided to Atonomi servers as the entire body of an HTTP PUT request, and the server will return the corresponding secure, packed response as the body of an HTTP 200 (OK) response. No special content encoding is required for the HTTP transaction, and only two headers are required: "`Content-Type: application/octet-stream`", and "`Content-Length: DDD`", where `DDD` is the numeric length of the packed request in bytes, expressed in base-10 decimal form.

Corresponding endpoint URLs for the supported request types are:

Action	**Endpoint URL**
Activation	`http://device.atonomi.net/activation`
Validation	`http://device.atonomi.net/validation`
Reputation	`http://device.atonomi.net/reputation`

3 API Reference

3.1 Common Library Context

An `atmi_context_t` provides a context to API functions with keying information. This structure must be populated with public and private keys for use during the packing or unpacking of messages.

Context Structure

```
typedef struct {
        uint8_t   publicKey[32];
        uint8_t   privateKey[32];
} atmi_context_t;
```

The `atmi_context_t` structure is only needed for read access and may reside in constant / non-volatile memory if desired. Alternatively, a developer may prefer to construct this on the stack prior to each use and explicitly clear it afterwards, minimizing exposure to the private key. Both methods are acceptable.

```
void ATMIsign_device_id(const atmi_context_t *ctx,
                        atmi_session_t *ssn,
                        uint8_t       idsgn_out[72],
                        const uint8_t devid_in[32]);
```

The `ATMIsign_device_id` function is used to cross-sign another device's Device ID, `devid_in`, used to create Device Validation and Device Reputation requests. See those sections below for further details.

```
extern void ATMI_memrand(void *p, size_t n);
```

The CENTRI component of the Atonomi packet requires a source of entropy in order to create any new packages (i.e. an RNG). Due to a limitation in how they are currently generated, this is exposed not by function pointer but by declared symbol that the linker is expected to locate.

The `ATMI_memrand` symbol has been aliased to an Atonomi-specific symbol name for improved clarity and must be provided by the developer: he or she must declare a function with this exact signature below. The function must have C linkage and should obtain the specified number of bytes of random entropy from a hardware RNG or other similar source and write them into the location provided.

Note: A CSPRNG (cryptographically secure PRNG) could also suffice in some cases. All implementation details and choices are left up to the developer, whom is expected to understand the ramifications and potential security impacts resulting from said choice.

3.2 Session-based Messaging

The Atonomi network protocol uses session-based messages, where every request requires a response to be generated and processed. In order to accomplish this on the device, some memory must be reserved to store session data for use in processing the response to a corresponding request. Note this data is unique to each request packet generated. Once the response is received for that request, however, the state data may be discarded.

Session Structure

```
typedef struct {
        uint8_t    state[ATMI_SESSBUF_STATE_SIZE];
        uint8_t    packet[ATMI_SESSBUF_SIZE];
} atmi_session_t;
```

The developer is responsible for allocating an `atmi_session_t` structure, which provides storage for packet state during a transfer, as well as a working buffer for operations. All message operations clobber the contents of this buffer; all packing operations place their output data in this buffer.

3.3 Messages for Device Activation

Activation Request and Response Structures

```
typedef struct {
        uint8_t  id_requestor[32];
} atmi_act_request_t;

typedef struct {
        int32_t  success;
} atmi_act_response_t;
```

The `id_requestor` field should be populated with the device ID that corresponds to the device making the request.

The `success` field returned from the server will be zero if the request was successful, and negative otherwise.

Activation Request Packing and Response Unpacking Functions

```c
int ATMIpack_act_request(const atmi_context_t *ctx,
                         atmi_session_t *ssn,
                         const atmi_act_request_t *act);

int ATMIunpack_act_response(const atmi_context_t *ctx,
                            atmi_session_t *ssn,
                            const void *pinbuf, size_t nin,
                            atmi_act_response_t *act);
```

The pack request and unpack response functions require `ctx`, a pointer to the Atonomi library context structure with key data, and `ssn`, a pointer to a new or current session structure. The pack function will construct an encrypted message with the data in `act`. The unpack function will read `nin` bytes of packed, encrypted data from `pinbuf`, decrypt it, and place the response contents in `act`.

3.4 Messages for Device Validation

Validation Request and Response Structures

```c
typedef struct {
        uint8_t  id_requestor[32];
        uint8_t  id_requestor_xsigned[72];
        uint8_t  id_subject[32];
} atmi_val_request_t;

typedef struct {
        int32_t  success;
        int8_t   as_initiator;
        int8_t   as_responder;
        int8_t   as_consumer;
        int8_t   as_provider;
} atmi_val_response_t;
```

The `id_requestor` field should be populated with the device ID that corresponds to the device making the request. The `id_subject` field should be populated with the device ID of the device to validate. The value of the `id_requestor` field must be sent to the device associated with `id_subject` to be cross-signed; this cross-signed result must be used to populate the `id_requestor_xsigned` field. Additionally, the caller must keep a local copy of this value if a reputation amendment request will be submitted, since the reputation amendment request message also requires this value. This cross-signed result is non-constant and must be repeated for each new Validation request.

The `success` field returned from the server corresponds to the subject's reputation score. This will be negative if the request was unsuccessful.

The `int8_t` fields indicate the current reputation state for the subject device.

Validation Request Packing and Response Unpacking Functions

```
int ATMIpack_val_request(const atmi_context_t *ctx,
                         atmi_session_t *ssn,
                         const atmi_val_request_t *val);

int ATMIunpack_val_response(const atmi_context_t *ctx,
                            atmi_session_t *ssn,
                            const void *pinbuf, size_t nin,
                            atmi_val_response_t *val);
```

The pack request and unpack response functions require `ctx`, a pointer to the Atonomi library context structure with key data, and `ssn`, a pointer to a new or current session structure. The pack function will construct an encrypted message with the data in `val`. The unpack function will read `nin` bytes of packed, encrypted data from `pinbuf`, decrypt it, and place the response contents in `val`.

3.5 Messages for Device Reputation

Reputation Request and Response Structures

```
typedef struct {
        uint8_t  id_requestor[32];
        uint8_t  id_requestor_xsigned[72];
        uint8_t  id_subject[32];
        uint8_t  comms_initiator;
        uint8_t  comms_replyreceived;
        uint8_t  comms_successful;
        uint32_t comms_noreplytmout_s;
} atmi_rep_request_t;

typedef struct {
        int32_t  success;
} atmi_rep_response_t;
```

The `id_requestor` field should be populated with the device ID that corresponds to the device making the request. The `id_subject` field should be populated with the device ID of the device to validate. The value of the `id_requestor` field must have been sent to the device associated with `id_subject` to be cross-signed; this cross-signed result is used to populate the `id_requestor_xsigned` field, and must match exactly the contents used to populate the preceeding Validation request. The cross-signed result is good for exactly one set of Validation and Reputation requests for a particular `id_subject` and can be destroyed by the caller after the Reputation request has been submitted.

The `comms_initiator` field should be populated with a non-zero value if the device making the reputation amendment request initiated communications with the other device.

The `comms_replyreceived` field should be populated with a non-zero value if any sort of reply was ever received from the other device.

The `comms_successful` field should be populated with a non-zero value if all communications with the other device completed successfully.

The `comms_noreplytmout_s` field should be set to the communication timeout length. This is the number of seconds after attempting to initiate communication with another device before the other device's response was received.

The `success` field returned from the server will be zero if the request was successful, and negative otherwise.

Reputation Request Packing and Response Unpacking Functions

```
int ATMIpack_rep_request(const atmi_context_t *ctx,
                         atmi_session_t *ssn,
                         atmi_rep_request_t *rep);

int ATMIunpack_rep_response(const atmi_context_t *ctx,
                            atmi_session_t *ssn,
                            const void *pinbuf, size_t nin,
                            atmi_rep_response_t *rep);
```

The pack request and unpack response functions require `ctx`, a pointer to the Atonomi library context structure with key data, and `ssn`, a pointer to a new or current session structure. The pack function will construct an encrypted message with the data in `rep`. The unpack function will read `nin` bytes of packed, encrypted data from `pinbuf`, decrypt it, and place the response contents in `rep`.