Bundle Protocol Security Library (BSL) v1.1 User Guide

DOC-005922, Prepared by The Johns Hopkins University Applied Physics Laboratory

License

This work was performed for the Jet Propulsion Laboratory, California Institute of Technology, sponsored by the United States Government under the prime contract 80NM0018D0004 between the Caltech and NASA under subcontract 1700763.

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
Initial	11 November 2025	Initial release for BSL v1.0.0. Affected sections or pages: All	
Revision A	15 May 2026	Updates for BSL v1.1.0. Affected sections or pages: Sec 2.4.1 removed EID callback limitation. Sec 3.2 added for secondary interactions. Sec 3.3 and 3.4 clarified BSL backend registrations. Sec 3.4.1 added for crypto discussion. Sec 5 relocated Default SC. Sec 6 relocated Example PP. Sec 7 relocated Mock BPA.	

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This User Guide provides an overview of the application programming interface (API) and high-level workflows of the Bundle Protocol Security Library (BSL), which is part of the NASA Advanced Multi-Mission Operations System (AMMOS) suite of tools.

1.1 Identification

Property	Value
Configuration ID (CI)	681.4
Element	Systems Engineering Office (SEO)
Program Set	Bundle Protocol Security Library (BSL)
Version	1.1

1.2 Scope

This document describes and explains the API and workflows of the BSL. For technical details about the BSL architecture, installation, upgrade, monitoring, and maintenance see the BSL Product Guide. Details about specific API structures and cross-relationships is provided in the online BSL API Docs.

1.3 Terminology

The following are Generic terms:

Application Programming Interface (API) The programming language-level interface defined by a library. For the C-language this is defined by header files in a specific directory structure.

Application Binary Interface (ABI) A combination of instruction set architecture (ISA) and operating-system-specific binary forms for libraries and executables. A built library has a specific ABI that can be different on different platforms even if its API does not change.

Concise Binary Object Representation (CBOR) A binary encoding defined in [RFC8949] which follows a superset of the JSON data model (see below) and enables both small encoded size as well as efficient encoding and decoding. The BSL itself uses CBOR to encode the contents of BPSec ASBs.

JavaScript Object Notation (JSON) A text encoding defined in [RFC8259] which allows a limited data model to be encoded in a human-readable form. The BSL does not use JSON directly, but the example ION-heritage Policy Provider uses JSON for encoding policy configuration.

JSON Web Key (JWK) A JSON data structure defined in [RFC7517] which represents a set of cryptographic keys and their parameters. The BSL does not use JWK directly, but the Mock BPA uses JWK for its key store configuration.

The following are BP- and BPSec-related terms:

Bundle Protocol (BP) The overlay network protocol used to transport BPSec blocks and target blocks between nodes defined in [RFC9171].

Bundle Protocol Security (BPSec) The mandatory-to-implement security mechanism to protect blocks of a BP bundle defined in [RFC9172]. This is the principal scope of behavior implemented in the BSL.

BP Agent (BPA) The instantiation of a BP node with a unique administrative Endpoint ID. A single BPA may have any number of additional endpoints registered to various applications on its node.

BP Endpoint The source or destination of a BP bundle, identified by a BP Endpoint ID (EID).

BP Endpoint ID (EID) The identifier of a BP Endpoint; names the source and destination for a BP bundle.

Bundle (per BPv7) The protocol data unit of Bundle Protocol, which uses a CBOR-encoding of its data.

Block (per BPv7) Each sub-element of a bundle. All bundles contain a mandatory primary block, any number of extension blocks, and a mandatory payload block. Each extension block has an explicit block type identifier.

Block-Type-Specific Data (BTSD) The arbitrary-length binary data containing the contents of a block which is block-type-specific.

Block Integrity Block (BIB) A well-known block type used for integrity operations in BPSec.

Block Confidentiality Block (BCB) A well-known block type used for integrity operations in BPSec.

Abstract Security Block (ASB) A the block-type-specific data for one of the security block types: BIB or BCB, which contains an encoded CBOR sequence.

Security Operation (per BPSec) A single security operation is a combination of choosing a type of security (integrity or confidentiality), a single role, a single target (block), a single security context, and a set of options that are context-specific.

Role (per BPSec) This determines the action of a security operation, as one of:

- **Source** This role causes a security operation to be added to a security block.
- **Verifier** This role verifies, but does not modify, a security operation within a security block.
- **Acceptor** This role verifies and then removes a security operation within a security block.

Security Context (per BPSec) Each security operation has a single associated BPSec context, identified by its Context ID. Context IDs can either be well-known, and registered with IANA, or taken from a reserved block for private or experimental use.

Target (per BPSec) Each security operation has a single target block identified by its unique-to-the-bundle block number.

Parameter (per BPSec) Each security block (the entire ASB) has a set of parameters which apply to all operations in the block.

Result (per BPSec) Each target of a security block has a set of results which apply to a single operation associated with one target.

The following are BSL-specific terms:

BSL Context An container of state and memory allocation for each instance of the BSL. Each BSL context is not thread safe, it must be used within a single thread exclusively.

Bundle Context A container of state and memory allocation for each bundle being processed by a BSL Context.

Policy Provider (PP) An abstract interface (and a C callback descriptor struct) for providing security policy to a BSL Context. The BSL dynamic backend contains a run-time-variable PP registry.

Security Context (SC) An abstract interface (and a C callback descriptor struct) for providing BPSec security context processing to a BSL Context. The BSL dynamic backend contains a run-time-variable SC registry.

Security Action Each action contains an ordered sequence of security operations and their internal configuration. PPs produce sets of actions when inspecting a bundle and operate on the same set of actions when finalizing a bundle.

Security Option An option is an internal-to-BSL item which communicates intent for a single Security Operation between PP and SC.

1.4 References

Title	Document Number
Software Development	57653 rev 10

Table 1.1: Applicable JPL Rules Documents

Title	Document Number
MGSS Implementation and Maintenance Task Requirements (MIMTaR)	DOC-001455 rev I
BSL Software Requirements Document (SRD)	DOC-005735
BSL Software Interface Specification (SIS)	DOC-005835
BSL v1.1 Product Guide	DOC-005921

Table 1.2: Applicable MGSS Documents

Title	Reference
BSL Source	GitHub project BSL
BSL Documentation Source	GitHub project BSL-docs
BSL API Documentation — Main Branch	GitHub Pages for BSL
Programming Languages — C	ISO/IEC 9899:1999
IEEE Standard for Information Technology - Portable Operating System Interface (POSIX®)	IEEE Std 1003.1-2008
M*LIB: Generic type-safe Container Library for C language	GitHub project for M*LIB
QCBOR Library	GitHub project for QCBOR
OpenSSL Library	OpenSSL Project
Jansson Library	GitHub project for Jansson
Unity Test Library	GitHub project Unity
NASA Interplanetary Overlay Networking (ION) software	GitHub project for ION-DTN
Bundle Protocol Security (BPSec) Policy Configuration for ION Open Source (ION-IOS)	ION BPSec Policy Manual
Wireshark Project	https://www.wireshark.org/
JSON Web Key (JWK)	IETF RFC 7517
The JavaScript Object Notation (JSON) Data Interchange Format	IETF RFC 8259
Concise Binary Object Representation (CBOR)	IETF RFC 8949
Bundle Protocol Version 7	IETF RFC 9171
Bundle Protocol Security (BPSec)	IETF RFC 9172
Default Security Contexts for Bundle Protocol Security (BPSec)	IETF RFC 9173

Table 1.3: Applicable Other Documents

Chapter 2

Architecture

The BSL is a set of software libraries and plugin modules which together perform the functions required by Bundle Protocol Security (BPSec) [RFC9172] and its Default Security Contexts [RFC9173] in a way which can be instantiated from and used by a Bundle Protocol Agent (BPA) operating according to the BPv7 specification [RFC9171].

The BSL is made to interact with its environment (BPA, libraries, host OS, *etc.*) through function calls into and out of the BSL library based on a [C99] Application Programming Interface (API) from header declarations and corresponding Application Binary Interface (ABI) for compiled libraries.

2.1 Factory Default Configuration

The "factory default" BSL is configured to operate with a "dynamic backend" which uses dynamic heap allocation and variable-sized data containers (arrays, lists, maps, *etc.*). An alternative backend could be developed for specific BPA needs, but that is outside the scope of the BSL project.

The factory default BSL also builds example Policy Providers and example Security Contexts in order to be able to fully exercise the BSL behaviors. Alternative Policy Providers are expected to be developed for each deployment. Alternative Security Context implementations are expected to be developed for future contexts, and to adapt to deployment-specific needs such as for key management or specialized cryptographic interfaces.

2.2 Library Associations

The BSL project is based on a single source tree but is subdivided into separate libraries, each with their own cross-dependencies (for compiling and linking). The dependencies between areas are shown in Figure 2.1, where the single "BSL" block is meant to represent both the frontend and backend libraries for simplicity.

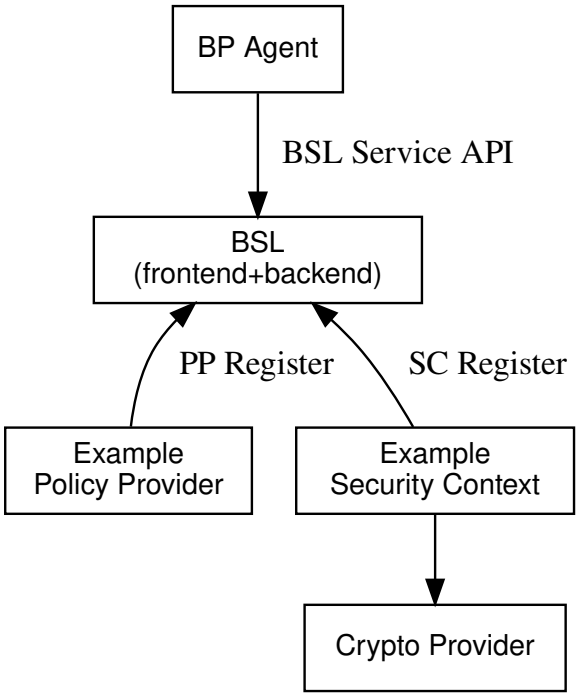

Figure 2.1: Logical Structure and Dependencies

The BSL is internally separated into two layers of implementation: an API-centric abstract Frontend library and a host-binding concrete Backend library.

The Frontend library provides the service API for the BSL to be called by its associated BPA as needed and for stable public APIs used by Policy Provider implementations and Security Context implementations. The Backend library implements forward-declared structs and functions from the Frontend using specific concrete data containers, algorithms, *etc.*

2.3 BPA Interaction Points

Most interactions with the BSL/frontend API occur within the context of a single bundle. There are four points along bundle traversal where BSL interaction from the BPA is necessary:

1. After bundle **transmission** from an application source (APPIN).
2. Before bundle **delivery** to an application destination (APPOUT).
3. After bundle **reception** via a CLA (CLIN).
4. Before bundle **forwarding** via a CLA (CLOUT).

This is depicted in Figure 2.2, where each of the edges in that diagram indicates a call from the BPA into the BSL to process security on that single bundle at the specific location. A more detailed view of the BSL processing sequence at each of those interaction points is described later in Section 4.3.

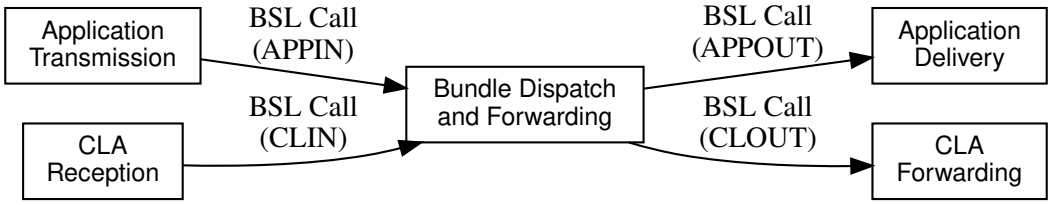

Figure 2.2: Interaction Points from the BPA into BSL

In addition to the bundle-related calls into the BSL and from the BSL, there are also other calls related to:

1. Heap memory management callbacks used by the BSL. If not provided, the standard `libc` functions for heap allocation are used.
2. Logging callbacks related to determining if logging is enabled at a specific severity level and to record a log event.
3. Telemetry counter access from each BSL instance through a synchronous call from the BPA.

These secondary callbacks are registered by the BPA as part of the initialization of Section 4.1 and explained in more detail in Section 3.2.

2.4 Calling Directions

Separate from the structural aspects of the BSL and its external APIs, there are different directions that calls are being made for different interactions between the same entities. For the factory configuration, using the dynamic backend, these interactions take two forms:

Frontend API These are calls into the BSL based on functions declared in its library headers. These functions are all declared by the BSL frontend library and defined by its backend library. The portion of the frontend API used by the BPA to initiate security processing is called its **Service API**.

Callback API Function calls from the BSL backend into its environment, either to its host BPA or to one of its registered PP or SC instances. The signature of these callback functions is declared by the BSL, but their definitions are made outside the BSL. The specific callback function (pointers) registered to the BSL are determined by the host BPA at the time of BSL context initialization (see Section 4.1).

The initiating calls are directed from the BPA into the BSL, and then into PPs (to inform what security operations need to be done) and SCs (to actually execute the security operations). This is depicted in Figure 2.3, where the BPA initiates the sequence using the frontend API and the BSL calls into each PP and necessary SC based on callbacks injected into the BPA at their times of registration.

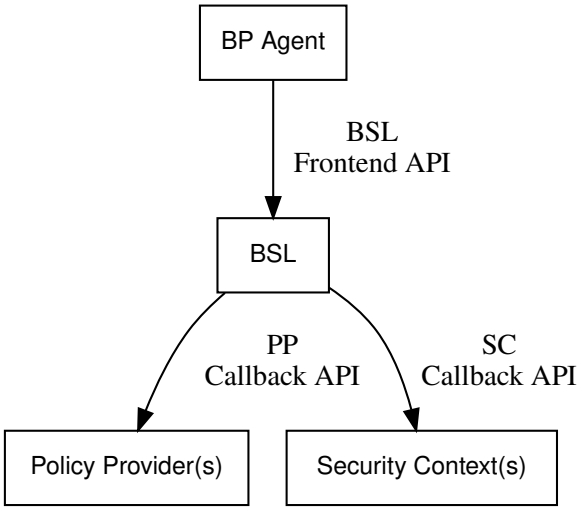

Figure 2.3: Calls Directed From the BPA

As part of normal PP and SC processing, they will need to obtain data from the bundle, access its various blocks, and obtain some data from the BPA itself. Some of these calls will originate from the BSL itself and some will originate from the PP and SC instances, passing through the BSL based on its own frontend API. This is depicted in Figure 2.4, which elides the ultimate source of each of these calls (which is always the BPA).

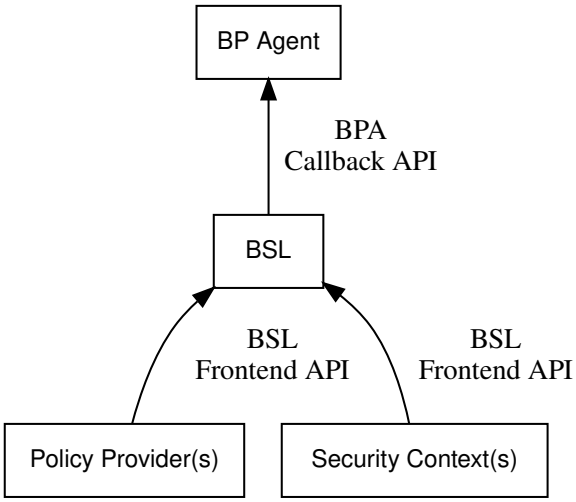

Figure 2.4: Calls Directed To the BPA

Chapter 3

API Overview

The following section provides an overview of the BSL frontend, dynamic-backend, and crypto APIs and references to specific sections and structures from the online BSL API Docs for more detailed C-language documentation. This implementation corresponds to the header-only library `bsl_front` and shared libraries `bsl_dynamic` and `bsl_crypto` as explained in the BSL Product Guide.

The information model for how the BSL operates is built upon the BPSec terminology listed in Section 1.3 with its own additional terms needed for internal logic.

3.1 Bundle Interactions

All of the discussion in this subsection is at the level of logical entities and information models. It does not map one-for-one with the actual APIs of the BSL, but is useful for explaining terminology and framing explanations at a higher level than the C-language details.

Note
This document uses UML diagrams to depict the logical structure and associations within the BSL. Because the implementation is based in the C language, there is no such concept as an abstract class, inheritance, or virtual function override. The BSL uses the concept of a "descriptor" struct to implement this behavior, which is simply a C struct containing a set of callback function pointers and some user data pointer used as a "self" context to each of the callbacks.

The BSL proper is embodied as a "BSL Context" state, for which a single process can have any number of instances. One BPA option is to have a single BSL Context for all of its security processing, which will save on memory use but will act as a bottleneck if the BPA performs its own bundle processing concurrently at each of the interaction points. Another BPA option is to use a separate BSL Context for each interaction point and operate them independently and possibly concurrently.

Because the function of the BSL is perform security processing on individual bundles, all of the processing of the BSL Context operates on a single "Bundle Context" at a time. The purpose of a Bundle Context is to both relate back to some form of BPA-specific *handle* used to identify the bundle within the BPA, as well as keeping BSL-specific state derived from the BPA-supplied bundle data such as an efficient look-up table for block types or block numbers.

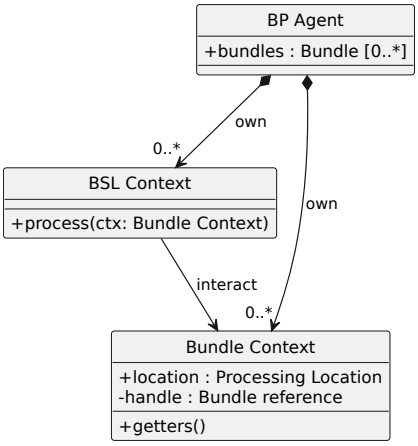

Figure 3.1: BPA and Bundle Context Associations

Each BSL Context instance is associated with one or more Policy Provider instances and one or more Security Context instances, as depicted in Figure 3.2. The Policy Providers are used to control *what* the BSL needs to do for a specific bundle, as discussed in more detail in Section 3.3. The Security Contexts are used to validate and actually execute each security operation, as discussed in more detail in Section 3.4.

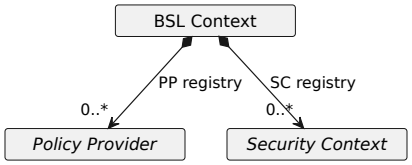

Figure 3.2: BSL Structure Breakdown

In addition to the externally-visible information about security operations and their *target*, *parameters*, and *results* the BSL adds the notion of a security Action which is an ordered sequence of specific operations. This is necessary because some policies require, for example, some operations to be accepted before others are sourced which would refer to the same target block.

Another internal information item is the security Option, which is used to communicate configuration of individual security operations between a Policy Provider and an associated Security Context. Some options are converted by the SC into Parameters or Results that get encoded into the ASB when acting as the Source role. Some options, like key identifiers for the default security contexts, do not have representation in the ASB but are necessary for correct processing of the security operation.

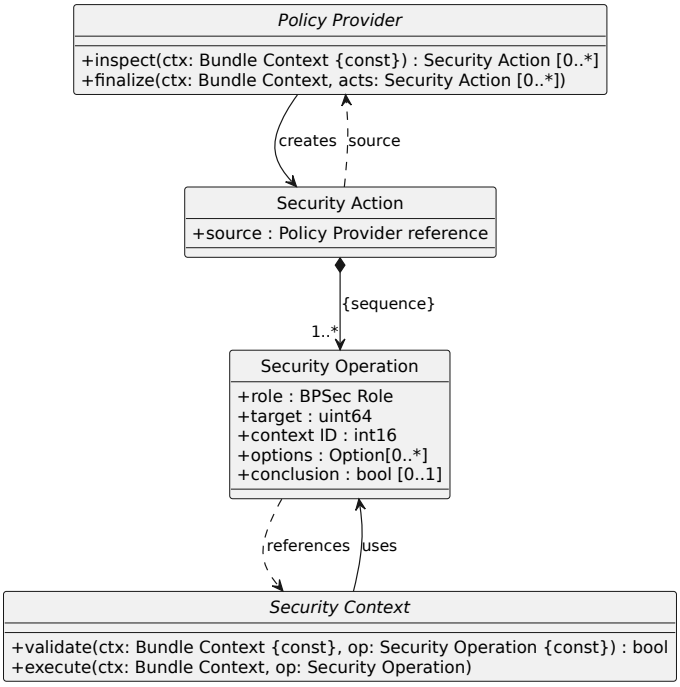

Figure 3.3: Security Operation Associations

3.2 Secondary Interactions

Separately from the bundle-related interactions described in Section 3.1, the BPA and BSL interact through several other interfaces related to infrastructure needs described below and depicted in Figure 3.4. For the dynamic backend, the callbacks are registered during initialization of the process in Section 4.1.

Heap Allocation For the dynamic backend, heap memory management callbacks are contained in a memory descriptors struct. As a safeguard for consistency, the custom memory functions will only be used when all of the callback pointers are non-NULL. If not provided, the standard `libc` functions for heap allocation are used.

Event Logging There are two callbacks in the dynamic backend related to logging. One is a function to determine if logging is enabled at a specific severity level, which is useful to avoid resource-intensive processing that is only actually recorded at low severity. The other is a function to actually process and record log events, each event includes a human-readable text message as well as metadata about the event such as its severity and the timestamp at which the event occurred. The event interface is similar to the POSIX `syslog` API but tailored to BSL use.

Telemetry Counters Telemetry from each BSL context is accessed through calls into the BSL context via the synchronous function `BSL_LibCtx_AccumulateTlmCounters()`. It is up to the host BPA to determine if, when, and how often to collect BSL telemetry and either log or aggregate the values itself. The content of the obtained `BSL_TlmCounters_t` struct is designed to be aggregated across multiple BSL contexts to make it easy for a BPA to report on statistics at individual interaction points as well as aggregated to whole-node statistics.

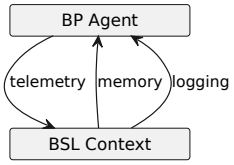

Figure 3.4: Secondary Associations with BPA

3.3 Policy Provider Interface

Policy Providers need to be registered with a library context via the dynamic backend before they can be used. Policy Providers must implement the function headers of the dynamic backend `BSL_PolicyDesc_t` struct defined in the `BPSecLib_Private.h` header file.

Policy Providers must inspect each bundle to produce an Action Set, containing Security Operations. Policy Providers also must finalize over a bundle after each Security Operation has been executed by the security context.

The BSL includes a simple rule-based example PP that may be utilized by any BPA (see Chapter 6), and is used by the Mock BPA for BSL testing (see Chapter 7). This policy provider's data is mutex-protected and may be re-used among multiple threads / BSL contexts (see the Mock BPA for an example).

3.4 Security Contexts

Security Contexts need to be registered with a library context via the dynamic backend before they can be used. Security Contexts must implement the function headers of the dynamic backend `BSL_SecCtxDesc_t` struct defined in the `SecurityContext.h` header file.

Security Contexts operate in the context of a single Security Operation over a bundle. Security Contexts must validate Security Operations for consistency, and process Security Operations on bundles to produce security outcomes.

The BSL includes two Default Security Context implementations, as explained in Chapter 5, which are also used by the Mock BPA for BSL testing (see Chapter 7).

3.4.1 Cryptographic Processing

Both of the default contexts use the BSL frontend for abstracting cryptographic processing. It is expected that alternative and/or future contexts will also use the BSL frontend for abstracting such processing, and that that frontend API will evolve as those needs change.

The BSL backend cryptographic interface utilizes OpenSSL to perform HMAC-signing, encryption, and decryption operations through its "EVP" primitives APIs [OpenSSL].

3.5 Preprocessor Define Directives

The BSL library frontend and its dynamic backend do not rely on any externally configured preprocessor defines to operate normally.

The dynamic backend does use defines to configure how the M*LIB library allocates memory, routing all allocation calls to the host heap allocation callbacks (see Section 3.2).

Chapter 4

Workflows

A simple BPA that utilizes the example policy provider, default security contexts, and dynamic backend could operate with the following workflow:

4.1 Initialization of BPA Callbacks

The following steps are not thread safe and must be performed before any BSL context instances are initialized (in Section 4.2).

1. **Set & Initialize Host Descriptors**: The BSL backend relies on host-specific information from the BPA, such as EID registering and encoding information. The function-pointer fields of a `BSL_HostDescriptors_t` struct should be set with host-implemented functions and initialized with `BSL_HostDescriptors_Set()` for successful BSL operation. See the Mock BPA for a simple example of implementing host descriptors.

4.2 Initialization of a BSL Context

The following steps contain BSL initialization instructions to be performed once (per-thread). The correct operation relies on the host BPA configuration from Section 4.1 to be in-place.

1. **Initialize the Library Context**: Each runtime instance of the BSL is isolated for thread safety within a host-specific struct referenced by a `BSL_LibCtx_t` pointer. Each instance should be initialized using `BSL_LibCtx_Init()`.
2. **Initialize EIDs**: BPAs can register one or more nodes, each of which has a unique endpoint ID (EID). Each EID must be registered with the host using `BSL_HostEID_Init()`.
3. **Register Example Policy Provider with the Library Context**: Register the example Policy Provider with the Library Context.
4. **Initialize Cryptographic State & Register Default Security Contexts with the Library Context**: Initialize the backend cryptographic interface with `BSL_CryptoInit()`. Then, register the two Default Security Contexts ("BIB-HMAC-SHA2" and "BCB-AES-GCM") with the Library Context.

4.3 Single-Bundle Workflow

The following steps should be performed for each bundle being processed, their entity relationships are depicted in Figure 4.1. All of these actions operate within a BSL library context, initialized in Section 4.2.

1. **Initialize Bundle Context for each Bundle**: For each bundle being processed by BPA at one of the four points of interaction (APPIN, APPOUT, CLIN, CLOUT), initialize a bundle context. The bundle context will keep track of a bundle's state throughout its interaction with the BSL. The context must utilize the host-specific struct `BSL_BundleCtx_t`.
2. **Inspect Bundles with Policy Providers**: Utilize the example Policy Provider's inspection function to produce an Action Set that contains Security Operations (Security Operations) to perform on the current bundle context.
3. **Validate Security Operations with Security Contexts**: For each Security Operation contained within the Action Set, utilize the validate function from the relevant Default Security Context to ensure validity and feasibility of the operation.
4. **Execute Security Operations with Security Contexts**: For each Security Operation contained within the Action Set, utilize the execute function from the relevant Default Security Context to perform the operations on the bundle context. The Security Context will produce Security Outcomes which will be returned to the BPA.
5. **Finalize Bundles with Policy Providers**: Utilize the example Policy Provider's finalize function to verify successful security operations, handle unsuccessful operations, and verify bundle consistency.
6. **Free Bundle Context**: The bundle has now completed the required BSL interactions, and the bundle context resources can be released. The bundle can now be forwarded within the BPA.

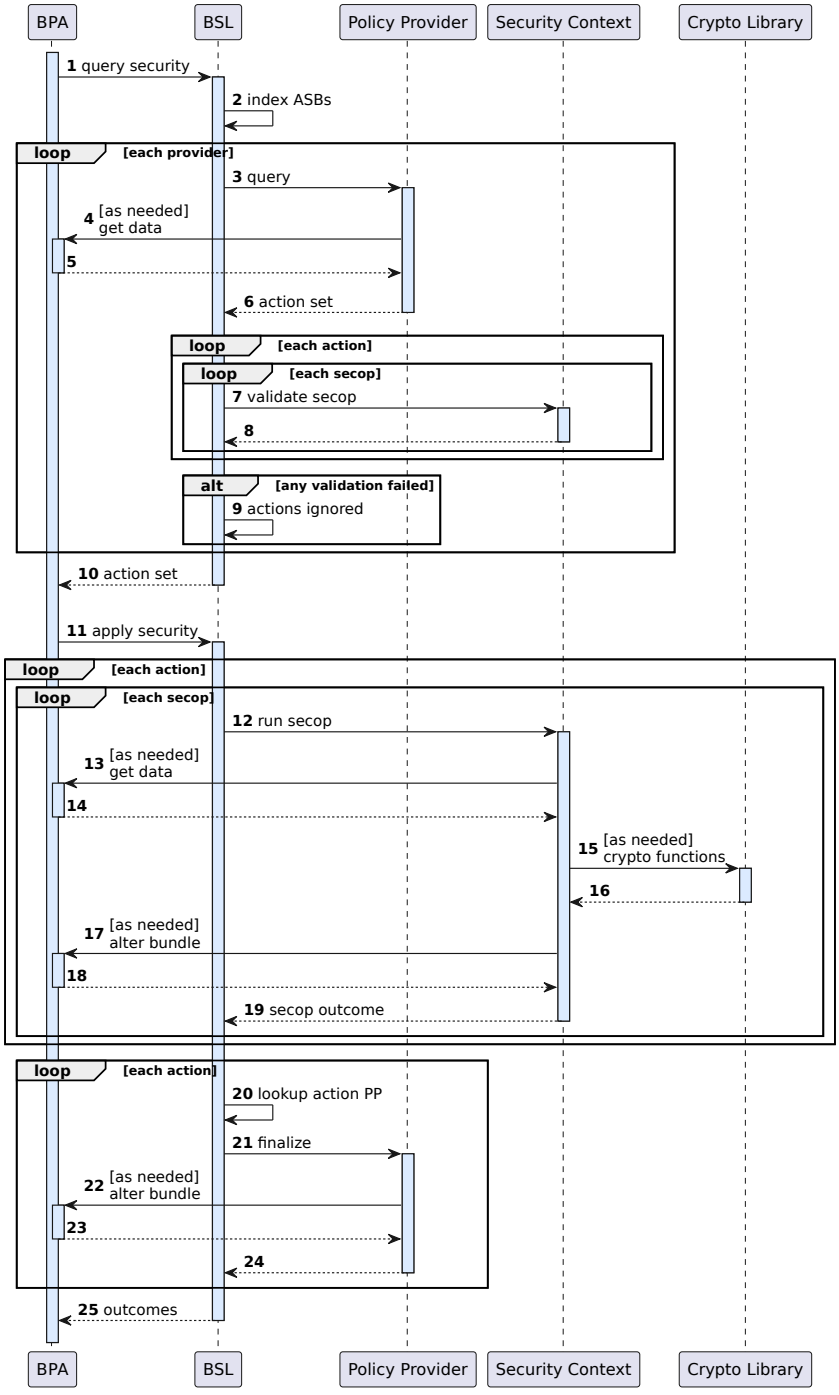

Figure 4.1: Sequence Diagram of Per-Bundle Workflow

The BSL Frontend API consists of two primary functions for per-bundle operation:

1. `BSL_API_QuerySecurity` covers steps 1 through 10 above. The function first utilizes the policy providers to query on a bundle. Next, each security operation in the resulting Action Set is validated using the security context associated

with that security operation.

2. `BSL_API_ApplySecurity` covers steps 11 through 25 above. The function first executes each security operation with its associated security context. Next, the results of each action are finalized using the policy provider which was the source of the action.

4.4 De-initialization of a BSL Context

Before joining or termination of an associated work thread, each `BSL_LibCtx_t` instance should be de-initialized with `BSL_LibCtx_Deinit()` to free its resources.

Each BSL Context is independent of all others, so there is no need to coordinate activities of one with any other.

Chapter 5

Example Default Security Contexts

The BSL source and default build includes implementations of the two Default Security Contexts [RFC9173] as an working example of how to use the BSL frontend and crypto APIs.

This implementation corresponds to the shared library `bsl_default_sc` as explained in the BSL Product Guide and the API Docs subsection on Example Default Security Contexts.

The default security contexts are:

- Context ID 1 "BIB-HMAC-SHA2" for Block Integrity
- Context ID 2 "BCB-AES-GCM" for Block Confidentiality

5.1 Preprocessor Define Directives

The following are preprocessor define directives that limit certain capabilities within the security contexts.

`BSL_CRYPTO_AESGCM_AUTH_TAG_LEN` = 16 The length of an Authentication Tag for AES-GCM encryption and decryption as specified by [RFC9173].

`BSLX_MAX_AES_PAD` = 64 Maximum size of padding added to AES operation by crypto finalize operation. AES-GCM will not produce extra padding, and this value is likely inconsequential.

`RFC9173_BCB_DEFAULT_IV_LEN` = 12 The default initialization vector length as specified by [RFC9173].

Chapter 6

Example ION-Heritage Policy Provider

The BSL source and default build includes a policy provider which has heritage in the ION BPA and is used to configure options for the Default Security Context implementations of Chapter 5. Alternative SC implementations can have other internal options that the example policy provider is unaware of and cannot configure.

This implementation corresponds to the shared library `bsl_example_pp` as explained in the BSL Product Guide and the API Docs subsection on Example Policy Providers.

6.1 JSON-Defined Policy Provider Rules

The sample policy provider has the option to parse and load JSON-encoded ION-like policy rules into the sample policy provider from a file. Policy rules are as specified in [ION-BPSec-Policy-Manual] with some notable differences/modifications mentioned as follows:

`policyrule_set` Each file's JSON encoding must have the attribute `policyrule_set`, which contains a JSON list of ION-like JSON-encoded `policyrule` as specified in [ION-BPSec-Policy-Manual].

`policy_action_on_fail` A full policy action set implementation does not exist yet within BSL. For now, a temporary `policy_action_on_fail` attribute should be set to "delete_bundle", "drop_block", or "do_nothing".

`key_wrap` An extra security context parameter id `key_wrap` will be parsed. When the value is set to "0", key wrapping will be skipped. For all other values, key wrapping will be enabled. Decision to key wrap is a required parameter for security operations when executed within the implementations of the Default Security Contexts.

`loc` An attribute of `filter` that specifies the interaction point (APPIN, APPOUT, CLIN, CLOUT) where the policy rule should be applied.

6.2 Steps to initialize policy provider from JSON

This procedure is how a BPA making use of the example policy provider initializes and registers with each instance of the BSL.

1. Initialize policy provider data with `BSLP_PolicyProvider_Init()`.
2. Initialize policy rules from JSON by calling `BSLP_RegisterPolicyFromJSON()`.
3. Register the policy with BSL by calling `BSL_API_RegisterPolicyProvider()`, the policy where the `user_data` of the policy provider descriptor is set to the initialized policy provider.

6.3 Policy Configuration Examples

An example with two policy rules is shown below.

Example JSON-Encoded Policy Provider

```
{
  "policyrule_set": [
    {
      "policyrule": {
      "desc": "Integrity source rule",
      "filter": {
        "rule_id": "1",
        "role": "s",
        "tgt": 1,
        "loc": "appin",
        "sc_id": 1
      },
      "spec": {
        "svc": "bib-integrity",
        "sc_id": 1,
        "sc_parms": [
          {
            "id": "key_name",
            "value": "9100"
          },
          {
            "id": "sha_variant",
            "value": "7"
          },
          {
            "id": "scope_flags",
            "value": "0"
          },
          {
            "id": "key_wrap",
            "value": "0"
          }
        ]
      },
      "es_ref": "d_integrity",
      "policy_action_on_fail": "delete_bundle"
    },
    "event_set": {
      "es_ref": "d_integrity",
      "events": [
        {
          "event_id": "sop_processed",
          "actions": ["remove_all_target_sops"]
        }
      ]
    }
  },
  {
      "policyrule": {
      "desc": "Confidentiality source rule",
```

```
      "filter": {
        "rule_id": "2",
        "role": "s",
        "tgt": 1,
        "loc": "appin",
        "sc_id": 2
      },
      "spec": {
        "svc": "bcb-confidentiality",
        "sc_id": 2,
        "sc_parms": [
          {
            "id": "key_name",
            "value": "9103"
          },
          {
            "id": "aes_variant",
            "value": "1"
          },
          {
            "id": "aad_scope",
            "value": "0"
          },
          {
            "id": "key_wrap",
            "value": "1"
          }
        ]
      },
      "es_ref": "d_confidentiality",
      "policy_action_on_fail": "delete_bundle"
    },
    "event_set": {
      "es_ref": "d_confidentiality",
      "events": [
        {
          "event_id": "sop_processed",
          "actions": ["remove_all_target_sops"]
        }
      ]
    }
  }]
}
```

6.4 Preprocessor Define Directives

The Example PP does not rely on any externally configured preprocessor defines to operate normally. It does rely on the defines from the Default Security Context to control options of security operations used by the PP.

Chapter 7

The BSL Mock BPA

This is an executable used to provide a test fixture and example BPA integration of specific security context and policy provider instances. This implementation corresponds to the executable `bsl-mock-bpa` and shared library `bsl_mock_bpa` as explained in the BSL Product Guide and the API Docs subsection on Mock BPA.

The Mock BPA does not provide any of the normal processing required of a real BPA by [RFC9171], it is limited to decoding and encoding BPv7 protocol data unit (PDU) byte strings, processing specific BPv7 primary block fields, providing BSL-required integration callbacks, and calling into the BSL for each bundle being processed at each interaction point. The Mock BPA communicates with "the outside" at each interaction point using UDP/IP socket binds configured by command options explained in detail in BSL API Docs.

Users may find it useful to reference the Mock BPA for a working example of library and bundle workflow, and working examples of initializing, registering, and operating the Default Security Context (of Chapter 5) and the Example Policy Provider (of Chapter 6). Exercising of the Mock BPA is part of normal BSL continuous integration (CI) testing and release testing, so it is always in-sync with the BSL APIs.

7.1 Preprocessor Define Directives

The following are preprocessor define directives that limit certain capabilities within the Mock BPA.

`MOCK_BPA_LOG_QUEUE_SIZE` = 100 Number of logging events to buffer before output.

`DATA_QUEUE_SIZE` = 100 Size of the Mock BPA ingress and egress queues for each thread.

Chapter 8

Product Support

There are two levels of support for the BSL: troubleshooting by a system administrator, which is detailed in Section 8.1, and upstream support via the BSL public GitHub project, accessible as described in Section 8.2. Attempts to troubleshoot should be made before submitting issue tickets to the upstream project.

8.1 Troubleshooting

The following provides troubleshooting guidance for the BSL from the perspective of a normal or administrative user. Each situation consists of an observed state followed by a recommended troubleshooting activity.

Why is BSL writing logs to standard error?

The host callback for handling log events has not been set and BSL is falling back to default `stderr` logging. Set the `log_event` (and optionally the `log_enabled_for`) callbacks in the registered `BSL_HostDescriptors_t` struct to handle logging via the BPA.

Why is BSL not using my memory management functions?

All of the members of the `BSL_DynMemHostDescriptors_t` must non-NULL to be used. If some but not all members are non-NULL that is treated as an error (along with a log event) and the default `libc` functions are used.

8.2 Contacting or Contributing

The BSL is hosted on a GitHub repository [bsl-source] with submodule references to several other repositories. There is a `CONTRIBUTING.md` document in the BSL repository which describes detailed procedures for submitting tickets to identify defects and suggest enhancements.

Separate from the source for the BSL proper, the BSL Product Guide and User Guide are hosted on a GitHub repository [bsl-docs], with its own `CONTRIBUTING.md` document for submitting tickets about either the Product Guide or User Guide.

While the GitHub repositories are the primary means by which users should submit detailed tickets, other inquiries can be made directly via email to the the support address dtnma-support@jhuapl.edu.