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

DOC-005921, 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 3.1 to correct package building. Sec 3.4.1, 3.4.2, 4.1.1 to clarify `CMAKE_INSTALL_PREFIX` use.	

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This Product Guide provides architectural and maintenance details about 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 technical details about the BSL installation, upgrade, monitoring, and maintenance. For details about the logical structure, workflows, and application programming interface (API) of the BSL see the BSL User Guide.

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 Design Document (SDD)	DOC-005834
BSL Software Requirements Document (SRD)	DOC-005735
BSL Software Interface Specification (SIS)	DOC-005835
BSL v1.1 User Guide	DOC-005922

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
Security Requirements for Cryptographic Modules	NIST FIPS 140-3
Using SELinux	RHEL9 SELinux Documentation
Packaging and distributing software	RHEL9 Packaging Documentation
Fedora Packaging Guidelines	Fedora Packaging Documentation
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
CMake Reference Documentation	CMake Project
The Ninja build system	Ninja manual
Tito: A tool for managing rpm based git projects	GitHub project Tito
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

BSL Architecture

The BSL is purposefully designed to be a software library independent of any specific Bundle Protocol Agent (BPA) implementation and runtime environment. It is intended to be linked to and used by a BPA during runtime to process BPSec security blocks according to local security policy.

The location of the BSL as a subsystem within a BP Node, operated by a BPA is shown in Figure 2.1. The interactions between the BSL and BPA are twofold: calls into the BSL to provide its security services, and calls from BSL into the BPA to provide agent, bundle, and block data and metadata.

Additionally, BSL security services are needed at four distinct points during bundle processing procedures within the BPA. These are depicted in Figure 2.2 and correspond to the following

- After bundle creation from an application source, augmenting the Transmission procedure of [RFC9171].
- Before bundle delivery to an application destination, augmenting the Delivery procedure of [RFC9171].
- After bundle reception via a CLA, augmenting the Reception procedure of [RFC9171].
- Before bundle forwarding via a CLA, augmenting the Forwarding procedure of [RFC9171].

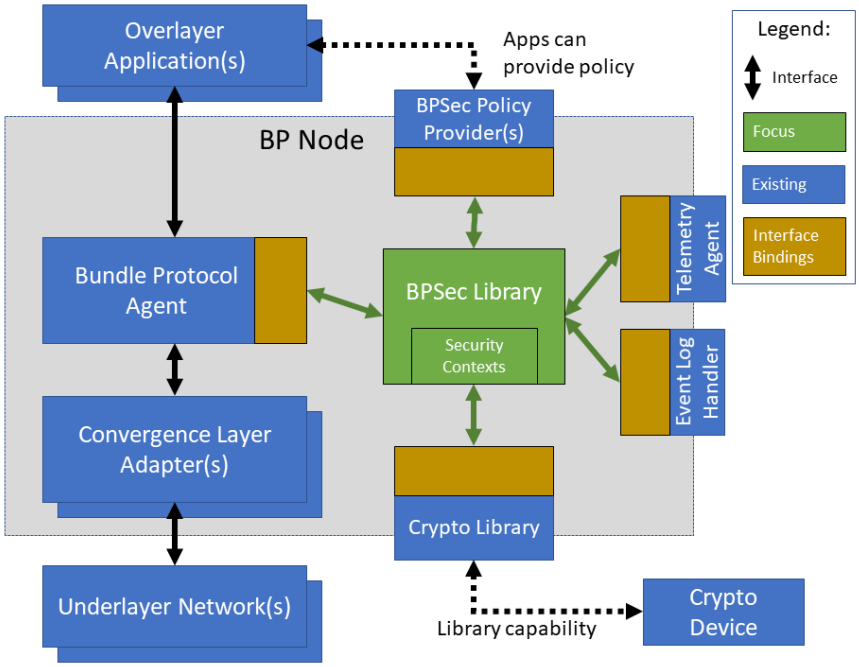

Figure 2.1: BSL System Context

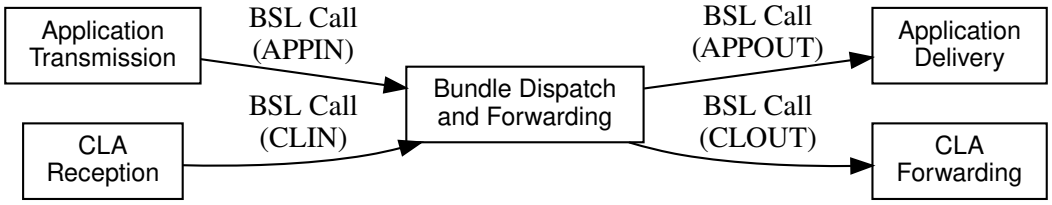

Figure 2.2: Interaction Points from the BPA into BSL

2.1 BSL Components

The BSL source is separated into several different components, each of which is explained in detail in the inline API Documentation BSL API Docs. A summary of the components is below.

BSL Frontend A C99 library used by a BPA integration and used by each Policy Provider and Security Context to access BSL and BPA behavior and data. This is the base of the BSL and is intended to be common for all deployments.

Dynamic Backend An implementation of the frontend suitable for general-purpose, non-constrained deployments which uses heap-allocated, dynamically-sized data structures and runtime registration of policy providers and security contexts. This component can be replaced by a deployment-specific alternative if needed.

Example Policy Provider An implementation of a configurable policy provider based on the syntax and semantics of the BPSec configuration from the NASA ION software suite [NASA-ION].

Default Security Contexts Implementations of the two Default Security Contexts (Context ID 1 and 2) from [RFC9173] using cryptographic functions provided by the OpenSSL library [OpenSSL].

Crypto Library An API for security contexts to isolate themselves from cryptographic processing and key handling. The default configuration uses [OpenSSL] to implement this library, which allows the BSL to operate in FIPS 140-3 environments.

Test Utilities A set of additional utility functions helpful for unit and fuzz testing but not needed by the operational BSL components.

Mock BPA An executable used to provide a test fixture and example BPA integration. This 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.

These components are represented as targets (libraries and executables) in the diagram of Figure 2.3, which was auto-generated from the BSL CMake project.

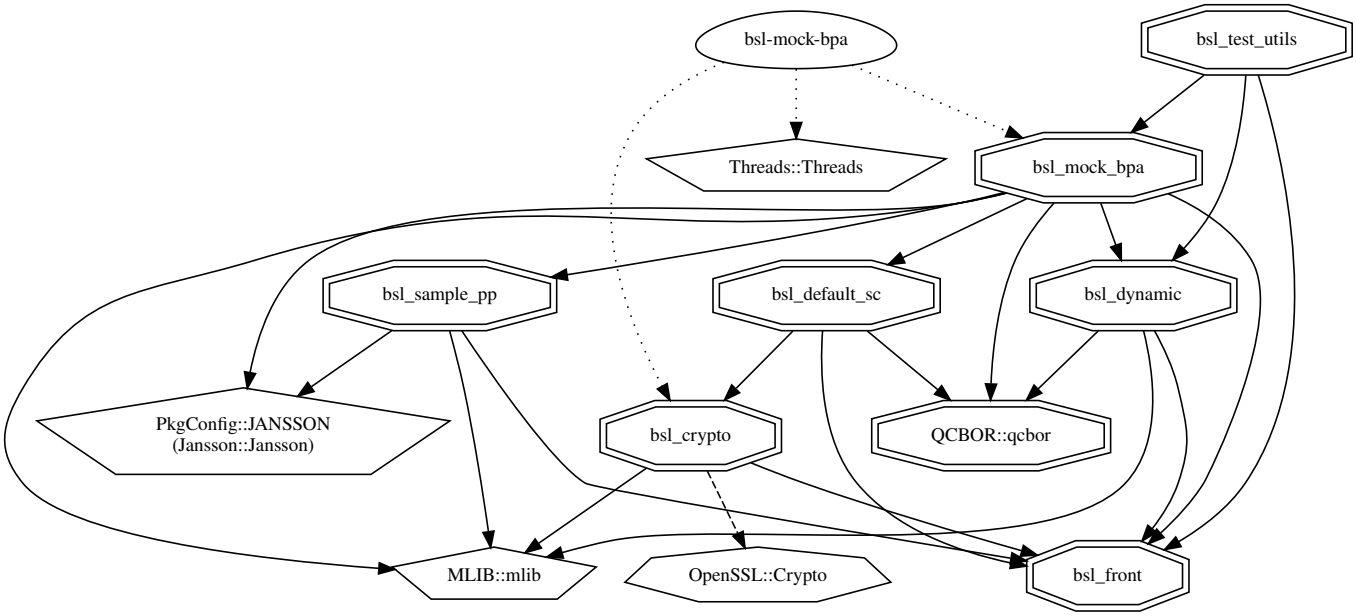

Figure 2.3: CMake-Generated Target Graph

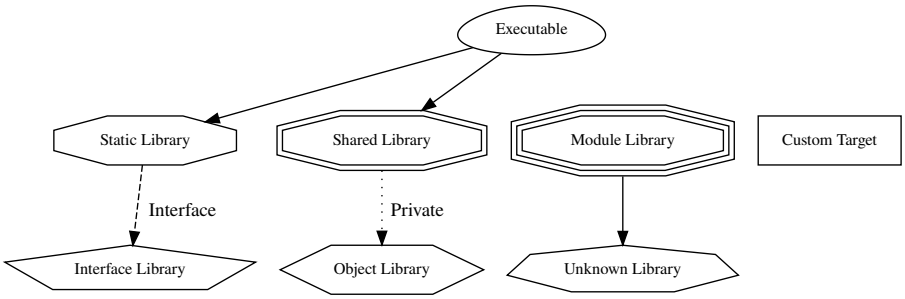

Figure 2.4: CMake Graph Legend

2.2 Build and Runtime Environments

The basic requirements in the BSL SRD are that the build environment use a C compiler, with its standard headers and libraries [C99], and include POSIX headers and libraries [POSIX].

The BSL dynamic backend uses the [MLIB] library for heap-allocated data containers, including dynamic arrays, linked lists, sorted trees, and hash maps. The BSL backend uses the [QCBOR] library for encoding and decoding of ASB sequences within security blocks.

The example JSON-based ION-heritage policy provider and Mock BPA JWK registration tool distributed with the BSL both use the [Jansson] library for JSON parsing.

The example security contexts distributed with the BSL uses the [OpenSSL] library for all cryptographic functions.

The Mock BPA distributed with the BSL uses POSIX UDP/IP sockets for BPv7 PDU transport, both as a test CLA and a test application interface. This allows traffic into and out of the Mock BPA to be captured by tools such as `pcap` and inspected with tools such as Wireshark and `tshark` [wireshark]. The Mock BPA also uses [QCBOR] for encoding and decoding of whole bundle PDUs, as well as the [Jansson] library for decoding JWK key stores.

Unit tests for each of the BSL components use the [unity-test] library for defining test fixtures and assertion logic.

The entire BSL repository tree is managed using the [CMake] tool and by default is configured to use the [Ninja] build tool. The official releases of the BSL use default CMake options, but developers can use other options as described in Section 3.4.1.

2.3 Software Packaging

The official releases of the BSL are packaged and distributed as RPM packages intended to be usable within a YUM/DNF repository [rhel9-packaging]. Packages are version marked based on the latest git tag in the working copy's commit history and revision marked based on the specific latest git commit hash of the working copy along with the distribution tag (see the "Versioning" and "Dist Tag" sections of [fedora-packaging]).

For example, a pre-release build of the BSL is marked with RPM version-revision of `0.0.0-0.g71ab437.el9` indicating it does not follow a release version tag (so gets marked with version `0.0.0`), it is zero commits from that (non-)tag, it is from commit hash `71ab437`, and it was built on RHEL-9 (or equivalent).

BSL packages can also built from the source tree, either under RHEL-9 directly or using a (Docker or Podman) container to provide an RHEL-9 environment. Details on these procedures are provided in Section 3.1.

The set of packages for each BSL release (or local package build) contains the following:

bsl The runtime files needed for the library itself. This contains versioned shared objects.

Major files are installed under `/usr/lib64/`.

bsl-devel Development files needed to build and link against the BSL. This contains C headers and shared object version links.

Major files are installed under `/usr/include/` and `/usr/lib64/`.

bsl-apidoc Doxygen-generated API documentation derived from in-source markup.

Major files are installed under `/usr/share/doc/bsl/`, which contains an `html` directory.

bsl-debuginfo Runtime debug information for the `bsl` package. This relies on `bsl-debugsource` for tracing to individual source lines for interactive debugging.

bsl-debugsource Copies of the original source files used along with the `*-debuginfo` packages to support interactive debugging.

bsl-test Unit test and Mock BPA executables and support libraries.

Major files are installed under `/usr/bin/`, containing the `bsl-mock-bpa` daemon executable, `/usr/lib64/` for its libraries, and `/usr/libexec/bsl/` which contains each unit test executable for the BSL.

bsl-test-devel Development files needed to build and link against the Mock BPA of the BSL. This contains C headers and shared object version links, including the Unity test library.

Major files are installed under `/usr/include/` and `/usr/lib64/`.

bsl-test-debuginfo Runtime debug information for the `bsl-test` package. This relies on `bsl-debugsource` for tracing to individual source lines for interactive debugging.

2.4 File System

The BSL itself does not require any specific input or configuration files for its normal operation. It relies on the host BPA to perform any configuration file management, loading, parsing, *etc.*.

As a Linux shared library, it does relate to the host file system in the following paths:

/usr/lib64/ The OS-standard path for all shared library files. The BSL installs its core and example libraries here.

/usr/include/ The OS-standard path for all library header files. The BSL installs its own headers under the `bsl` sub-directory, and its inbuilt (non-OS) dependencies under `QCBOR` and `m-lib` sub-directories.

/usr/bin/ The OS-standard path for all non-privileged executable files. The BSL installs its Mock BPA as the executable `bsl-mock-bpa` here.

/usr/libexec/ The OS-standard path for context-dependent executable files. The BSL installs its unit tests under the `bsl` sub-directory.

2.5 Networking

The BSL itself does not require any specific OS networking configuration or API interfaces. It relies on the host BPA to perform any network configuration or runtime use.

The Mock BPA distributed with the BSL uses UDP/IP sockets, configured by command-line options, to communicate bundles into and out of the Mock BPA process (see Section 3.5).

2.6 Cryptographic Functions

The BSL itself does not require any specific OS or middleware cryptographic functions.

The example implementation of the default security contexts distributed with the BSL uses the [OpenSSL] library for performing all cryptographic functions.

Chapter 3

Procedures

This chapter includes specific procedures related to managing an BSL deployment from source and for development of BSL changes.

3.1 Building Packages

The BSL source is composed of a top-level repository `BSL` [bsl-source] and a number of submodule repositories; all of them are required for building the BSL.

The following procedure is targeted for the RHEL-9 environment. Other conditions and procedures are discussed in more detail in the source repository `README.md` document.

1. The top-level checkout can be done with:

   ```
   git clone --recursive --branch <TAGNAME> https://github.com/NASA-AMMOS/BSL.git
   ```

2. Optional: switching to a different tag or branch can be done with the sequence:

   ```
   git checkout <TAGNAME>
   git submodule update --init --recursive
   ```

3. If necessary, dependency OS packages can be installed with:

   ```
   sudo dnf install -y epel-release
   sudo crb enable
   sudo dnf install -y \
       rsync cmake git ninja-build gcc ruby \
       openssl-devel jansson-devel valgrind-devel \
       doxygen graphviz plantuml texlive-bibtex \
       asciidoctor \
       tito rpm-build rpmlint
   ```

   The packages `doxygen graphviz plantuml texlive-bibtex asciidoctor` are optional, and used only for the `bsl-docs` subpackage.

4. The BSL packages are then built (using the Tito packaging tool [tito]) with:

```
./build.sh rpm-build
```

5. The resulting packages are placed in the directory `build/default/pkg` and can be seen by the listing:

```
find build/default/pkg/rpmbuild -name '*.rpm'
```

6. Optionally: A check and test install of the packages can be performed using:

```
./build.sh rpm-check
```

An alternative to the procedure for building on an RHEL host is to build within a container with an RHEL base image. This can be done using the following on any host with docker available using the command:

```
./build.sh rpm-container
```

With package files copied into the same `build/default/pkg` as the native build. The specific docker executable is configured through the `DOCKER` environment.

3.2 Installation

Once packages are built locally, they can all be installed by running:

```
pushd build/default/pkg/rpmbuild/RPMS/x86_64
dnf install -y bsl-*.rpm
popd
```

Or by some more discriminate choice of packages, such as only the two necessary to integrate the BSL library: `bsl bsl-devel`

Or if pre-built packages are available on an enabled YUM/DNF repository, they can be installed (more simply by name) using:

```
dnf install -y bsl bsl-devel
```

Once installed, the BSL library can be linked with and built against as any other OS-installed C library.

3.3 Upgrading

Because the BSL is deployed in an RPM package form, the normal operating system tools and procedures for dealing with software library upgrading apply to the BSL. The BSL provides SOVERSION information in its libraries, so RPM management tools such as DNF which are cross-dependence-aware will ensure that the correct needed SOVERSION of the BSL is installed.

Individual BSL releases may identify pre-upgrade or post-upgrade steps in their specific Release Description Document (RDD) which would augment this OS-standard procedure.

3.4 Development Building

When modifying the BSL itself (or one of its example Policy Provider or Security Context implementations or the Mock BPA) a more varied set of procedures is necessary, because RPM packages are not used as intermediate forms because of the time and resources it takes to build them and the separation they then have from the original BSL sources.

3.4.1 CMake Project Options

The BSL CMake Project has several options to control what and how the components of the project are build, tested, and installed.

Some built-in CMake options which are useful for development are:

CMAKE_PREFIX_PATH Path specification to use for searching for external libraries and CMake packages. The BSL build script sets this to the local tree `./testroot/` which is where the dependencies are installed by default.

CMAKE_INSTALL_PREFIX Path specification to use for installing the BSL libraries, executables, documentation, and other files. The BSL build script sets this to the local tree `./testroot/` by default.

CMAKE_BUILD_TYPE Choose between standard types "Debug", "Release", "RelWithDebInfo" The BSL build script uses build type Debug, while the RPM spec builds packages with the RPM-default type.

BUILD_SHARED_LIBS Build using shared libraries (default ON)

The BSL-specific options (defined at the top of the root `CMakeLists.txt`) are:

BUILD_LIB Build the library itself (default ON)

BUILD_DOCS_API Enable API documentation building (default OFF)

BUILD_DOCS_MAN Enable manpage building (default OFF)

BUILD_MOCK_BPA Enable building the Mock BPA library and executable (default ON)

BUILD_UNITTEST Enable building unit tests (default ON)

TEST_MEMCHECK Enable test runtime memory checking (default ON)

BUILD_COVERAGE Enable runtime coverage logging and reporting (default OFF)

BUILD_FUZZING Enable building fuzzing executables (default OFF)

BUILD_PACKAGE Enable building package outputs (default OFF)

3.4.2 Local Building and Testing

The coarse-grained procedure for building and testing the BSL locally is the following, based on a working copy of the BSL repository and starting shell commands within that working copy root directory.

1. Build the local dependencies an install into the local target tree `./testroot/` using:

   ```
   ./build.sh deps
   ```

2. Prepare the CMake project, with optional additional options, using:

   ```
   ./build.sh prep -DBUILD_COVERAGE=ON
   ```

   One option available to control the installation is the standard `CMAKE_INSTALL_PREFIX` (see Section 3.4.1).

3. Build the artifacts into the out-of-source tree `./build/default/` using:

```
./build.sh
```

4. Install artifacts into the local target tree `./testroot` using:

```
./build.sh install
```

   If a non-default install prefix is used, this install command may need to be run under `sudo`.

5. Optionally, execute registered unit tests using:

```
./build.sh check
```

   a. If BSL sources are updated and need re-testing, they must be built-and-installed using:

```
./build.sh && ./build.sh install
```

6. Optionally, execute Mock BPA built-item tests using the following sub-procedure.

   a. Install and enter a local Python virtualenv using:

```
python3 -m venv venv
source venv/bin/activate
pip install -r mock-bpa-test/requirements.txt
```

   b. Run the test suite using:

```
python3 -m pytest mock-bpa-test --log-cli-level=info
```

   c. If BSL sources are updated and need re-testing, they must be built-and-installed using:

```
./build.sh && ./build.sh install
```

   d. Leave the virtualenv using:

```
deactivate
```

7. Collect and report on coverage metrics using:

```
./build.sh coverage
./build.sh coverage-summary
```

8. The coverage outputs under the build tree can be viewed in a Web Browser using:

```
xdg-open build/default/coverage-html/index.html
```

3.4.3 Local API Documentation Building

The full API documentation can be built and edited locally using the following procedure.

1. Build the local dependencies (the same as a normal build) using:

```
./build.sh deps
```

2. Prepare the CMake project, with options to enable API documentation building, using:

```
./build.sh prep -DBUILD_DOCS_API=ON
```

3. Build the documentation itself, observing warnings, using:

```
./build.sh docs
```

4. The documentation outputs under the build tree can be viewed in a Web Browser using:

```
xdg-open build/default/docs/api/html/index.html
```

3.5 Monitoring

The BSL itself, as a software library, does not directly make use of any OS-level logging or monitoring facilities.

As discussed more in the BPA integration portion of the BSL User Guide, one form of monitoring output from the BSL is its log events and another form is polling for BSL telemetry counters.

Because the Mock BPA uses "normal" BPv7/UDPCL it can be monitored using off-the-shelf Wireshark since version 4.0 [wireshark] with the protocols "BPv7" and "UDPCL" enabled, and the appropriate UDP ports used by the Mock BPA set to "Decode As…" the UDPCL.

3.5.1 SELinux Audit Events

The procedures in this section are a summary of more detail provided in Chapter 5 of the RedHat [rhel9-selinux] document.

By default, the `setroubleshootd` service is running, which intercepts SELinux audit events

To observe the system audit log in a formatted way run:

```
sudo sealert -l '*'
```

Some SELinux denials are marked as "don't audit" which suppresses normal audit logging when they occur. They are often associated with network access requests which would flood an audit log if they happen often and repeatedly. To enable logging of `dontaudit` events run:

```
sudo semanage dontaudit off
```

3.5.2 FIPS-140 Denials

The effect of FIPS-140 enforcement and denied behavior appear as 'normal' failures of the corresponding cryptographic API and must be observed from logs of the BSL or of the cryptographic library itself. Because the BSL operates as a library, its logging is routed through the callback API to its host application (*i.e.* the BPA). Determining how to access the host application logs is outside the scope of this guide.

3.6 Checkout Procedures

The BSL packaging procedure includes built unit tests within the `bsl-test` RPM package which allows executing unit tests on the BSL library after build time on any other host.

The `bsl-mock-bpa` executable distributed as part of that package also enables verification of the installed BSL libraries using an example policy provider and example security contexts and real BPv7 PDUs exchanged via UDP sockets (equivalent to the un-framed transfer of the UDPCL).

All other checkout of the BSL requires a specific BPA integration in order to exercise its *service interface* from a running BPA instance.

Chapter 4

Product Support

There are two levels of support for the BSL: troubleshooting by a system administrator, which is detailed in Section 4.1, and upstream support via the BSL public GitHub project, accessible as described in Section 4.2. Attempts to troubleshoot should be made before submitting issue tickets to the upstream project.

4.1 Troubleshooting

The following situations provide troubleshooting guidance for the BSL from the perspective of a package maintainer or BSL developer, typically working from a local clone of the BSL git repository. Each situation consists of an observed state followed by a recommended troubleshooting activity.

4.1.1 Building and Continuous Integration

This section covers issues that can occur during packaging (see Section 3.1) or during development (see Section 3.4) of the BSL.

Portions of the CMake project are giving errors during preparation

It is important that all CMake options (see Section 3.4.1) given at prep time are consistent. When the same option is supplied on the command multiple times, the last value is the one actually used.

Mock BPA tests are failing inconsistently or due to symbol errors

Before running Mock BPA tests, the built artifacts must be installed to the `testroot` tree using `./build.sh install`.

Files are not installed where I want them

This can be controlled with the `CMAKE_INSTALL_PREFIX` option (see Section 3.4.1) for the BSL itself. Dependencies also need to be configured by setting the standard `DESTDIR` and `PREFIX` environment variables before running `./build.sh deps`.

The API documentation target complains about misspellings

If it is a false positive (the word is correctly spelled, just not in the current dictionary) edit the file `docs/api/dictionary.txt` to add the word.

4.1.2 Installation

This section covers issues that can occur during installation (see Section 3.2) of the BSL.

Permission is denied by YUM/DNF to install packages

Because the RPM packages are installed to the OS, their use requires privileged user account or the use of `sudo`.

4.1.3 Operations

This section covers issues that can occur after successful installation (see Section 3.2) and checkout (see Section 3.6) of the BSL.

Behavior is reported or suspected to be blocked by SELinux

If there is any behavior of the BSL not working correctly and there is suspicion that it is being blocked because of local SELinux policy, the procedures of Section 3.5.1 should be used to troubleshoot.

Behavior is reported or suspected to be blocked by FIPS-140 enforcement

The example security contexts maintained as part of the BSL make use of a FIPS-approved version of OpenSSL with algorithms and security parameters also compliant with FIPS-140. So these default security contexts should not run afoul of any blocks caused by enabling "FIPS mode" on the host OS.

Any additional security contexts registered with a specific BSL instance may not be FIPS-140 compliant and should be carefully considered before use in an expected FIPS-enabled environment.

4.2 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.