Cell Broadband Engine

Security Software Development Kit 3.0

Installation and User's Guide

Version 3.01

September 10, 2007

Printed in the United States of America August 2007

IBM Systems and Technology Group

2070 Route 52, Bldg. 330

Hopewell Junction, NY 12533-6351

The IBM home page can be found at http://www.ibm.com

The IBM Cell Broadband Engine resource center: http://www.ibm.com/developerworks/power/cell/

September 10, 2007

Contents

1 Overview

The IBM Security Software Development Kit (SDK) for Cell Broadband Engine™ (Cell/B.E.™) is a complete package of tools for security-sensitive applications. Broadly speaking, users can choose between two approaches. With the first approach, the application developer simply signs and encrypts SPU (Synergistic Processing Unit) applications. The applications will remain encrypted until immediately before execution thus thwarting reverse engineering and piracy attempts. Furthermore, the application is verified for authenticity and integrity immediately before execution. This feature makes tampering, reverse engineering, and piracy much more difficult for the adversary. In essence, the platform and the tools provide a secure communication channel between the application developer and the legitimate application user.

For applications which require a higher level of protection, there is the second approach which builds upon the first approach. This approach invokes the *hardware SPU isolation mode* whereby the hardware provides a "vaulted" execution environment for the SPU application. The applications are decrypted and verified immediately before execution just as in the first approach, but the decrypted and verified application is further protected *during* execution. This feature is unique to the Cell/B.E. and is considered a processor architecture differentiator. Details of the processor Cell/B.E. security architecture can be found in publications listed in section 1.5, "Related Documentation".

1.1 What's New for SDK 3.0

1.1.1 Two Versions

For the SDK 3.0 release, there are two versions of the Cell/B.E. security package. One version, "the publicly available" version can be downloaded from IBM developerWorks. The other version, "the confidentiality and disclosure agreement (CDA)" version requires the customer to sign a CDA with IBM to obtain the package. This is how the two versions compare:

The publicly available version:

- Can be determined by the version number of the RPMs: cell-spu-isolation-<>-0.3-x.<>.rpm
- Does application signing and verification using asymmetric cryptography.
- Replaces symmetric cryptographic encryption with logical XOR function for application encryption and for data encryption library functions.

The CDA version:

- Can be determined by the version number of the RPMs: cell-spu-isolation-<>-3.0-x.<>.rpm
- Does application signing and verification using asymmetric cryptography.

- Uses symmetric cryptography for application encryption and for data encryption library functions.
- Supplies a security-enabled Cell/B.E. simulator with which the user can simulate the hardware SPU isolation mode.
- Additional library routines which assist with the isolated SPE programming.

1.1.2 Emulated Isolation Mode

With previous releases, users were constrained to use a security-enabled simulator to execute the run-time Cell/B.E. Security SDK stack. However, with the introduction of the emulated isolation mode in this release, users may develop and execute the security software stack on an IBM QS20/21 or a regular (non-security-enabled) simulator.

This is the mode that should be used by developers who will only encrypt and sign applications and do not intend to use the hardware SPU isolation feature.

The emulated isolation mode allows the GDB (GNU project debugger) to work with the security application during its development.

1.1.3 Key Hierarchy

The key hierarchy and management have been much improved for SDK 3.0. Industry standard features such as Certificate Revocation Lists (CRL) and tiered Certificate Authority (CA) hierarchy is now part of the architecture.

1.1.4 Secure File Storage

A new set of APIs for file storage allow users to read and write an encrypted and verified file. The interface is near identical to the standard file I/O API and therefore, users can use this facility transparently.

1.2 How to obtain the CDA version

Please contact your IBM customer representative.

1.3 Supported platforms

The Cell/B.E. Security SDK 3.0 is supported on all of the platforms supported by the Cell/B.E. SDK 3.0, including:

- x86

- x86-64
- 64-bit PowerPC (PPC64)
- Cell/B.E.-based blade server (QS20, QS21)

An application developed on any one of the above platforms can be executed on the Cell/B.E. blade server or the Full-System Simulator on any of the other above platforms

1.4 Getting questions answered

Please post your questions to the developerWorks Cell Broadband Engine Architecture forum online.

1.5 Related Documentation

- "Cell Broadband Engine processor security architecture", http://www-128.ibm.com/developerworks/power/library/pa-cellsecurity/
- "SPE Runtime Management Library", installed in /opt/cell/sdk/docs/lib/

2 Installing the Cell/B.E. Security SDK

Before installing the Cell/B.E. Security SDK, you need to install the Cell/B.E. SDK. Please see the Cell/B.E. SDK Installation and User's Guide for instructions for installing the Cell/B.E. SDK. This chapter assumes that you have installed the Cell/B.E. SDK on your host system and are now ready to install the Cell/B.E. Security SDK.

2.1 Security Subcomponents

The Cell/B.E. Security SDK consists of a set of optional Cell/B.E. SDK RPMs that are installed using yum or a graphical installer (e.g., pirut or pup). The security RPMs are optional subcomponents of the following Cell/B.E. SDK components

Cell Development Tools	
	Build Tool – binaries and source
Cell Development Libraries	
	Libraries & Headers
	Secure Loader (cross development)
Cell Runtime Environment	
	Secure Loader (ppc)
Cell Programming Examples	
	Samples

The table below shows a complete list of RPMs that can be installed for each supported platform.

Component Area	x86	PPC64	Cell-based Blade Server
Build Tool	`cell-spu-isolation-tool-0.3-5.i386.rpm`	`cell-spu-isolation-tool-0.3-5.ppc.rpm`	
Build Tool - source	`cell-spu-isolation-tool-source-0.3-5.noarch.rpm`		
Libraries & Headers	`cell-spu-isolation-cross-devel-0.3-5.noarch.rpm`	`cell-spu-isolation-devel-0.3-5.ppc.rpm`	
Secure Loader	`cell-spu-isolation-loader-cross-0.3-5.noarch.rpm`	`cell-spu-isolation-loader-0.3-5.ppc.rpm`	
Samples	`cell-spu-isolation-emulated-samples-0.3-5.noarch.rpm`		

If desired, all of the security SDK RPM's can be easily installed using the command

```
yum install cell-spu-isolation*
```

The samples rpm installs only source code. Once installed, the samples are built by the following command sequence:

```
cd /opt/cell/sdk/prototype/src/examples/isolation
make
```

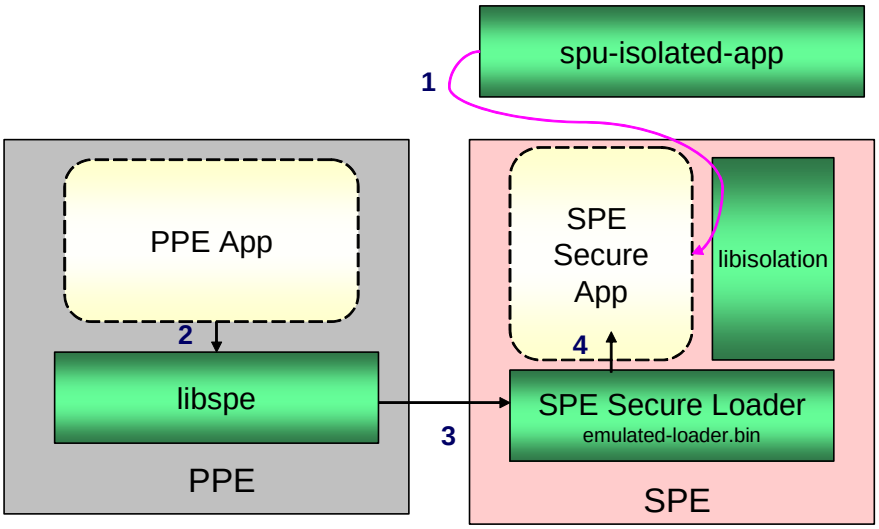
If any cross development rpm's are installed or the samples are built for a cross development environment, then the sysroot needs to be synchronized for the simulator environment. This is accomplished by executing the script

```
/opt/cell/cellsdk_sync_simulator
```

3 Overview of the Cell/B.E. Security SDK

3.1 Components

This section describes the contents of the SDK and how the various components work together.



The spu-isolated-app tool: In step 1 of the diagram, spu-isolated-app encrypts and signs the SPE application during build-time. The user can specify the keys used for signing and encryption. Because of this step, the SPE application is now a *SPE secure application*. The tool binary can be found at /opt/cell/sdk/prototype/usr/bin/spu-isolated-app.

Libspe: libspe is the runtime SPE thread management library. The PPE code which invokes the SPU secure application must specify that the SPE thread run in emulated-isolation or isolation mode (in step2 of the diagram).

The flag to be used for emulated-isolation mode is SPE_ISOLATE_EMULATE.

The flag to be used for isolation mode is SPE_ISOLATE. This flag should only be used on security-enabled Cell/B.E. platforms or the security-enabled simulator.

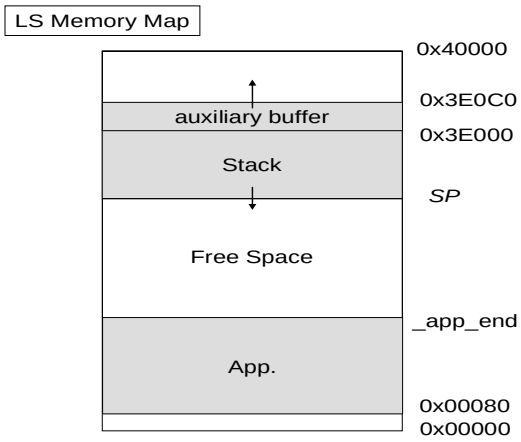
More details can be found in the libspe documentation (reference in section 1.5).

SPE Secure Loader: because the SPE thread was specified to run in emulated-isolation or isolation mode, the SPE Secure Loader is loaded and started on the SPE (step 3). It loads the SPE Secure Application, cryptographically verifies its integrity and authenticity, and decrypts it. If the verification succeeds, the SPE Secure Loader starts the application's execution. If the verification fails, the SPE Secure Loader does not execute the application and returns an error via libspe (more about error code in section 3.3). The secure loader for emulated-isolation mode can be found in /usr/lib/spe.

Libisolation: libisolation must be linked with the SPE Secure Application when the functions listed in the API section (section 8) and PPE-assisted library calls (standard C library functions) are used in the SPE Secure Application. The library can be found in /opt/cell/sdk/prototype/usr/spu/lib/libisolation.a on a native Cell/B.E. platform and in /opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/lib/libisolation.a on a cross-compiling environment.

3.2 LS Memory Map for SPE Secure Applications

The SPE Secure Loader loads the application code at LS location 0x00080, which is the same address as for a non-isolated application. After the verification and (optional) decryption of the application, the SPE Secure Loader will branch to the secure application, and the Local Store has the following layout.



The application stack starts at address 0x3E000 and grows towards the lower addresses.

The area marked “auxiliary buffer” is the buffer area for PPE-assisted library functions. This buffer area is at least 192 bytes and the area between 0x3E000 and 0x3E0C0 is reserved for this buffer. There is an API available (described in section 8) for growing the auxiliary buffer area. The area will grow towards the higher addresses.

For this release of the Cell/B.E. Security SDK, a secure application, including code and static data, is limited to a maximum size of 167KB. However, the runtime application space is extended to 247KB.

3.3 Return Error Codes

The libspe functions will return errors that will guide the programmer on where problems are happening. There are three libspe functions that are called in order to launch a secure SPE thread. In section 7, a “hello world” example is introduced to illustrate how these functions are used to launch a secure SPE thread.

3.3.1 spe_context_create()

No errors specific to the SPE_ISOLATE_EMULATE are returned. However, if ENODEV[1] is returned, the programmer should verify that the SPE_ISOLATE flag was not used instead of the SPE_ISOLATE_EMULATE flag. ENODEV will be returned if the Cell/B.E. platform is not hardware security enabled and SPE_ISOLATE flag is used.

3.3.2 spe_program_load()

If the SPE_ISOLATE_EMULATE or SPE_ISOLATE flag is set, and the SPE application is not formatted as the SPE secure application, this function will return an ENOEXEC and a return value of "-1". In other words, the programmer had requested an isolated thread on a program that has not been signed by the spu-isolated-app tool.

3.3.3 spe_context_run()

If libspe returns stop_reason value of "7", this indicates that it is an SPE_ISOLATION_ERROR type. The spe_exit_code will further indicate what kind of isolation error has been returned.

spe_ exit_ code	Error reason	Possible problems & fixes
1	Application image is too large	Application image may be larger than the permitted 200KB.
2	Application header is incorrect	An application built in a "public SDK" development environment may be executing in a "CDA SDK" runtime environment. Verify that the application has been rebuilt since the "CDA SDK" was installed.
3	Application decryption has failed.	
4	Application authentication has failed	The certificates and keys that are used by the SPE Secure Loader at runtime may be missing on the system. The directory location of these are listed in section 4.2.6

[1] An older kernel may cause libspe to return EFAULT instead.

3.4 Usability of Tools in Emulated Isolation Mode

Because the emulated isolation mode, unlike the hardware-based isolation mode, does not physically "lock-up" the LS, it is possible to use diagnostic tools such as GDB (the GNU debugger). GDB can be used in the same manner for emulated isolation mode SPU applications as it is for normal SPU applications. The SPU extensions for ppu-gdb is documented in the Cell/B.E. Programmer's Guide which can be found under /opt/cell/sdk/docs.

It is expected that this feature is only used during development and when the application is ready to be deployed, the developer will strip out the symbols in the application binary.

For this release of the SDK, the performance profiling tool, *oprofile*, cannot be used for emulated isolation mode SPU programs. It is expected this will be supported in future releases.

4 The Key Hierarchy

For many customers, the key hierarchy is important from both a flexibility and security perspective. Most application developers will interface with the application-level keys, and thus, the key hierarchy must be designed to meet their needs. At the same time, customers need to be convinced that the key hierarchy solution that we are providing is indeed secure and protects against the attack profiles that they are concerned with.

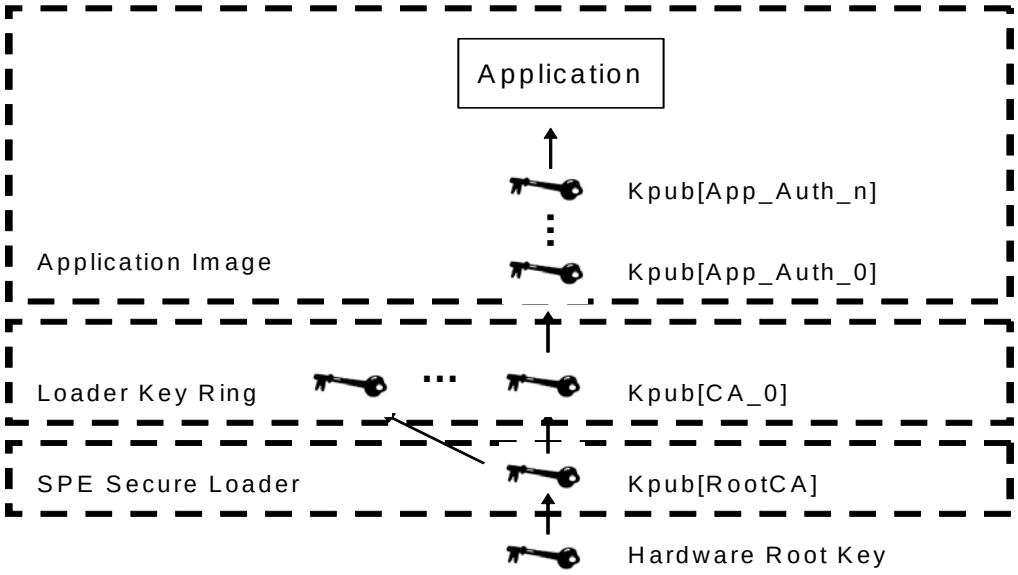
The key hierarchy is grounded in the unique Cell Security architecture which relies on a hardware root key. The hardware root key is the "grandfather key of them all" and the security of all the other keys in the key hierarchy comes down to the hardware root key. For users who are comfortable without the hardware solution, the root of trust starts in the SPE Secure Loader.

Although the foundational layers of the key hierarchy are unique to Cell, the application layers use an industry standard such as X.509 for better industry and customer adoption.

4.1 Key Naming Convention in this document

- Kpub[*Name*] is the public key of an RSA key pair of the name *Name*. Unless otherwise noted, 2048-bit size key is used in this release of the SDK.
- Kpriv[*Name*] is the private key of an RSA key pair of the name *Name*. Unless otherwise noted, 2048-bit size key is used in this release of the SDK.
- CA is short for "Certificate Authority", a common concept in a public key infrastructure system.

4.2 Application Trust Chain



4.2.1 Overview

At run-time, the goal is to verify that the application image has not been tampered since shipment and that the application is authorized to execute on the platform. Every time an SPE Secure Application is about to be executed, the trust chain is re-verified from the root of trust[2] all the way up to the signature on the application.

1. The hardware root key is used to verify the signature on the Public Key of the Root Certificate Authority (CA), Kpub[RootCA]. This is to check that Kpub[RootCA] has not been tampered with and is authorized to be used as the root CA for the system.[3]
2. Once the integrity and authenticity of the Kpub[RootCA] is verified, it is used to verify the signature of one of the second level CA public keys, Kpub[CA_i].
3. Once the appropriate Kpub[CA_i] is verified, it is used to check the first-level Application Authentication Key, Kpub[App_Auth_0] embedded in the application image.
4. There can be an arbitrary length chain of Application Authentication Keys.
5. The last Application Authentication Key, Kpub[App_Auth_n] is used to verify the Application image.
6. At this point, the application verification is complete, and the application will start executing.

[2] for emulated isolation mode, the root is the key in the SPE secure loader and for hardware isolation mode, the root is the key embedded in the hardware

[3] This step is skipped for emulated isolation mode.

Some Key Points:

- At any point, if the verification fails, the execution stops. The application will not be started.
- Because the application keys must be signed by a CA key, this feature provides control to the platform owner on which secure SPE application can run on this platform. Applications with application keys that are not signed by the approved CAs will not be able to execute on the SPE using the isolation (hardware-based or emulated) modes.

4.2.2 Component View

4.2.2.1 SPE Secure Loader

The Kpub[RootCA] key that is embedded in the SPE Secure Loader image is a static key for the lifetime of the platform. It cannot be revoked.

4.2.2.2 Loader Key Ring

Instead, the flexibility is provided in the SPE Loader Key Ring which is stored in a file independently from the SPE Secure Loader and is read from and written to (only) by the SPE Secure Loader. The Loader Key Ring contains the set of (non-Root or second-level) CA public keys.

By adding a key onto the key ring, the particular Cell/B.E. system is effectively entered into a group administrated by the CA; if the key is removed, the device is removed from the group or the key has been revoked due to a security breach.

4.2.2.3 Application Image

The Application Image format contains the application binary, the Application Authentication Keys Certificate in X.509 format, and the signature value. The Application Authentication keys can be revoked and/or expired.

How many applications use the same Application Authentication key is determined by the developer. Some developers may choose to have many applications have the same Application Authentication Key. This implies that these applications implicitly trust each other. Conversely, applications that do not trust each other should not have the same Application Authentication key.

4.2.3 The Players

4.2.3.1 Root CA

The Root CA is most likely the manufacturer or the distributor of the Cell-based system. It has the power to authorize which CA can accept or reject application developers' certificates for those systems.

4.2.3.2 CA

The CA organization has the role of signing the application developer certificates. By signing these certificates, the CA is authorizing the developer's applications to run in SPE emulated or hardware isolation mode.

4.2.3.3 Application Developers

Application developers create applications to be deployed on the Cell/B.E. They have the option of signing and/or encrypting the application for secure delivery of the code. The public key counterpart to the application signing key must be signed by one of the approved CAs.

4.2.4 Summary

Key Name	Symbol	Function	Owner	Location
Application Authentication Key (Private)	Kpriv[AppAuth_i]	Sign the application.	Developer	Developer
Application Authentication Key (Public)	Kpub[AppAuth_i]	Verify the application.	Developer	Embedded in application image.
(Second-level) CA Key (Private)	Kpriv[CA_i]	Sign the Application Key (Public) Certificate	CA	CA
(Second-level) CA Key (Public)	Kpub[CA_i]	Verify the Application Key Certificate	CA	SPE Secure Loader Key Ring (see below)
Root CA Key (Private)	Kpriv[RootCA]	Sign the (Second-level) CA certificates	Root CA, Platform Owner	Root CA
Root CA Key (Public)	Kpub[RootCA]	Verify the (Second-level) CA certificates	Root CA	Embedded in SPE Secure Loader
Certificate Revocation List for CA	CRL[CA]	Revoke the (Second-level) CA certificates	Root CA	Filesystem (see below)
Certificate Revocation List for Applications	CRL[AppAuth]	Revoke the Application Authentication Key Certificates	(Second-level) CA	Filesystem (see below)

4.2.5 Usage Guide on Application Authentication Keys

Users can simply use the sample key provide with the SDK to sign their application. In the sample directory, a sample signing key and its corresponding public key certificate is provided. In section 6.2, we discuss how to modify the Makefile to point to the signing key and the public key certificate. Please see below for the location of the keys.

Alternatively, users may generate their own application signing key (i.e. Application Authentication Key pair). In this case, the user must sign the application verification key (i.e. Application Authentication, Public) with a CA signing key (i.e. CA key, private). A CA signing key is provided as part of the SDK, and the user may use a tool such as OpenSSL to do the RSA signing. Please see below for the location of the key.

4.2.6 File locations

Key Name	Path name
Sample Application Authentication Key (Private)	/opt/cell/sdk/prototype/src/examples/isolation/keystore/user_sign_key.pem
Sample Application Authentication Key (Public)	/opt/cell/sdk/prototype/src/examples/isolation/keystore/user_signed_crt.pem
CA signing key	/etc/pki/cell-spu-isolation/CA/ca.signing_key.private.pem
Certificate Revocation List for CA	/etc/pki/cell-spu-isolation/loader/rootca.revoked_ca.crl
Certificate Revocation List for Applications	/etc/pki/cell-spu-isolation/loader/ca.revoked._app_cert.crl
SPE Secure Loader Key Ring	/etc/pki/cell-spu-isolation/loader/rootca.approved_ca.keyring

4.3 Application Encryption Chain

There are many advantages to encrypting the application. For one, it makes code reverse-engineering more difficult and secrets in the code easier to protect. Furthermore, it gives the application distributor more control over the application deployment. For example, the application distributor can force authentication to always happen before the application is executed so that a hacked version is never executed. Or, application encryption gives the distributor the ability to allow only approved users to run the application in the manner of DRM (digital rights management).

Without encryption and with only a digital signature, an adversary can remove the signature so that the binary is freely executable just as any non-secure binary.

4.3.1 Application at Build Time

At build-time, the application is encrypted by a derivative of two keys.

The first key is the SPE Secure Loader's *Application Decryption Key (Public Key).* The *Application Decryption Key* pair is an RSA key pair used for encrypting application so that only the SPE Secure Loader with the key pair can decrypt it.

The second key is the application's *Application Authentication Key (Public Key).* By encrypting with a derivative of this key, the decryption will be tied to the application. This prevents a rogue application from decrypting the encrypted section of another application.

For further details, please see the example described in section 9.6

4.3.2 Application at Execution Time

At execution time, the SPE Secure Loader will decrypt the application image using a derivative of two keys.

The first key is the SPE Secure Loader's *Application Decryption Key (Private Key).* This is the private key counterpart to the public key that was used to encrypt the application.

The second key is the application's *Application Authentication Key (Public Key)* which is the same key used to authenticate the application. This ensures that only applications that have correctly authenticated are decrypted. A rogue application that authenticates with a different key cannot have the encrypted data decrypted by the SPE Secure Loader.

4.3.3 Summary

Key Name	Symbol	Function	Owner	Location
Application Decryption Key (Private)	Kpriv[Loader_Decrypt]	Used by the SPE Secure Loader to decrypt the application	SPE Secure Loader	SPE Secure Loader
Application Decryption Key (Public)	Kpub[Loader_Decrypt]	Used by the application developer to encrypt the application	SPE Secure Loader	File system[4]
Application Authentication Key (Public)	Kpub[AppAuth_i]	Used in the encryption process so that the encryption is tied to that particular application	Developer	Embedded in application image.

Note: In the public release of the SDK, encryption is replaced with XOR. Users should obtain the CDA version of the Cell/B.E. Security SDK in order to use the encryption capabilities. However, in the public SDK, the keys are still placed in the user's environment and the build and run-time environment interface is identical to the CDA version.

[4] Kpub[Loader_Decrypt] is installed in /etc/pki/cell-spu-isolation/loader/loader.app_encrypt_key.public.pem

4.4 Application Visible Keys

During run-time, keys are generated by the SPE Secure Loader to be passed to the SPE Secure Application. These keys are passed via the API of the calling interface.

The calling interface for an SPE Secure Application is as follows:

```
int (*secure_app_main_func_t) (
       unsigned long long                              /* spe_id */,
       unsigned long long                              /* param */,
       unsigned long long                              /* env */,
       void *                                          /* secure_env_t * */);
```

The parameters passed to a secure application are always `NULL`; that is, parameter passing from the PPE application to the SPE secure application is not supported.

The SPE Secure Loader passes additional parameters to the secure application using the fourth parameter, secure_env. The structure of this parameter is as follows:

```
typedef struct {
    vector unsigned char app_set_shared_key;
    vector unsigned char app_version_specific_key;
} secure_env_t;
```

These keys are described in the following sections.

4.4.1 Application Set Shared Key (128-bit)

`app_set_shared_key`

This is a 128-bit secret value that is passed to applications which share the same Application Authentication Key and are running on the same platform. If two different applications have different Application Authentication Keys, they will be passed different Application Set Shared Keys. If two different applications are intended to run on different systems (system owned by A and system owned by B for example), they will be passed different Application Set Shared Keys.

The intent for this Application Set Shared Key is for two or more applications to have a shared key because these applications are intended to work together and are developed by the same application developer, the owner of the Application Authentication Key. The Application Set Shared Key is tied to a particular system so that the data encrypted by the key cannot be decrypted by an owner of another system.

One of the advantages of this key to the application developer is that it is given to the application “for free” and does not need to be explicitly managed and stored by the application.

Summary : Application Set Shared Key is tied to a specific SPE Secure Loader instance and to a specific Application Authentication Key (Public).[5]

[5] For the public SDK, because encryption could not be used, these Application Visible Keys are not robustly tied to the parent keys. A user would need the CDA version of the SDK, to obtain the robust implementation of this design.

All SPEs on the same Cell/B.E. system will use the same SPE Secure Loader.

4.4.2 Application Version Specific Key (128-bit)

```
app_version_specific_key
```

This is a 128-bit secret value that is dependant on the version of the application. Thus, when the application is upgraded, the key value that is passed to the application will also change. Therefore, any data that is encrypted with this key by a newer version of the application can not be decrypted by an older version of the application. This is valuable when an older version of the application is known to have a security hole.

As with the Application Set Shared Key, this key is also tied to a specific platform and is not a shared value across multiple systems. And again, one of the advantages of this key is that it does not need to be explicitly managed or stored by the application.

Summary : Application Set Shared Key is tied to a specific SPE Secure Loader instance, a specific Application Authentication Key (Public), and to a specific application binary.

All SPEs on the same Cell/B.E. system will use the same SPE Secure Loader.

5 Secure File System Resource

The Secure File System (SFS) resource provides a set of APIs that implement reading and writing an encrypted and verified file. The model follows the standard I/O fopen, fread, fwrite, and fclose model, but the data is encrypted before it is written to the file system and decrypted after it is read from the file system. In addition, each data block is compared against separate verification data to ensure that it has not been corrupted since it was written.

5.1 File Layout

An SFS file comprises a header section written by the resource, the data section with content from the caller, and the verification section written by the resource. A caller will not have any access to the the header or verifier sections through any of the SFS API's; rather, the caller will read and write only the data section; all data verification occurs within the SFS API's and the results of data verification are reported only if there is a verification failure (e.g., data tampering was detected).

The header section is read during the `sfs_fopen` call to verify the encryption key passed on the call. If the encryption key does not verify, then the file is immediately closed and a NULL stream is returned to the caller with `errno` set to `EINVAL`.

The verification section contains a verifier for each data block and a file verifier that is the hash over the header section and all other verifiers. This section is read and the file verifier is checked as part of the `sfs_fopen` call. If the computed file verifier does not match the verifier read from the file, then the file is immediately closed and a NULL stream is returned to the caller with `errno` set to `EIO`.

The header section and verification section are updated as part of the `sfs_fclose` call for any SFS file opened for writing. Otherwise, the header and verification sections are buffered in the SPE local storage, which limits the total size of SFS files (see section 5.3).

5.2 Encrypted Contents

All data written to a Secure File System file is encrypted with the key provided by the caller on the `sfs_fopen` call; this includes the header section, the verification section, and the contents of the data section. Two different secure applications may share the contents of an SFS file if they share the encryption key. Thus, the management of the encryption keys determines which applications have access to the contents of an SFS file.

It is still the responsibility of the file system to manage access to the SFS files. However, the contents should not be decipherable to an application without the encryption key, and any tampering of the file contents will be detected by the verification of the file header or checking the verification data when the file is opened, or finding a verification fault when data is read from the file.

5.3 Using the SFS API's

The easiest method to write programs using the SFS API's is to use the standard I/O API's fopen, fread, fwrite, and fclose. The SFS API's were designed to closely mimic the corresponding standard I/O API, and to return the same error codes when this made sense. This means that porting an existing application to use the SFS resource can be reasonably easy.

It is important that the error codes used for verification failures (EINVAL for failure to verify the encryption key, and EIO for conflicts with verification data) are considered and appropriate action is taken to avoid leakage of information or use of corrupted information.

5.4 Implementation

For the publicly available version, XOR with a static key is used for the data encryption and a cyclic redundancy check (CRC) is used for data verification. For the CDA version, symmetric cryptography is used for data encryption and SHA-1 is used for data verification.

The SFS resource maintains information about each open file. This includes the current header section, the verification section, a data buffer, and additional navigation information. The total overhead for this information is approximately

```
overhead = 604 bytes + (number of packets + 1) * 20
```

where the number of packets is equal to the size of the data content divided by 512 (the data content is organized into 512 packets). The CDA version has an additional 240 bytes of overhead.

6 Building Secure Applications

The normal process flow would comprise the following steps:

1. Building and testing the application
2. Securing the application
3. Running the application in emulated isolation mode

6.1 Building and Testing the Application

The most common way to build and test a secure application would be to build and test the application as a non-secure application using all of the standard tools and libraries of the SDK, and then modify the application to invoke the SPE emulated isolation mode. Please see the Cell Broadband Engine Programming Tutorial for instructions and tips for programming using the non-secure SDK.

6.2 Securing the Application

Modifying an existing application to run in SPE isolation mode involves the following steps:

1. Modify the `spe_context_create` of the existing PPE application to add the `SPE_ISOLATE_EMULATE` flag as a parameter. These changes signal to libspe that the SPE is to be placed into emulated isolation mode.
2. Replace the include of `make.footer` with `make.iso.footer` in your Makefile for your SPU code, and add the defines for `ISO_SIGN_KEY` and `ISO_SIGN_CERT`, for the files containing the application signing key and application signing certificate. Both the signing key and signing certificate must be in PEM format. Optionally, add the defines for `ISO_ENCRYPT_SEC` if a section of the application is to be encrypted. `ISO_ENCRYPT_SEC` may be the name of an ELF section of the SPE application or `ALL` if all code and data sections are to be encrypted. This step invokes the Build Tool to sign and (optionally) encrypt the SPE application prior to its being embedded within the PPE application. (Please see the Makefiles in the examples directories for concrete examples.)
3. Finally, build the application.

6.2.1 SPE Secure Application Build Tool, spu-isolated-app

The isolated SPE application build tool is a standalone application that signs and (optionally) encrypts the SPE secure application using the supplied keys. The specifics of the key usage model and hierarchy are described in the “Key Hierarchy” section.

The tool binary is installed as `/opt/cell/sdk/prototype/usr/bin/spu-isolated-app`. The source, installed by the `cell-spu-isolation-tool-source rpm`, is installed in the directory `/opt/ibm/cell-sdk/prototype/src/tools/isolation`

The flow at build-time is as follows:

1. The developer codes the application. At this point, the developer can annotate one block of code or data that will be encrypted.
2. Using the regular compiler and linker, an SPE-ELF (Executable and Linkable Format) executable is generated.
3. The build tool takes the SPE-ELF executable and, using keys provided for signing, generates an image that is signed and (optionally) encrypted.
4. The output format of the build tool is SPE-ELF compatible.

The application is called with the following parameter list:

```
spu-isolated-app <infile> <outfile> <signKey> <signCert> [<encryptSec>]
```

where

`infile` is the name of the input file (SPE ELF executable)

`outfile` is the name of the output file (secure SPE ELF executable)

`signKey` is the name of the file containing the Application Authentication Private Key for signing

`signCert` is the name of the file containing the signed X.509 Application Authentication Public Key certificate for certifying the digital signature of the file

`encryptSec` is the name of the section to encrypt or ALL if all sections, both code and data, are to be encrypted

The build tool expects keys and certificates in PEM format. If you have keys or certificates in DER format, then you can convert to PEM format using openssl or other tool.

7 “Hello, World!” Programming Example

Consider the code for a standard Hello World! example. The example can be found in the directory `/opt/cell/sdk/prototype/src/examples/isolation/iso_simple`.

The standard part of the “Hello, World!” example will execute from within the SPU; the largest change from a standard program is the PPE code which starts the SPE code.

7.1 SPU Example Code

The SPU code for `spu/hello_spu.c` would be as follows:

```
#include <stdio.h>

int main()
{
  printf("Isolate Sample: Hello Cell!\n");
  return 0;
}
```

Even though the library code for printing is substantially different for the SPE environment, the basic code is very familiar.

The `spu/Makefile` for the SPU application follows:

```
#             Target
PROGRAMS_spu   := hello_spu

SECURE_LIBRARY_embed  := lib_hello_spu.a

#             Local Defines
ISO_SIGN_KEY   = ../../keystore/user_sign_key.pem
ISO_SIGN_CERT  = ../../keystore/user_signed_crt.pem

#             make.iso.footer
include $(CELL_TOP)/buildutils/make.iso.footer
```

The `PROGRAMS_spu` defines the SPU target executable. The `SECURE_LIBRARY_embed` defines the name of the library which is to contain the SPU executable in a form that can be embedded in the PPE executable.

An isolated SPU executable needs to be signed by the user and then verified by the SPE Secure Loader. `ISO_SIGN_KEY` defines the private RSA signing key. `ISO_SIGN_CERT` defines the X.509 certificate that is used by the SPE Secure Loader to verify the digital signature on the SPU executable; this is included within the PPE embeddable object. The keys used here are the prototype keys delivered with the samples code.

Finally, `make.iso.footer` replaces `make.footer` for the isolated SPU Makefile.

7.2 PPE Example Code

Our PPE code for `iso_simple.c` is responsible for starting the isolated SPU function, waiting for the SPE program to complete, executing a printf with the SPE result code, and then cleanly exiting. This is very similar to the PPE code for a non-secure application, except for the use of the SPE_ISOLATE_EMULATE flag. So, our PPE code would be as follows:

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>

extern spe_program_handle_t hello_spu;

/* arguments list of a PPU thread */
typedef struct ppu_pthread_data {
    spe_context_ptr_t spe_id;
    pthread_t pthread;
    unsigned int entry;
    unsigned int flags;
    void *argp;
    void *envp;
    spe_stop_info_t stopinfo;
} ppu_pthread_data_t;


/**
 * The entry point of a PPU thread to start an SPE application.
 */
void *ppu_pthread_function(void *arg) {
    ppu_pthread_data_t *datap = (ppu_pthread_data_t *)arg;
    int rc;
    do {
        if((rc = spe_context_run(datap->spe_id, &datap->entry, datap->flags, datap-
>argp, datap->envp, &datap->stopinfo)) < 0) {
            perror("failed running context");
            exit(1);
        }
    } while( rc > 0 ); // loop until exit
    pthread_exit(NULL);
}

/**
 * Entry point of 'iso_simple'
 */
int main(int argc __attribute__ ((unused)), char **argv __attribute__ ((unused)) )
{
    ppu_pthread_data_t data;
    int rc;

    /* Create an SPE-threads in isolation mode to execute 'hello_spu'. */
    /* Create conext */
    if ((data.spe_id = spe_context_create (SPE_ISOLATE_EMULATE, NULL)) == NULL) {
        fprintf(stderr, "Failed spe_context_create(errno=%d)\n", errno);
        if(errno == ENODEV) {
            perror("Failed creating context, requires an SPE enabled for isolation
mode");
        } else {
            perror("Failed creating context");
        }
        exit(1);
    }

    /* Load program */
    if ((rc = spe_program_load(data.spe_id, &hello_spu)) != 0) {
        fprintf(stderr, "Failed spe_program_load(errno=%d)\n", errno);
        perror("Failed loading program");
```

```
            exit(1);
        }

        /* Initialize data */
        data.entry = SPE_DEFAULT_ENTRY;
        data.flags = 0;
        data.argp = NULL;
        data.envp = NULL;

        /* Create thread */
        if ((rc = pthread_create(&data.pthread, NULL, &ppu_pthread_function, &data)) != 0)
    {
            fprintf(stderr, "Failed pthread_create(errno=%d)\n", errno);
            perror("Failed creating thread");
            exit(1);
        }

        /* Wait for SPU-thread to complete execution. */
        /* Join thread */
        if ((rc = pthread_join (data.pthread, NULL)) != 0) {
            fprintf(stderr, "Failed pthread_join(rc=%d, errno=%d\n", rc, errno);
            perror("failed joining thread");
            exit(1);
        }

        /* Destroy context */
        if ((rc = spe_context_destroy (data.spe_id)) != 0) {
            fprintf(stderr, "Failed spe_context_destroy(rc=%d, errno=%d)\n", rc, errno);
            perror("failed destroying thread");
            exit(1);
        }

        /* Check stop reason of the SPE application */
        if (data.stopinfo.stop_reason == SPE_EXIT) {
            printf("spe thread has exited(stopinfo.spe_exit_code=%d)\n",
    data.stopinfo.result.spe_exit_code);
            return (data.stopinfo.result.spe_exit_code);
        } else {
            fprintf(stderr, "stopinfo.stop_reason=%x, stopinfo.spe_exit_code=%x \n",
    data.stopinfo.stop_reason, data.stopinfo.result.spe_exit_code);
            return -1;
        }
    }
```

The `Makefile` for the PPE application follows:

```
#               Subdirectories
DIRS            :=     spu

#                         Target
PROGRAM_ppu     :=     iso_simple

#                         Local Defines
IMPORTS          := spu/lib_hello_spu.a -lspe2

INSTALL_DIR    = $(SDKPRBIN_ppu)/examples
INSTALL_FILES  = $(PROGRAM_ppu)

#               make.footer
include $(CELL_TOP)/buildutils/make.footer
```

The Makefile for the PPE application is almost exactly what would be prepared for a non-isolated SPU application.

7.3 Building the Sample Application

First, define the following environment variable.

```
export CELL_TOP=/opt/cell/sdk
```

Then build your application using:

```
make
```

This creates the SPU application, converts it to the SPU memory image and signs it, and then creates the PPU executable, iso_simple, with the SPU application embedded.

7.4 Execute the Sample Application

Executing the command:

```
./iso_simple
```

will produce the following output:

```
Isolate Sample: Hello Cell!
spe thread has exited(stopinfo.spe_exit_code=0)
```

8 API

8.1 Data Transfer through Open Area of LS

As described in the Cell/B.E. Security architecture document (see section 1.5 for references), when an SPU is in hardware isolation mode, there is an open area of LS much like a “window”, whereby an application can DMA in and out data. The following functions below assist the programmer with this programming model. The copyin/copyout functions allow user to transfer data between main memory and LS via the open area. The decrypt_in and encrypt_out functions are based on copyin/copyout but additionally, applies an XOR mask to the data to mimic encryption.

There is no need for programmers who intend to only use the emulated isolation mode to use these functions. They are only for programmers who eventually want to move up to using the hardware isolation mode.

8.1.1 copyin – Data Transfer Utility Functions

C specification

```
#include <libisolation.h>

int copyin(uint64_t ea, void *ls, uint32_t size)
```

Description

The *copyin* subroutine copies data from the main memory into the LS. *ea* and *ls* must be 16-byte aligned, and *size* must be a multiple of 16 bytes.. If all data are successfully transferred to the LS, this subroutine returns success (value *0*). Otherwise, it returns an error (*-1*).

Dependencies

None.

See also

copyout

8.1.2 copyout – Data Transfer Utility Functions

C specification

```
#include <libisolation.h>

int copyout(uint64_t ea, void *ls, uint32_t size)
```

Description

The *copyout* subroutine copies data out from the LS to the main memory. *ea* and *ls* must be 16-byte aligned, and *size* must be a multiple of 16 bytes. If all data are successfully transferred to the main memory, this subroutine returns success (value *0*). Otherwise, it returns an error (*-1*).

Dependencies

None.

See also

copyin

8.1.3 decrypt_in – Data Transfer Utility Functions

C specification

```
#include <libisolation.h>

int decrypt_in(void *dst_ls, const uint64_t ea, const uint32_t message_size,
    const vector unsigned char shared_key)
```

Description

The *decrypt_in* subroutine copies the message on *ea* into *dst_ls*, and XOR'es the message at *dst_ls* with a library static mask. The XOR'ed results are stored in *dst_ls*. *ea* and *dst_ls* must be 16-byte aligned, and *message_size* must be a multiple of 16 bytes. *shared_key* is ignored in this release. If all data are correctly XOR'ed into the LS, it returns success (*0*). Otherwise, this subroutine returns errors.

Dependencies

copyin

See also

encrypt_out

8.1.4 encrypt_out – Data Transfer Utility Functions

C specification

```
#include <libisolation.h>

int encrypt_out(const uint64_t ea, void *src_ls, const uint32_t
    message_size, const vector unsigned char shared_key)
```

Description

The *encrypt_out* subroutine XOR'es the message at *src_ls* with a library static mask and copies the XOR'ed message to *ea*. *ea* and *src_ls* must be 16-byte aligned, and *message_size* must be a multiple of 16 bytes. *shared_key* is ignored in this release. If all data are correctly XOR'ed and copied out from the LS, it returns success (*0*). Otherwise, this subroutine returns errors.

Dependencies

copyout

See also

decrypt_in

8.2 Secure File Storage

The Secure File Storage APIs provide a facility for reading and writing sensitive information to an encrypted file, with the added function that the content is verified after reading, so that any tampering is detected. The Secure File Storage API's mirror fopen and fread/fwrite for file streams, but with a slightly more limited set of semantics; in particular, not all of the file api's are supported for the encrypted files.

When a secure file is created or opened, an encryption key is provided. Files may be shared between applications by sharing the encryption key. If the file is not to be shared, then the encryption key must be kept secret; this may be best done by encrypting the key as part of the encrypted section of the secure SPE application.

8.2.1 SFS_FILE * – Pointer to Secure File Storage file structure

C specification

```
#include <libisolation.h>

SFS_FILE *sfs_file_var;
```

Description

The SFS_FILE * pointer is the Secure File Storage analog of the FILE * for file I/O. It references a structure that is allocated, maintained, and freed by the Secure File Storage api's, and it contains all of the information necessary to access the Secure File Storage.

Dependencies

None.

See also

None.

8.2.2 sfs_fopen – Create or open a Secure File Storage file

C specification

```
#include <libisolation.h>

SFS_FILE *sfs_fopen(char *filename, char *mode,
     const vector unsigned char key);
```

Description

sfs_fopen creates or opens the Secure File Storage file associated with the string filename, and associates a stream, referenced by the SFS_FILE structure, with the file.

The argument *mode* is interpreted as if for an fopen call to stdio; it must point to a string beginning with one of the following sequences:

'r' Open the file for reading. The stream is positioned at the beginning of the file. It is an error if the file does not exist.

‘r+’ Open the file for reading and writing. The stream is positioned at the beginning of the file. It is an error if the file does not exist.

‘w’ Open the file for writing. Truncate an existing file or create a new file. The stream is positioned at the beginning of the file.

‘w+’ Open the file for writing and reading. Truncate an existing file or create a new file. The stream is positioned at the beginning of the file.

‘a’ Open the file for writing. The file is created if it does not exist. The stream is positioned at the end of the file and all writes occur at the end of the file.

‘a+’ Open the file for writing and reading. The file is created if it does not exist. The stream is positioned at the end of the file and all writes occur at the end of the file.

The argument *key* points to a vector of unsigned characters which contains the 16 character encryption key. This key is used for all encryption and decryption of the Secure File Storage.

During the open of the encrypted file, sfs_fopen reads the file header and verification data, and may write additional verification data. As a result of these operations, sfs_fopen may set errno to any of the error codes associated with fopen, fread, and fwrite; it may also set errno to EINVAL if the encryption key can not correctly decrypt the file header, or EIO if the verification data does not correctly verify.

Dependencies

None.

See also

sfs_fclose.

8.2.3 sfs_fclose – Close a Secure File Storage file

C specification

```
#include <libisolation.h>

int sfs_fclose(SFS_FILE *stream);
```

Description

sfs_fclose closes the Secure File Storage associated with *stream*. If the stream is open for writing, any buffered data is first written to the file.

sfs_fclose return 0 if successful, EOF otherwise. In either case, no further access to the stream is allowed.

During this operation, sfs_fclose updates the file header and verification data if the file was open for writing. As a result of these operations, sfs_fclose may set errno to any of the error codes associated with fwrite and fclose.

Dependencies

None.

See also

sfs_fopen.

8.2.4 sfs_fread – Read data from a Secure File Storage file

C specification

```
#include <libisolation.h>

int sfs_fread(void *buffer, size_t element_size, size_t count,
     SFS_FILE *stream);
```

Description

sfs_fread will read *count* elements from the *stream*, each of size *element_size*, into the memory referenced by *buffer*. If less than count elements are remaining in the stream, then fewer bytes will be returned. If the number of remaining bytes is not a multiple of *element_size*, then an error is returned.

Upon successful completion, sfs_fread returns the number of elements of size element_size that were read and returned. The stream location is incremented by the number of bytes returned.

As part of this operation, sfs_fread reads both the requested data and the associated verification information. sfs_fread may return with errno set to any of the error codes associated with fread. In addition, if the verification data does not agree with the computed verification over the requested data, then sfs_fread returns with errno set to EIO.

Dependencies

None.

See also

sfs_fwrite.

8.2.5 sfs_fwrite – Write data to a Secure File Storage file

C specification

```
#include <libisolation.h>

int sfs_fwrite(void *buffer, size_t element_size, size_t count,
     SFS_FILE *stream);
```

Description

sfs_fwrite will write *count* elements to the *stream*, each of size *element_size*, from the memory referenced by *buffer*.

sfs_fwrite returns the number of elements written. If the return value is less than *count*, then a write error occurred.

During this operation, sfs_fwrite writes the data to the encrypted file and computes the new verification data. As a result of these operations, sfs_fwrite may set errno to any of the error codes associated with fwrite.

Dependencies

None.

See also

sfs_fread.

8.2.6 sfs_fseek – Set the current position for a Secure File Storage file

C specification

```
#include <libisolation.h>

int sfs_fseek(SFS_FILE *stream, long int offset, int wherefrom);
```

Description

sfs_fseek sets the current stream location to that specified by *offset*. The interpretation of *offset* is determined by *wherefrom*.

SEEK_SET Interpret *offset* as an absolute offset from the start of the file.

SEEK_CUR Interpret *offset* as a relative offset from the current stream location.

SEEK_END Interpret offset as a relative offset from the stream position immediately following the last data byte in the file.

If successful, sfs_fseek return 0; otherwise, sfs_fseek returns -1. sfs_fseek may return with errno set to any of the error codes associated with fseek.

Dependencies

None.

See also

sfs_ftell.

8.2.7 sfs_ftell – Return the current position for a Secure File Storage file

C specification

```
#include <libisolation.h>

int sfs_ftell(SFS_FILE *stream);
```

Description

sfs_ftell returns the current position offset within the *stream*.

If successful, sfs_ftell returns 0 or a positive value; otherwise, sfs_ftell returns -1. sfs_ftell may return with errno set to any of the error codes associated with ftell.

Dependencies

None.

See also

sfs_fseek.

8.3 PPE-Assisted Functions

PPE-serviced SPE C library functions are described in chapter 4.1 of “Cell Broadband Engine SDK Libraries Overview and Users Guide”. However, the implementation does not operate as described when the application is executing in the isolation mode due to environment differences. Analogous replacement functions for the SPE functions are provided in `libisolation.a`, which work around these differences and provide equivalent function. Note that restrictions described in this section apply only to PPE-serviced C library functions; SPE-serviced C library functions operate as normal.

The replacement functions make use of an auxiliary buffer, which is located in the area of LS reserved for system use (LS address range 0x3E000-0x3E0C0). The size of auxiliary buffer is initially 192 bytes, but the size may be changed using the `change_ppuassist_buf_len` function (see section8.3.1). Buffer size can be set to no more than 8016 bytes, thus placing a limit on the combined size of function parameters. This implies, for example, that it is not possible to fread()/fwrite() more than ~7900 bytes at a time, and it is not possible to printf more than ~7900 bytes of data at a time.

The size of auxiliary buffer must be at least as large as the sum of the following:

1. 16 bytes,
2. 16 bytes for each function parameter,
3. for each string, array, or struct referenced, the size in bytes of the referenced item rounded up to the next multiple of 16 bytes, and
4. for printf, fprintf, vprintf, and vfprintf, an additional 32 bytes.

For example, consider the function call,

```
printf(“This is a %s call with %d parameters.\n”, “PPE-assisted call”,
    (int) 3);
```

This call has 3 parameters: two strings and one integer. The format string requires 48 bytes and the parameter string requires 32 bytes. An additional 32 bytes are required for the printf function. The total buffer size required for the call is

```
size of auxiliary buffer = 16 + 3*16 + (48 + 32) + 32 = 176 bytes
```

If size of auxiliary buffer was set to an insufficient value, PPE-assisted call will terminate early, with errno variable set to ENOBUFS.

Developers must ensure that, when invoking such a PPE-assisted call, they have no outstanding DMAs with their source and/or destination overlapping with auxiliary buffer, to avoid incorrect execution of PPE-assisted call and/or DMA.

PPE-assisted functions that are unsupported in emulated isolation mode will produce a missing link error at application build time, by using stubs which link to a non-existent function.

To use PPE-serviced calls in emulated isolation mode, libisolation.a must be linked after libc.a and libgloss.a, but only after defining the symbol __send_to_ppe as being wrapped (redefined to __wrap___send_to_ppe) at link time. Including make.iso.footer in the Makefile will resolve the correct linkage steps.

8.3.1 change_ppuassist_buf_len – PPE Assisted Functions

C specification

```
#include <libisolation.h>

int change_ppuassist_buf_len(unsigned int new_len)
```

Description

The *change_ppuassist_buf_len* function sets the size of the buffer area to be used for PPE-assisted functions. These are the functions that require assistance from code that executes on the PPE. Parameters to the PPE code are passed using a buffer in the area of LS reserved for system use (LS address range 0x3E000-0x3E0C0), and results are returned using the same buffer. Since this buffer competes with other uses for this reserved area of LS, this function can be used to set the size of the buffer smaller when not needed, or larger for larger parameters (e.g., large strings). Please see section 8.3 “PPE-Assisted Functions” for more details.

Minimal and maximal sizes buffer can be set to are 80 and 8016 bytes, respectively. Value of 0 is returned on successful change, or -1 is returned on failure.

Dependencies

None.

See also

None.

8.3.2 Unsupported libc.a Functions

The following PPE-assisted functions from libc.a are not supported in isolation mode:

```
gets

fscanf

scanf

setbuf

setvbuf

snprintf

sprintf

sscanf

tmpnam
```

```
vfscanf
vscanf
vsnprintf
vsprintf
vsscanf
system
```

8.3.3 Unsupported libgloss.a Functions

The following PPE-assisted functions from libgloss.a are not supported in isolation mode:

```
ftok_ea
mmap_ea
mremap_ea
msync_ea
munmap_ea
shmat_ea
shmctl_ea
shmdt_ea
shmget_ea
shm_open
shm_unlink
mktemp
utimes
readv
writev
```

8.3.4 Unsupported libea.a Functions

EA functionality is not supported in this release.

8.3.5 Limited Support libc.a Functions

The following PPE-assisted functions from libc.a have limited support in isolation mode. This support is limited as the types of arguments are determined by scanning the format string; if the format string is wrong or does not match the argument types, then it is possible to leak information.

```
fprintf
```

```
printf
vfprintf
vprintf
```

Additional limitation is that conversion character '%n' is not supported. This is the conversion character that makes *printf functions write back number of characters written so far into original parameter list.

9 SDK Programming Examples

The source code and Makefile's for the samples associated with the Cell/B.E. Security SDK are installed into the `/opt/cell/sdk/prototype/src/examples/isolation` directory. Each of the samples has an associated README file. There is also a top-level README in the `src/samples/isolation` directory introducing the structure of the sample code source tree. In addition, there are a number of useful PDF documents in the `/opt/cell/sdk/docs` directory including a programming tutorial.

Code specific to a given Cell/B.E. processor unit type is in a corresponding place within a given sample's directory:

- sample's directory for code compiled for execution on the PPE
- spu/ subdirectory for code compiled for execution on an SPE

9.1 Changing the default compiler

In the `/opt/cell/sdk/buildutils` are some top level makefiles that control the build environment for all of the samples, and `make.iso.footer`, that controls the build environment for the emulated isolated SPU. The Cell/B.E. Security SDK samples are built using `/opt/cell/sdk/prototype/src/examples/isolation/Makefile`; these samples are not built using any makefile present above this directory (except `make.iso.footer`). All of the samples have their own makefile but the common definitions are in the top level makefiles. The build environment makefiles are documented in `/opt/cell/sdk/README_build_env.txt`

Environment variables in the `/opt/cell/sdk/buildutils/make.env` makefile are used to determine which compiler is used to build the samples.

9.2 System root directory (for simulator users only)

Building the libraries and samples copies output files into a special directory named `/opt/cell/sysroot`. This directory has a very similar structure to a normal system root directory (that is, /) and contains the usual subdirectories such as `/bin`, `/usr` and `/etc`. Compiled binaries of examples are deployed into directory `/opt/cell/sysroot/opt/cell/sdk/prototype/usr/bin/examples`.

After logging on as `root` this `sysroot` directory can be synched up with the simulator sysroot image file using the install script with the synch task. The command is

```
/opt/cell/cellsdk_sync_simulator
```

and is useful whenever a library or sample has been recompiled. This script reduces user error by providing a standard mechanism to mount the system root image, sync the contents of the two corresponding directories, and unmounting the system root image.

9.3 File I/O Programming Example

This example demonstrates how an SPU program can perform file operations (i.e. creation, reading, updating, deletion) using either POSIX or C99 functions. It is described in the code for the iso_fileio sample provided as part of the SDK in the directory
`/opt/cell/sdk/prototype/src/examples/isolation/iso_fileio`

9.4 Copying Encrypted Data Example

This example demonstrates how to utilize the system memory as a secure shared buffer. The SPU application encrypts a plaintext in LS, and copies out the ciphertext to the system memory. It also shows 'decrypt_in' usage where the SPU application copies the cipher text from the system memory back into the LS, and decrypts it. Code for this sample is located under
`/opt/cell/sdk/prototype/src/examples/isolation/iso_enccopy`

9.5 Copying Encrypted Data with Replay Protection Example

This example is a more advanced version of the previous one, it utilizes the nonce to prevent replay attacks on the data copied out. Code for this sample is located under directory
`/opt/cell/sdk/prototype/src/examples/isolation/iso_enccopy2`

9.6 Encrypted SPU Application Example

This sample demonstrates a fully encrypted SPU program. It utilizes additional variables in the `spu/Makefile`:

```
     ISO_ENCRYPT_SEC := ALL
     ISO_KERN_KEY    = /etc/pki/cell-spu-isolation/loader/loader.app_encrypt_key.public.pem
```

Variable `ISO_ENCRYPT_SEC` specifies which sections of SPU program image to encrypt, and can be set to `ALL` or `.text`. Variable `ISO_KERN_KEY` specifies the public key to which SPU program image is encrypted.

Code for this sample is located under directory
`/opt/cell/sdk/prototype/src/examples/isolation/iso_encrypted`

Please note that with the publicly available Security SDK, cryptographic encryption is not being used and thus the key is ignored by the SPE Secure Application build tool which simply XORs the application image with a static value. In the CDA version of the SDK, cryptographic encryption is used and the ISO_KERN_KEY is used in the process.