AOCL Crypto IPP Plugin Documentation

IPP Plugin Documentation

Contents

List of Figures

Part I

Key Terminologies

1) IPP-CP - Intel Performance Primitives - CP
2) Asymmetric Cryptography - Cryptography which uses single key
3) Symmetric Cryptography - Cryptography which uses two keys, one private key and one public key for encryption and decryption.
4) Hashing - Creating a value which represents a data which is a trap door(cant be de-coded/converted back to the original file), which is mainly used for checking the integrity.
5) RNG - Random Number Generator
6) BRNG - Base Random Number Generator
7) PRNG - Pseudo Random Number Generator
8) AES - Advanced Encryption Standard

Part II

Introduction

Chapter 1

About IPPCP

IPP-CP is an opensource cryptographic primitives library. It is specifically optimized for Intel CPU.

It supports Symmetric Cryptography, Hashing Primitives, Authentication Primitives, Public Key (Asymmetric) Cryptography.

Key algorithms supported by IPP-CP are as follows.

- Symmetric Cryptography Primitive Functions:
    - AES (ECB, CBC, CTR, OFB, CFB, XTS, GCM, CCM, SIV)
    - SM4 (ECB, CBC, CTR, OFB, CFB, CCM)
    - TDES (ECB, CBC, CTR, OFB, CFB)
    - RC4
- One-Way Hash Primitives:
    - SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
    - MD5
    - SM3
- Data Authentication Primitive Functions:
    - HMAC
    - AES-CMAC
- Public Key Cryptography Functions:
    - RSA, RSA-OAEP, RSA-PKCS_v15, RSA-PSS
    - DLP, DLP-DSA, DLP-DH
    - ECC (NIST curves), ECDSA, ECDH, EC-SM2
- Multi-buffer RSA, ECDSA, SM3, x25519
- Finite Field Arithmetic Functions
- Big Number Integer Arithmetic Functions
- PRNG/TRNG and Prime Numbers Generation

To read more about IPP-CP click here

Currently used version of IPP-CP is `ipp-crypto_2021_8` when this document was written.

Chapter 2

Usage of IPP-Plugin - Brief

2.1 Building examples

To build examples, simply invoke `make -j` from the root of the package directory. Set LD_LIBRARY_PATH properly to lib directory in the package to avoid loader issues while executing. Executables for examples should be found in bin directory after make is successful.

For more information please read BUILD_Examples.md

2.2 Preloading IPP-Compat Lib

```
export LD_LIBRARY_PATH=/path/to/libalcp.so:$LD_LIBRARY_PATH
LD_PRELOAD=/path/to/libipp-compat.so ./program_to_run
```

- Export Path should be a directory.
- Preload Path should be the .so file itself.
- Any command can follow LD_PRELOAD.

For more details, please refer to Preloading_IPP (Preloading IPP-CP wrapper plugin)

Part III

Preloading

When you specify LD_PRELOAD=/path/to/somelib.so, loader will load this library first before loading the actual program into memory. This dynamic linking in any program which is running with preloaded library will try to find the symbol in preloaded library first before attempting to search LD_LIBRARY_PATH for the library specified in the ELF executable. This means even though another lib which may have same symbol is available, preloaded library overrides the linkage during run time.

To read more about preloading, please check ld.so man page and ld man page.

Chapter 3

Temporary Preloading

To preload temporarily, one can modify the environment variable LD_PRELOAD. This can be setup in bashrc or zshrc or any rc file of your shell leading to semi permanent preloading. Even if we can setup the same concept in /etc/environment, for a more permanent setup, LD_PRELOAD is not recommended.

LD_PRELOAD can be used for on demand preloading to test out if preloading works as indented. Temporary preloading is also recommended because it does not modify the loader parameters for programs that do not require the preload.

If you are looking for a more perminant setup which is not recommended, you can look below.

Chapter 4

Permanent Preloading

Warning: This type of preloading may break some other program which may load symbols with same name and parameter list as the preloaded library. Only use this if you know what you are doing and is really sure that there would be no such conflicts.

To preload permanently, you would need to either set LD_PRELOAD environment globally as discussed above but its not the way it is supposed to be done.

In order to preload globally there is a config file `/etc/ld.so.preload` this file by default do not exist in any machine as its not a good idea to preload library globally. You can create/edit this file and add path of each `.so` file you wish to preload line by line.

Example /etc/ld.so.preload

```
/path/to/somelib.so
/path/to/someotherlib.so
```

Lines are parsed in order that means `somelib.so` will override all the symbols exported by `someotherlib.so`.

Chapter 5

Precedence of Loading

Precedence of loading is determined by is it preloaded, does the lib come first in the list.

Let's say for example you are preloading `lib1.so lib2.so` while loading `lib3.so lib4.so` because inside elf its specified to load it.

```
LD_LIBRARY_PATH=/some/path/lib1.so:/some/path/lib2.so ./someexecutable
```

If some executable is linked to lib3.so and lib4.so in order. Then the symbol lookup order will be

1) `lib1.so`
2) `lib2.so`
3) `lib3.so`
4) `lib4.so`

You can say that `lib1.so` and `lib2.so` will override both `lib3.so` and `lib4.so` as they are preloaded. lib1.so will override `lib2.so` as its the first in the list. `lib3.so` will override `lib4.so` as `lib3.so` comes first.

Chapter 6

Editing linkage

In Linux you can edit the linkage of an executable, this can be used to remove the linkage to existing library to replace with a better version of it or some other library which exports the same symbols and does the same thing as the other library but better.

Editing linkage is not recommended if the program has multiple executables, as every executable's linkage needs to be edited.

In order to edit the linkage of a program or library, you need to install a package known as `patchelf`.

6.1 Important Arguments of patchelf

1. `--remove-needed` - removed a lib as needed, thereby loader does not load it anymore.
2. `--add-needed` - add a lib as needed, loader loads if any symbols are there in this which can be linked during run time then it will link it.
3. `--replace-needed` - replaces already needed lib with a new lib, this can be seen as a combination of above two arguments.

To know more about the parameters do

```
patchelf --help
```

or click here

Chapter 7

Preloading IPP-CP wrapper plugin.

Assuming that you are in the package root directory and you have IPPCP setup and in the environment.

```
LD_PRELOAD=$PWD/lib/libipp-compat.so executable_path
```

Example with Intel IPP AES CTR Encryption.

```
wget https://raw.githubusercontent.com/intel/ipp-crypto/ipp-crypto_2021_6/examples/aes/a
g++ aes-256-ctr-encryption.cpp -o aes-ctr -lippcp
LD_PRELOAD=$PWD/lib/libipp-compat.so ./aes_ctr
```

Part IV

Appendix

Chapter 8

Compiling and installing IPPCP

For official guide on how to compile and install IPPCP click here

```
git clone https://github.com/intel/ipp-crypto -b ipp-crypto_2021_6
cd ipp-crypto
mkdir build
cd build
cmake ../ -DCMAKE_INSTALL_PEFIX=/usr/local -DARCH=intel64
make -j$(nproc --all) # Low memory, override the -j parameter.
sudo make install
```

For setting up environment.

This is normally not required, if you face errors please add these to your bashrc/zshrc/somerc

```
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
export LIBRARY_PATH=/usr/local/lib:$LIBRARY_PATH
export PATH=/usr/local/bin:$PATH
export C_INCLUDE_PATH=/usr/local/include:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=/usr/local/include:$CPLUS_INCLUDE_PATH
```