FREEWAVE

PUBLICLY AVAILABLE FREEWAVE PRODUCT ENCRYPTION SOURCE CODE

24 February 2022

Per BIS Part 734.3(b)(3), 734.7 and 742.15(b), FreeWave is publishing and making publicly available free of charge, standard encryption source code used in its Zum and Fusion family of products.

- The following table details the FreeWave product families that include the encryption source code located at the listed URLs.

FreeWave Products	Source Code Location	Manufacturer
Zum Family Products	https://github.com/openssl/openssl	OpenSSL Project
Fusion Family Products	https://github.com/openssl/openssl	OpenSSL Project
FreeWave general standard encryption Code used in Zum and Fusion Family Products	RadioCryptoThorv3cpp: https://freewave.com RadioCryptoThorv2h: https://freewave.com	FreeWave Technologies, Inc.
FreeWave general standard encryption code used in FreeWave Edge products	Securitygo: https://freewave.com	FreeWave Technologies, Inc.

- Per 742.15(b), by making the above listed source code publicly available it is no longer subject to the EAR and removes the encryption export controls in Category 5 Part 2 for the listed final end-item products. The final end-items in the Zum and Fusion families remain subject to the EAR, but in a Category outside of Category 5 Part 2.

- This document will be updated as required to reflect any additions, modifications or updates to the table above.

If there are any questions about this document, please contact FreeWave Compliance at compliance@freewave.com

/*

This license governs use of the accompanying software. If you use the software, you accept this license. If you do not accept the license, do not use the software.

1. Definitions
The terms "reproduce," "reproduction" and "distribution" have the same meaning here as under U.S. copyright law.

"You" means the licensee of the software.

"Your company" means the company you worked for when you downloaded the software.

"Reference use" means use of the software within your company as a reference, in read only form, for the sole purposes of debugging your products, maintaining your products, or enhancing the interoperability of your products with the software, and specifically excludes the right to distribute the software outside of your company.

"Licensed patents" means any Licensor patent claims which read directly on the software as distributed by the Licensor under this license.

2. Grant of Rights
(A) Copyright Grant- Subject to the terms of this license, the Licensor grants you a non-transferable, non-exclusive, worldwide, royalty-free copyright license to reproduce the software for reference use.

(B) Patent Grant- Subject to the terms of this license, the Licensor grants you a non-transferable, non-exclusive, worldwide, royalty-free patent license under licensed patents for reference use.

3. Limitations
(A) No Trademark License- This license does not grant you any rights to use the Licensor's name, logo, or trademarks.

(B) If you begin patent litigation against the Licensor over patents that you think may apply to the software (including a cross-claim or counterclaim in a lawsuit), your license to the software ends automatically.

(C) The software is licensed "as-is." You bear the risk of using it. The Licensor gives no express warranties, guarantees or conditions. You may have additional consumer rights under your local laws which this license cannot change. To the extent

```
permitted under your local laws, the Licensor excludes the
implied warranties of merchantability, fitness for a particular
purpose and non-infringement.
*/

#include <sys/stat.h>
#include <sys/types.h>
#include <fstream>
#include <iomanip>
#include <cassert>
#include <sstream>
#include "CliHandler.h"

#include <openssl/aes.h>
#include <openssl/modes.h>
#include <openssl/rand.h>
#include <openssl/rsa.h>
#include <openssl/evp.h>

#include "gen_shared_memory.h"
#include "print.h"
#include "sm_api.h"
#include "utils.h"
#include "utilsCpp.h"
#include "RadioCryptoThor.h"

#define SECURITY_LEVEL_0 0
#define SECURITY_LEVEL_4 4
#define SECURITY_LEVEL_7 7
#define SZ_KEY_AES_128 16
#define SZ_KEY_AES_256 32
#define SZ_AUX_SEC_HDR_KEYIDMODE0 5
#define SZ_MIC_0 0
#define SZ_MIC_16 16

using namespace std;

#ifdef __cplusplus
extern "C" {
#endif
    extern int SetThorEncryptionMode(cliControlBlock_t* pCb,
sharedMemoryItem_t* item, uint8_t* parameterValue, uint8_t**
pReturnValue);
    extern int GetThorEncryptionMode(cliControlBlock_t* pCb,
sharedMemoryItem_t * item, uint8_t** pReturnValue);
    extern int SetThorActiveKey(cliControlBlock_t* pCb,
sharedMemoryItem_t* item, uint8_t* parameterValue, uint8_t**
pReturnValue);
    extern int GetThorActiveKey(cliControlBlock_t* pCb,
sharedMemoryItem_t * item, uint8_t** pReturnValue);
    extern int SetThorKeyGetter(cliControlBlock_t* pCb,
sharedMemoryItem_t * item, uint8_t** pReturnValue);
```

```
    extern int SetThorKey(cliControlBlock_t* pCb,
sharedMemoryItem_t* item, uint8_t* parameterValue, uint8_t**
pReturnValue);
    extern int GetThorKey(cliControlBlock_t* pCb,
sharedMemoryItem_t* item, uint8_t* parameterValue, uint8_t**
pReturnValue);
    extern int SetThorKeyx(cliControlBlock_t* pCb,
sharedMemoryItem_t* item, uint8_t* parameterValue, uint8_t**
pReturnValue);
    extern int GetThorKeyx(cliControlBlock_t* pCb,
sharedMemoryItem_t * item, uint8_t** pReturnValue);
#ifdef __cplusplus
}
#endif

extern RSA* createRSA(uint8_t* key, int usePublic, BIO** ppBIO);
extern const char* rInfo;
extern const char defaultDCPrivateKey[];
extern const char defaultDCPublicKey[];

const char* rInfo = defaultDCPrivateKey;

RSA* createRSA(uint8_t* key, int usePublic, BIO** ppBIO) {
    RSA* rsa = NULL;
    BIO* keybio = NULL;

    keybio = BIO_new_mem_buf(key, -1);
    if (keybio == NULL) {
        printf("Failed to create key BIO");
        return 0;
    }

    if (usePublic) {
        rsa = PEM_read_bio_RSA_PUBKEY(keybio, &rsa, NULL, NULL);
    }
    else {
        rsa = PEM_read_bio_RSAPrivateKey(keybio, &rsa, NULL,
NULL);
    }

    if (rsa == NULL) {
        printf("Failed to create RSA");
    }

    if (ppBIO != nullptr)
    {
        *ppBIO = keybio;
    }

    return rsa;
}
```

```
RadioCryptoThor::RadioCryptoThor(CryptoMode mode) :
    _mode(mode),
    _activeKey(-1)
{
    _nonce =
    {
        { 0x00, 0x00, 0x00, 0xFF, 0xFE, 0x00, 0x00, 0x00 },
        htonl(0UL),
        SECURITY_LEVEL_0
    };
    LoadKeys();
}

RadioCryptoThor::~RadioCryptoThor()
{
}

RadioCryptoThor* RadioCryptoThor::Instance(void)
{
    static RadioCryptoThor cryptoThor(AES_CCM);
    return &cryptoThor;
}

void RadioCryptoThor::SetMode(CryptoMode mode)
{
    std::lock_guard<std::mutex> lock1(_mutexEncrypt);
    _mode = mode;
    if (_nonce.securityLevel != 0)
    {
        EVP_CIPHER_CTX_free(_ctxEncrypt_1_1);
        EVP_CIPHER_CTX_free(_ctxDecrypt_1_1);
    }

    _ctxEncrypt_1_1=EVP_CIPHER_CTX_new();
    _ctxDecrypt_1_1=EVP_CIPHER_CTX_new();

    if (_activeKey>=0)
    {
        vector<uint8_t>& key = _key[_activeKey];
        if (key.size() == 0)
        {
            _nonce.securityLevel = SECURITY_LEVEL_0;
        }
        else if (key.size() <= SZ_KEY_AES_128)
        {
            key.resize(SZ_KEY_AES_128, 0);
            if (AES_CTR == _mode)
            {
                _nonce.securityLevel = SECURITY_LEVEL_4;
                EVP_EncryptInit_ex(_ctxEncrypt_1_1, EVP_aes_128
_ctr(), nullptr, nullptr, nullptr);
                EVP_DecryptInit_ex(_ctxDecrypt_1_1, EVP_aes_128
```

```
_ctr(), nullptr, nullptr, nullptr);
            }
            else if (AES_CCM == _mode)
            {
                _nonce.securityLevel = SECURITY_LEVEL_7;
                EVP_EncryptInit_ex(_ctxEncrypt_1_1, EVP_aes_128
_ccm(), nullptr, nullptr, nullptr);
                EVP_DecryptInit_ex(_ctxDecrypt_1_1, EVP_aes_128
_ccm(), nullptr, nullptr, nullptr);
            }
            else
            {
                _nonce.securityLevel = SECURITY_LEVEL_0;
                return;
            }
        }
        else
        {
            key.resize(SZ_KEY_AES_256);
            if (AES_CTR == _mode)
            {
                _nonce.securityLevel = SECURITY_LEVEL_4;
                EVP_EncryptInit_ex(_ctxEncrypt_1_1, EVP_aes_256
_ctr(), nullptr, nullptr, nullptr);
                EVP_DecryptInit_ex(_ctxDecrypt_1_1, EVP_aes_256
_ctr(), nullptr, nullptr, nullptr);
            }
            else if (AES_CCM == _mode)
            {
                _nonce.securityLevel = SECURITY_LEVEL_7;
                EVP_EncryptInit_ex(_ctxEncrypt_1_1, EVP_aes_256
_ccm(), nullptr, nullptr, nullptr);
                EVP_DecryptInit_ex(_ctxDecrypt_1_1, EVP_aes_256
_ccm(), nullptr, nullptr, nullptr);
            }
            else
            {
                _nonce.securityLevel = SECURITY_LEVEL_0;
                return;
            }
        }
    }
}

RadioCryptoThor::CryptoMode RadioCryptoThor::GetMode(void)
{
    std::lock_guard<std::mutex> lock1(_mutexEncrypt);
    CryptoMode mode = _mode;
    return mode;
}

void RadioCryptoThor::SetActiveKey(int i)
```

```
{
    unique_lock<mutex> lock1(_mutexEncrypt);
    bool modeRefresh = false;

    if (_activeKey != i)
    {
        modeRefresh = true;
    }

    _activeKey = i;

    if (modeRefresh)
    {
        lock1.unlock();
        SetMode(_mode);
    }
}

int RadioCryptoThor::GetActiveKey(void)
{
    std::lock_guard<std::mutex> lock1(_mutexEncrypt);
    return _activeKey;
}

void RadioCryptoThor::SetKeyVal(int i, const vector<uint8_t>&
key)
{
    if (i > NUM_STORED_KEYS - 1)
    {
        return;
    }
    if (i < 0) { return; }
    unique_lock<mutex> lock1(_mutexEncrypt);
    _key[i] = key;

    if (_activeKey == i)
    {
        lock1.unlock();
        SetMode(_mode);
    }
}

void RadioCryptoThor::GetKeyVal(int i, vector<uint8_t>& key)
{
    std::lock_guard<std::mutex> lock1(_mutexEncrypt);
    key = _key[i];
}

bool RadioCryptoThor::Encrypt(uint16_t srcAddr, uint8_t* inbuf,
uint32_t inbufSize, vector<uint8_t>& outbuf)
{
    std::lock_guard<std::mutex> lock(_mutexEncrypt);
```

```
    if (_activeKey < 0) { return false; }

    if (SECURITY_LEVEL_0 == _nonce.securityLevel)
    {
        return false;
    }
    else if (SECURITY_LEVEL_4 == _nonce.securityLevel)
    {
        outbuf.resize(SZ_AUX_SEC_HDR_KEYIDMODE0 + inbufSize, 0);
    }
    else if (SECURITY_LEVEL_7 == _nonce.securityLevel)
    {
        EVP_CIPHER_CTX_ctrl(_ctxEncrypt_1_1,
EVP_CTRL_CCM_SET_IVLEN, 15 - L, nullptr);
        outbuf.resize(SZ_AUX_SEC_HDR_KEYIDMODE0 + inbufSize +
SZ_MIC_16, 0);
        EVP_CIPHER_CTX_ctrl(_ctxEncrypt_1_1,
EVP_CTRL_CCM_SET_TAG, SZ_MIC_16, nullptr);
    }
    else
    {
        outbuf.clear();
        return false;
    }
    UpdateNonce(srcAddr);
    outbuf.data()[0] = _nonce.securityLevel & 0x07;
    *(uint32_t*)&outbuf.data()[1] = _nonce.frameCounter;

    int tmpLen = 0;
    if (SECURITY_LEVEL_4 == _nonce.securityLevel)
    {
        unsigned char aesCtrNonce[16];
        memcpy(&aesCtrNonce[1], &_nonce, sizeof(_nonce));
        aesCtrNonce[0] = 0xFF;
        aesCtrNonce[14] = 0x00;
        aesCtrNonce[15] = 0x00;
        EVP_EncryptInit_ex(_ctxEncrypt_1_1, nullptr, nullptr,
_key[_activeKey].data(), aesCtrNonce);
    }
    else
    {
        EVP_EncryptInit_ex(_ctxEncrypt_1_1, nullptr, nullptr,
_key[_activeKey].data(), (unsigned char*)&_nonce);
        EVP_EncryptUpdate(_ctxEncrypt_1_1, nullptr, &tmpLen,
nullptr, inbufSize);
        EVP_EncryptUpdate(_ctxEncrypt_1_1, nullptr, &tmpLen,
_key[_activeKey].data(), _key[_activeKey].size());
    }
    unsigned char tmpBuf[2048];
    EVP_EncryptUpdate(_ctxEncrypt_1_1, tmpBuf, &tmpLen, inbuf,
inbufSize);
```

```
    memcpy(outbuf.data() + SZ_AUX_SEC_HDR_KEYIDMODE0, tmpBuf,
inbufSize);
    EVP_EncryptFinal_ex(_ctxEncrypt_1_1, nullptr, &tmpLen);
    if (_nonce.securityLevel == SECURITY_LEVEL_7)
    {
        EVP_CIPHER_CTX_ctrl(_ctxEncrypt_1_1,
EVP_CTRL_CCM_GET_TAG, SZ_MIC_16, outbuf.data() +
SZ_AUX_SEC_HDR_KEYIDMODE0 + inbufSize);
    }
    return true;
}

bool RadioCryptoThor::Decrypt(uint16_t srcAddr, uint8_t* inbuf,
uint32_t inbufSize, vector<uint8_t>& outbuf)
{
    if (_activeKey < 0) { return false; }
    std::lock_guard<std::mutex> lock(_mutexEncrypt);
    size_t cipherTextLen = 0;
    int tempLen = 0;
    outbuf.clear();
    if (SECURITY_LEVEL_0 == _nonce.securityLevel)
    {
        return false;
    }

    if (SECURITY_LEVEL_4 == _nonce.securityLevel)
    {
        if ((inbufSize) < SZ_AUX_SEC_HDR_KEYIDMODE0 + 1)
        {
            return false;
        }
        cipherTextLen = (inbufSize) - SZ_AUX_SEC_HDR_KEYIDMODE0;
    }
    else if (SECURITY_LEVEL_7 == _nonce.securityLevel)
    {
        if (1 != EVP_CIPHER_CTX_ctrl(_ctxDecrypt_1_1,
EVP_CTRL_CCM_SET_IVLEN, 15 - L, nullptr))
        {
            outbuf.clear();
            return false;
        }
        if ((inbufSize) < SZ_AUX_SEC_HDR_KEYIDMODE0 + SZ_MIC_16 +
1)
        {
            return false;
        }
        if (1 != EVP_CIPHER_CTX_ctrl(_ctxDecrypt_1_1,
EVP_CTRL_CCM_SET_TAG, SZ_MIC_16, (void*)(&inbuf[inbufSize -
SZ_MIC_16])))
        {
            return false;
        }
```

```
        cipherTextLen = (inbufSize) - SZ_AUX_SEC_HDR_KEYIDMODE0 -
SZ_MIC_16;
    }

    Nonce nonce;
    memset(&nonce, 0, sizeof(nonce));
    *(uint16_t*)&nonce.extendedSourceAddress[3] = htons(srcAddr);
    nonce.frameCounter = *(uint32_t*)&inbuf[1];
    nonce.securityLevel = inbuf[0] & 7;

    if (SECURITY_LEVEL_4 == _nonce.securityLevel)
    {
        unsigned char aesCtrNonce[16];
        memcpy(&aesCtrNonce[1], &nonce, sizeof(nonce));
        aesCtrNonce[0] = 0xFF;
        aesCtrNonce[14] = 0x00;
        aesCtrNonce[15] = 0x00;

        if (1 != EVP_DecryptInit_ex(_ctxDecrypt_1_1, nullptr,
nullptr, _key[_activeKey].data(), aesCtrNonce))
        {
            return false;
        }
    }
    else
    {
        if (1 != EVP_DecryptInit_ex(_ctxDecrypt_1_1, nullptr,
nullptr, _key[_activeKey].data(), (unsigned char*)&nonce))
        {
            return false;
        }
        if (1 != EVP_DecryptUpdate(_ctxDecrypt_1_1, nullptr,
&tempLen, nullptr, cipherTextLen))
        {
            return false;
        }
        if (1 != EVP_DecryptUpdate(_ctxDecrypt_1_1, nullptr,
&tempLen, _key[_activeKey].data(), _key[_activeKey].size()))
        {
            return false;
        }
    }
    outbuf.resize(cipherTextLen, 0);
    unsigned char tmpBuf[2048];
    memcpy(tmpBuf, inbuf + SZ_AUX_SEC_HDR_KEYIDMODE0,
min(sizeof(tmpBuf), cipherTextLen));
    if (EVP_DecryptUpdate(_ctxDecrypt_1_1, outbuf.data(),
&tempLen, tmpBuf, cipherTextLen) > 0)
    {
        return true;
    }
    outbuf.clear();
```

```
    return false;
}

int SetThorEncryptionMode(cliControlBlock_t* pCb,
sharedMemoryItem_t* item, uint8_t* parameterValue, uint8_t**
pReturnValue)
{
    if (nullptr != pReturnValue)
    {
        *pReturnValue = (uint8_t*)"";
    }

    int isValid = 0;
    RadioCryptoThor::CryptoMode mode =
(RadioCryptoThor::CryptoMode)stringToUIntWithLabelGroup((const
char *)parameterValue, &isValid, item->labelGroup);

    if ((isValid))
    {
        RadioCryptoThor::Instance()->SetMode(mode);
    }
    else
    {
        return SM_PARAMETER_NOT_VALID;
    }
    return GetThorEncryptionMode(pCb,item,pReturnValue);
}

int GetThorEncryptionMode(cliControlBlock_t* pCb,
sharedMemoryItem_t * item, uint8_t** pReturnValue)
{
    if (nullptr != pReturnValue)
    {
        *pReturnValue = (uint8_t*)
getLabel(RadioCryptoThor::Instance()->GetMode(), item->
labelGroup);
    }

    return SM_STATUS_OK;
}

int SetThorActiveKey(cliControlBlock_t* pCb, sharedMemoryItem_t*
item, uint8_t* parameterValue, uint8_t** pReturnValue)
{
    int isValid = 0;
    int activeKey = stringToInt((const char*)parameterValue,
&isValid);
    if (!isValid)
    {
        return SM_PARAMETER_NOT_VALID;
    }
    if ((activeKey < 0) || (activeKey > NUM_STORED_KEYS))
```

```
    {
        return SM_PARAMETER_NOT_VALID;
    }
    RadioCryptoThor::Instance()->SetActiveKey(activeKey-1);
    return GetThorActiveKey(pCb, item, pReturnValue);
}

int GetThorActiveKey(cliControlBlock_t* pCb, sharedMemoryItem_t *
item, uint8_t** pReturnValue)
{
    if (nullptr != pReturnValue)
    {
        *pReturnValue = (uint8_t*)getLabel((uint32_t)
(RadioCryptoThor::Instance()->GetActiveKey() + 1), item->
labelGroup);
    }
    return SM_STATUS_OK;
}

void RadioCryptoThor::LoadKeys(void)
{
    BIO* pBio = nullptr;
    RSA* rsa = createRSA((uint8_t*)rInfo, 0, &pBio);

    if (!rsa)
    {
        return;
    }
    unsigned char* pDecrypted = new unsigned char[RSA_size(rsa)];

    for (int i = 0; i < NUM_STORED_KEYS; i++)
    {
        ifstream ifs(ConvertPathToOs(std::string(KEYSTORE_PATH) +
"/key" + to_string(i + 1)), ios::in | ios::binary);

        if (ifs)
        {
            std::vector<char>
encData((std::istreambuf_iterator<char>(ifs)),
                std::istreambuf_iterator<char>());
            int szDecrypted = RSA_private_decrypt(encData.size(),
(unsigned char*)encData.data(), pDecrypted, rsa, RSA_PKCS1
_PADDING);
            vector<uint8_t> key;
            if (szDecrypted > 0)
            {
                key.resize(szDecrypted);
                memcpy(key.data(), pDecrypted, szDecrypted);
                SetKeyVal(i, key);
            }
        }
    }
```

```
    delete[] pDecrypted;
    RSA_free(rsa);
    BIO_free(pBio);
}

void RadioCryptoThor::SaveKey(int i)
{
    mkdir(ConvertPathToOs(KEYSTORE_PATH).c_str(), S_IRWXU);
    ofstream ofs(ConvertPathToOs(std::string(KEYSTORE_PATH) +
"/key" + to_string(i+1)), ios::out | ios::binary);

    if (!ofs)
    {
        return;
    }

    BIO* pBIO = nullptr;
    RSA* rsa = createRSA((uint8_t*)defaultDCPublicKey, 1, &pBIO);
    if (rsa)
    {
        unsigned char* pEncrypted = new unsigned
char[RSA_size(rsa)];
        int szEncrypted = RSA_public_encrypt(_key[i].size(),
_key[i].data(), pEncrypted, rsa, RSA_PKCS1_PADDING);
        if (szEncrypted > 0)
        {
            ofs.write((char*)pEncrypted, szEncrypted);
        }
        delete[] pEncrypted;
        RSA_free(rsa);
    }
    BIO_free(pBIO);
}

int SetThorKeyGetter(cliControlBlock_t* pCb, sharedMemoryItem_t *
item, uint8_t** pReturnValue)
{
    if (nullptr != pReturnValue)
    {
        *pReturnValue = (uint8_t*)"";
    }
    return SM_STATUS_OK;
}

int SetThorKey(cliControlBlock_t* pCb, sharedMemoryItem_t* item,
uint8_t* parameterValue, uint8_t** pReturnValue)
{
    int result = SM_STATUS_OK;
    if (nullptr != pReturnValue)
    {
        *pReturnValue = (uint8_t*)"";
    }
```

```
    if (parameterValue && *parameterValue == '\0')
    {
        return SM_STATUS_OK;
    }
    int isValid = 0;
    uint32_t keyIdx;
    vector<uint8_t> key(34,0);
    keyIdx = stringToUIntWithLabelGroup((const char *)
parameterValue, &isValid, LABEL_GROUP_ENCRYPTION_KEYS);

    if (!isValid)
    {
        return SM_PARAMETER_NOT_VALID;
    }
    if (keyIdx>0)
    {
        int len = 0;
        uint32_t numPrams = getNumParameters((char *)
parameterValue);
        if (numPrams == 2)
        {
            len = stringToByteArray((const char*)parameterValue,
key.data(), key.size(), 1, &isValid);

            if (!isValid || ((len != SZ_KEY_AES_128) && (len !=
SZ_KEY_AES_256)))
            {
                len = 0;
                result = SM_PARAMETER_NOT_VALID;
            }
        }
        else if (numPrams > 2)
        {
            len = 0;
            result = SM_PARAMETER_NOT_VALID;
        }

        key.resize(len);
        RadioCryptoThor::Instance()->SetKeyVal(keyIdx - 1, key);
        RadioCryptoThor::Instance()->SaveKey(keyIdx - 1);
    }

    GetThorKey(pCb,item,parameterValue,pReturnValue);
    return result;
}

int SetThorKeyx(cliControlBlock_t* pCb, sharedMemoryItem_t* item,
uint8_t* parameterValue, uint8_t** pReturnValue)
{
    int isValid = 0;
    uint32_t keyIdx;
    vector<uint8_t> key(34, 0);
```

```
    keyIdx = stringToUIntWithLabelGroup((const char *)item->
extra, &isValid, LABEL_GROUP_ENCRYPTION_KEYS);

    if (!isValid)
    {
        return SM_PARAMETER_NOT_VALID;
    }

    if (keyIdx>0)
    {
        int len = 0;

        if (stringCompareI((const char*)parameterValue, "clear")
== 0)
        {
            key.resize(0);
            RadioCryptoThor::Instance()->SetKeyVal(keyIdx - 1,
key);
            RadioCryptoThor::Instance()->SaveKey(keyIdx - 1);
        }
        else
        {
            len = stringToByteArray((const char*)parameterValue,
key.data(), key.size(), 0, &isValid);
            if (isValid && ((len == SZ_KEY_AES_128) || (len ==
SZ_KEY_AES_256)))
            {
                key.resize(len);
                RadioCryptoThor::Instance()->SetKeyVal(keyIdx -
1, key);
                RadioCryptoThor::Instance()->SaveKey(keyIdx - 1);
            }
        }
    }
    return GetThorKeyx(pCb,item,pReturnValue);
}

int GetThorKeyx(cliControlBlock_t* pCb, sharedMemoryItem_t *
item, uint8_t** pReturnValue)
{
    int isValid = 0;
    uint32_t keyIdx;
    std::vector<uint8_t> key;
    static std::string _result = "";

    keyIdx = stringToUIntWithLabelGroup((const char *)item->
extra, &isValid, LABEL_GROUP_ENCRYPTION_KEYS);
    if (!isValid)
    {
        return SM_PARAMETER_NOT_VALID;
    }
```

```
    _result = "Key has not been set.";

    if (keyIdx > 0)
    {
        RadioCryptoThor::Instance()->GetKeyVal(keyIdx - 1, key);
        if (key.size())
        {
            _result = std::to_string(key.size() * 8) + "-bit
key.";
        }
    }

    if (nullptr != pReturnValue)
    {
        *pReturnValue = (uint8_t*)_result.c_str();
    }

    return SM_STATUS_OK;
}

int GetThorKey(cliControlBlock_t* pCb, sharedMemoryItem_t* item,
uint8_t* parameterValue, uint8_t** pReturnValue)
{
    vector<uint8_t> key;
    int isValid = 0;
    uint32_t keyIdx;

    keyIdx = stringToUIntWithLabelGroup((const char *)
parameterValue, &isValid, item->labelGroup);

    if (!isValid)
    {
        return SM_PARAMETER_NOT_VALID;
    }
    if (keyIdx == 0)
    {
    }
    else if (keyIdx>0)
    {
        RadioCryptoThor::Instance()->GetKeyVal(keyIdx - 1, key);
        printString(pCb, "Keys are write only. Position ");
        printS32(pCb, keyIdx);
        if (key.size() == 0)
        {
            printLine(pCb, " is empty. Use setKey to store a
key.");
        }
        else
        {
            printString(pCb, " has a ");
            printS32(pCb, key.size() * 8);
            printLine(pCb, "-bit key.");
```

```
        }
    }

    if (nullptr != pReturnValue)
    {
        *pReturnValue = (uint8_t*)"";
    }

    return SM_STATUS_OK;
}

void RadioCryptoThor::UpdateNonce(uint16_t srcAddr)
{
    *(uint16_t*)&_nonce.extendedSourceAddress[3] =
htons(srcAddr);
    _nonce.frameCounter = htonl(ntohl(_nonce.frameCounter) + 1);
}
```

/*

This license governs use of the accompanying software. If you use the software, you accept this license. If you do not accept the license, do not use the software.

1. Definitions
The terms "reproduce," "reproduction" and "distribution" have the same meaning here as under U.S. copyright law.

"You" means the licensee of the software.

"Your company" means the company you worked for when you downloaded the software.

"Reference use" means use of the software within your company as a reference, in read only form, for the sole purposes of debugging your products, maintaining your products, or enhancing the interoperability of your products with the software, and specifically excludes the right to distribute the software outside of your company.

"Licensed patents" means any Licensor patent claims which read directly on the software as distributed by the Licensor under this license.

2. Grant of Rights
(A) Copyright Grant- Subject to the terms of this license, the Licensor grants you a non-transferable, non-exclusive, worldwide, royalty-free copyright license to reproduce the software for reference use.

(B) Patent Grant- Subject to the terms of this license, the Licensor grants you a non-transferable, non-exclusive, worldwide, royalty-free patent license under licensed patents for reference use.

3. Limitations
(A) No Trademark License- This license does not grant you any rights to use the Licensor's name, logo, or trademarks.

(B) If you begin patent litigation against the Licensor over patents that you think may apply to the software (including a cross-claim or counterclaim in a lawsuit), your license to the software ends automatically.

(C) The software is licensed "as-is." You bear the risk of using it. The Licensor gives no express warranties, guarantees or conditions. You may have additional consumer rights under your local laws which this license cannot change. To the extent

```
permitted under your local laws, the Licensor excludes the
implied warranties of merchantability, fitness for a particular
purpose and non-infringement.
*/


#ifndef _RADIO_CRYPTO_THOR_H_
#define _RADIO_CRYPTO_THOR_H_

#include <stdint.h>
#include <mutex>
#include <vector>
#include <openssl/ssl.h>
#include <openssl/evp.h>

#define NUM_STORED_KEYS 16

class RadioCryptoThor
{
public:
    typedef enum CryptoMode
    {
        AES_CTR,
        AES_CCM
    } CryptoMode;

    static RadioCryptoThor* Instance(void);

    void SetMode(CryptoMode mode);
    CryptoMode GetMode(void);

    void SetActiveKey(int i);
    int GetActiveKey(void);

    void SetKeyVal(int i, const std::vector<uint8_t>& key);
    void GetKeyVal(int i, std::vector<uint8_t>& key);

    void SaveKey(int i);

    bool Encrypt(uint16_t srcAddr, uint8_t* inbuf, uint32_t
inbufSize, std::vector<uint8_t>& outbuf);
    bool Decrypt(uint16_t srcAddr, uint8_t* inbuf, uint32_t
inbufSize, std::vector<uint8_t>& outbuf);

    void UnitTestCCM(void);
    void UnitTestCTR(void);
protected:
    RadioCryptoThor(CryptoMode mode);
    virtual ~RadioCryptoThor();

private:
    void LoadKeys(void);
```

```
    void UpdateNonce(uint16_t srcAddr);

#if defined __GNUC__
#define PACKED_STRUCT __attribute__ ((__packed__))
#else
#define PACKED_STRUCT
#endif

    #define L 2
    typedef struct Nonce
    {
        uint8_t extendedSourceAddress[8];
        uint32_t frameCounter;
        uint8_t securityLevel;
    } PACKED_STRUCT Nonce;

    CryptoMode _mode;
    EVP_CIPHER_CTX *_ctxEncrypt_1_1=NULL;
    EVP_CIPHER_CTX *_ctxDecrypt_1_1=NULL;
    int _activeKey;
    std::vector<uint8_t> _key[NUM_STORED_KEYS];
    std::mutex _mutexEncrypt;
    bool _configured;
    Nonce _nonce;
};

#endif
```

package database

```
import (
        "crypto/aes"
        "crypto/cipher"
        "crypto/rand"
        "crypto/sha256"
        "encoding/hex"
        "fwlog"
        "io"

        "golang.org/x/crypto/pbkdf2"
)

var dbKey []byte

func CreateKey(passphrase string, salt string) []byte {
        key := pbkdf2.Key([]byte(passphrase), []byte(salt), 1000, 32, sha256.New)
        return key
}

func encrypt(plaintextBytes []byte) []byte {
        block, err := aes.NewCipher(dbKey)
        if err != nil {
                fwlog.Debugf("encryption error: %v", err)
        }
        gcm, err := cipher.NewGCM(block)
        if err != nil {
                fwlog.Debugf("encryption error: %v", err)
        }
        nonce := make([]byte, gcm.NonceSize())
        if _, err := io.ReadFull(rand.Reader, nonce); err != nil {
                fwlog.Debugf("encryption error: %v", err)
        }
        cipherBytes := gcm.Seal(nonce, nonce, plaintextBytes, nil)
        return cipherBytes
}

func encode(plainText string) string {
        plainBytes := []byte(plainText)
        cipherBytes := encrypt(plainBytes)
        cipherEncoded := hex.EncodeToString(cipherBytes)
        return cipherEncoded
}

func decrypt(nonceAndCipherBytes []byte) []byte {
        block, err := aes.NewCipher(dbKey)
        if err != nil {
                fwlog.Debugf("decryption error: %v", err)
        }
        gcm, err := cipher.NewGCM(block)
        if err != nil {
```

```
                fwlog.Debugf("decryption error: %v", err)
        }
        nonceSize := gcm.NonceSize()
        nonce := nonceAndCipherBytes[:nonceSize]
        cipherBytes := nonceAndCipherBytes[nonceSize:]
        plainBytes, err := gcm.Open(nil, nonce, cipherBytes, nil)
        if err != nil {
                fwlog.Debugf("decryption error: %v", err)
        }
        return plainBytes
}

func decode(encodedText string) string {
        encodedBytes, err := hex.DecodeString(encodedText)
        if err != nil {
                fwlog.Debugf("decoding error: %v", err)
        }
        plainBytes := decrypt(encodedBytes)
        plainText := string(plainBytes)
        return plainText
}
```