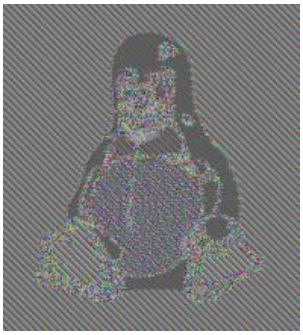
05_AES_CBC.md

AES-CBC

Der AES-ECB Modus ist nachgewiesenerweise unsicher, es wird Empfohlen einen anderen Blockmodus zu verwenden. Das Problem mit AES-ECB ist, dass mit demselben Key und derselben Eingabe, immer der gleiche Ciphertext berechnet wird.

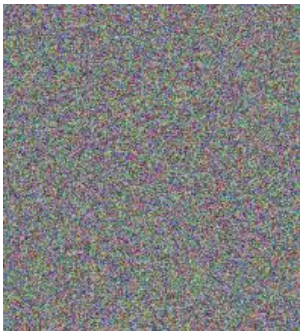
Beispiel:

Das Bild mit AES-ECB verschlüsselt sieht nachher so aus:



In diesem Tutorial werden wir und anschauen, wie man in RIOT Daten mit AES-CBC verschlüsselt. AES-CBC nimmt neben dem Klartext und dem Schlüssel noch einen sogenannten Initialisierungsvektor (IV) als dritten Parameter an. Bevor der erste Block verschlüsselt wird, wird der Block mit dem IV XORed, für alle anderen Blöcke wird der letzte Ciphertext Block als IV verwendet.

Das selbe Bild mit AES-CBC verschlüsselt, sieht nachher so aus:



Anpassung der Makefile

Wir fügen die Module `cipher_modes` sowie `random` hinzu:

```
# Name der Anwendung
APPLICATION = aes_cbc_example

# Standardboard
BOARD ?= native

USEMODULE += shell_commands  # RIOT Shell Commands
USEMODULE += shell           # RIOT Shell Modul
USEMODULE += crypto_aes      # Verschlüsselung mithilfe von AES
USEMODULE += od              # Object Dump
USEMODULE += od_string       # Object Dump String representation
+  USEMODULE += cipher_modes # Um AES-CBC zu verwenden
+  USEMODULE += random       # Um einen zufälligen IV zu generieren

# Pfad zur RIOT installation
RIOTBASE ?= ${HOME}/RIOT

include $(RIOTBASE)/Makefile.include
```

Der `crypto/modes/cbc.h` Header

Um in unserem Program Daten mit AES-CBC zu verschlüsseln müssen wir den Header `crypto/modes/cbc.h` includen.

Der Header definiert noch die Funktionen `cipher_encrypt_cbc` und `cipher_decrypt_cbc`.

`cipher_encrypt_cbc`

```
/**
 * @brief Encrypt data of arbitrary length in cipher block chaining mode.
 *
 * @param cipher     Already initialized cipher struct
 * @param iv         16 octet initialization vector. Must never be used more
 *                   than once for a given key.
 * @param input      pointer to input data to encrypt
 * @param input_len  length of the input data
 * @param output     pointer to allocated memory for encrypted data. It has to
 *                   be of size data_len + BLOCK_SIZE - data_len % BLOCK_SIZE.
 *
 * @return            <0 on error
 * @return            CIPHER_ERR_INVALID_LENGTH when input_len % BLOCK_SIZE != 0
 * @return            CIPHER_ERR_ENC_FAILED on internal encryption error
 * @return            otherwise number of input bytes that aren't consumed
 */
int cipher_encrypt_cbc(const cipher_t *cipher, uint8_t iv[16], const uint8_t *input,
                       size_t input_len, uint8_t *output);
```

Die `cipher_encrypt_cbc` ist ähnlich wie die `cipher_encrypt` Funktion aus dem vorherigen Kapitel. Es gibt ein paar Unterschiede:

- Es wird der Initialisierungsvektor (IV) als Parameter mitgegeben
- Der Klartext darf nun größer als ein AES Block (16 Bytes) sein. Wenn die Eingabe größer als 16 Bytes ist, wird ge-chained.
- Die Länge des Klartext muss mitgegeben werden und muss ein vielfaches der AES Blockgröße (16) sein.
- Der Ausgabepointer muss groß genug sein um den gesamten verschlüsselten Text zu speichern. Die Größe dieses Buffer lässt sich einfach durch die angegebene Formel `data_len + BLOCK_SIZE - data_len % BLOCK_SIZE` berechnen
- Als Rückgabewert wird bei erfolgreichem Verschlüsseln nicht mehr `1` zurück, sondern die Menge an Bytes, die aus dem Input nicht verarbeitet wurden.

`cipher_decrypt_cbc`

```
/**
 * @brief Decrypt encrypted data in cipher block chaining mode.
 *
 * @param cipher     Already initialized cipher struct
 * @param iv         16 octet initialization vector.
 * @param input      pointer to input data to decrypt
 * @param input_len  length of the input data
 * @param output     pointer to allocated memory for plaintext data. It has to
 *                   be of size input_len.
 *
 * @return            <0 on error
 * @return            CIPHER_ERR_INVALID_LENGTH when input_len % BLOCK_SIZE != 0
 * @return            CIPHER_ERR_DEC_FAILED on internal decryption error
 * @return            otherwise number of bytes decrypted
 */
int cipher_decrypt_cbc(const cipher_t *cipher, uint8_t iv[16], const uint8_t *input,
                       size_t input_len, uint8_t *output);
```

Die `cipher_decrypt_cbc` Funktion verhält sich ebensfalls wie die `cipher_decrypt` Funktion aus dem vorherigen Kapitel. Der einzige Unterschied zu `cipher_encrypt_cbc` ist, dass der Input nicht verschlüsselt, sondern entschlüsselt wird. Der Rückgabewert ist die Anzahl an entschlüsselten Bytes.

Program zum Verschlüsseln von beliebig langen Nachrichten

```
#include <stdio.h>

#include "crypto/ciphers.h"
#include "crypto/aes.h"
#include "crypto/modes/cbc.h"

#include "od.h"

#include "random.h"

static const uint8_t key[] = {
        0x64, 0x52, 0x67, 0x55,
        0x6B, 0x58, 0x70, 0x32,
        0x73, 0x35, 0x75, 0x38,
        0x78, 0x2F, 0x41, 0x3F};

int main(void)
{
    /*
     * Die Nachricht enthält 6 mal alle Hexadezimalen Ziffern hintereinander aufgereiht.
     * Dadurch kann man im Ciphertext gut erkennen, dass derselbe Klartextblock mit CBC verschlüsselt unterschiedliche Ergebnisse
     */
    const char* message = "0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF";

    /* ======== Initialisierung des Ciphers ======== */

    cipher_t cipher;
    int err;

    if ((err = cipher_init(&cipher, CIPHER_AES_128, key, AES_KEY_SIZE)) != CIPHER_INIT_SUCCESS) {
        printf("Failed to init cipher: %d\n", err);
        return err;
    }

    /* ======== Berechnen der Längen ======== */

    int data_len = strlen(message) + 1;
    int n_required_blocks = data_len / AES_BLOCK_SIZE;
```

```
    if (data_len % AES_BLOCK_SIZE) {
        n_required_blocks++;
    }

    size_t total_len = n_required_blocks * AES_BLOCK_SIZE;

    /* ======== Erstellen der Buffer ======== */

    uint8_t* input = calloc(n_required_blocks, AES_BLOCK_SIZE);
    uint8_t* output = calloc(n_required_blocks, AES_BLOCK_SIZE);
    uint8_t* decrypted = calloc(n_required_blocks, AES_BLOCK_SIZE);

    memcpy(input, message, data_len);

    /* ======== Erstellen des IV  ======== */

    uint8_t iv[16] = {0};
    random_bytes(iv, 16); // WICHTIG: In Produktionscode, einen kryptographisch sicheren RNG nehmen!

    /* ======== Verschlüsseln und Entschlüsseln  ======== */

    if ((err = cipher_encrypt_cbc(&cipher, iv, input, total_len, output)) < 0) {
        printf("Failed to encrypt data: %d\n", err);
        return err;
    }

        if ((err = cipher_decrypt_cbc(&cipher, iv, output, total_len, decrypted)) < 0) {
        printf("Failed to decrypt data: %d\n", err);
        return err;
    }

    /* ======== Ausgabe ======== */

    printf("IV: ");
    od_hex_dump(iv, 16, 0);
    printf("\n\n");

    printf("Plaintext:\n");
    od_hex_dump(input, total_len, AES_BLOCK_SIZE);
    printf("\n\n");

    printf("Ciphertext:\n");
    od_hex_dump(output, total_len, AES_BLOCK_SIZE);
    printf("\n\n");

    printf("Decrypted Ciphertext:\n");
    od_hex_dump(input, total_len, AES_BLOCK_SIZE);
    printf("\n\n");

    /* ======== Aufräumen ======== */

    free(input);
    free(output);
    free(decrypted);

    return 0;
}
```

Ausgabe

```
IV: 00000000  13  E9  4A  77  4D  D2  F9  59  C7  21  D0  F5  24  25  1F  10  ..JwM..Y.!..$%..


Plaintext:
```

```
00000000  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000010  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000020  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000030  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000040  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000050  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000060  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  ................


Ciphertext:
00000000  AF  71  EA  E1  6D  87  EA  38  CE  14  BC  39  4E  21  88  26  .q..m..8...9N!.&
00000010  5E  E2  28  B8  8A  9B  30  2B  53  2B  11  AD  2C  8A  1F  83  ^.(...0+S+..,...
00000020  FA  86  20  33  D4  48  12  B6  D6  3E  DA  11  C7  85  C9  A8  .. 3.H...>......
00000030  8C  CE  15  F1  7A  53  B7  F8  61  5C  7C  F8  36  70  63  E3  ....zS..a\|.6pc.
00000040  20  2D  1A  A6  3F  AE  0B  B3  9B  0F  55  B9  9C  C6  29  C4   -..?.....U...).
00000050  72  83  BF  72  CE  8E  F9  E1  19  21  44  B9  E5  80  91  ED  r..r.....!D.....
00000060  F5  0C  4F  6E  13  3D  B6  E0  F5  C0  50  25  63  22  8A  AE  ..On.=....P%c"..


Decrypted Ciphertext:
00000000  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000010  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000020  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000030  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000040  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000050  30  31  32  33  34  35  36  37  38  39  41  42  43  44  45  46  0123456789ABCDEF
00000060  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  ................
```

Der IV und Ciphertext wird bei jedem mal ausführen anders aussehen

Anderes Beispielprogramm

Hier ist ein Beispielprogramm, welches einen Command beinhält, dass eine im Command angegebene Nachricht mit AES-CBC verschlüsselt:

Syntax: `encrypt <message>`

Um eine Nachricht mit Leerzeichen zu verschlüsseln einfach den gesamten Text in Anführungszeichen setzen.

z.B: `encrypt "Meine Nachricht"`

Zurück zum Index

Zurück zu Teil 4: AES im Electronic Codebook (ECB) Modus

Weiter zu Teil 6: Exkurs UDP