07_Relic.md

Relic

Neben AES-ECB und AES-CBC kann man unter RIOT auch das RELIC Toolkit verwenden, um verschiedene Kryptographische Algorithmen anzuwenden. In diesem Tutorial werden wir und auf RSA konzentrieren.

Makefile

```
# name of your application
APPLICATION = relic-cpp

# If no BOARD is found in the environment, use this default:
BOARD ?= native

# This has to be the absolute path to the RIOT base directory:
RIOTBASE ?= ${CURDIR}/../.base

+  export RELIC_CONFIG_FLAGS=-DARCH=NONE -DALLOC=DYNAMIC -DQUIET=off -DWORD=32 -DFP_PRIME=255 -DWITH="BN;MD;DV;FP;EP;CP;BC;EC" -D
+
+  USEPKG += relic

USEMODULE += od
USEMODULE += od_string
USEMODULE += shell
USEMODULE += shell_commands

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

Mit der `export RELIC_CONFIG_FLAGS=...` Zeile geben wir Optionen zur Verwendung von RELIC an. Mithilfe der `USEPKG += relic` Anweisung fügen wir RELIC zu unserem Projekt hinzu.

Der `rsa_t` Typ

Wenn wir mithilfe von RELIC RSA Encryption verwenden wollen, müssen wir mit dem `rsa_t` Typ arbeiten. Da wir in unserer Makefile (spezifisch in den `RELIC_CONFIG_FLAGS`) `ALLOC` auf `DYNAMIC` gesetzt haben, ist `rsa_t` definiert als ein Pointer zu einer Struktur Namens `relic_rsa_st` . Diese Struktur enthält mehrere Pointer zu sog. Multiple Precision Integers (In RELIC: `bn_t`) welches die Parameter der RSA Schlüssel sind. Unter anderem die Primzahlen p und q, deren Produkt n, usw.

RELIC verwendet mehrere Makros um mit diesem Datentyp zu arbeiten. Z.B. wird der Makrobefehl `rsa_null` verwendet, um die Pointer auf `NULL` zu setzen.

Der `rsa_new` Makrobefehl wird verwendet, um den notwendigen Speicher für die RSA Schlüssel zu allocaten und diesen zu Initialisieren.

`cp_rsa_gen` ist ein Makrobefehl, der verwendet wird, um ein RSA Schlüsselpaar zu generieren. Der Makrobefehl nimmt drei Parameter an: Den öffentlichen Schlüssel, den privaten Schlüssel und die Menge an Bits. Als Rückgabewert liefert der Makrobefehl ein int, welches `STS_OK` bei erfolgreicher Generation ist.

Der `rsa_free` Makrobefehl wird verwendet, um den durch die RSA Schlüssel verwendeten Speicher wieder freizugeben.

`cp_rsa_enc`

```
/**
 * Encrypts using the RSA cryptosystem.
 *
 * @param[out] out          - the output buffer.
 * @param[in, out] out_len  - the buffer capacity and number of bytes written.
 * @param[in] in            - the input buffer.
 * @param[in] in_len        - the number of bytes to encrypt.
 * @param[in] pub           - the public key.
 * @return STS_OK if no errors occurred, STS_ERR otherwise.
 */
int cp_rsa_enc(uint8_t *out, int *out_len, uint8_t *in, int in_len, rsa_t pub);
```

Die Funktion `cp_rsa_enc` wird verwendet, um Daten mit RSA zu verschlüsseln. Die Funktion nimmt 5 Parameter an: Den Ausgabebuffer, einen Pointer zur Länge des Ausgabebuffers, den Eingabebuffer, die Größe des Eingabebuffers und den öffentlichen Schlüssel.

Als Rückgabewert liefert die Funktion bei erfolgreichen Verschlüsseln `STS_OK`. Bei Fehlern gibt die Funktion `STS_ERR` zurück.

Die Länge des Ausgabebuffers wird als Pointer übergeben, da die Funktion denselben Pointer nutzt um dort die Menge an in den Ausgabebuffer geschriebenen Bytes zu speichern.

Verwendung von RELIC im Programm

```
#include "relic.h"
#include "od.h"

int main(void)
{
    core_init(); // WICHTIG!: Initialisierung der RELIC Library

    /* ============ Initialisierung der RSA Schlüssel ============ */

    rsa_t pub, priv; // Definieren der Pointer zu den RSA Schlüsseln (Wegen ALLOC = DYNAMIC ist rsa_t ein Pointer)

    // Setze die Pointer auf NULL
    rsa_null(pub);
    rsa_null(priv);

    // Initialisiere die Pointer. WICHTIG!: rsa_new ruft calloc auf!
    rsa_new(pub);
    rsa_new(priv);

    /* ============ Generierung der RSA Schlüssel ============ */

    int err;

    if ((err = cp_rsa_gen(pub, priv, RELIC_BN_BITS)) != STS_OK) {
        printf("Failed to generate RSA Key Pair: %d\n", err);

        // Gebe den Speicher wieder frei
        rsa_free(pub);
        rsa_free(priv);

        return err;
    }

    /* ============ Verschlüsseln einer einfachen Nachricht ============ */

    const char* data = "Hallo!";
    uint8_t out[RELIC_BN_BITS / 8 + 1]; // RELIC_BN_BITS gibt die sowohl Präzision in Bits der Multiple Precision Integers, als a

    // Der out_len Parameter der cp_rsa_enc ist ein int Pointer, von ihm wird die Größe des Output-buffers gelesen,
    // die Funktion schreibt dort auch die Menge an geschriebenen Bytes hin
```

```
    int out_len = RELIC_BN_BITS / 8 + 1;

    if ((err = cp_rsa_enc(out, &out_len, (uint8_t*) data, strlen(data), pub)) != STS_OK) {
        printf("Failed to encrypt with RSA: %d\n", err);

        // Gebe den Speicher wieder frei
        rsa_free(pub);
        rsa_free(priv);

        return err;
    }

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

    printf("RSA Encrypted Text:\n");
    od_hex_dump(out, out_len, 0);

    /* ============ Aufräumen ============ */

    rsa_free(pub);
    rsa_free(priv);

    core_clean();

    return 0;
}
```

Beispielprogramm

Hier ist ein C++ Beispielprogramm, welches ein zufälliges Schlüsselpaar generiert, diese dann im PEM Format ausgibt und einen Command bereitstellt, Daten mit diesen Schlüsseln zu verschlüsseln.

Befehl: `rsa <message>`

Als Ausgabe des Befehls gibt es ein oder mehrere (auch im PEM Format) verschlüsselte Blöcke.

Das in dem Verzeichnis enthaltene Shell Skript `decrypt_test.sh` kann in einer Linux Maschine (mit openssl installiert) verwendet werden, um die Ausgabe des Befehls zu entschlüsseln. Dafür muss zuerst der Private Schlüssel aus der Konsolenausgabe in einer Datei Namens `key.private` abgespeichert werden.

Beispiel

In RIOT:

- *Ausgabe -----BEGIN RSA PRIVATE KEY----- bis -----END RSA PRIVATE KEY-----* kopieren und in Datei `key.private abspeichern`
- Befehl ausführen: `rsa <Nachricht>`
- Ausgabe des Befehls kopieren und Newlines entfernen

In Linux:

- Befehl ausführen `./decrypt_test.sh <Base64 von Ausgabe>`

Zurück zum Index

Zurück zu Teil 6: Exkurs UDP

Weiter zu Kapitel 3: Benchmarking und Ergebnisse