
Aktuelle Themen der IT-Sicherheit: RIOT Challenges

WS 22/23: Prof. Dr. Jan Seedorf

Thomas Jakkel (1001594), Severin Nonenmann (1001599),

Lukas Reinke (1001213), Lars Weiß (1001596)

29. Januar 2023



Inhalt

1	Woche 1	2
1.1	Challenge 1: VM installation	2
1.2	Challenge 2: Erste schritte mit RIOT	2
1.2.1	First_test Application	2
1.2.2	Simple Network communication	3
2	Woche 2	5
2.1	Challenge 1	5
2.1.1	Task 01	5
2.1.2	Task 02	5
2.1.3	Task 03	5
2.1.4	Task 04	6
2.2	Challenge 2	6
2.3	Challenge 3	7
2.3.1	Review AES-CBC	7
2.3.2	Additional Block Cipher Mode: CTR	7
2.4	Challenge 4	10
2.4.1	4.2.: Understanding existing benchmarking code	10

1 Woche 1

1.1 Challenge 1: VM installation

Das VM setup wird hier nicht genauer beschrieben.

1.2 Challenge 2: Erste schritte mit RIOT

Die Anleitung zum Setup von RIOT OS aus den RIOT Tutorials wurde durchlaufen und ein funktionierender Workspace erstellt.

Wir haben das Setup insofern verändert, dass unser Code in einem separaten Ordner neben dem von GitHub geklonten RIOT Dateien liegt.

1.2.1 First_test Application

Das Ziel ist ein erstes RIOT-OS selber zu kompilieren, mit einer eigen Funktion zu versehen und zu starten.

Im default Makefile müssen zwei Änderungen vorgenommen werden:

1. In der Variable `APPLICATION` der Name der Ausführbaren binary zu setzen
2. Die `RIOTBASE`, dem Pfad zu den Hauptdateien des RIOT-OS, zu setzen.

```
1 ...  
2 APPLICATION = First_test  
3  
4 BOARD ?= native  
5  
6 RIOTBASE ?= $(CURDIR)/../../RIOT/  
7 ...
```

Es soll eine Shell Kommando geschrieben werden das bei Aufruf einen String zurück gibt. Zugrunde liegt eine einfache C Funktion mit einem `printf()` statement:

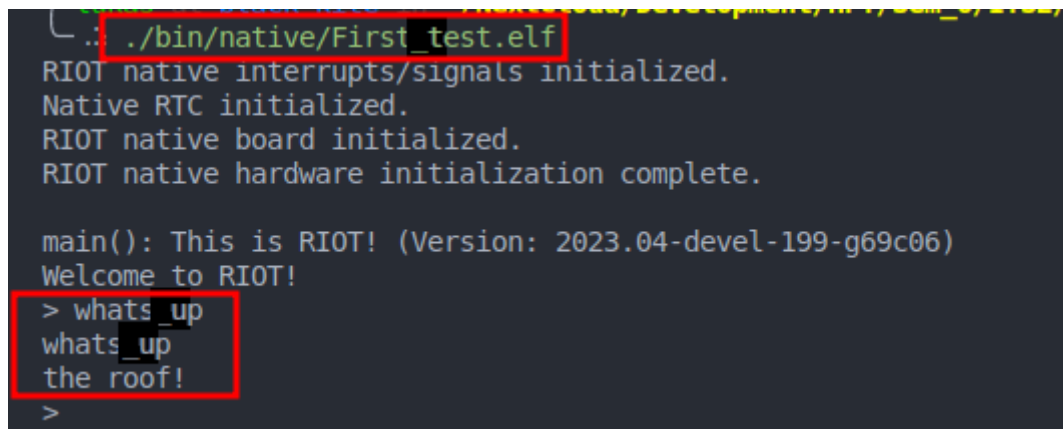
```
1 static int whats_up(int argc, char **argv) {  
2     (void)argc;  
3     (void)argv;  
4  
5     printf("The roof!\n");  
6     return 0;  
7 }
```

Des weiteren muss die Funktion in einem Array eingetragen und dieses Array als Quelle für Shell-befehle in der `main` Funktion registriert werden.

```
1 const shell_command_t shell_commands[] = {  
2     {"whats_up", "prints the roof", whats_up},  
3     { NULL, NULL, NULL}  
4 };
```

```
1 shell_run(shell_commands, line_buf, SHELL_DEFAULT_BUFSIZE);
```

Nun kann mithilfe des `make`-Kommandos ein build gestartet werden und die resultierende Binary mit dem Namen **First_test.elf** ausgeführt werden. In der RIOT Shell kann nun der Befehl `whats_up` ausgeführt werden.



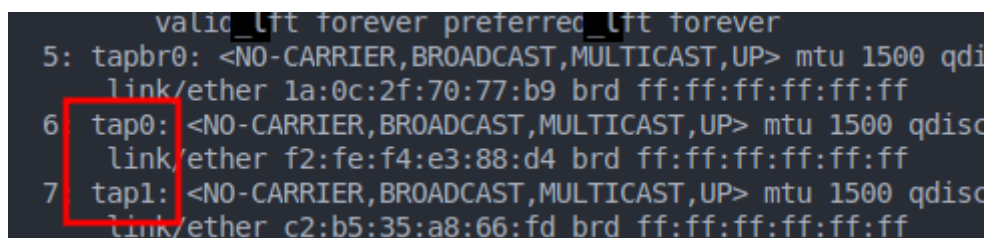
```
./bin/native/First_test.elf  
RIOT native interrupts/signals initialized.  
Native RTC initialized.  
RIOT native board initialized.  
RIOT native hardware initialization complete.  
  
main(): This is RIOT! (Version: 2023.04-devel-199-g69c06)  
Welcome to RIOT!  
> whats_up  
whats_up  
the roof!  
>
```

Abbildung 1: `whats_up` Befehl in RIOT shell

1.2.2 Simple Network communication

Das Ziel dieser Challenge war, zwei RIOT-OS Instanzen über Netzwerk kommunizieren zu lassen.

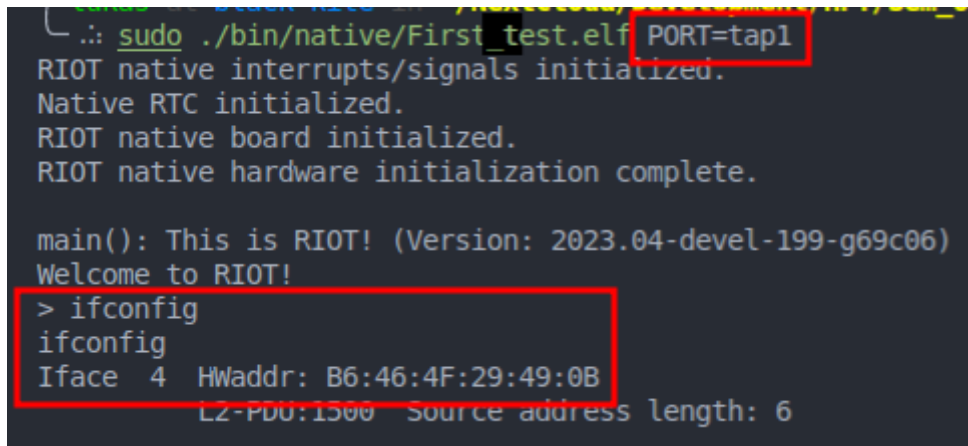
Das im RIOT Repo mitgelieferte Script `tapsetup` kann genutzt werden um in der Linux Umgebung zwei interfaces (`tap0` und `tap1`) anzulegen.



```
valid_lft forever preferred_lft forever  
5: tapbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc  
   link/ether 1a:0c:2f:70:77:b9 brd ff:ff:ff:ff:ff:ff  
6: tap0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc  
   link/ether f2:fe:f4:e3:88:d4 brd ff:ff:ff:ff:ff:ff  
7: tap1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc  
   link/ether c2:b5:35:a8:66:fd brd ff:ff:ff:ff:ff:ff
```

Abbildung 2: Zwei virtuelle Interfaces mit verbindender “bridge”

Wird nun eine RIOT-OS Instanz mit dem Zusatz `PORT=tap0` ist das Interface tap0 Verbunden und kann intern mit dem Befehl `ifconfig` gefunden werden. Neben dem Interface wird die Hardware-Adresse auf der das Interface später angesprochen werden kann angezeigt.



```
./bin/native/First_test.elf PORT=tap1
RIOT native interrupts/signals initialized.
Native RTC initialized.
RIOT native board initialized.
RIOT native hardware initialization complete.

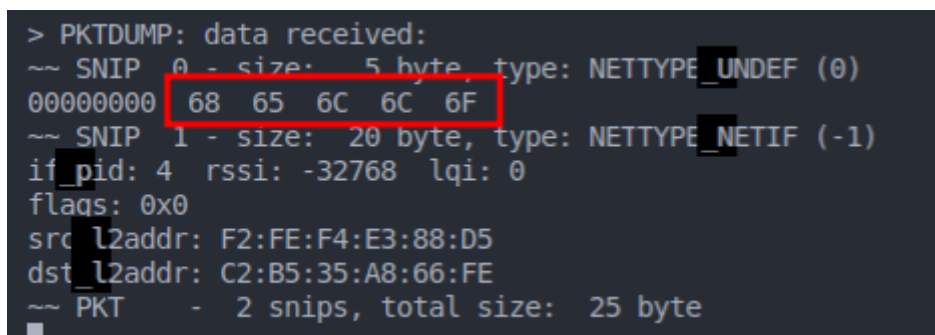
main(): This is RIOT! (Version: 2023.04-devel-199-g69c06)
Welcome to RIOT!
> ifconfig
ifconfig
Iface 4 HWaddr: B6:46:4F:29:49:0B
L2-PDU:1500 Source address length: 6
```

Abbildung 3: tap1 in RIOT-OS

Nun kann mit Hilfe des Befehls `txtsnd 4 C2:B5:35:A8:66:FE hello` eine Nachricht an ein anderes Interface gesendet werden. Der Befehl beinhaltet:

1. Die Interface Nummer auf der gesendet werden soll
2. Die Hardware Adresse des Ziels
3. Die Nachricht

Auf dem zweiten Instanz kann die gesendete Nachricht nun in Hexadezimaler Form empfangen werden.



```
> PKTDUMP: data received:
~~ SNIP 0 - size: 5 byte, type: NETTYPE_UNDEF (0)
00000000 68 65 6C 6C 6F
~~ SNIP 1 - size: 20 byte, type: NETTYPE_NETIF (-1)
if_pid: 4 rssi: -32768 lqi: 0
flags: 0x0
src_l2addr: F2:FE:F4:E3:88:D5
dst_l2addr: C2:B5:35:A8:66:FE
~~ PKT - 2 snips, total size: 25 byte
```

Abbildung 4: Empfangene Nachricht

2 Woche 2

2.1 Challenge 1

Das Ziel ist die ersten vier Tutorials von RIOT durchzuarbeiten

2.1.1 Task 01

Einfügen der Code-Zeile `printf("This application runs on %s\n", RIOT_BOARD);` gibt den Hardware Typ aus für den RIOT Kompiliert wurde.

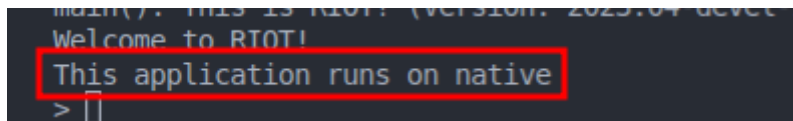
A terminal window with a dark background. The text 'main(): THIS IS RIOT! (VERSION: 2023.04-devel-155-g03c00)' is at the top. Below it is 'Welcome to RIOT!'. The line 'This application runs on native' is highlighted with a red rectangular box. The prompt '>' is visible at the bottom.

Abbildung 5: Ausgabe der neuen codezeile

2.1.2 Task 02

Die Funktion `echo()` aus dem Tutorial Code muss um die Zeile `printf("%s", argv[1]);` erweitert werden, um das erste Argument der Funktion auszugeben.

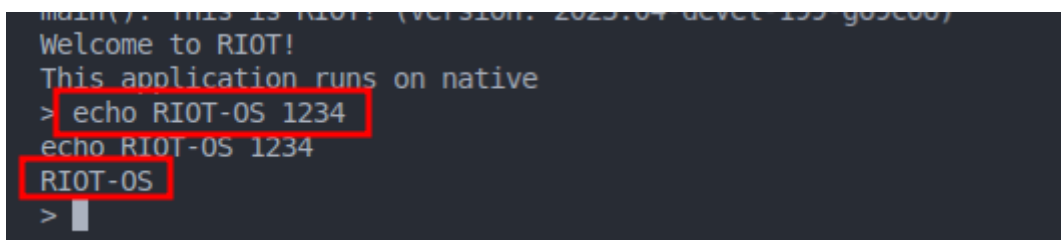
A terminal window with a dark background. The text 'main(): THIS IS RIOT! (VERSION: 2023.04-devel-155-g03c00)' is at the top. Below it is 'Welcome to RIOT!'. The line 'This application runs on native' is highlighted with a red rectangular box. Below that is the command '> echo RIOT-OS 1234'. The output 'echo RIOT-OS 1234' is shown. The line 'RIOT-OS' is highlighted with a red rectangular box. The prompt '>' is visible at the bottom.

Abbildung 6: Ausgabe des Ersten Arguments

2.1.3 Task 03

Zum erstellen eines Threads muss die Multithreading library importiert und ein Array als Stack für den Thread erstellt werden.

```
1 ...  
2 #include "thread.h"  
3  
4 char rcv_thread_stack[THREAD_STACKSIZE_MAIN];
```

5 ...

Jetzt kann mit `thread_create()` ein neuer Thread erstellt werden. Die Parameter enthalten eine Referenz auf den Stack und die Funktion die aufgerufen werden soll, diese wurde im Tutorial Code vorgegeben.

```
main(): This is RIOT! (Version: 2023.04-devel-199-g69c06)
This is Task-03
I'm in "thread" now
> []
```

Abbildung 7: Ausgabe des Threads

2.1.4 Task 04

Um Timer in RIOT-OS Nutzen zu Können muss im Makefile das Modul **xtimer** importiert werden:
`USEMODULE += xtimer`

In einem Thread wird nun mit `xtimer_now_usec()` die aktuelle Systemzeit in Millisekunden Ausgegeben und dann mit `xtimer_sleep(2)` zwei Sekunden geschlafen.

```
1  ...
2  void *system_time(void *arg)
3  {
4      int time = 0;
5      while(true){
6          time = xtimer_now_usec();
7          printf("%d \n",time);
8          xtimer_sleep(2);
9      }
10
11     (void)arg;
12     return NULL;
13 }
14 ...
```

2.2 Challenge 2

Lorem Ipsum

2.3 Challenge 3

2.3.1 Review AES-CBC

Unter Chapter_2_Crypto befindet sich die Dokumentation zu 05_AES_CBC_en.md. Zu Beginn werden die Module cipher_modes (Für die verschiedenen AES cipher modes) und random (zur Generierung der IV) zum makefile hinzugefügt. Anschließend werden die zwei header Dateien für die Verschlüsselungs- und Entschlüsselungsmethoden vom CBC Mode hinzugefügt.

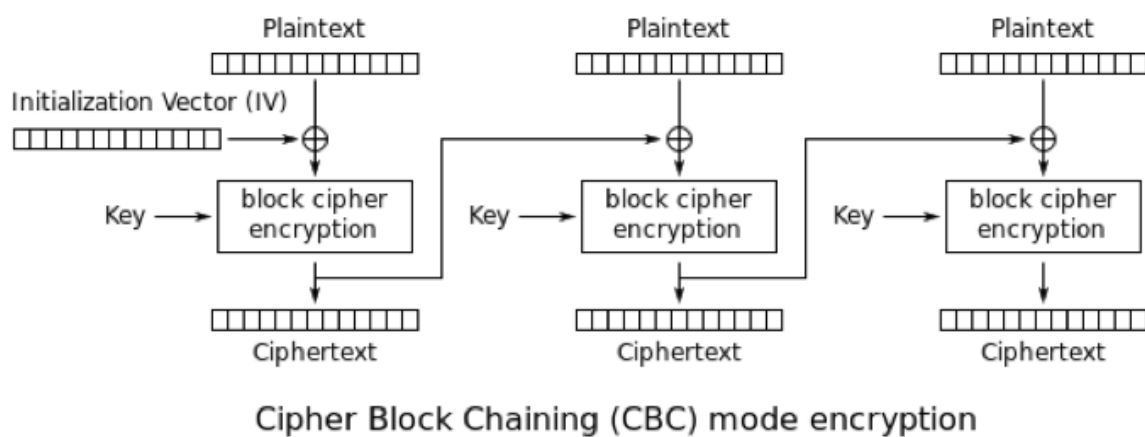


Abbildung 8: AES CBC Mode

Für die Generierung eines cipher textes c1 Im CBC Mode wird der Plaintext mit der IV xor verknüpft und anschließend mit dem key verschlüsselt. Die IV ist dabei zufällig. Für den nächsten cipher text c2, dient c1 als IV.

Im main code wird nun zunächst ein Schlüssel, die message (plaintext) und cipher initialisiert. Anschließend folgen buffer Initialisierungen für input, output und decrypt.

Die IV ist eine zufällige, 16 byte große Zahl. Hier ist ein Verweis, dass man im productive mode einen kryptographischen Random Number Generator benutzen sollte, da die IV für jeden cipher text einzigartig sein muss.

Nun werden die encrypt_cbc und decrypt_cbc Methoden aufgerufen.

2.3.2 Additional Block Cipher Mode: CTR

In diesem Abschnitt wird der Code der obigen Review (AES CBC) zu AES_CTR abgeändert.

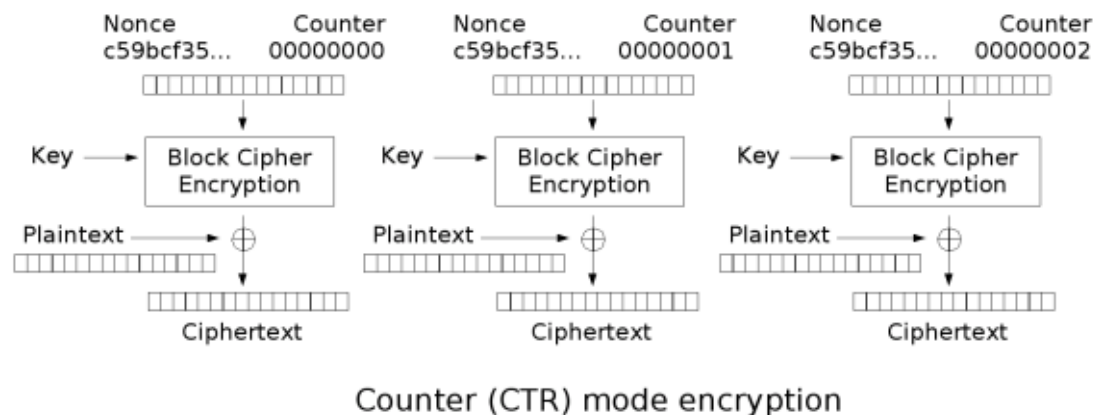


Abbildung 9: AES CTR Mode

Dafür definieren wir die folgenden Begriffe: - Nonce: Ist eine unique number. Diese wird oft Synonym mit der IV aus den anderen AES modes benutzt. - Counter: Eine weitere unique number, welche in nachfolgenden Runden mit einer Inkrementier-Methode inkrementiert wird. - Counter Block: Ist die nonce konkatiert mit dem counter.

Der Counter Block ist wie die IV aus dem AES-CBC 16 byte groß, also 128 bit. Dieser muss in zwei Teile geteilt werden für die Nonce + Counter. Der Counter darf dabei nicht zu klein gewählt werden, da je nach Inkrementierungs-Methode ein Overflow entstehen kann. In NIST-SP800-38A Appendix B.2 ist die zweite Herangehensweise zur Initialisierung des Counter Blocks, dass die Nonce und der Counter genau die Hälfte des Counter Blocks einnehmen. Deswegen wählen wir eine 64 bit Nonce und einen 64 bit Counter. Diese werden wie in der Review des AES-CBC mit einer random number initialisiert, wobei in der Praxis ein kryptographischer Random Number Generator benutzt werden sollte.

Der Counterblock wird zur Erzeugung des Keystreams (Counter Block mit Key verschlüsseln) verwendet, der dann mit dem Plaintext XOR-verknüpft wird, um den Ciphertext zu erzeugen. Also anders wie im CBC Mode, wo erst eine XOR-Verknüpfung des Plaintext mit der IV und anschließend eine Verschlüsselung stattfindet, wird bei CTR der Counter Block verschlüsselt und anschließend eine xor Verknüpfung mit dem Plaintext durchgeführt.

Für die Implementierung werden die Header Dateien mit den cipher_encrypt_ctr und cipher_decrypt_ctr hinzugefügt:

```
1 encrypt_ctr
2 int cipher_encrypt_ctr(const cipher_t *cipher, uint8_t counter_block
   [16], const uint8_t *input, size_t input_len, uint8_t *output);
3
4 //decrypt_ctr
5 int cipher_decrypt_ctr(const cipher_t *cipher, uint8_t counter_block,
```

```
const uint8_t *input, size_t input_len, uint8_t *output);
```

Entsprechend werden folgende Zeilen des CBC-Codes für die main abgeändert:

- Anstatt von `Create IV`

```
1 /* ===== Create IV ===== */
2
3 uint8_t iv[16] = {0};
4 random_bytes(iv, 16); // IMPORTANT: In productive environment, use
   a cryptographically secure RNG!
```

wird ein Counter Block initialisiert: “`uint8_t nonce[8] = {0}; random_bytes(nonce, 8); // IMPORTANT: In productive environment, use a cryptographically secure RNG!`”

`uint8_t counter[8] = 1; // IMPORTANT: In productive environment, use a cryptographically secure RNG!`

`//Concat uint8_t* counter_block[16] = {0}; memcpy(counter_block, nonce, 8); memcpy(counter_block + 8, counter, 8);`“

- Nun müssen die richtigen Methoden aufgerufen werden und im Output wird zusätzlich der Counter Block mit ausgegeben.

```
1 /* ===== Encryption and Decryption ===== */
2
3 if ((err = cipher_encrypt_ctr(&cipher, counter_block, input,
   total_len, output)) < 0) {
4     printf("Failed to encrypt data: %d\n", err);
5     return err;
6 }
7
8 if ((err = cipher_decrypt_ctr(&cipher, counter_block, output,
   total_len, decrypted)) < 0) {
9     printf("Failed to decrypt data: %d\n", err);
10    return err;
11 }
12
13 /* ===== Output ===== */
14
15 printf("Counter Block: ");
16 od_hex_dump(counter_block, 16, 0);
17 printf("\n\n");
18
19 printf("Plaintext:\n");
20 od_hex_dump(input, total_len, AES_BLOCK_SIZE);
21 printf("\n\n");
22
23 printf("Ciphertext:\n");
24 od_hex_dump(output, total_len, AES_BLOCK_SIZE);
```

```
25 printf("\n\n");
26
27 printf("Decrypted Ciphertext:\n");
28 od_hex_dump(input, total_len, AES_BLOCK_SIZE);
29 printf("\n\n");
```

2.4 Challenge 4

2.4.1 4.2.: Understanding existing benchmarking code

Die Funktion “executeAesCbc()” in aes-cbc.c nimmt drei Parameter entgegen: numberOfRounds, key-Size und messageLength. Sie führt dann AES-CBC für die angegebene Anzahl von Runden wie folgt aus:

- Jede Runde generiert die Funktion eine zufällige Plaintext-Nachricht. Dann erzeugt sie einen zufälligen Initialisierungsvektor (IV). Danach wird das AES-Chiffrierobjekt mit einem zufällig erzeugten Chiffrierschlüssel initialisiert.
- Der Verschlüsselungsprozess wird mit der Funktion cipher_encrypt_cbc durchgeführt, die den IV, den Plaintext und den outputBuffer als Eingabe erhält.
- Der Entschlüsselungsprozess wird mit cipher_decrypt_cbc durchgeführt und erhält wie eben IV, Output der encrypt-Funktion und den decryptBuffer
- Zum Ende werden jeweils die Rundenzeiten für encrypt und decrypt ausgegeben.

Die Funktion “executeAesEcb()” in aes-ecb.c ist gleich aufgebaut, wie die obere: Beide Funktionen erzeugen zufällige Eingabedaten und Verschlüsselungsschlüssel und führen die Ver- und Entschlüsselung durch, wobei der einzige Unterschied in der Art des verwendeten Algorithmus besteht (CBC vs. ECB).