1 Introduction

We have worked on one of the sub-module of **Random Number Generator** (RNG) module. There are many threats and security issues if the application has week policy of random number generator. RNG module provides the function for random number generation in a effective way such that number generated can not be breached easily.
It has 3 sub-modules `ctr_drbg, entropy, havege` in which our focus is on `ctr_drbg` sub-module.

2 Our Work

We have changed the `ctr_drbg.c, ctr_drbg.h` and some other header files to Rust. You can check it out here.
File `ctr_drbg.c` is in src/rng/ as `ctr_drbg.rs` which has the working of the sub-module and `ctr_drbg.h` is in src/rng/header as `ctr_drbg.rs` which has all the macros declared as well as the definition of the structure used in the sub-module.

3 Description

The `ctr_drbg` is a standardized way of building a pseudo-random number generator from a block-cipher in counter mode operation. In `ctr_drbg` *(Counter-mode block-cipher-based Deterministic Random Bit Generator)*, **entropy** is gathered with the help of a callback function which is provided at the time of initialization. This callback is invoked on initialisation as well as when reseeding is required.
Also, there are some initialization which are required :

- **Entropy Length** : It's value is 48 bytes. This is the amount of entropy to be used on each and every seed and reseed. This is of 48 bytes because the entropy module uses SHA-512.
- **Reseed Interval** : It's default value is 10000, this means reseed is given in every 10000 calls. In other words, the number of calls to `ctr_drbg` after which it is reseeded.
- **Personalisation String** : It is an *optional string* used to differentiate the instants. It is appended on the seed during the initialisation so that you can uniquely identify the current instant.
- **Prediction Resistance** : It is initially (as a default) is disabled. If this is disabled reseeding is done on the specified interval but is it is enabled then reseeding occurs in every call, not only on the mentioned interval.
- **Block & Key Size** : The value of block size used by cipher is 16 bytes and key size used by the cipher is of 32 bytes.
- **mbedtls_ctr_drbg_context** : This is a structure used in the sub-module to represent the internal state consisting of some variables like counter, entropy length, reseed counter *(the number of requests that have been made since the last (re)seeding)*, interval, and AES context defined in AES-256 used by the `ctr_drbg` algorithm, and the settings specified on initialisation.

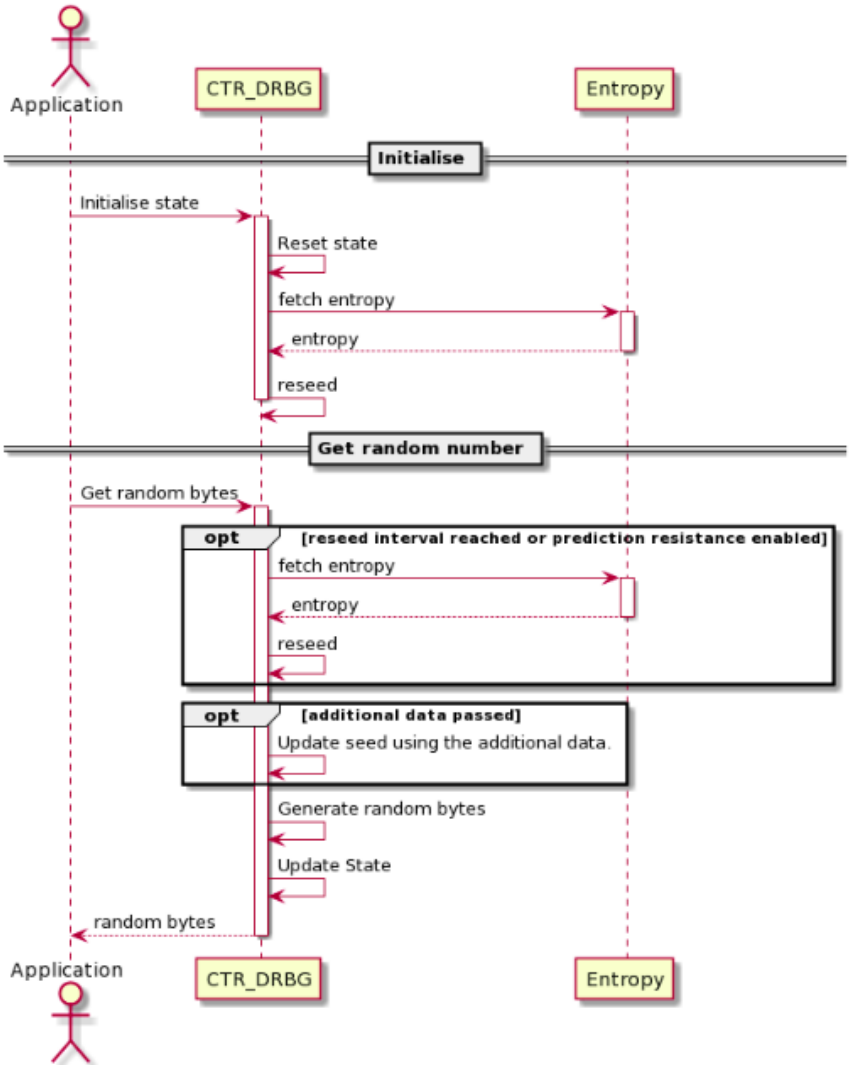

Figure 1: Working of *ctr_drbg* (Source : tls.mbed.org)

Some important functions:

- **block_cipher_df** : It constructs IV (16 bytes) and S in buffer such that IV is the counter (in 32-bits) padded to 16 with zeroes and S is the length of input string (in 32-bits) or length of output (in 32-bits) or data or 0x80 and total is padded to a multiple of 16-bytes with zeroes. After that it reduces data to MBEDTLS_CTR_DRBG_SEEDLEN bytes (i.e, 48 bytes) of data and finally, do the encryption with the reduced data.
- **mbedtls_ctr_drbg_reseed_internal** : It gathers the entropy_len bytes of entropy to seed state and also entropy for a nonce if requested. After that it adds the additional data if it is provided. Finally, it reduces the modified seed into 384 bit data and update the internal states.
- **mbedtls_ctr_drbg_seed** : It checks the desired amount of entropy to grab for a nonce. If

it is -1, that indicates the entropy nonce length was not set explicitly and use a sufficiently large nonce for security. After that it initializes a empty key with the initial seed and returns 0 if successfully done.

- **mbedtls_ctr_drbg_random_with_add** : This function updates a ctr_drbg instance by incrementing the counter, crypting the counter block, and copying the random block to destination with additional data and uses it to generate random data and returns 0 on success.

After the initialisation of the module, random numbers can be requested from it. During the request of random numbers, an additional data (optional) can be passed, which can be included while updating the internal state of `ctr_drbg` algorithm.
There are many conditions on which the modules returns an error such as :

- If the entropy source fails for some reason.
- If the requested amount of random is too large.
- If the input data buffer used to seed the algorithm is too large.

4 Challenges

- **GOTO & EXIT :** In the `ctr_drbg.c` there are many goto and exit calls which are considered to be unsafe. So, we have made a extra function for that code segment.
- **String Functions :** Some string functions like `memset` and `memcpy` are used too frequently but they don't have a direct alternative in Rust.
- **VOID :** Void function conversion was easy but the void pointers and typecasting from void and to void typecasting was bit difficult. At last we have resolved this issue by using a package `std::ffi::c_void`.
- **MACROS Conversion :** Changing from C Macros to RUST was bit difficult and confusing. So, we changed the variable with default values described as a macro in C into Rust as a global immutable variable.
- **Function Pointer :** Function pointer is declared and defined little bit complicated as compared to C.
- **File Handling :** There is no good documentation of file handling in Rust and there is one function in the sub-module which works on file.
- **No direct conversion :** As there is no direct conversion from C to RUST. So, we have to understand the basic functionalities of the pre-defined functions of C as well as time whole function.
- **Dependencies :** `ctr_drbg` module is dependent on many other sub-modules of some different module (like AES).

5 Securities Added

- **Variables :** There is a large number of variable in which some are declared as non-constant in C but whose value is not changing during the whole function call. This vulnerability may rise to some exploit. So, we have modified it into immutable variable as in the implemented code such that during the whole function execution no one can change the value of the variable.
- **MEMCPY :** `memcpy` function is not safe to use because doesn't check for overflow or \0. As well as it leads to problems when source and destination addresses overlap. So, we removed this issue by doing with simply a control loop.
- **SIZEOF :** In Rust, we can't do `sizeof(var)` if var is mutable(means whose value can be manipulated). It is good for security reasons in terms of memory allocation.
- **Borrowing :** When we give the references during some function call in C it may be vulnerable for the application. So, we have used the borrow checkers for passing the variable as reference. It is acheived due to Rust's ownership model and Borrow Checker.
- **Pointers to Mutable Variables :** If a mutable reference `(&mut)` to an object exists, no other reference can exist for that object. It is only possible due to the **memory safeties** in Rust. Also in Rust, we can create a pointer to any location, but we can't dereference it. Any reference we create is always bound to an object.

6 Conclusion

We have completed the conversion of the sub-module `ctr_drbg` from C/C++ to Rust successfully. Also, we removed some of the vulnerabilities from the code and made it more secure.

References

[1] `High Level Design of mbedTLS :` here

[2] `RNG Module Level Design of mbedTLS :` here

[3] `arm MBED API Documentation :` here

[4] `RUST Documentation :` here

[5] `C++ Documentation :` here