flea cryptographic library v1.0
manual

cryptosource GmbH, Darmstadt
flea@cryptosource.de
www.cryptosource.de

cryptosource
Cryptography. Security.

January 16, 2016

Contents

1 Introduction

flea, the acronym of "flexible lightweight efficient algorithms" is a cryptographic library written in C and intended for strongly resource constrained devices. In the current version it comes with a basic set of the most fundamental symmetric and asymmetric algorithms. It is especially intended to run on bare microcontrollers without any operating system support. However, it can be readily run on standard PCs as well. The current release of flea only supports 32-bit architectures but compiles and runs also on 64-bit machines.

The flexibility of the library is manifold: it begins with the availability of the library under two licenses allowing its free use, the GPL v3 and the flea license. The latter allows the use of the library in closed source applications free of charge with even less requirements than in a BSD license but without permission of redistribution of the source code.

Turning to the implementation itself, the library offers various compile-time configuration options, such as

- the set of algorithms to include in the library
- use of stack memory only or the additional use of the heap,
- configuration of the maximal key sizes for the determination of the buffer sizes in stack-only mode,
- buffer overwrite detection through canary values at the start and end of each buffer,
- various trade-offs concerning code-size, RAM demand and speed.

2 Getting started with flea

In order to get easily started with flea, the library is shipped with a CMake configuration to build it on standard Linux or Windows PCs. On Linux, in the flea directory, the command sequence

```
cmake .
make -j4
./build/unit_test
```

can be issued to build the library, the unit tests, and execute the tests. Empty output indicates successful completion of the tests.

However, the library is shipped in a state in which it produces two compilation errors. This is due to the fact that for the secure operation of the random number generator (RNG) of the library the user has to implement two functions for the persistence of the RNG state. The source code contains guidance how to quickly repair this compilation errors with a workaround, which, however, leads to an insecure build of the library that must not be used in production code. In Section 3 we give guidance on how to use the RNG safely.

The compile-time configuration of the library is completely managed by a preprocessor framework. All configuration options can be set in the file `include/internal/common/build_config.h`, making it easy to configure the library on embedded platforms without any dependency on the build system.

3 Random Number Generation

flea features a simple but secure random number generator which is based on the management of an RNG state file. In order to use the RNG safely in an embedded firmware, the following steps must be taken.

3.1 Necessary Steps During Implementation

During implementation, the in the file `src/user/prng_user.c` the functions for loading the PRNG state from the non-volatile memory (NVM) and saving it to the NVM have to be implemented. The developer must allocate an area within the device's NVM for this purpose, the size of which is 32 bytes.

A second requirement is that during the device's startup, prior to the use of any other functions of the flea library, the function

```
flea_err_t THR_flea_lib__init()
```

must be called, which internally loads the RNG state from NVM into the corresponding RAM variable.

3.2 Necessary Steps During Production and Activation

When the devices running the firmware using flea are produced, before the execution of any cryptographic operation, the PRNG state must be seeded with a high entropy seed different for each device. This seed should be generated on an external computer which has appropriate entropy sources available for this purpose. The function

```
flea_err_t THR_flea_rng__reseed_persistent(const flea_u8_t* seed, flea_dtl_t seed_len);
```

must be called once during the production or activation process with that seed data. From that point on, the RNG state will be managed internally by the flea library. During startup, when the function `THR_flea_lib__init()` is called, the stored RNG state is loaded from NVM to initialize the RNG, then a fresh RNG state is created from the RNG's output and stored in the NVM. From that point on, the RNG operates efficiently with its RAM state and secure, non-repetitive random number generation is ensured even after a reset.

4 Using flea's API

flea uses a macro framework to abstract object and buffer initialization as well as error handling. This allows for greater flexibility and security for a number of reasons:

- it is possible to switch between stack and heap usage for buffer allocation by changing a single compiler flag,
- the life cycles of buffers and objects are formalized and integrate seamless with the error handling, guaranteeing safety from resource leaks and errors such as use-after-free if the programming standards are adhered to.

From the following sections, only Section 4.1 is essentially important for users of the library since the other sections describe the internal design of flea which is not necessary for the use of its API. But since the test code shipped with flea, which can serve as a source of example code, is written in the same framework, it is helpful to understand also the approaches to error handling and buffer management outlined in the subsequent sections.

4.1 flea's Objects

flea's objects have the following life cycle states

- UNINITIALIZED. This state is entered by simply declaring an object on stack or heap without assigning any value to it. From this state, only the transition to the state INITIALIZED is allowed.

- INITIALIZED. This state is entered by calling the macro `FLEA_DECL_OBJ(symbol_name, object_type)` for the object declaration and simultaneous initialization. Alternatively, an object in the UNINITIALIZED can enter this state is the corresponding initialization macro, following the naming pattern `flea_<type>__INIT(symbol_address)`, for that object is called. In this state, only two actions may be performed on the object: The object's destructor (dtor) function or one of its constructor (ctor) functions may be called.

- CREATED. This state is entered by the calling one of the object's ctor functions. It can only be left by calling the object's dtor function.

- DESTROYED. This state is entered by calling the object's dtor function. In this state, the dtor function may be called repeatedly (without any effect) or the object may be created again using a ctor function.

This is the contract offered by all types of flea's API. The dtor calls should be made even in the stack-only mode, because they wipe secret values from the RAM before deallocation.

4.2 Error Handling

The error handling in flea is also abstracted by macros. Any function of flea that potentially returns an error, referred to as a throwing function, can be identified by starting with the string `THR_`. Such a function returns `FLEA_ERR_FINE` upon success or an error code otherwise. Any throwing function in flea is structured by the following skeleton of macros:

```
<declarations-section>
FLEA_THR_BEG_FUNC();
<initializations-section>
...
FLEA_THR_FIN_SEC(cleanup-code);
```

where `cleanup-code` is the code that shall be executed whenever the function ends, either by naturally running to the location of the macro call `FLEA_THR_FIN_SEC()` or because between the two macros calls an error was thrown and not handled. This can happen mainly due to two incidents:

- An error was raised by calling the macro `FLEA_THROW()`. This causes the routine to directly jump to the cleanup code and the function returns the error code raised by `FLEA_THROW()`.

- An error was returned by a throwing function called as `FLEA_CCALL(THR_flea_some_function(args))`. The behaviour is the same as for the `FLEA_THROW()` macro. Here, the error code returned by the called function is returned to the caller.

In order to have correct life cycle management even in the presence of errors thrown or returned by called functions, one essential rule must be followed:
Before any code that potentially raises errors is executed, all objects must be initialized. If only the `FLEA_DECL_OBJ()` is used for the object declaration, then this is naturally achieved. If, however, any objects are declared in the standard way and only initialized in the initialization-section indicated above, then it is vital that no error throwing code is executed in the initialization section.
Following this rule ensures that the jump to the cleanup section, which typically executes the dtors for all local objects, is possible at any time during error-throwing code.

4.3 Buffer Management

The buffer management in flea is abstracted by macros in order to enable both the abstraction from heap/stack usage and in order to enable the use of canary values for buffer overwrite detection.

The following macros are mainly employed for the buffer management:

- `FLEA_DECL_BUF(symbol_name, type, size)` declares a buffer,
- `FLEA_ALLOC_BUF(symbol_name, size)` allocates a buffer,
- `FLEA_FREE_BUF_FINAL(symbol_name)` frees a buffer,
- `FLEA_FREE_BUF_SECRET_ARR(symbol_name, size)` also frees a buffer, but also overwrites the memory contents prior to that.

All buffer sizes are specified in element counts, not in bytes. However, the user of the library, who does not intend to make use of flea's features for switching between stack and heap mode in his own code or use the buffer canaries, may certainly use any explicit heap or stack buffer allocation as he is used to.

5 Support

cryptosource provides commercial support for the flea library. Please visit `http://cryptosource.de/product_flea_en.html` for further information or contact us at `flea@cryptosource.de`.