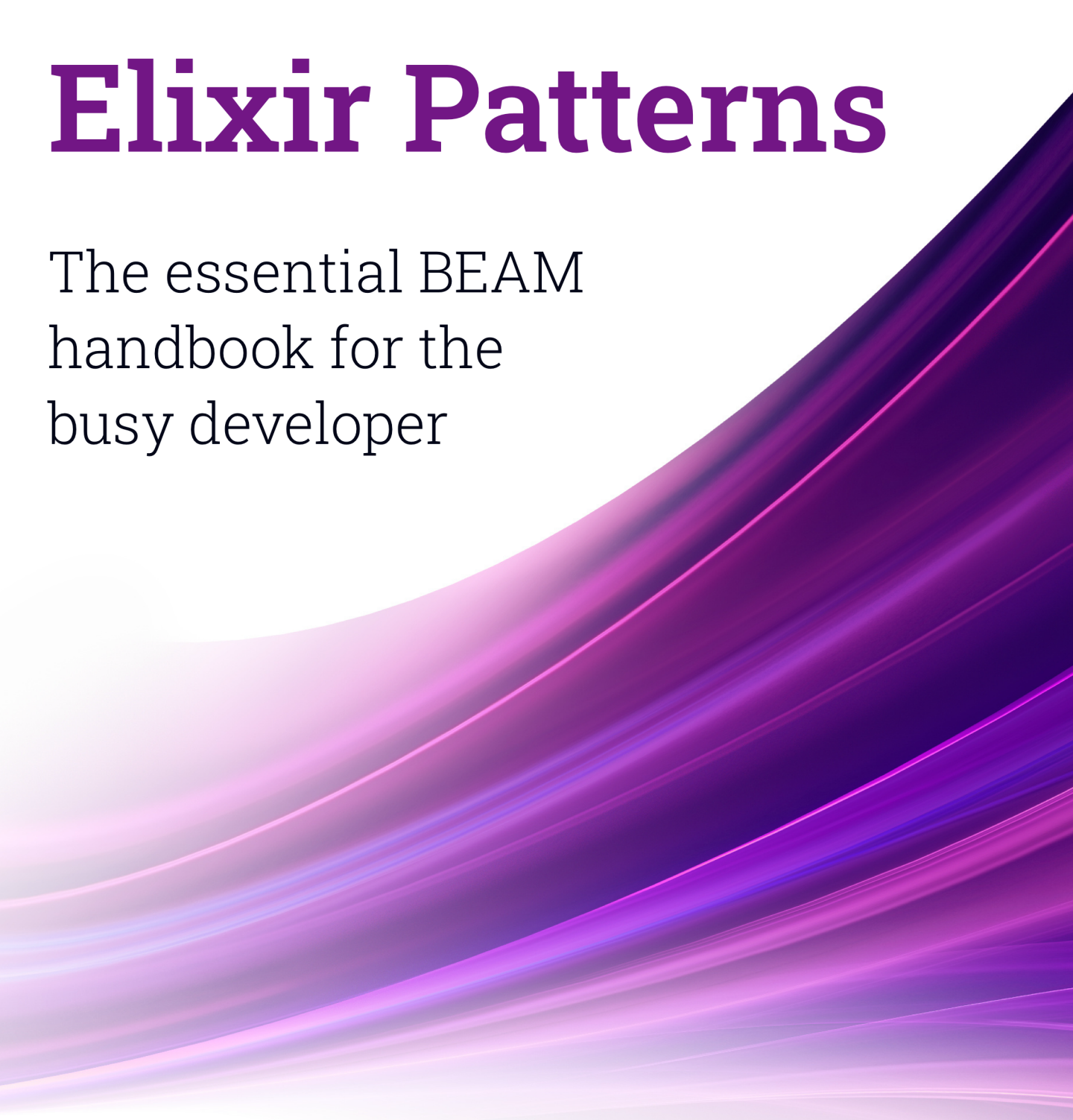
Elixir Patterns

The essential BEAM handbook for the busy developer



Alexander Koutmos
Edited by Hugo Baraúna

Contact team@elixirpatterns.dev for regarding errata and support

Alexander Koutmos
Visit my website at https://akoutmos.com

Hugo Baraúna
Visit my website at https://elixir-radar.com

Printed in the United States of America

First Printing: March 2025
Printed by KDP

ISBN 979-83-07689-76-9

2.6. Keeping Things Secret with the Crypto Module

Cryptography in and of itself is a very complex and mathematically involved field. While we won't be getting into the weeds as to how the various cryptography algorithms work, it is important to know what the various tools are in the cryptography toolbox and when to reach for them. In general, cryptography is all about secure communication between two or more parties.

The Erlang `:crypto` module helps us ensure secure communication by providing us with tools for computing hashes from data, for validating message authentication codes (MACs), and for encrypting data both symmetrically and asymmetrically. If you are specifically interested in asymmetric cryptography, be sure to also check out the Erlang `:public_key` module[5].

Before diving into the code, let's unpack what these four groups of tools are and how we can leverage them in our day to day programming endeavors.

1. Hash functions - Simply put, hash functions can deterministically compute an output (i.e the same) of a fixed length, regardless of the size of the input. In other words, given a hash function `X`, and input data `Y` you will always get an output of `Z` (or if you prefer it as an equation `X(Y) = Z`). This also means that you cannot reliably reconstruct the input from the output given that hash functions are one-way functions and reduce an arbitrary input space into a finite output space. The fact that these functions work one way is why hash functions are used to store passwords. In the case of a database leak, the attacker would need to attempt a large number of permutations in order to find some value that would yield the same output to log in to someone else's account.
2. Message Authentication Codes - Message authentication codes, or MACs for short, allow message senders and recipients to verify that the messages that are shared are both authentic and have not been tampered with. Given a shared secret key between the sender and receiver, the two parties can pass a message payload through a MAC function and generate an authentication code that can be used to

verify the message once it is received. One example where this is particularly useful is when your application supports webhook functionality. The best way to ensure that the payload that you received has not been tampered with and is indeed authentic is to have your sending application provide to you their result of the MAC function and you can compare that to what you compute on your side. If the two values match then you know that the inbound message can be safely processed.

3. Symmetric Encryption - Symmetric encryption is probably the category of cryptographic tools that people most associate with cryptography. With symmetric cryptography, a message is encrypted with a secret key, at which point it is no longer discernible what the original message was. In order to derive the original message, the same secret key must be applied to the encrypted message through a function that decrypts the encrypted message. If you are encrypting data at rest in a database, this is generally how it is done. It is encrypted via a symmetric encryption algorithm and then written to the database.

4. Asymmetric Encryption - Asymmetric encryption is slightly more complicated than symmetric encryption but equally important. Whereas symmetric encryption relies on the same key to encrypt/decrypt a piece of data, asymmetric encryption relies on a pair of keys to encrypt and decrypt messages. One of the keys is known as the public key which can be freely distributed to anyone without compromising security. The other key is the private key and this key (as the name implies) should be kept secret and not distributed. Using the public key, message senders can encrypt messages and send them to their intended recipients. Only the person who has the private key can decrypt the message. This way, even if the sender's machine is compromised, all the attacker would be able to get access to is the public key which is not useful in decrypting messages anyways. If you have ever accessed a website over HTTPS, then you are using asymmetric encryption under the hood to ensure that your communication with the server is secure.

With an understanding of what these different types of tools are and how they work, let's take them for a test drive using the `:crypto` module.

2.6.1. Hash Function

As mentioned earlier, hash functions are one-way functions that produce the same fixed-length hash for the same input. In the example below we pass a few different binaries to the hash `:crypto.hash/2` function and apply different hash algorithms each time:

Listing 29. Computing the hash from data

```
iex(1) > :crypto.hash(:blake2s, "This is some data") ①
...(1) > |> Base.encode16()
"B060EC95F0BEF0F967671AB6A47196685CC241BF3CD329A8DB3A09E253C7CCA9"

iex(2) > :crypto.hash(:blake2s, "This is some other data") ②
...(2) > |> Base.encode16()
"170959DFB421768A00EFDC6C70580C5B473E54691A6E5EF32711D4D102CE8AE8"

iex(3) > :crypto.hash(:sha256, "This is some other data") ③
...(3) > |> Base.encode16()
"07F6BFDB1BC57D898DD8A9022BF01BB581529323071E21337628C3EF6AB29BD1"
```

① Generate the hash of the data using the `:blake2s` algorithm

② Using the same algorithm but a different payload, a different hash is generated

③ In this case we generate the hash of the data using `:sha256`

As you can see, by using the `:crypto.hash/2` function we can generate the hash for a piece of data. In the previous examples, we leveraged the `:blake2s` and `:sha256` algorithms to generate the hash on some strings. Next, let's take a look at some different ways to use MACs.

2.6.2. Message Authentication Codes

MACs allow us to validate that a message came from a known source and that the message has not been tampered with. They are able to do this given that the sender and receiver share a secret key and when the key is applied to the message, an equal

hash should be generated. Let's take this for a test drive for a better understanding:

Listing 30. MAC helper functions

```
iex(1) > generate_hmac = fn secret_key, payload -> ①
...(1) > :hmac
...(1) > |> :crypto.mac(:sha256, secret_key, payload)
...(1) > |> Base.encode64()
...(1) > end
#Function<43.65746770/2 in :erl_eval.expr/5>

iex(2) > validate_hmac = fn your_key, payload, expected_hash -> ②
...(2) > :hmac
...(2) > |> :crypto.mac(:sha256, your_key, payload)
...(2) > |> Base.encode64()
...(2) > |> Kernel.==(expected_hash)
...(2) > end
#Function<42.65746770/3 in :erl_eval.expr/5>
```

① The `generate_hmac` helper function will generate a MAC hash using the SHA256 algorithm

② The `validate_hmac` helper function will check to see if the secret key provided yields the expected MAC hash

Let's leverage these helper functions by running some data through them and checking to see what happens when we attempt to validate the data payload:

Listing 31. Validate data integrity with a MAC

```
iex(3) > payload = :erlang.term_to_binary(%{some: "Data", i: "Need"}) |>
Base.encode64()
"g3QAAAACZAABaW0AAAAETmVlZGQABHNvbWVtAAAABERhdGE=" ①

iex(4) > secret_key = "this_is_a_secret_and_secure_key" ②
"this_is_a_secret_and_secure_key"
```

```
iex(5) > correct_hmac_hash = generate_hmac.(secret_key, payload)
"gp1QB0rWy4m5DDoDBqZU4hwyVhBdwMXc0gnGELup1Ow="

iex(6) > validate_hmac.("INVALID KEY", payload, correct_hmac_hash) ③
false

iex(7) > validate_hmac.(secret_key, payload, correct_hmac_hash) ④
true
```

① Here we serialize some data using `:erlang.term_to_binary/1`

② We set the correct secret key to `this_is_a_secret_and_secure_key`

③ Given an invalid key, the `validate_hmac/3` function returns false

④ Given a valid key, the `validate_hmac/3` function return true

At first glance, it may seem as though this type of cryptography tooling does not provide a lot of utility when all the data you are working with is local to the machine. Once you consider transmitting data on the public internet, then this becomes far more useful. For example, APIs as a service such as Stripe[6] and Twilio[7] leverage MACs in order to give you the assurance that the data you are operating on is authentic and has not been tampered with. Now that you have a good understanding of how MACs work, let's take a look at symmetric encryption.

2.6.3. Symmetric Encryption

Symmetric encryption is usually what comes to mind when people talk about cryptography. With symmetric encryption, a secret key is required both to encrypt and decrypt messages. Full disk encryption usually relies on symmetric encryption to secure data at rest on the disk and when it comes time to read data off the disk, you must provide the same secret key. Let's see how we can do this using the Erlang `:crypto` module. First we'll create a couple of helper functions in order to make the code more concise:

Listing 32. Helper functions to encrypt and decrypt

```
iex(1) > encrypt =
...(1) >   fn message, key ->
...(1) >     opts = [encrypt: true, padding: :zero] ①
...(1) >     :crypto.crypto_one_time(:aes_256_ecb, key, message, opts)
...(1) >   end
#Function<43.65746770/2 in :erl_eval.expr/5>

iex(2) > decrypt =
...(2) >   fn payload, key ->
...(2) >     opts = [encrypt: false] ②
...(2) >
...(2) >     :aes_256_ecb
...(2) >     |> :crypto.crypto_one_time(key, payload, opts)
...(2) >     |> String.trim(<<0>>) ③
...(2) >   end
#Function<43.65746770/2 in :erl_eval.expr/5>
```

① When we encrypt data, we need to be sure that we pass the `encrypt: true` option. In addition, we need to pad the message using the `padding: :zero` option to ensure that our data does not get trimmed because it is not a block size aligned payload.

② When we decrypt the payload, we need to pass the `encrypt: false` value to signal to the `:crypto.crypto_one_time/4` function that we want to decrypt.

③ Given that we padded the encrypted payload, we also need to trim any trailing null bytes.

With our helper functions in place, we can start to leverage them and see what we get back:

Listing 33. Symmetrically encrypting and decrypting data

```
iex(3) > message = "This is a very very important message. Keep it
secret...keep safe"
"This is a very very important message. Keep it secret...keep safe" ①
```

```
iex(4) > secret_key = :crypto.strong_rand_bytes(32) ②
<<97, 176, 26, 172, 231, ...>>

iex(5) > encrypted_message = encrypt.(message, secret_key) ③
<<219, 216, 102, 130, 75, ...>>

iex(6) > try do
...(6) >   decrypt.(encrypted_message, "INVALID_KEY") ④
...(6) > rescue
...(6) >   error -> error
...(6) > end
%ErlangError{original: {:badarg, {'api_ng.c', 244}, 'Bad key size'}}

iex(7) > decrypt.(encrypted_message, :crypto.strong_rand_bytes(32)) ⑤
<<194, 121, 32, 55, 79, 163, ...>>

iex(8) > decrypt.(encrypted_message, secret_key) ⑥
"This is a very very important message. Keep it secret...keep safe"
```

① This is the information that we would like to keep secret

② Using `:crypto.strong_rand_bytes/1` we are able to generate a 32 byte secret key

③ Using our secret key and the original message we encrypt it

④ Providing a key of the wrong length yields an error

⑤ providing a key of the correct size but incorrect value yields random data

⑥ Using the encrypted data and the secret key, we can derive the original message

As you can see, it is relatively straightforward to encrypt and decrypt data using the `:crypto.crypto_one_time/4` function. One thing that should be noted is the value of the last argument to the function. Be sure that you flip the `:encrypt` boolean accordingly and also set the `:padding` option or else your data will be truncated to fit in the cipher's block (if you are using a block cipher like `:aes_256_ecb`). Aside from that, just make sure that your key is the correct size and that you do not lose the key as you won't be

able to recover your data.

Asymmetric encryption is a bit more complicated and out of scope for this book. If you are interested in the topic though, I would suggest looking at the Erlang Getting Started guide for public key encryption[8].

2.7. What's Next?

While we covered quite a bit of the Erlang standard library in chapters 1 and 2, there are still plenty of gems that are useful in your day-to-day programming that we did not cover here. We urge you to dive into the Erlang docs[9] and do some exploring. You may be surprised by what you find!

With a solid overview of the Erlang standard library, it's time to shift our focus to the Elixir standard library. Similar to the Erlang standard library, there is quite a bit in the Elixir standard library. So instead of going over every function in the Enum or Map module, we'll be looking at useful ways to compose functions from these modules in order to perform various tasks. With that being said, let's jump right to it!

[1] https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

[2] https://www.erlang.org/doc/man/erpc.html

[3] https://www.erlang.org/doc/apps/erts/match_spec.html

[4] https://www.erlang.org/doc/man/dets.html

[5] https://www.erlang.org/doc/man/public_key.html

[6] https://stripe.com/docs/webhooks/signatures

[7] https://www.twilio.com/docs/usage/security

[8] https://www.erlang.org/doc/apps/public_key/using_public_key.html

[9] https://www.erlang.org/doc