Krypcak

Password Tool

CRYPTOGRAPHIC SPECIFICATION

version 1.0

Zerosum Security
www.zerosumsecurity.nl

Contents

1 Overview

This document describes the cryptographic details of the Krypcak Password Tool. Krypcak assists in managing and using your passwords.

The full source of Krypcak can be inspected at [ZS13a].

1.1 Document history

Version	Date	Comments
1.0	2013-12-26	Initial release

1.2 References

[BDP+12] G. Bertoni, , J. Daemen, M. Peeters and G. Van Assche - *Duplexing the sponge: single-pass authenticated encryption and other applications*, Selected Areas in Cryptography, Lecture Notes in Computer Science, Volume 7118, 2012, pp 320-337

[BDP+13a] G. Bertoni, , J. Daemen, M. Peeters and G. Van Assche. Keccak Code Package. July 2013. Available from https://github.com/gvanas/KeccakCodePackage

[BDP+13b] G. Bertoni, , J. Daemen, M. Peeters and G. Van Assche – A Software Interface for Keccak. July 2013. Available from http://keccak.noekeon.org/NoteSoftwareInterface.org

[JPT13] S. Josefsson, Tarsnap, C. Percival - *The scrypt Password-Based Key Derivation Function* (draft-josefsson-scrypt-kdf-00). Available from http://tools.ietf.org/html/draft-josefsson-scrypt-kdf-00 (retrieved 2013-06-23)

[NM13] Nick Mathewson - What has gone wrong with RNGs in practice. http://www.ietf.org/mail-archive/web/dsfjdssdfsd/current/msg00007.html

[P09] C. Percival - *Stronger key derivation via sequential memory-hard functions*, BSDCan'09. http://www.tarsnap.com/scrypt/scrypt.pdf (retrieved 2012-06-23), 2009.

[ZS13a] Zerosum Security – Krypcak Source code. Available at https://github.com/zerosumsecurity/Krypcak-for-windows

[ZS13b] Zerosum Security – Krypcak Password Tool / File Formats Specification. Available at www.zerosumsecurity.nl/krypcak.

2 Keccak

All cryptography is Krypcak is centered around Keccak in duplex mode [BDP+12]. In fact, Krypcak should be partly seen as a tribute to the concept of sponge functions (of which Keccak is an example). Many known cryptographic constructions can be simplified and/or made more elegant by using sponge functions, either in “classic” absorb/squeeze mode or in duplex mode.

Keccak is used with an internal state of 1600 bits is used with a rate of 1027 bits and a capacity of 573 bits. Per application of Keccak-f, blocks of size upto 1024 bits of data are absorbed and/or squeezed.

Note: each input block is padded to a 1027-bit block before being absorbed. This padding is implicit and is not shown on the schematics in this document. See [BDP+13b] for more details.

In Krypcak, Keccak in duplex-mode is mainly used for

1. Authenticated encryption (see §3), when protecting notebooks, encrypted files and encrypted messages. This is done with Keccak in Spongewrap mode (which uses the duplex construction).
2. All the instances of authenticated encryption mentioned in 1. are keyed with passwords, so we need a form of password hashing. Keccak in duplex-mode is used for this (see §3.1);
3. Pseudo-random generation (see §4). Keccak in duplex-mode is used to collect and process different sources of random (from timings, the Windows cryptographic pseudo random generator and user mouse movements).

2.1 Implementation

We use the implementation of Keccak in duplex mode as provided by the designers of Keccak in [BDP+13a] and described in [BDP+13b].

3 Authenticated encryption

All data that is encrypted in Krypcak is also provided with a message authentication code (MAC). The following data is encrypted in Krypcak:

- the content of a password notebook;
- files;
- messages.

This is all done with Keccak in SpongeWrap mode, as proposed in [BDP+12]. Keying of the SpongeWrap is done with a password. The way a SpongeWrap is keyed with a password is described in §3.1.

3.1 Password hashing

The password hashing scheme with Keccak in duplex mode is based on the SCRYPT password hashing scheme (see [P09], [JPT13]). Here the versatility of sponge functions eases the construction: there is no need for separate primitives such as a stream cipher and a hash function as is the case in SCRYPT – Keccak is happy to take on both roles.

3.1.1 Initialization

The SpongeWrap Keccak duplex object is initialized with a 16-byte salt and a password.

First, the salt is prepended with the byte 0x10, indicating the length of the salt in bytes. After the salt the length of the password is appended (number of characters, encoded in a single byte), followed by the password itself (in unicode encoded, taking two bytes per character). The resulting byte array is padded to end up with a multiple of 128 bytes. Assume we end up with k 128-byte blocks, which we denote $A_0,..A_{k-1}$. Hence

$$A_0 \,||..||\, A_{k-1} = \text{len(salt)} \,||\, \text{salt} \,||\, \text{len(password)} \,||\text{password (unicode)} \,||\, \text{pad}$$

where "||" denotes concatenation. In k steps, each of these blocks is absorbed by the SpongeWrap Keccak object.

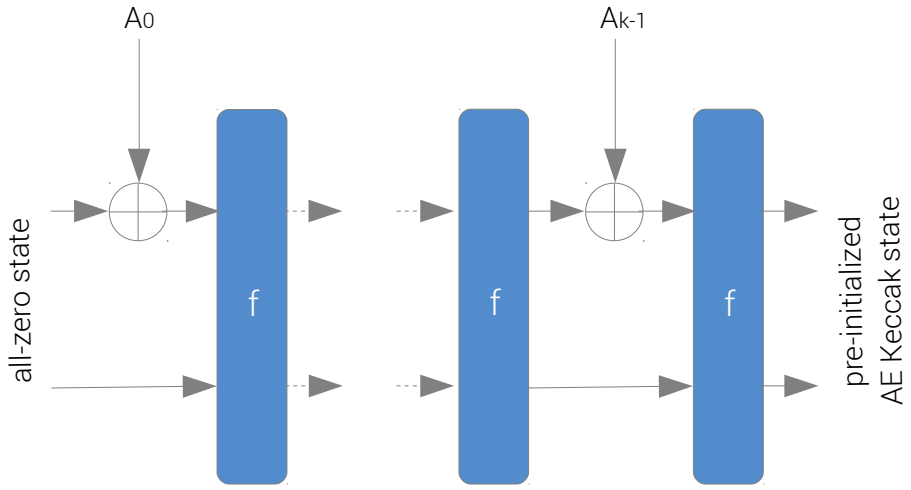

Illustration 1: Absorbing salt and password

3.1.2 Expensive loop

Now we could use this SpongeWrap Keccak object for data encryption. However, to hinder password brute-forcing, the SpongeWrap Keccak object is fist transformed in a way that requires

- many CPU cycles;
- a non-negligible amount of RAM.

This is done in two steps:

- in n (with n a large number defined below) iterations, n blocks of 128-bytes each are squeezed from the SpongeWrap Keccak object. These blocks are denoted by $B_0,..,B_{n-1}$.
- In n subsequent iterations these n blocks are again absorbed by the same SpongeWrap Keccak object.

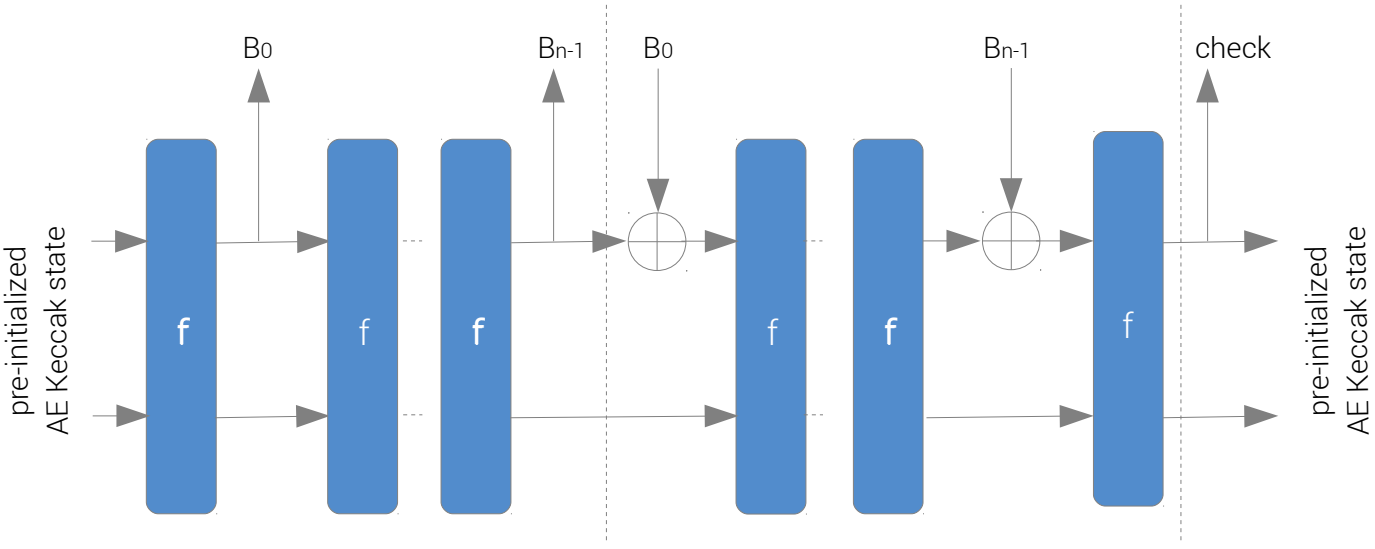

Illustration 2: Expensive password hashing

In Krypcak we set $n = 2^{19} = 524{,}288$. This implies a total of $2^{20} = 1{,}048{,}576$ invocations of the Keccak-f function. Since all the blocks $B_0,..,B_{n-1}$ have to be held in memory together, this requires a total of $2^{19} \times 128$ bytes = 64 MB of RAM.

After the last block B_{n-1} is absorbed, a 32-bit password check is squeezed from the Keccak state. This 32-bit check will be part of the output of the encryption process. It allows for the detection of an incorrect password without having to decrypt all the data and then concluding (upon an incorrect MAC) that the supplied password might have been wrong.

After the check has been squeezed out, the SpongeWrap object is ready to encrypt or decrypt data.

3.2 Data encryption

After the SpongeWrap object is initialized with a 16-byte salt and a password and after a 32-bit password check has been generated (as described in §3.1), it is ready to process the data to be protected.

Denote the data to be encrypted with D and suppose D consists of M bytes. For encryption D is padded to end up with a multiple of 128 bytes. Assume we end up with m 128-byte blocks, which we denote $P_0,..P_{m-1}$. Thus

$$P_0||.. \,||\, P_{m-1} = D \,||\, pad$$

First the 28-byte header H is absorbed. This header contains (see also [ZS13b]):

- the 4-byte file type identifier;
- the 4-byte version information block;
- the 16-byte salt;
- the 4-byte password check.

Then m iterations of the following form are executed:

- Squeeze out 128 bytes of keystream from the Keccak state;
- Exclusive-or these bytes with block P_i to form the ciphertext block C_i.
- Absorb the plaintext block P_i.

After all m blocks P_i have been processed, a final 16-byte block is squeezed out. This block forms the message authentication code (MAC).

The output consists now of the M leftmost bytes of $C_0,..,C_{m-1}$ concatenated with the 16-byte MAC.

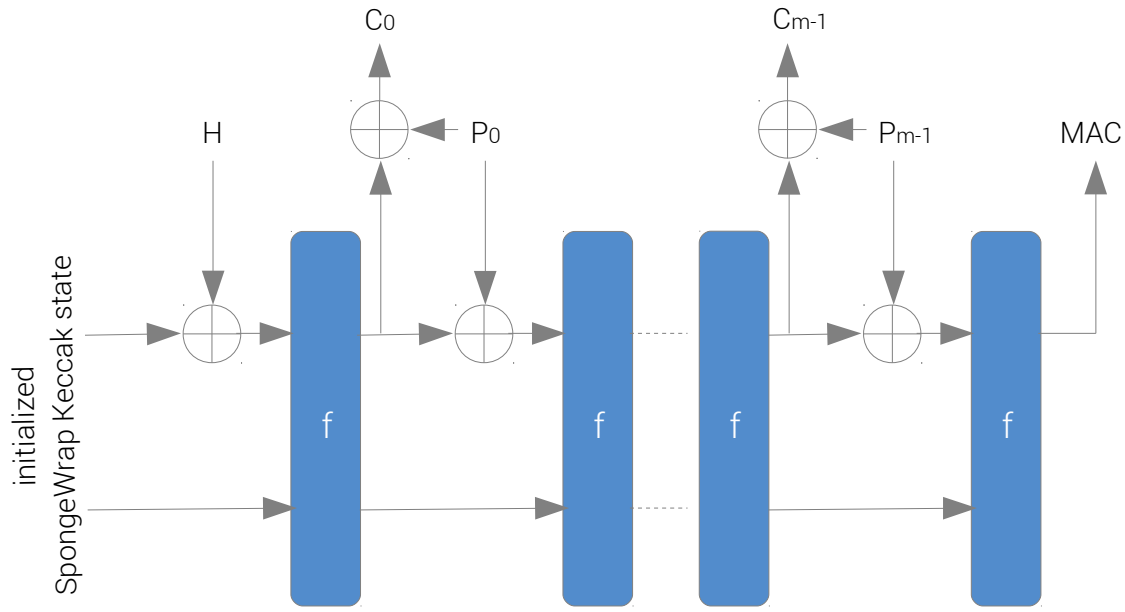

Illustration 3: Data encryption

3.3 Data decryption

After the SpongeWrap object is initialized with the 16-byte salt and the password, and a 32-bit password check is re-generated (as described in §3.1), this check is compared with the 32-bit check included in header of the encrypted data.

If the two checks match, the SpongeWrap object is ready to decrypt the data.

Also the last 16 bytes (containing the MAC) are stripped and stored for later comparison.

Assume the total number of bytes of encrypted data is M.

First the 28-byte header H is absorbed. This header contains (see also [ZS13b]):

- the 4-byte file type identifier;
- the 4-byte version information block;
- the 16-byte salt;
- the 4-byte password check.

Then m iterations of the following form are executed:

- Squeeze out 128 bytes of keystream from the Keccak state;
- Exclusive-or these bytes with block C_i to form the plaintext block P_i.
- Absorb the plaintext block P_i.

After this, a final 16-byte block is squeezed out. This block forms the message authentication code (MAC). This is compared with the MAC as included in the encrypted data.

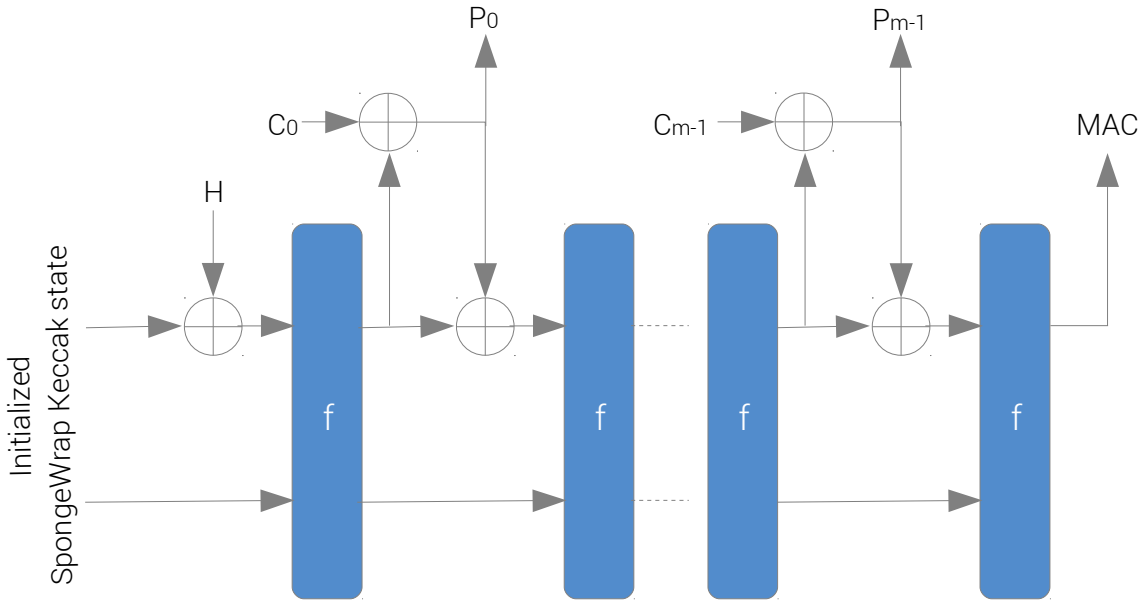

Illustration 4: Data decryption

4 Pseudo random generator

Random bytes are generated in Krypcak for the following purposes:

- Generating passwords and passphrases;
- Generating salts for password hashing (as described in §3.1).

4.1 Design rationale

The pseudo-random generator is Krypcak was designed to stay away from all known pitfalls, as e.g. mentioned in [NM13]. More specifically the PRNG was designed to:

- Not depend on a single entropy source;
- Not use random sources directly but include a cryptographically strong post-processing;
- Be forward secure – compromising the state of the PRNG should not affect previously generated random;
- Recover quickly from a state compromise – include fresh entropy as frequently as possible to recover as quickly as possible from a temporary compromise of the internal state.

4.2 Internal state

The pseudo random generator in Krypcak has an internal state of 328 bytes. These bytes are divided over two buffers:

- A 128-byte entropy buffer;

- A 200-byte internal state of a Keccak duplex object.

The entropy buffer serves as a dump site for entropy gathered during operation of the application.

The Keccak PRNG duplex object is created at startup of the application and is persistent until the application is closed. It serves as the main collector of entropy and requests for random data ar handled by supplying output from this duplex object.

4.3 Initialization

On initialization of the PRNG, the 200-byte internal state of the Keccak duplex object is set to the all-zero state, as is the 128-byte entropy buffer.

The following low-entropy system information is copied into the entropy buffer:

- the two-byte process id;
- the two-byte thread id;
- four bytes of information from the free disc space of the system.

Then the PRNG is updated as described in §4.5.

4.4 Sources of entropy

During operation, the entropy buffer gathers entropy from different sources:

- Mouse movements from the user (see §4.4.1);
- Timing information (see §4.4.2).

Besides the entropy accumulated in the entropy buffer, random from the Windows operating system is mixed directly in the PRNG Keccak state (see §4.4.3).

4.4.1 Mouse movements

User mouse movements over a window of the Krypcak application are detected and captured. The mouse position is then converted into a byte by concatenating the lower four bits of the x-coordinate and the lower four bits of the y-coordinate.

This byte is then copied into the entropy buffer.

4.4.2 Timing information

When the PRNG is updated (see §4.5) and whenever the application requests random from the PRNG (see §4.6), twelve bytes of timing information is copied into the entropy buffer:

- four bytes from GetTickCount();
- four bytes from GetLocalTime();
- four bytes from QueryPerformanceCounter().

4.4.3 Windows random

The Windows operating system contributes entropy via a call to the WINAPI routine CryptGenRandom(). Each time this function is called a total of 128 bytes is requested. These bytes are directly into the PRNG Keccak state.

The function is called when the PRNG is updated (see §4.5) and *twice* whenever the application requests random from the PRNG (see §4.6).

4.5 Updating internal state

When the PRNG is signaled to update its state it will

- fetch and absorb 128 bytes of Windows random (see §4.4.3);
- add 12 bytes of timing information to the entropy buffer (see §4.4.2)
- absorb the content of the entropy buffer

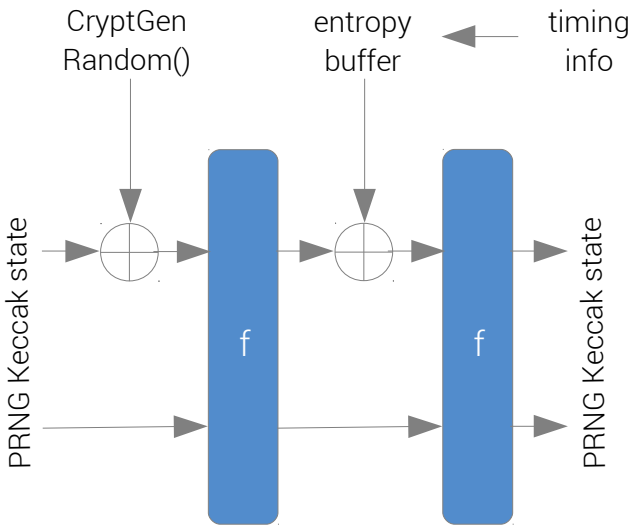

Illustration 5: Updating the PRNG

The PRNG is signaled to perform this internal update

- on start-up of the application;
- whenever the entropy buffer has accumulated 128 fresh bytes of data from mouse movements and timing information;
- whenever the user activates one of the GUI controls (such as a button or a check-box).

4.6 Generating random

When random is requested the PRNG will

- fetch and absorb 128 bytes of Windows random (see §4.4.3);
- add 12 bytes of timing information to the entropy buffer (see §4.4.2)
- absorb the content of the entropy buffer;
- squeeze out the requested number of random bytes;
- again, fetch and absorb 128 bytes of Windows random (see §4.4.3).

Note absorbing the last block of output from CryptGenRandom() serves to make the PRNG forward secure. In order for an attacker to guess the previously generated output from the PRNG, given a full compromise of the internal state of the PRNG, he would also have to predict the output of the last CryptGenRandom() call that was used.

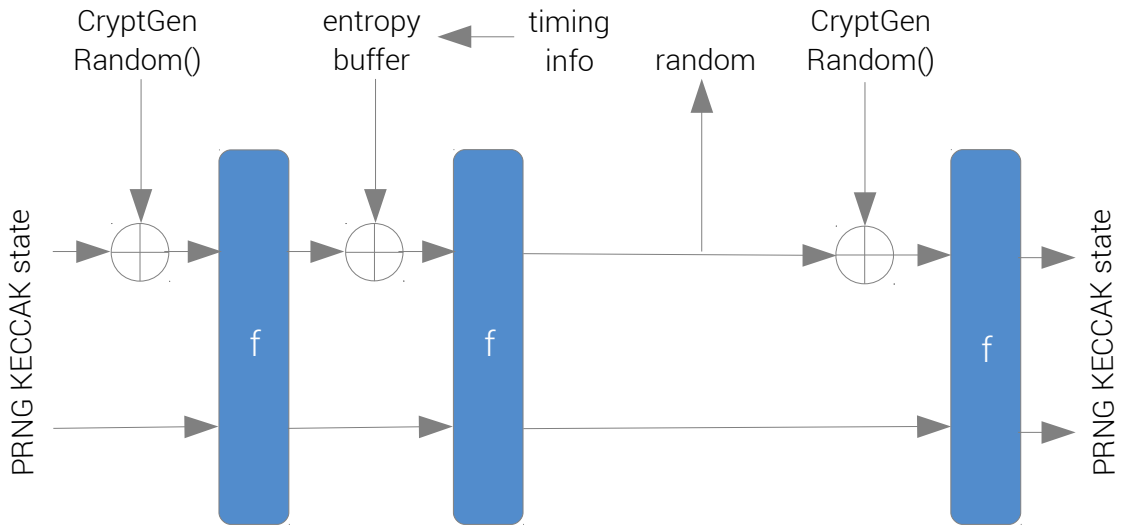

Illustration 6: Random generation

4.7 Entropy estimation

Before random can be generated, Krypcak ensures sufficient entropy has been gathered into the random pool. For each source we estimate (lower-bound) the entropy it can provide per call. Also we pose upper limits to how much we estimate the given source can contribute in total. This to ensure we will not depend too much on a single source.

Source	Entropy per call	Maximum attributed entropy contribution
Combination Process and thread id	8	8
Free disc space	16	16
CryptGenRandom()	32	64
Timing information	8	16
User mouse movement	2	N/A

Table 1: Entropy estimations

When a call for random data is made, and the entropy (in number of bits) is less than 80, the user is requested to provide additional entropy by moving the mouse cursor over a dedicated Krypcak window.

Illustration 7: Requiring user generated random (mouse)