How to correctly use and implement Cryptographic Primitives

In our modern digital world, use of cryptographic services, functions and mechanisms is a common rule. Initially largely dedicated to the protection of data confidentiality, cryptographic algorithms are now used to verify data integrity, data origin and to provide non-repudiable evidence of transactions for instance.

In this article, we survey different cryptographic primitives, their goal and we show, using examples with TMS Cryptography Pack[1], how to use them in a secure manner. Five families of algorithms are discussed: symmetric algorithms, asymmetric algorithms, hash functions, key derivation functions and random number generators.

The last section discusses the critical aspect of key management to properly activate the various primitives.

Symmetric encryption algorithms

These algorithms owe their name to the fact that the same key is used to both encrypt and decrypt file, messages and streams. They use a series of invertible functions that enable the production of a cryptogram from clear text.

The most famous symmetric encryption algorithm is the Data Encryption Standard (DES), closely followed by the Advanced Encryption Standard (AES). DES using only 56 bit keys, it shall be banned to protect any sensitive[2] data.

AES shall be preferred to encrypt data whereas SPECK may be used with low power chips, such as found in embedded systems at large, with the reservation on the fact that it has been designed by NSA and therefore it should not be used to encrypt highly sensitive data with current knowledge on its design.

Symmetric encryption algorithms fall into two categories: block encryption (e.g., AES, SPECK) and stream encryption (e.g., SALSA20). The latter are designed for voice and video stream protection. It is however quite simple to turn a block encryption algorithm into a stream encryption algorithm but throughput is generally lesser.

Also, block size for the input is an important factor to counter “birthday paradox” attacks. To this end, SPECK should only be used with 128 bit blocks for sensitive data.

The **birthday paradox** is a probabilistic estimate of the number of people that have to be in a group to stand a 50% chance that two members of the group were born on the same day of the year. This number is 23 and seems to be counter intuitive. In a group of 57 persons or more, the probability of a match exceeds 99 %.

[1] http://www.tmssoftware.com/site/tmscrypto.asp
[2] Il y a de nombreuses autres considérations que la taille de clé pour considérer qu’un algorithme est sûr mais elles dépassent le cadre de cet article.

Applied to cryptography, and depending on the encryption mode and its properties, it implies that for an n bit block size, about $2^{n/2}$ blocks of encrypted data are required to obtain a collision, i.e., that two distinct messages provide the same cryptogram with an identical key. It is then possible to retrieve the encryption key[3].

[FRAME] Birthday paradox[4]

```
aes := TAESEncryption.Create;

aes.AType := atcbc; // select CBC mode

aes.KeyLength := kl128; // 128 bit key

aes.OutputFormat := base64;

aes.key := '0123456789012345'; // could be generated with Argon2

aes.IVMode := TIVMode.rand; // random Initialization vector

aes.paddingMode := TPaddingMode.PKCS7;

aes.Unicode := yesUni;

s:= aes.Encrypt('test ');

aes.key := #0#0#0#0#0#0#0#0#0#0#0#0#0#0#0#0;

aes.Free;
```

[EXAMPLE 1] AES encryption then (basic and naive) key zeroization

In this example, it is worth noting that cryptographic variables have to be "cleaned up" following use. This measure is essential but not always sufficient because it is not possible to control program execution and memory management performed by the Operating System.

From an operational stand point, it is paramount that the encryption keys are as random as possible and regularly changed.

Asymmetric cryptographic algorithms

With no surprise, asymmetric algorithms owe their name to the fact that encryption keys and decryption keys are different (together with the internals of the algorithms). From an operational stand point, these algorithms, of which RSA is the most famous, are very slow compared to their symmetric siblings and are not used to encrypt (large amounts of) data.

They are mostly used to sign and verify signatures and to encrypt short messages such as encryption keys (then used by symmetric algorithms), in a key exchange mechanism for example, when their cryptographic engine allows it.

```
ecc := TECCEncSign.Create;
```

[3] For a practical attack, see https://sweet32.info/
[4] https://en.wikipedia.org/wiki/Birthday_problem

```
ecc.ECCType := cc25519; // we pick the Ed25519 curve

ecc.OutputFormat := hexa;

ecc.PublicKey := BobPublicKey; // hex format selected

ecc.Unicode := yesUni;

ecc.NaCl := naclNo; // no libsodium compatibility requested

s := ecc.Encrypt('test');

ecc.Publickey := '0000000000000000000000000000000000000000000000000000000000000000'; // hex format

ecc.Free;
```

[EXAMPLE 2] signature with elliptic curve 25519

Public and private keys are usually referred to as key pairs.

Hash functions

A cryptographic hash function enables the generation of a hash for a file or a data block with a close to nil probability to get the same hash from two different files or data blocks[5]. The most famous hash functions probably are SHA1 and MD5, but they are both to be banned because they are too weak from a cryptographic stand point, especially for MD5. SHA2 and SHA3 shall be used because they are more secure and, for the latter, much more modern and flexible.

The hash, either stored or exchanged using another communication channel, enables the verification of the data integrity. A simple hash re-computation and a comparison with the original result are enough.

Signature functions use a hash for their operations. For the sake of efficiency, a file or block hash is computed then signed and or verified (if formerly signed).

```
sha3 := TSHA3Hash.Create;

sha2.HashSizeBits := 256;

sha3.OutputFormat := base64url;

sha3.Unicode := yesUni;

s := sha3.Hash('test');

sha3.Free;
```

[EXAMPLE 3] use of SHA3

[5] Hash functions have other properties but a discussion is beyond the scope of this article.

A variant of a hash is a hash with a secret obtained from a function called *keyed-Hash Message Authentication Code* or HMAC. In this type of function, the hash is computed from a message or a block and a secret key is used as a parameter to the computation. The HMAC can only be computed by another party if the key is shared.

Key derivation functions

Keys for symmetric algorithms shall be random, or shall not be generated from an easy to guess sequence. It is close to impossible to request the user to directly provide a 16 or 32 byte (128 or 256 bit) key and it is much better to use a key derivation function taking a password as a parameter.

The two most known functions to derive keys are likely PBKDF2 and Argon2, the latter being an emerging standard.

These functions require a password and a "salt" as input. The "salt" is a value used to increase the entropy and make an exhaustive password search a lot more complicated for an attacker.

A key derivation function can also be used to protect a password: to this end, it is necessary to store the resulting value as well as the salt then to compare, for instance for each connection attempt, the computed value for the user entered password with the stored value (using the same salt in both cases indeed) and then grant or deny access to the system or application.

```
argon2 := TArgon2KeyDerivation.Create;

argon2.OutputFormat := base32;

argon2.OutputSizeBytes := 16;

argon2.Counter := 10;

argon2.Memory := 16;

argon2.StringSalt := '0123456789012345';

argon2.Unicode := yesUni;

k := argon2.GenerateKey('password123:'); // the user could be requested to enter a password using a dialog box
```

[EXAMPLE 4] ARGON2 use

Random number generators (RNGs)

Cryptography would be much less useful without random number generators (RNG). As it is extremely difficult to generate truly random numbers with software[6], many specific functions have been developed over time and some RNGs are considered cryptographically secure. This is the case of Fortuna, implemented by Microsoft in their CryptoAPI and functions such as CryptGenRandom.

[6] We don't address hardware RNGs and True RNGs for which specific technologies are used.

Unix and Linux like operating systems provide the /dev/random and /dev/urandom devices that can be read to get a random number. The difference between these two devices is their response time (the former is blocking) and there is no evidence showing that one is more secure than the other.

Another family of PRNGs use a symmetric algorithm with a shared initial key between two or more participants. The partners just need to agree on an initial key to start the generator (altogether with an IV) and to use a counter to know the correct "random" key to be used at any given time.

```
// RandomUBuffer:  return a random Buffer
// Input : unsigned int len (the length of the Buffer)
// unsigned char* MyBuffer (the output Buffer)
// Output : code of error
// Windows only
int RandomUBuffer(unsigned int len, unsigned char* MyBuffer) {
	int err;
	HCRYPTPROV hProvider = 0;

	if (!CryptAcquireContext(&hProvider, 0, 0, PROV_RSA_FULL,
		CRYPT_VERIFYCONTEXT | CRYPT_SILENT)) {
		err = ER_ERROR_RANDOM_INT_1;
		return err;
	}

	if (!CryptGenRandom(hProvider, (DWORD)len, (BYTE*)MyBuffer)) {
		CryptReleaseContext(hProvider, 0);
		err = ER_ERROR_RANDOM_INT_1;
		return err;
	}
	if (!CryptReleaseContext(hProvider, 0)) {
		err = ER_ERROR_RANDOM_INT_1;
		return err;
	}
	return ER_SUCCESS;
}
```

[EXAMPLE 5] use of CryptGenRandom in C on Windows

```
// RandomUBuffer:  return a random Buffer
// Input : unsigned int len (the length of the Buffer)
// unsigned char* MyBuffer (the output Buffer)
// Output : code of error
// OSX, Unix, Linux
int RandomUBuffer(unsigned int len, unsigned char* MyBuffer) {
	int err;
#include <stdio.h>

	bool test = false;
```

```
        int compt = 0;
        int fHandle = open("/dev/urandom", O_RDONLY);
        int result;
        if (fHandle == -1) {
                // error, unable to read /dev/urandom
                err = ER_ERROR_RANDOM_INT_1;
                return err;
        }

        result = read(fHandle, MyBuffer, len);
        if (result == len) {
                test = true;
        }
        compt += result;
        while (test == false) {
                result = read(fHandle, MyBuffer + compt, len - compt);
                compt += result;
                if (compt >= len) {
                        test = true;
                }
        }
        close(fHandle);
        return ER_SUCCESS;
}
```

[EXAMPLE 6] use of /dev/urandom on OSX/UNIX/Linux

RNG output is used to generate symmetric keys, usually after being processed by a hash function, as well as in the generation process of asymmetric keys (to generate prime numbers for instance).

Key management

With the assumption that the cryptographic algorithms are secure and when their implementation is correct, an adversary is not left with many options to conduct an attack that can be successful in recovering the clear text. This adversary must either take control of the keys or guess them.

To prevent an attacker from guessing keys, we need to use a good PRNG and to ensure proper key management (and associated certificates), we need to use a Public Key Infrastructure (PKI), and a key exchange algorithm for symmetric keys.

Key and hash size recommendations

When computing power allows it, it is recommended to use the maximum possible key size for a given algorithm, e.g., 256 bits for the AES.

The case of RSA is more problematic because the key size has a very significant impact of speed, but a minimum size of 2048 bits is recommended.

www.keylength.com provides a comparison of recommendations on key and hash sizes to protect data over a selected period of time.

Method	Date	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash
[1] Lenstra / Verheul	2017	83	1717 1344	147	1717	157	166
[2] Lenstra Updated	2017	80	1300 1435	159	1300	159	159
[3] ECRYPT II	2016 - 2020	96	1776	192	1776	192	192
[4] NIST	2016 - 2030	112	2048	224	2048	224	224
[5] ANSSI	2014 - 2020	100	2048	200	2048	200	200
[6] IAD-NSA	-	256	3072	-	-	384	384
[7] RFC3766	-	-	-	-	-	-	-
[8] BSI	2017 - 2022	128	2000	250	2000	250	256

Figure 1: key and hash size comparison

Note that TMS Cryptography Pack authorizes all figure 2 cases and caters for the maximum sizes proposed in the recommendations of figure 1 and very often way beyond, except for SPECK that, by design, has a maximum key size of 128 bits (but bear in mind it was designed for the IoT and low power chips).

	AES	SPECK	SALSA20	RSA	EC25519	EC511187	SHA2	SHA3	BLAKE2
Type	sym.	sym.	sym.	asym.	asym.	asym	hash	hash	hash
Min	128	64	256	2048	255	511	256	256	256
Max	256	128	256	4096	255	511	512	1024	512
Recommendation (max.)	256	256	256	3072	250	250	384	384	384

Figure 2: key and hash sizes as enables in TMS Cryptography Pack

Conclusion

Cryptographic algorithm design and implementation are complex matters and require solid math skills as well as experience in attack techniques on different types of algorithms. The use of proven standards such as the one highlighted in this article is more than recommended and it is of utmost importance to stay away from homemade algorithms, except for specific organizations.

Similarly, the implementation and concrete use of algorithms is not exempt of traps and errors of all sorts. It is therefore necessary to follow a few basic rules to properly code cryptographic algorithms. A developer shall at least:

- Check that the algorithm passes all test cases from the standard if they exist (FIPS, RFC, reference implementation)
- Complement these tests with specific tests (empty string, empty file, strings containing nil values, huge file, etc.)
- Clean up internal variables as soon as possible and before they are freed in the code

- Test upper and lower input bounds
- Etc.

With respect to the use of functions implementing these algorithms in software, a developer needs to:

- use an algorithm in the correct context and with the appropriate mode
- correctly initialize the relevant parameters
- clean up variables containing secret values after use
- regularly change encryption keys
- revoke compromised certificates (lost, stolen or possibly with “broken” keys)
- constrain every user input
- validate every output (check what can be checked)
- anticipate and address code failures
- etc.