The Long Journey from Papers to Software: Crypto APIs

Pascal Junod
HES-SO / HEIG-VD

IACR School on Design and Security of Cryptographic Algorithms and Devices
October 18-23, 2015 - Sardinia / Italy

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Agenda

- Introduction
- Security models and primitives
- Choice of security parameters / configuration
- Generation of parameters
- Conclusion

Introduction

At least 99.99999857% of the users of crypto libraries are not djb.

Yet, many people use crypto libraries on a daily basis.

They often do it in a wrong way.

Why?

What this talk is *not* about

- Blaming arrogant **cryptographers** that do not know or do not care about programming and software development

- Trashing ignorant crypto libraries **developers** that do not know everything about cryptography

- Laughing at 99.9995% of crypto libraries **users** that do horrible mistakes most of the time

- Security of software cryptographic implementations with respect to grey- and white-box adversaries

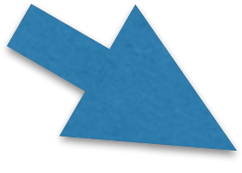
What this talk is about

- Exposing a real problem that arises every day in practice: the lack of security-related usability in most cryptographic libraries
- Showing a sample of today's good and bad practices
- Sketch usability requirements on cryptographic libraries

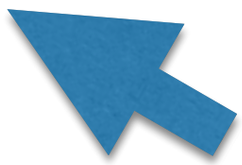
Security of a Cryptographic Library

Security Model

Crypto Primitives



Good crypto design



Security Parameter

Crypto Parameters

Security Models and Crypto Primitives

Security Models

- In the private-key setting, cryptographers are used to deal with different security models:
  - Indistinguishability in the presence of an eavesdropper
  - Indistinguishability under a CPA
  - Indistinguishability under a CCA

Security Models

- In practice, facing a **passive** adversary **only** is **extremely infrequent**.

- Very likely, what one is looking for is **authenticated encryption** (or a **MAC** if no confidentiality is required)

- Two reasonable options for a user:

  - Use an authenticated encryption mode equipped with a proper block cipher (e.g. AES-GCM)

  - Use an encrypt-then-authenticate construction (e.g. AES-CTR + CBC-MAC-AES + HKDF)

What Could Go Wrong?

- Use AES in ECB mode
- Use AES in CBC mode with constant IV
- Use AES in CBC mode with random IV
- Use AES in CBC mode with random IV, CBC-MAC on the ciphertext, all with the same key
- etc.

Facts

- Many SW crypto libraries offer the possibility to use:
  - insecure ciphers (e.g. DES, RC4, …);
  - secure ciphers without modes of operations;
  - secure ciphers with insecure modes of operations (e.g. AES-ECB);

Facts

- Many SW crypto libraries offer the possibility to use:
  - secure ciphers with secure modes of operation, but with insecure parameters (e.g. AES-CBC with constant IV);
  - secure ciphers with secure modes of operation, but without authentication (e.g. AES-CBC with random IV);
  - the same key for ciphering and authentication;
  - etc.

How it Should Work?

- Offer only **high-level** APIs (like `data_seal()`, `data_encrypt()`) providing:
  - secure choice of security model (i.e. authenticated encryption);
  - secure choice of primitives;
  - secure handling of IVs and keys.

How it Should Work?

- If willing to offer more capabilities:
  - Force the developper to build library with **`-DUSE_LOW_LEVEL_API`** for having access to CPA-secure modes of operations.
  - Warn at compilation time and in debug configuration for every use of a low-level API.

How it Should Work?

In Practice: Botan

Ciphers and cipher modes

- Authenticated cipher modes EAX, OCB, GCM, SIV, CCM, and ChaCha20Poly1305
- Unauthenticated cipher modes CTR, CBC, XTS, CFB, OFB, and ECB
- AES (including constant time SSSE3 and AES-NI versions)
- AES candidates Serpent, Twofish, MARS, CAST-256, RC6
- Stream ciphers Salsa20/XSalsa20, ChaCha20, and RC4
- DES, 3DES and DESX
- National/telecom block ciphers SEED, KASUMI, MISTY1, GOST 28147
- Other block ciphers including Threefish-512, Blowfish, CAST-128, IDEA, Noekeon, TEA, XTEA, RC2, RC5, SAFER-SK
- Large block cipher construction Lion

Is more algos really better?

Hash functions and MACs

- SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512
- SHA-3 winner Keccak-1600
- SHA-3 candidate Skein-512
- Authentication codes HMAC, CMAC, Poly1305, SipHash
- RIPEMD-160, RIPEMD-128, Tiger, Whirlpool
- Hash function combiners (Parallel and Comb4P)
- National standard hashes HAS-160 and GOST 34.11
- Non-cryptographic checksums Adler32, CRC24, CRC32
- Obsolete algorithms MD5, MD4, MD2, CBC-MAC, X9.19 DES-MAC

In Practice: `libsodium`

```c
#define MESSAGE ((const unsigned char *) "test")
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_secretbox_MACBYTES + MESSAGE_LEN)

unsigned char nonce[crypto_secretbox_NONCEBYTES];
unsigned char key[crypto_secretbox_KEYBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];

randombytes_buf(nonce, sizeof nonce);
randombytes_buf(key, sizeof key);
crypto_secretbox_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
if (crypto_secretbox_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce, key) != 0) {
    /* message forged! */
}
```

Algorithms:

- Chacha20 + Poly1305 MAC
- HMAC-SHA512/256

```c
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char key[crypto_auth_KEYBYTES];
unsigned char mac[crypto_auth_BYTES];

randombytes_buf(key, sizeof key);
crypto_auth(mac, MESSAGE, MESSAGE_LEN, key);

if (crypto_auth_verify(mac, MESSAGE, MESSAGE_LEN, key) != 0) {
    /* message forged! */
}
```

Facts

- Many SW crypto libraries offer the possibility to use:
  - non-cryptographic algorithms (e.g. CRC32, ADLER, Mersenne Twister)
  - insecure ciphers/algorithms (e.g. DES, RC4, MD5, SHA1, insecure variants of CBC-MAC);
  - insecure protocols (e.g. SSL 2.0, SSL 3.0)
  - insecure cipher suites

How it Should Work?

- If really willing to offer more capabilities:
  - Force the developer to build lib with `-DUSE_WEAK_CIPHERS` for having access to weaker or obsolete ciphers and protocols (64-bit blocks, RC4, DES, SSL 2.0, 3.0, etc.)
  - Additionally, warn at compilation time and in debug configuration at every use of a weak cipher, hash, MAC protocol or non-cryptographic function

Idea of Self-Defending Primitives

- Triple-DES: self-defending implementation should check that `!(k1 == k2 == k3)`

```
/*
* TripleDES Key Schedule
*/
void TripleDES::key_schedule(const byte key[], size_t length)
   {
   des_key_schedule(&round_key[0], key);
   des_key_schedule(&round_key[32], key + 8);

   if(length == 24)
      des_key_schedule(&round_key[64], key + 16);
   else
      copy_mem(&round_key[64], &round_key[0], 32);
   }

}
```

Idea of Self-Defending Primitives

- Other example: PyCrypto counter mode API

```
def new(nbits, prefix=b(""), suffix=b(""), initial_value=1, overflow=0, little_endian=False,
        allow_wraparound=False, disable_shortcut=_deprecated):
    """Create a stateful counter block function suitable for CTR encryption modes.

Each call to the function returns the next counter block.
Each counter block is made up by three parts::

  prefix || counter value || postfix

The counter value is incremented by 1 at each call.
```

Idea of Self-Defending Primitives

```
:Parameters:
  nbits : integer
    Length of the desired counter, in bits. It must be a multiple of 8.
  prefix : byte string
    The constant prefix of the counter block. By default, no prefix is
    used.
  suffix : byte string
    The constant postfix of the counter block. By default, no suffix is
    used.
initial_value : integer
  The initial value of the counter. Default value is 1.
overflow : integer
  This value is currently ignored.
little_endian : boolean
  If *True*, the counter number will be encoded in little endian format.
  If *False* (default), in big endian format.
allow_wraparound : boolean
  If *True*, the counter will automatically restart from zero after
  reaching the maximum value (``2**nbits-1``).
  If *False* (default), the object will raise an *OverflowError*.
disable_shortcut : deprecated
  This option is a no-op for backward compatibility.  It will be removed
  in a future version.  Don't use it.
```

Idea of Self-Defending Primitives

Open **cryptopathe** opened this issue on 14 Nov 2014 · 1 comment

cryptopathe commented on 14 Nov 2014

Dear pyCrypto developpers,

I can't figure out any useful situation where the `allow_wraparound` flag defined in `lib/Crypto/Util/Counter.py` would be useful in practice. Instead, allowing the counter to wrap in the CTR mode can open a serious security issue (one can XOR the portions of the ciphertext generated with the same counter values, and one immediately gets the XOR of the corresponding parts of the plaintexts, which is a serious problem for non-random plaintexts).

If you split the CTR initial value into a nonce and a random initial counter value, and that you get exceptions because of the counter wrapping around, then it is because your counter has probably a too small bit width for your application. With a proper counter size, this situation should only happen with a probability that is negligible in practice.

I have quickly searched on Github examples where allow_wraparound would be set to True, and found none.

In summary, this option, introduced in 2.1.0alpha2, can IMHO only help developers to write bad crypto code, and it should be removed.

Idea of Self-Defending Primitives

dlitz commented on 10 Jul — Owner

I think you're probably right.

When I designed `Counter` 7 years ago, I was probably worried about a split nonce being too small and I imagined that people might deal with this by just using a full 128-bit random nonce and relying on their sparse distribution to avoid block collisions. Another reason it might have been useful was to help avoid a timing side-channel in code where authentication happens after decryption.

At this point, though, I don't know of any protocol that actually does either of those things, and the world has moved on to better things like GCM, SIV, and chacha20poly1305. Meanwhile, I *have* seen confusion about the `allow_wraparound` option, and it's also one of those weird PyCrypto-specific options that doesn't exist in other libraries, so it'd probably be wise to just drop it.

Thanks!

How it Should Work?

Offer only capabilities that are useful for developers and that, at the same time, do not allow him/her to generate security-related mistakes.

Choice of Security Parameter

Idea of Self-Defending Primitives

- Blowfish, RC5, etc.: emit a warning, or better, refuse to key-schedule in normal conditions when the key length is smaller than 80 bits.

- For instance, **`libgcrypt`** implementation of Blowfish key-schedule does a self-test, looks for weak keys, but allows 32-bit keys.

RSA

- Most libraries don’t tell anything when you generate an RSA key with a too small key length.
- In 2015, we know that 1024 bits is a strict minimum only valid for data with a very short life, and that we should use at least 1536-bit keys, better 2048-bit ones.
- Yet…

RSA

```
nowhere:apps pjunod$./openssl version

OpenSSL 1.0.2d 9 Jul 2015

nowhere:apps pjunod$ ./openssl genrsa

Generating RSA private key, 2048 bit long modulus

.....................................................................
.....................................+++..+++

e is 65537 (0x10001)

-----BEGIN RSA PRIVATE KEY-----

[…]
```

RSA

```
nowhere:apps pjunod$ openssl version

OpenSSL 0.9.8zg 14 July 2015

nowhere:apps pjunod$ openssl genrsa

Generating RSA private key, 512 bit long modulus

..............++++++++++++

.......++++++++++++

e is 65537 (0x10001)

-----BEGIN RSA PRIVATE KEY-----

[…]
```

RSA

```
nowhere:apps pjunod$ ./openssl genrsa 128

Generating RSA private key, 128 bit long modulus

..++++++++++++++++++++++++++++

e is 65537 (0x10001)

-----BEGIN RSA PRIVATE KEY-----

MGMCAQACEQDCz4LBaILQw62vAnvJcGYXAgMBAAECEA3z+vOLXsNBALIVFQBUmLEC

CQDkWMf0wgBiuwIJANpnCdIa3/pVAgkAqaQLaRp3juECCHWEfARR8bMBAgkAzEYS

ZrcEG/w=

-----END RSA PRIVATE KEY-----
```

RSA

```
nowhere:apps pjunod$ ./openssl genrsa 64

Generating RSA private key, 64 bit long modulus

.++++++++++++++++++++++++++++++

.++++++++++++++++++++++++++++

e is 65537 (0x10001)

-----BEGIN RSA PRIVATE KEY-----

MD4CAQACCQDFAmoNN7zQsQIDAQABAgkAukcZG/LFpR0CBQDvmYDHAgUA0n6axwIE

ANgyqwIEX7qeTwIEZw+/Gw==

-----END RSA PRIVATE KEY-----
```

RSA

```
nowhere:apps pjunod$ ./openssl genrsa 32

Generating RSA private key, 32 bit long modulus

.++++++++++++++++++++++++++++++

.++++++++++++++++++++++++++++++

e is 65537 (0x10001)

-----BEGIN RSA PRIVATE KEY-----

MCwCAQACBQDSdAJdAgMBAAECBFjnEKkCAwD6BwIDANd7AgIYfwICMeMCAwCMFQ
==

-----END RSA PRIVATE KEY-----
```

RSA

```
openssl:openssl pjunod$ pwd

/Users/pjunod/data/science/work/crypto-APIS/libs/libressl/libressl-2.3.0/
apps/openssl

eduroam-3-153:openssl pjunod$ ./openssl genrsa 32
Generating RSA private key, 32 bit long modulus
.++++++++++++++++++++++++++++
.++++++++++++++++++++++++++++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MCsCAQACBQCcGCAdAgMBAAECBAv8q50CAwDL/wIDAMPjAgIFswICWYkCAkV5
-----END RSA PRIVATE KEY-----
```

RSA / OpenSSL

```
for (;;) {
    /*
     * When generating ridiculously small keys, we can get stuck
     * continually regenerating the same prime values. Check for this and
     * bail if it happens 3 times.
     */
    unsigned int degenerate = 0;
do {
    if (!BN_generate_prime_ex(rsa->q, bitsq, 0, NULL, NULL, cb))
        goto err;
} while ((BN_cmp(rsa->p, rsa->q) == 0) && (++degenerate < 3));
if (degenerate == 3) {
    ok = 0;                 /* we set our own err */
    RSAerr(RSA_F_RSA_BUILTIN_KEYGEN, RSA_R_KEY_SIZE_TOO_SMALL);
    goto err;
}
```

RSA / Botan

```
#include <botan/keypair.h>
#include <botan/internal/assert.h>

namespace Botan {

/*
* Create a RSA private key
*/
RSA_PrivateKey::RSA_PrivateKey(RandomNumberGenerator& rng,
                               size_t bits, size_t exp)
   {
   if(bits < 512)
      throw Invalid_Argument(algo_name() + ": Can't make a key that is only " +
                             to_string(bits) + " bits long");
   if(exp < 3 || exp % 2 == 0)
      throw Invalid_Argument(algo_name() + ": Invalid encryption exponent");
```

RSA / Crypto++

```
void InvertibleRSAFunction::GenerateRandom(RandomNumberGenerator &rng, const NameValuePairs &alg)
{
        int modulusSize = 2048;
        alg.GetIntValue(Name::ModulusSize(), modulusSize) || alg.GetIntValue(Name::KeySize(), modulusSize);

        if (modulusSize < 16)
                throw InvalidArgument("InvertibleRSAFunction: specified modulus size is too small");

        m_e = alg.GetValueWithDefault(Name::PublicExponent(), Integer(17));

        if (m_e < 3 || m_e.IsEven())
                throw InvalidArgument("InvertibleRSAFunction: invalid public exponent");
```

RSA / libgcrypt

```
static gpg_err_code_t
generate_std (RSA_secret_key *sk, unsigned int nbits, unsigned long use_e,
              int transient_key)
{
  gcry_mpi_t p, q; /* the two primes */
  gcry_mpi_t d;    /* the private key */
  gcry_mpi_t u;
  gcry_mpi_t t1, t2;
  gcry_mpi_t n;    /* the public key */
  gcry_mpi_t e;    /* the exponent */
  gcry_mpi_t phi;  /* helper: (p-1)(q-1) */
  gcry_mpi_t g;
  gcry_mpi_t f;
  gcry_random_level_t random_level;
```

```
if (fips_mode ())
  {
    if (nbits < 1024)
      return GPG_ERR_INV_VALUE;
    if (transient_key)
      return GPG_ERR_INV_VALUE;
  }

/* The random quality depends on the transient_key flag.  */
random_level = transient_key ? GCRY_STRONG_RANDOM : GCRY_VERY_STRONG_RANDOM;

/* Make sure that nbits is even so that we generate p, q of equal size. */
if ( (nbits&1) )
  nbits++;

if (use_e == 1)   /* Alias for a secure value */
  use_e = 65537;  /* as demanded by Sphinx. */

/* Public exponent:
   In general we use 41 as this is quite fast and more secure than the
   commonly used 17.  Benchmarking the RSA verify function
   with a 1024 bit key yields (2001-11-08):
   e=17    0.54 ms
   e=41    0.75 ms
```

RSA / wolfcrypt

```
nowhere:src pjunod$ pwd

/Users/pjunod/data/science/work/crypto-APIS/libs/
wolfSSL/wolfssl-3.6.8/wolfcrypt/src

nowhere:src pjunod$ grep RSA_MIN_SIZE *.c

rsa.c:    RSA_MIN_SIZE = 512,

rsa.c:    if (size < RSA_MIN_SIZE || size >
RSA_MAX_SIZE)

nowhere:src pjunod$ grep RSA_MAX_SIZE *.c

rsa.c:    RSA_MAX_SIZE = 4096,
```

Other Public Key Algos

- `./openssl gendh 128`
- `./openssl ecparams -list_curves`

```
secp112r1, secp112r2, secp128r1, secp128r2,
sect113r1, sect113r2, sect131r1, sect131r2,
wap-wsg-idm-ecid-wtls1, wap-wsg-idm-ecid-
wtls4, wap-wsg-idm-ecid-wtls6, wap-wsg-idm-
ecid-wtls8,   Oakley-EC2N-3, Oakley-EC2N-4,
…
```

- SSL/TLS obsolete or weak cipher suites

RSA / Keyczar

```
// static
RSAPrivateKey* RSAPrivateKey::GenerateKey(int size) {
  if (!KeyType::IsValidCipherSize(KeyType::RSA_PRIV, size))
    return NULL;
```

```
static const int kAESSizes[] = {128, 192, 256, 0};
#ifdef COMPAT_KEYCZAR_06B
static const int kHMACSHA1Sizes[] = {256, 0};
static const int kRSASizes[] = {512, 768, 1024, 2048, 3072, 4096, 0};
#else
static const int kHMACSizes[] = {160, 224, 256, 384, 512, 0};
static const int kRSASizes[] = {1024, 2048, 3072, 4096, 0};
#endif
static const int kDSASizes[] = {1024, 2048, 3072, 0};
static const int kECDSASizes[] = {192, 224, 256, 384, 0};
```

How it Should Work?

- Offer only security parameters, configuration options that are **meaningful** in terms of security!

- Offer **safe** values by default, always!

- If really required, weaker/obsolete variants could be made available, but requiring an **effort** from the developper, emitting warnings at compilation time, etc.

Generation of Non-Secret Parameters

Parameters

- Cryptographic algorithms and implementations use parameters all the time, that are secret or not:
  - Keys
  - IVs
  - Nonces

Parameters

- An issue of tremendous importance is the availability of cryptographically secure random numbers.

- Most cryptographic libraries do a rather good job in providing a secure CPRNG to the developer.

- However, most cryptographic libraries put **the responsibility to use the CPRNG** on the developer shoulders.

Parameters

- If the responsibility to use a CPRNG is left to the developers, two bad things can happen:
  - Forget to use random values
  - Use a non-cryptographic PRNG

Parameters

If you use the code in the authenticated encryption example, be sure each message gets a unique IV. The `DeriveKeyAndIV` produces predictable IVs for demonstration purposes, but it violates semantic security because two messages under the same key will produce the same ciphertext.

If you choose to generate a random IV and append it to the message, be sure to authenticate the {IV,Ciphertext} pair.

Parameters

- For instance:

```c
#define MESSAGE ((const unsigned char *) "test")
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_secretbox_MACBYTES + MESSAGE_LEN)

unsigned char nonce[crypto_secretbox_NONCEBYTES];
unsigned char key[crypto_secretbox_KEYBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];

randombytes_buf(nonce, sizeof nonce);
randombytes_buf(key, sizeof key);
crypto_secretbox_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
if (crypto_secretbox_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce, key) != 0) {
    /* message forged! */
}
```

Generating the nonce is the user's responsibility.

The nonce doesn't have to be confidential, but it should never ever be reused with the same key. The easiest way to generate a nonce is to use `randombytes_buf()`.

Parameters

GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One C/B

Shay Gueron[1] and Yehuda Lindell[2]

Trivial Nonce-Misusing Attack on Pure OMD

Tomer Ashur and Bart Mennink

Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance

Viet Tung Hoang[1,2] Reza Reyhanitabar[3] Phillip Rogaway[4] Damian Vizár[3]

[1] Dept. of Computer Science, Georgetown University, USA
[2] Dept. of Computer Science, University of Maryland, College Park, USA
[3] EPFL, Lausanne, Switzerland
[4] Dept. of Computer Science, University of California, Davis, USA

Parameters

Furthermore, it is well known that ECDSA's session keys are much less tolerant than the long-term key of slight deviations from randomness, even if the session keys are not revealed or reused. For example, Nguyen and Shparlinski in [61] presented an algorithm using lattice methods to compute the long-term ECDSA key from the knowledge of as few as 3 bits of r for hundreds of signatures, whether this knowledge is gained from side-channel attacks or from non-uniformity of the distribution from which r is taken.

EdDSA avoids these issues by generating $r = H(h_b, \ldots, h_{2b-1}, M)$, so that different messages will lead to different, hard-to-predict values of r. No per-message randomness is consumed. This idea of generating random signatures in a secretly deterministic way, in particular obtaining pseudorandomness by hashing a long-term secret key together with the input message, was proposed by Barwood in [9]; independently by Wigley in [79]; a few months later in a patent application [57] by Naccache, M'Raïhi, and Levy-dit-Vehel; later by M'Raïhi, Naccache, Pointcheval, and Vaudenay in [55]; and much later by Katz and Wang in [47]. The patent application was abandoned in 2003.

Parameters

Signing a string is as follows. Note that a PRNG is required because the Digital Signature Standard specifies a per-message random value.

```
ECDSA<ECP, SHA1>::PrivateKey privateKey;
privateKey.Load(...);
ECDSA<ECP,SHA1>::Signer signer( privateKey );

AutoSeededRandomPool prng;
string message = "Yoda said, Do or do not. There is no try.";
string signature;

StringSource s( message, true /*pump all*/,
    new SignerFilter( prng,
        signer,
        new StringSink( signature )
    ) // SignerFilter
); // StringSource
```

Generating the nonce is the API's responsibility.

```
sig = ECDSA_do_sign(digest, 20, eckey);
if (sig == NULL)
        {
        /* error */
        }
```

```
DSA_PrivateKey* dsakey = dynamic_cast<DSA_PrivateKey*>(key.get());

if(!dsakey)
   {
   std::cout << "The loaded key is not a DSA key!" << std::endl;
   return 1;
   }

PK_Signer signer(*dsakey, "EMSA1(SHA-1)");

DataSource_Stream in(message);
byte buf[4096] = { 0 };
while(size_t got = in.read(buf, sizeof(buf)))
   signer.update(buf, got);

sigfile << base64_encode(signer.signature(rng)) << std::endl;
```

```c
int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *aad,
        int aad_len, unsigned char *key, unsigned char *iv,
        unsigned char *ciphertext, unsigned char *tag)
{
        EVP_CIPHER_CTX *ctx;

        int len;

        int ciphertext_len;


        /* Create and initialise the context */
        if(!(ctx = EVP_CIPHER_CTX_new())) handleErrors();

        /* Initialise the encryption operation. */
        if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, NULL, NULL))
                handleErrors();

        /* Set IV length if default 12 bytes (96 bits) is not appropriate */
        if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_IVLEN, 16, NULL))
                handleErrors();

        /* Initialise key and IV */
        if(1 != EVP_EncryptInit_ex(ctx, NULL, NULL, key, iv)) handleErrors();

        /* Provide any AAD data. This can be called zero or more times as
         * required
         */
        if(1 != EVP_EncryptUpdate(ctx, NULL, &len, aad, aad_len))
                handleErrors();

        /* Provide the message to be encrypted, and obtain the encrypted output.
         * EVP_EncryptUpdate can be called multiple times if necessary
         */
        if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len))
                handleErrors();
        ciphertext_len = len;

        /* Finalise the encryption. Normally ciphertext bytes may be written at
         * this stage, but this does not occur in GCM mode
         */
        if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len)) handleErrors();
        ciphertext_len += len;

        /* Get the tag */
        if(1 != EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, 16, tag))
                handleErrors();

        /* Clean up */
        EVP_CIPHER_CTX_free(ctx);

        return ciphertext_len;
}
```

How it Should Work?

In encryption mode, the IVs/nonces should, **whenever possible**, be generated in a transparent way and **returned** by the API, as it is done for public-key crypto.

Error Handling

Error Handling

RETURN VALUES

ECDSA_size() returns the maximum length signature or 0 on error.

ECDSA_sign_setup() and ECDSA_sign() return 1 if successful or 0 on error.

ECDSA_verify() and ECDSA_do_verify() return 1 for a valid signature, 0 for an invalid signature and -1 on error. The error codes can be obtained by ERR_get_error(3).

```
if (!ECDSA_sign (…)) { handle_error (); }

if (ECDSA_verify (…)) { /* good signature */…}
```

Error Handling

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
unsigned char sk[crypto_sign_SECRETKEYBYTES];
crypto_sign_keypair(pk, sk);

unsigned char signed_message[crypto_sign_BYTES + MESSAGE_LEN];
unsigned long long signed_message_len;

crypto_sign(signed_message, &signed_message_len,
            MESSAGE, MESSAGE_LEN, sk);

unsigned char unsigned_message[MESSAGE_LEN];
unsigned long long unsigned_message_len;
if (crypto_sign_open(unsigned_message, &unsigned_message_len,
                     signed_message, signed_message_len, pk) != 0) {
    /* Incorrect signature! */
}
```

Errors Handling

```
nowhere:rsa pjunod$ grep ERR_REASON
rsa_err.c | wc -l

      68
```

How it Should Work?

- In debug mode, a crypto library should return detailed error messages.

- In production mode, a crypto library should return two types of return value: OK / NOK.

- Error handling should be as simple as possible.

Conclusion

Conclusions

- **In a crypto library, a non-secure functionality is a superfluous functionality that must be killed.**

- **Crypto libraries should stop assuming that its crypto library users are omniscient cryptographers. 99.999% of them are not.**

- **In consequence, crypto libraries should stop put security-related responsibilities on developers' shoulders as much as possible.**

Conclusions

- **A crypto library should categorically refuse to generate too small keys or cryptographic groups.**
- **A crypto library should refuse with all possible energy to use obsolete/weak cryptographic configurations.**
- **Whenever possible, a crypto library should generate itself the required IVs/nonces and return them to the user.**
- **A crypto library should have the cleanest and simplest possible error handling mechanism.**

The perfect cryptographic library does not exist yet, even if a few of them converge in the good direction!

The Long Journey from Papers to Software: Crypto APIs

Pascal Junod
HES-SO / HEIG-VD

IACR School on Design and Security of Cryptographic Algorithms and Devices
October 18-23, 2015 - Sardinia / Italy

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland