

Towards Developer-Proof Cryptography

Pascal Junod



EPFL-SURI, Lausanne (Switzerland), June 20th, 2016

Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

Motivations

- Dozens of open- or closed-source crypto libraries are available.
- Many people use [crypto libraries](#) on a daily basis.
- Most of the time, the final result **does not reach the expected cryptographic strength**.
- In average, I need 10-15' to identify a problematic piece of open-source cryptographic software that I can transform in an exam problem for my "Industrial Cryptography" lecture.

Not Convinced?

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

M. Egele et al. *An Empirical Study of Cryptographic Misuse in Android Applications*, ACM-CCS 2012.

Not Convinced?

ABSTRACT

Developers use cryptographic APIs in Android with the intent of securing data such as passwords and personal information on mobile devices. In this paper, we ask whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security, e.g., IND-CPA security. We develop program analysis techniques to automatically check programs on the Google Play marketplace, and find that 10,327 out of 11,748 applications that use cryptographic APIs – 88% overall – make at least one mistake. These numbers show that applications do not use cryptographic APIs in a fashion that maximizes overall security. We then suggest specific remediations based on our analysis toward improving overall cryptographic security in Android applications.

Not Convinced?

Rule 1: Do not use ECB mode for encryption. [6]

Rule 2: Do not use a non-random IV for CBC encryption. [6, 23]

Rule 3: Do not use constant encryption keys.

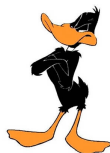
Rule 4: Do not use constant salts for PBE. [2, 5]

Rule 5: Do not use fewer than 1,000 iterations for PBE. [2, 5]

Rule 6: Do not use static seeds to seed `SecureRandom(·)`.

M. Egele et al. *An Empirical Study of Cryptographic Misuse in Android Applications*, ACM-CCS 2012.

This Talk is not about...



... blaming **cryptographers** that do not know or do not care about programming and software development issues.

This Talk is not about...



... trashing **crypto libraries developers** that do not know everything about cryptography.

This Talk is not about...



... trolling 99.9995% of the **crypto libraries users** that do horrible mistake most of the time.

This Talk is not about...

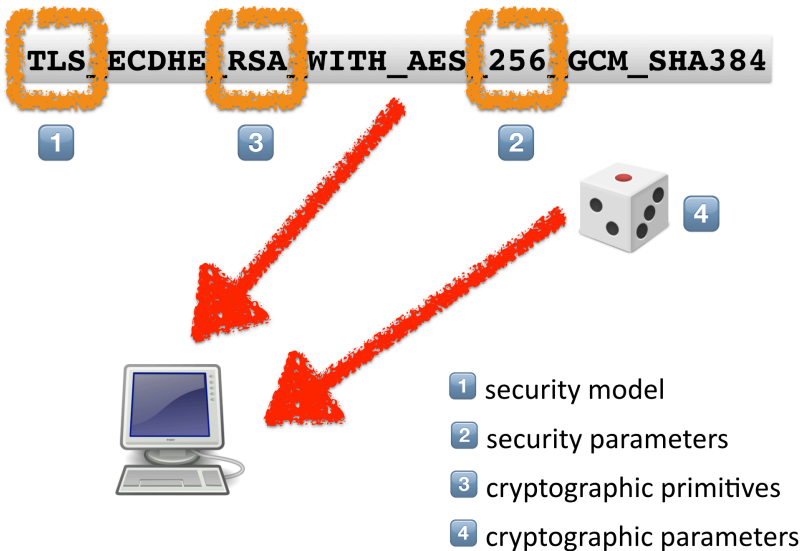


... the **other security aspects** of crypto libraries, namely attacks related to grey- and white-box adversaries: side-channel attacks, SW reverse engineering, exploiting and tampering, etc.

Goals of this Talk

- Exposing a real-world problem that has consequences every day in practice: the **lack of security-related usability** in most cryptographic libraries.
- Identify important interactions between the academic and the real world and their consequences.
- Showing a sample of current **good** and **bad** practices.
- Propose **usability requirements** on cryptographic libraries.

Outline



Outline

- 1 Motivations
- 2 Security Models**
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

Symmetric Encryption Security Models

- Just in the symmetric-key encryption setting, many theoretical security models exist:
 - **IND-CPA**: indistinguishability under chosen-plaintext attacks
 - **IND-CCA**: indistinguishability under chosen-ciphertext attacks
 - **INT-PTXT**: integrity of plaintexts
 - **INT-CTXT**: integrity of ciphertexts
 - **NM-CPA**: non-malleability under chosen-plaintext attacks
 - **NM-CCA**: non-malleability under chosen-ciphertext attacks
 - ...

Security Models for Symmetric Encryption

Composition Method	Privacy			Integrity	
	IND-CPA	IND-CCA	NM-CPA	INT-PTXT	INT-CTXT
<i>Encrypt-and-MAC</i>	insecure	insecure	insecure	secure	insecure
<i>MAC-then-encrypt</i>	secure	insecure	insecure	secure	insecure
<i>Encrypt-then-MAC</i>	secure	insecure	insecure	secure	insecure

Composition Method	Privacy			Integrity	
	IND-CPA	IND-CCA	NM-CPA	INT-PTXT	INT-CTXT
<i>Encrypt-and-MAC</i>	insecure	insecure	insecure	secure	insecure
<i>MAC-then-encrypt</i>	secure	insecure	insecure	secure	insecure
<i>Encrypt-then-MAC</i>	secure	secure	secure	secure	secure

Figure 2: Summary of security results for the composite authenticated encryption schemes. The given encryption scheme is assumed to be IND-CPA for both tables while the given MAC is assumed to be weakly unforgeable for the top table and strongly unforgeable for the bottom table.

Source: M. Bellare and C. Namprepmpre, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, Asiacrypt 2000.

Security Models for Symmetric Encryption

The Order of Encryption and Authentication for Protecting Communications (Or: How Secure is SSL?)*

Hugo Krawczyk**

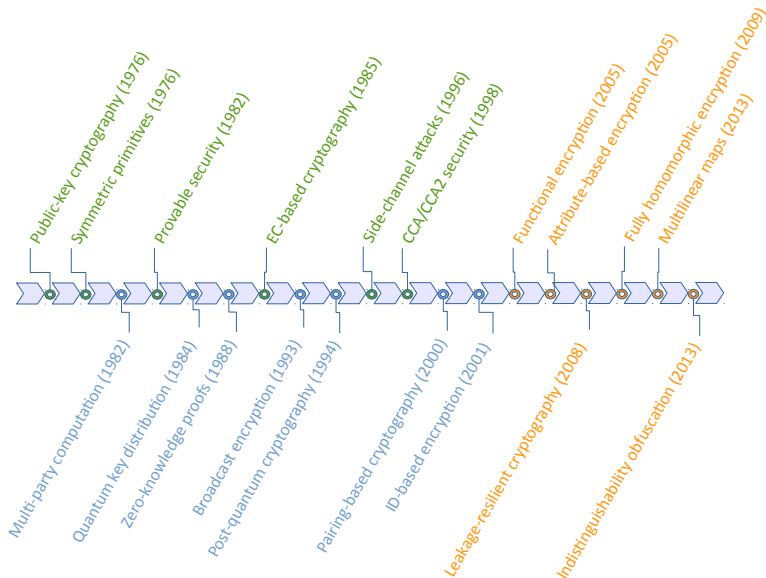
Abstract. We study the question of how to generically compose *symmetric* encryption and authentication when building “secure channels” for the protection of communications over insecure networks. We show that any secure channels protocol designed to work with any combination of secure encryption (against chosen plaintext attacks) and secure MAC must use the encrypt-then-authenticate method. We demonstrate this by showing that the other common methods of composing encryption and authentication, including the authenticate-then-encrypt method used in SSL, are not generically secure. We show an example of an encryption function that provides (Shannon’s) perfect secrecy but when combined with any MAC function under the authenticate-then-encrypt method yields a totally insecure protocol (for example, finding passwords or credit card numbers transmitted under the protection of such protocol becomes an easy task for an active attacker). The same applies to the encrypt-and-authenticate method used in SSH.

On the positive side we show that the authenticate-then-encrypt method is secure if the encryption method in use is either CBC mode (with an underlying secure block cipher) or a stream cipher (that xor the data with a random or pseudorandom pad). Thus, while we show the generic security of SSL to be broken, the current practical implementations of the protocol that use the above modes of encryption are safe.

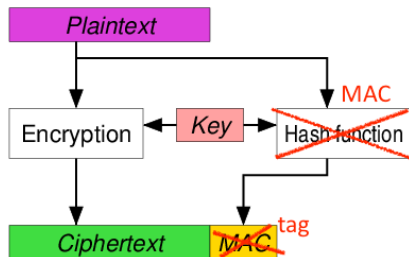
Source: H. Krawczyk, *The Order of Encryption and Authentication for Protecting Communications (Or: How Secure is SSL?)*,

Crypto 2001.

Some Great Crypto Inventions



Generic Composition: Encrypt-and-MAC



Source: Wikipedia

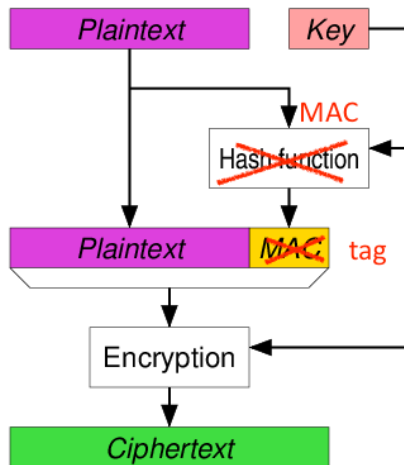
Security of Generic Composition: EaM

Security		Weak MAC		Strong MAC	
		Result	Reason	Result	Reason
Privacy	IND-CPA	Insecure	Proposition 4.1	Insecure	Proposition 4.1
	IND-CCA	Insecure	IND-CPA insecure and IND-CCA \rightarrow IND-CPA	Insecure	IND-CPA insecure and IND-CCA \rightarrow IND-CPA
	NM-CPA	Insecure	IND-CPA insecure and NM-CPA \rightarrow IND-CPA	Insecure	IND-CPA insecure and NM-CPA \rightarrow IND-CPA
Integrity	INT-PTXT	Secure	Theorem 4.3	Secure	Theorems 4.3 and 2.5
	INT-CTXT	Insecure	Proposition 4.4	Insecure	Proposition 4.4

Figure 4: Summary of results for the *Encrypt-and-MAC* composition method.

Source: M. Bellare and C. Namprepmpre, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, *Asiacrypt* 2000.

Generic Composition: MAC-then-Encrypt



Source: Wikipedia

Security of Generic Composition: MtE

Security		Weak MAC		Strong MAC	
		Result	Reason	Result	Reason
Privacy	IND-CPA	Secure	Theorem 4.5	Secure	Theorem 4.5
	IND-CCA	Insecure	NM-CPA insecure and NM-CPA \rightarrow IND-CCA	Insecure	NM-CPA insecure and NM-CPA \rightarrow IND-CCA
	NM-CPA	Insecure	Proposition 4.6	Insecure	Proposition 4.6
Integrity	INT-PTXT	Secure	Theorem 4.5	Secure	Theorems 4.5 and 2.5
	INT-CTXT	Insecure	IND-CPA secure and NM-CPA insecure and INT-CTXT \wedge IND-CPA \rightarrow NM-CPA	Insecure	IND-CPA secure and NM-CPA insecure and INT-CTXT \wedge IND-CPA \rightarrow NM-CPA

Figure 5: Summary of results for the *MAC-then-encrypt* composition method

Source: M. Bellare and C. Namprepre, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, *Asiacrypt* 2000.

MtE vs. SSL/TLS: CVE-2013-0169 / CVE-2016-2107

Filippo Valsorda @FiloSottile · 4 mai

Just published a zero-to-decryption analysis of how and why the OpenSSL padding oracle works [blog.cloudflare.com/yes-another-pa...](https://blog.cloudflare.com/yes-another-padding-oracle/)

303 341

kennyog @kennyog

@FiloSottile Was worried we'd missed this issue in the L13 paper. Turns out we didn't...

Voir la traduction

3. If $t + \text{padlen} + 1 > \text{plen}$, then the plaintext is not long enough to contain the padding (as indicated by the last byte of plaintext) plus a MAC tag. In this case, run a loop as

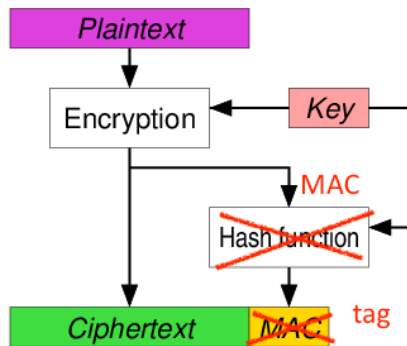
RETWEETS 8 J'AIME 7

18:31 - 4 mai 2016

8 7

Source: Twitter

Generic Composition: Encrypt-then-MAC



Source: Wikipedia

Security of Generic Composition: EtM

Security		Weak MAC		Strong MAC	
		Result	Reason	Result	Reason
Privacy	IND-CPA	Secure	Theorem 4.7	Secure	Theorem 4.9
	IND-CCA	Insecure	NM-CPA insecure and NM-CPA \rightarrow IND-CCA	Secure	Theorem 4.9
	NM-CPA	Insecure	Proposition 4.6	Secure	IND-CCA secure and IND-CCA \rightarrow NM-CPA
Integrity	INT-PTXT	Secure	Theorem 4.7	Secure	INT-CTXT secure and INT-CTXT \rightarrow INT-PTXT
	INT-CTXT	Insecure	IND-CPA secure and NM-CPA insecure and INT-CTXT \wedge IND-CPA \rightarrow NM-CPA	Secure	Theorem 4.9

Figure 6: Summary of results for the *encrypt-then-MAC* composition method

Source: M. Bellare and C. Nampreppe, *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, *Asiacrypt* 2000.

Symmetric Encryption in Practice

- In practice, facing a **passive adversary** only is extremely **infrequent**.
- The only viable symmetric encryption model is **authenticated encryption**, i.e. IND-CCA + unforgeability, than can be achieved by properly combining IND-CPA + INT-CTXT.
- There is only two reasonable (i.e., fool-safe) options:
 - use a well-equipped and secure authenticated encryption mode, like e.g. AES-GCM;
 - use an encrypt-then-authenticate scheme, like e.g. AES-CTR + HMAC-SHA256 + HKDF.

Generic Composition: What could Go Wrong?


- AES in ECB mode;
- AES in CBC mode with constant IV;
- AES in CBC mode with random IV;
- AES in CBC mode with random IV, HMAC-SHA256 on the ciphertext, both using the same key;
- etc.


Back to Software

- Many crypto libraries offer the following options to the developer:
 - secure ciphers without mode of operations;
 - secure ciphers with insecure modes of operations, like AES-ECB, AES-CTR or AES-CBC;
 - secure ciphers with secure modes of operations, but with insecure parameters, like AES-GCM with repeating nonces;
 - using the same key for encryption and authentication when not using authenticated encryption modes;
 - etc.

Do we Really Need Them?

Ciphers and cipher modes

- Authenticated cipher modes EAX, OCB, GCM, SIV, CCM, and ChaCha20Poly1305
- Unauthenticated cipher modes CTR, CBC, XTS, CFB, OFB, and ECB 

algorithm type	name
authenticated encryption schemes	GCM , CCM , EAX
high speed stream ciphers	ChaCha8 , ChaCha12 , ChaCha20 , Panama , Sosemanuk , Salsa20 , XSalsa20
AES and AES candidates	AES (Rijndael), RC6 , MARS , Twofish , Serpent , CAST-256
other block ciphers	IDEA , Triple-DES (DES-EDE2 and DES-EDE3), Camellia , SEED , RC5, Blowfish, TEA, XTEA, Skipjack, SHACAL-2
 block cipher modes of operation	ECB, CBC, CBC ciphertext stealing (CTS), CFB, OFB, counter mode (CTR)
message authentication codes	VMAC , HMAC , GMAC (GCM) , CMAC , CBC-MAC, DMAC, Two-Track-MAC

Dream Crypto Library Requirement #1

**Offer only high-level APIs
implementing authenticated
encryption or sealing.**

Dream Crypto Library Requirement #2

Force the developer to build the library with

- DUSE_LOW_LEVEL_API for having access to other modes; warn at compilation time and in debug mode for every use of an unsecure, low-level API.**

Example of Good Practice: libsodium

```
#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! */
}
```

Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives**
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

The Dark Side of Crypto Diversity

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

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 combinators (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

The Dark Side of Crypto Diversity

- Many crypto libraries offer the option to use:
 - non-cryptographic algorithms (e.g., CRC32, ADLER, Mersenne Twister)
 - insecure ciphers (e.g., DES, RC4, MD5, SHA1, insecure variants of CBC-MAC)
 - insecure protocols (e.g., SSL 2.0, SSL 3.0)
 - insecure ciphersuites (e.g.,
 `TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5`);
 - etc.

Dream Crypto Library Requirement #3

Force the developer to build the library with `-DUSE_WEAK_CIPHERS` for having access to weak or obsolete ciphers; warn at compilation time and in debug mode for every use of an unsecure, low-level API.

Self-Defending Implementations

Example of Triple-DES: a self-defending implementation should check that `!(key[0..7] == key[8..15] == key[16..23])`.

```
/*  
 * 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);  
}  
  
}
```


Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters**
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion

Self-Defending Implementations ?

- Ciphers designed to have a flexible key length (Blowfish, RC5, etc.): emit a warning, or even better, refuse to key-schedule in standard 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 ones.

Self-Defending Implementations ?

- Most libraries don't tell anything when you generate an RSA key with a too short key length.
- In 2016, one knows that 1024 bits is a strict minimum only valid for data with a very short expected lifetime, and that we should use at least 1536-bit keys, better 2048-bit ones.

RSA Case: openssl

```
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 Case: openssl

```
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 Case: openssl

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

Generating RSA private key, **128** bit long modulus

```
..+++++
```

```
..+++++
```

```
e is 65537 (0x10001)
```

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

RSA Case: openssl

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

Generating RSA private key, **64** bit long modulus

```
.+++++
```

```
.+++++
```

```
e is 65537 (0x10001)
```

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

RSA Case: openssl

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

Generating RSA private key, **32** bit long modulus

```
.+++++
```

```
.+++++
```

```
e is 65537 (0x10001)
```

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


RSA Case: 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) && (++denerate < 3));  
    if (denerate == 3) {  
        ok = 0;                /* we set our own err */  
        RSAerr(RSA_F_RSA_BUILTIN_KEYGEN, RSA_R_KEY_SIZE_TOO_SMALL);  
        goto err;  
    }  
}
```

RSA Case: libressl



[LibreSSL 2.4.0](#) released May 31st, 2016

LibreSSL is a version of the TLS/crypto stack forked from [OpenSSL](#) in 2014, with goals of modernizing the codebase, improving security, and applying best practice development processes.

```
eduroam-3-153:openssl pjunod$ ./openssl genrsa 32
```

Generating RSA private key, **32** bit long modulus

```
.+++++
```

```
.+++++
```

```
e is 65537 (0x10001)
```

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

RSA Case: BoringSSL

BoringSSL

BoringSSL is a fork of OpenSSL that is designed to meet Google's needs.

```
somewhere:tool pjunod$ ./bssl genrsa -bits 32
-----BEGIN RSA PRIVATE KEY-----
MCsCAQACBQC7lZ3NAgMBAAECBFg1hoECAwDdoQIDANitAgIEAQICQOECAmWN
-----END RSA PRIVATE KEY-----
somewhere:tool pjunod$ █
```

RSA Case: 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;
    }
}
```

RSA Case: Botan in 2015

```
#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 Case: Botan in 2016

```
namespace Botan {  
  
    /*  
    * Create a RSA private key  
    */  
    RSA_PrivateKey::RSA_PrivateKey(RandomNumberGenerator& rng,  
                                    size_t bits, size_t exp)  
    {  
        if(bits < 1024)  
            throw Invalid_Argument(algo_name() + ": Can't make a key that is only " +  
                                    std::to_string(bits) + " bits long");  
        if(exp < 3 || exp % 2 == 0)  
            throw Invalid_Argument(algo_name() + ": Invalid encryption exponent");  
    }  
}
```

RSA Case: 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};
```

Myriad of Other Examples

- Small Diffie-Hellman groups
- Weak elliptic curves
- Weak ciphersuites (EXPORT, involving DES or RC4 or MD5, ...)
- etc.

Dream Crypto Library Requirement #5

- **Offer only security parameters and configuration options that are meaningful in terms of security.**
- **Offer safe security parameters by default.**
- **If really required, weaker/obsolete variants could be made available, but such that it requires an effort from the developer and emitting warnings at compilation time.**

Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters**
- 6 API Simplicity
- 7 Conclusion

Cryptographic Parameters

- Without (secure) randomness, no cryptographic security is possible.
- Most cryptographic libraries do a rather good job in providing a cryptographically secure PRNG (CPRNG) to the developer.
- However, most cryptographic libraries put the **responsibility to use the CPRNG** on the developer shoulders.
- Two bad things can happen:
 - Forget to use random values;
 - Use a non-cryptographic PRNG.

Cryptographic Nonces

- The notion of **cryptographic nonce** (“number used only once”) dates back to defenses against replay attacks in cryptographic protocols.
- (**Wrong**) assumption: generating (cryptographically secure) randomness is hard; fortunately, **nonces are easier to manage**.
- In practice, the only fool-proof way to generate nonces ... is to ensure they have a sufficient length and to generate them at random!
- Depending on the cipher, reusing a nonce can have a catastrophic impact (e.g. ECDSA).

Cryptographic Nonces



Pascal Junod
@cryptopathe

What will happen first? [#crypto](#) [#predictions](#)

Crypto II by Dan Boneh

21 %

Univ. quantum computer

43 %

Non-repeating nonces

16 %

FHE

20 %

228 votes • Résultats définitifs

13.06.16 10:33

14 RETWEETS 21 J'AIME



Self-Defending Implementations

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.

Self-Defending Implementations

: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.



Self-Defending Implementations



cryptopathe opened this issue on 14 Nov 2014 · 1 comment



cryptopathe commented on 14 Nov 2014

Dear pyCrypto developers,

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.

Self-Defending Implementations



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!

Dream Crypto Library Requirement #4

Offer only capabilities that are useful for developers and that do not allow him/her to trigger security-related mistakes.

Cryptographic Parameters: User Responsibility

```
#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()`.

Cryptographic Parameters: Nonces Reuse

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

Shay Gueron¹ and Yehuda Lindell²

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 Reyhanitabar³ Phillip Rogaway⁴ Damian Vizár³

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

Cryptographic Parameters: EdDSA

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 $r = H(h_b, \dots, h_{2b-1}, M)$, 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'Raihi, and Levy-dit-Vehel; later by M'Raihi, Naccache, Pointcheval, and Vaudenay in [55]; and much later by Katz and Wang in [47]. The patent application was abandoned in 2003.

Cryptographic Parameters: API Responsibility

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
```

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

Generating the nonce is the API's responsibility.

```
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;
```

Dream Crypto Library Requirement #6

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.

Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity**
- 7 Conclusion

API Simplicity

```
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;
}
```

API Simplicity: 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 */...}
```

Dream Crypto Library Requirement #7

- **In debug mode, a crypto library could possibly return detailed error messages.**
- **In production mode, a crypto library should return only a boolean flag: OK/NOK.**
- **API and error handling should be as simple as possible.**

Outline

- 1 Motivations
- 2 Security Models
- 3 Primitives
- 4 Security Parameters
- 5 Cryptographic Parameters
- 6 API Simplicity
- 7 Conclusion**

Wishful Thinking?

- Crypto libraries users are not omniscient cryptographers.
- A non-secure functionality is a superfluous feature that must be killed.
- Stop put security-related responsibilities on developers' shoulders.
- Refuse to generate too small keys or cryptographic groups.
- Refuse with all possible energy to use obsolete/weak cryptographic configurations.
- Generating the required IV/nonces should be the library responsibility.
- Crypto libraries must be extremely simple to use.

Conclusion

The perfect cryptographic library **does not exist (yet)!**

Thank You!

pascal@junod.info
@cryptopathe