CrashCourseCrypto

Cryptography 101 for Developers

Mathias Tausig

Who am I?

- MSc in Mathematics (University of Technology Vienna)
- Professional experience as a Developer, Sysadmin, Security Officer, Computer retail
- Spent 8 years in the PKI business
- Teaching IT-Security at the FH Campus Wien
- Research at the Competence Centre of IT-Security

Who are you?

- Developer
- Having to do with security becoming ubiquitous
- Realising security involves cryptography
- Never learnt any cryptography
- Relying on Stackoverflow for all things crypto

Disclaimer

- All rules presented are written to prevent you from shooting yourself in the foot. There might very well be exceptions to them.
- Code snippets are written for brevity and might miss important aspects (especially error handling)
- If you need this talk, you really shouldn’t be doing this . . .

Basics

- Do not design your own crypto
- Do not implement your own crypto
- There is probably an existing scheme for your usecase. Use it
- *https://keylength.com*
- Protect your keys
  - Use your OS
- Stackoverflow answers: Check reputation on *security.stackexchange.com* or *crypto.stackexchange.com*

Kerckhoff Prinzip

> *Die Sicherheit eines Systems muss alleine von der Geheimhaltung des Schlüssels abhängen, und darf nicht von der Geheimhaltung des Systems abhängen.*
> *– Auguste Kerckhoff, La cryptographie militaire, 1883*

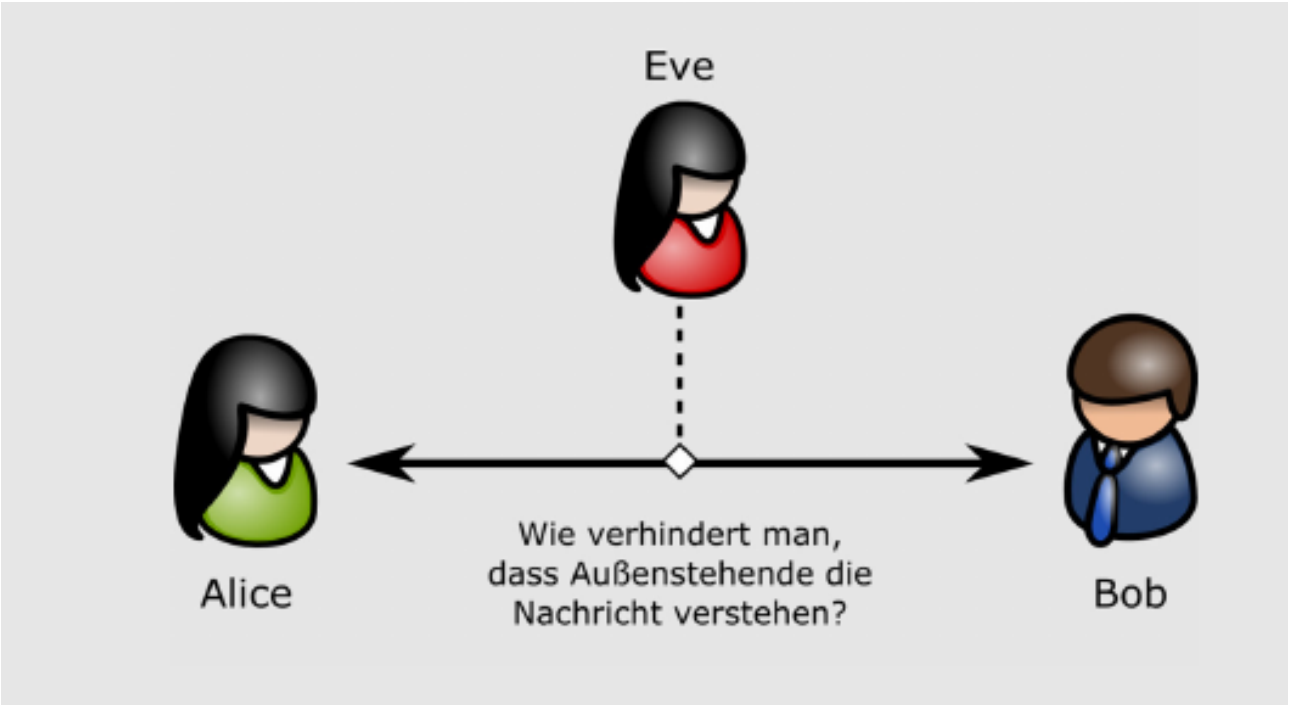
Gegenteil: *Security through obscurity*

What are we fighting for?

Security is never an absolute thing.
It is relative to your **Threat Model** and your **Security Targets**.

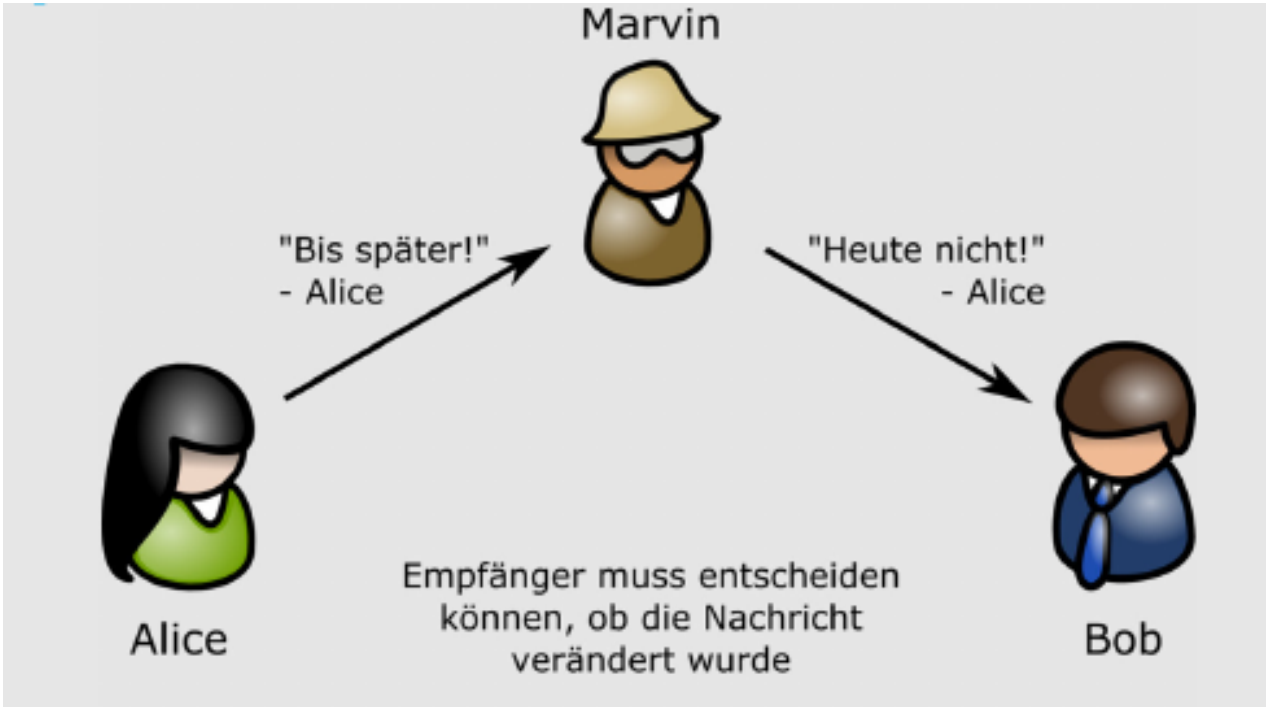
Confidentiality

Ensuring that only *authorized persons* are able to read a message's *content*.



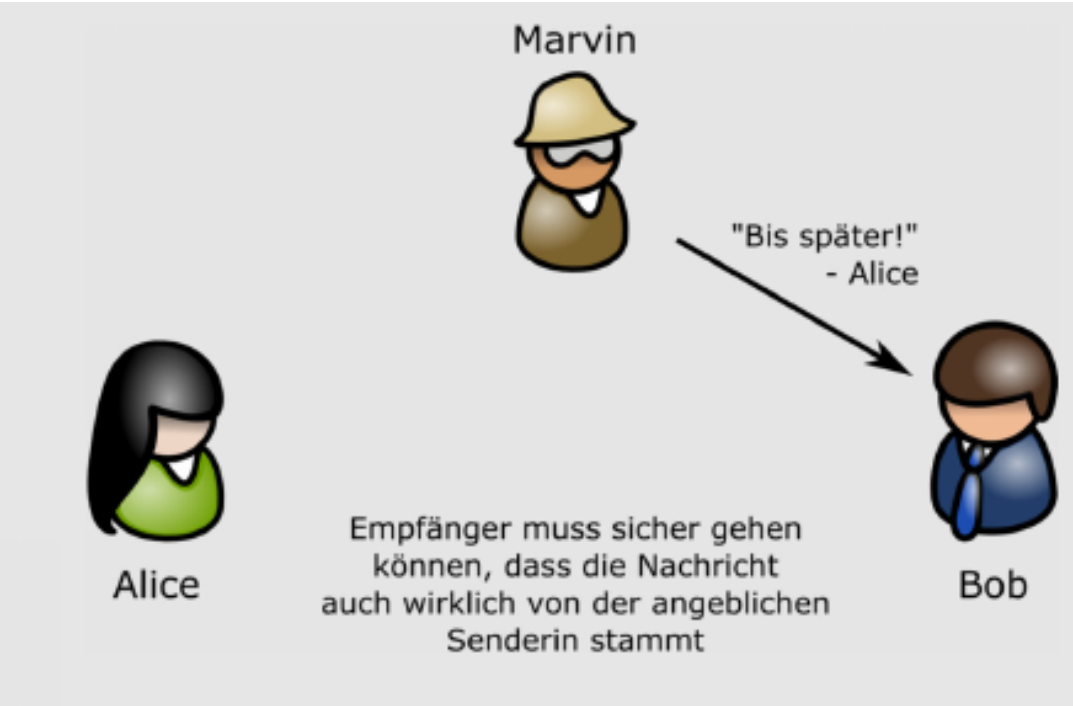
Integrity

Ensuring that a message cannot be altered undetected.



Authentication

Confirming the identity of a message's author.



Algorithms

Random numbers

Random numbers

Random numbers are at the foundation of most cryptographic algorithms. Getting them wrong will probably break your whole system.

Abbildung: Quelle: `http://dilbert.com/strip/2001-10-25`

Random numbers

- **True Random Numbers** Obtained from physical sources (clocks, sensors, hard drives, . . .)
- **Pseudo Random Numbers** Calculated deterministically from a random *seed value*
- **Entropy** Measures the amount of randomness within some random data

Random numbers

CSPRNG

An ordinary *Pseudorandom Number Generator (PRNG)* creates numbers which are statistically indistinguishable from truly random numbers.
For cryptographic usage, this is not enough. We need a *Cryptographically Secure Pseudorandom Number Generator (CSPRNG)*. That is a PRNG which is additionally *forward and backward secure*. An adversary cannot deduce future or past random values from observation of the random values.

Random numbers

OS

Preferably, you should just use the RNG provdided by your Operating system:

- `/dev/urandom` (*NIX)
- `CryptGenRandom` (Win)

Random numbers

Manual Workflow

1. Obtain a random seed value (possibly implicit)

- Random Numbers as provided by the OS: `/dev/urandom` (*NIX), `CryptGenRandom` (Win)
- Add another independent random source: Time (difference), tick value, tail of syslog, ...

Random numbers

Manual Workflow

1. Obtain a random seed value (possibly implicit)
2. Initialize a CSPRNG (with that seed)

Random numbers

Manual Workflow

1. Obtain a random seed value (possibly implicit)
2. Initialize a CSPRNG (with that seed)
3. Generate random numbers

Random numbers

Manual Workflow

1. Obtain a random seed value (possibly implicit)
2. Initialize a CSPRNG (with that seed)
3. Generate random numbers
4. (Reseed)

Random number generators

Don’t

- `rand()`, `random()`, Linear congruence generator, Mersenne Twister, ANSI X9.17

Random number generators

Don’t

- `rand()`, `random()`, Linear congruence generator, Mersenne Twister, ANSI X9.17

Do

- `SecureRandom`, `CryptoRandom`, NIST SP-800-90*, CTR-DRBG, HASH-DRBG, HMAC-DRBG, Fortuna

Random number generators

Beware

- Forks, threads
- Embedded systems
- Security level $\leq$ Length of seed

Hash Functions

A *Cryptographic Hash Function* identifies arbitrary data of arbitrary length with a deterministic identifier of fixed length, called the *hash* or *digest value* of the data.

Such hash functions have the *avalanche property*, meaning that the slightest change to the input will lead to a completely different output.

Examples

SHA-256(“Crypto is great.”) =
`d4467c5debce875f060936b91538798a62f77508090eaf72a354af82f18ee23c`
SHA-256(“Crypto is great:”) =
`7a738e6ebc68922bad14b28d410477eab7eb4cb3e3e1479eacc5d4a6c189ccf4`
SHA-256(4GB ISO Datei) =
`195baca6c5f3b7f3ad4d7984a7f7bd5c4a37be2eb67e58b65d07ac3a2b599e83`

Properties

- **Collision resistance**: No collission (two different inputs with the same digest) are known or can be computed
- **Preimage resistance**: It is not possible to calculate a preimage for a digest (*O*ne Way Functions)

Note

These properties only hold for *cryptographic* hash functions[1].

[1] there are non-cryptographic ones, too

Integrity

This property makes a hash function usable to ensure the *Integrity* of some data. Any change to the data will lead to an altered hash value and can thus be detected.

Integrity

This property makes a hash function usable to ensure the *Integrity* of some data. Any change to the data will lead to an altered hash value and can thus be detected.

Caveat

This is only true against errors during the transmission or if your adversary is only a *passive attacker (eavesdropper)*. Since no secret is needed for the hash calculation, an *active attacker* can just recalculate the digest for the manipulated data.

Usage

- Integrity checks for files
- Digital signatures
- Git
- Blockchain
- Identifiers in data structures
- Building block of other cryptographic functions

Do

- SHA-2
  - SHA-256 (256 bit = 32 byte digest length)
  - SHA-512 (512 bit = 64 byte digest length)
- SHA-3
  - SHA-3-256 or SHA-3-512

Do

- `SHA-2`
  - `SHA-256` (256 bit = 32 byte digest length)
  - `SHA-512` (512 bit = 64 byte digest length)
- `SHA-3`
  - `SHA-3-256` or `SHA-3-512`

Don't

- `MD5`, `SHA-1`, `RIPEMD-160`
- `CRC-32`
- use a key in the hash
- Hash passwords with these functions

Beware

- Encoding issues (UTF-8 vs. ISO 8859)
- Formating issues
  - Tabs vs. Spaces
  - automatic indentations
- Terminating characters

Password hashing

If you need to store password for authentication purposes, always store a hash of the password. This has to be done using special **Password Hashing** functions. They have the same basic properties as standard hash functions, but are furthermore designed to be very slow on all possible devices. This is done to prevent *Brute Force Attacks* to reverse the hash.

Salt

When hashing a password, you should always use a **Salt** value.

A salt is a random string[2] which is appended to the password upon hashing. Thus 2 users with the same password but different salts will have different password hashes.
This masks same password and prevents the usage of precalculated hash lists (*Rainbow tables*).

[2] which must not be kept secret

Do

- `SCrypt`
- `Argon2`
- Inidividual salt per user ($\sim$ 8 byte)
  - Must be stored alongside the hash
- (PBKDF-2, bcrypt for legacy purposes)

Do

- `SCrypt`
- `Argon2`
- Inidividual salt per user ($\sim$ 8 byte)
  - Must be stored alongside the hash
- (PBKDF-2, bcrypt for legacy purposes)

Don't

- Standard hash functions like SHA-2 or HMAC
- Reuse a salt for multiple users
- store passwords encrypted or in plain text

Beware

- All algorithms have parameters which can tweak their slowness
  - Store parameters alongside the hash
  - Use parameters from trusted sources
  - Document that sources
  - Check the parameters from time to time

Message authentication

To prove the authenticity of some message as well as to preserve integrity against an active attacker, a **Message Authentication Code** can be used. The properties and usage of a MAC are very similar to that of a cryptographic hash function with one important exception: A MAC requires a **secret key** for calculation. The resulting code is called a *MAC* or an *(authentication) tag*.

MAC usage

Alice		**Bob**
Message M, Key K		Key K
$T = S_K(M)$		
	$M, T \longrightarrow$	
		$T' = S_K(M)$
		$T' == T$?

Symmetric cryptography

MAC algorithms are part of *symmetric cryptography*, meaning that both sides need to know the same in advance. This key can be used to authenticate as well as to verify messages.
As a consequence, a MAC can not be used to proof who created a message, since at least two parties have the ability to create a tag. A MAC does not provide *Non-Repudiation*.

Do

- `HMAC`
  - Based on a cryptographic hash function
  - `HMAC-SHA256`, `HMAC-SHA512`
- `KMAC`
- Key at least 16 bytes long
- Use a separate key for authentication and encryption

Do

- `HMAC`
  - Based on a cryptographic hash function
  - `HMAC-SHA256`, `HMAC-SHA512`
- `KMAC`
- Key at least 16 bytes long
- Use a separate key for authentication and encryption

Don't

- Try to create your own MAC from a hash ($H(K\|M)$) or encryption
- Tag shorter than 16 bytes long

Encryption

Confidentiality, the property that only authorized entites may read a messages's content, can be achieved by **encrypting** the data.

Symmetric encryption

A key *K* is used to transform a *plaintext* message *P* into a *ciphertext* C. This process is called *encrypting* or *enciphering*.
The opposite process, *decrypting* or *deciphering*, uses the same key and must yield the original plaintext.

$$Enc_K(P) = C$$
$$Dec_K(C) = P$$

There are two types of *cipher* algorithms

- Blockciphers
  - AES
  - ~~DES~~
- Stream ciphers
  - ChaCha20
  - ~~RC4~~

To use a block cipher on data of arbitrary length, a *cipher mode* has to be used.

- AES-GCM
- ~~AES-CBC~~
- ~~AES-CTR~~
- ...

Probabilistic encryption

To prevent a cipher from producing a *deterministic output*, all modern algorithms require the specifications of a *Nonce* or *Initialization Vector (IV)*. That is a piece of data which hat to be unique[3] (somtimes even random, e.g. CBC mode) and must be transmitted in the clear alongside the ciphertext. *Nonce misuse* usually has disastrous consequences!

[3] per key/message combination

Authenticated encryption

Lots of recent attacks made it clear, that encryption alone is not sufficient. Security can be increased significantly if encryption and authentication are combined to build an **Authentication Encryption (AE)** scheme which guarantees both *confidentiality* and *authentication*. An AE scheme outputs an *authentication tag* alongside the ciphertext.

AEAD

An advanced form of AE is *Authenticated Encryption with Additional Data (AEAD)*. One can include *additional data* (or *authenticated data*) which is included in the calculation of the authentication tag but not enciphered.

Do

- Always authenticate the ciphertext
- Keys should be 128 bit or 256 bit long
- Tags should be $\geq$ 128 bit long
- `AES-GCM` (96 bit nonce)
- `AES-CCM`, `AES-OCB` (96 bit nonce)
- `ChaCha20-Poly1305`, `XSalsa20-Poly1305` (96 bit nonce)
- Ensure nonce is unique
  - Cryptographic random numbers
  - Construct using a timestamp and a system wide counter

Don’t

- Reuse a nonce
- Use all-zeroes or the key as the nonce
- Use, store or send the decrypted text if the authentication failed
- Send any specific error codes if decryption fails
- Invent your own construction
- Use an encryption-only mode (like CBC, CTR, ECB)
- Use RC4, DES, 3DES
- Encrypt too much (GB) data without changing key/nonce

Beware

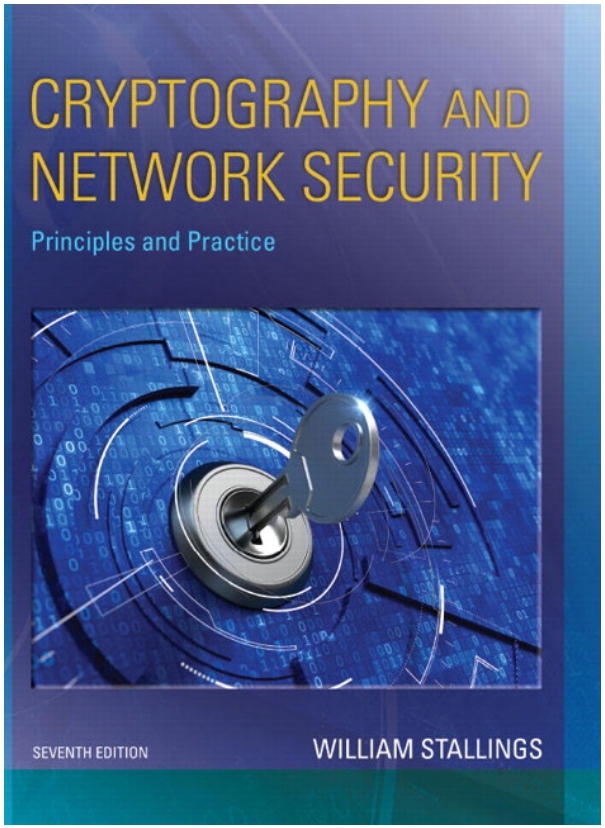
- Output is longer than plaintext (+ nonce length + tag length)

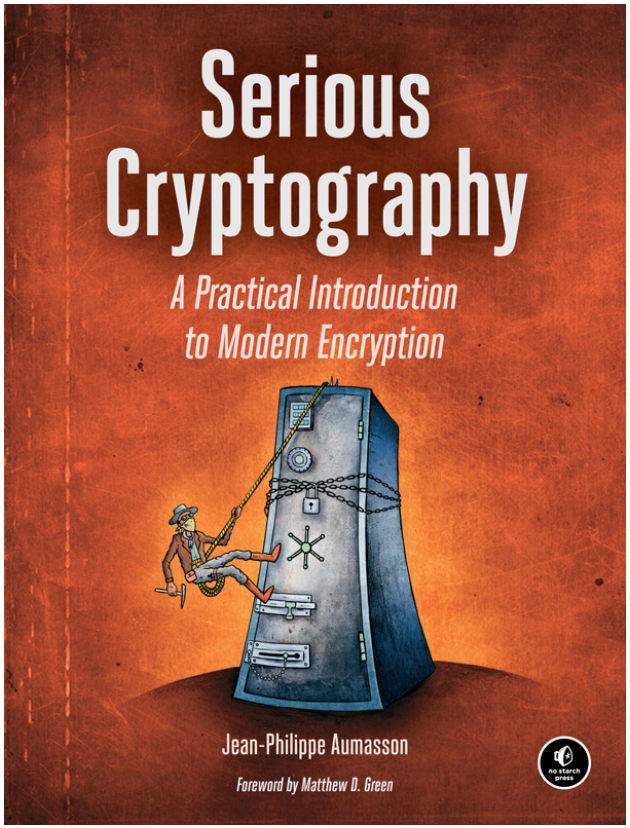
Beyond

Further topics

- Public key encryption
  - RSA-OAEP (keysize 3072 or 4096 bit)
- Digital signatures
  - RSA-PSS (keysize 3072 or 4096 bit)
  - ed25519
- Key derivation
  - HKDF
- Transport encryption with TLS
  - v1.3 or v1.2
  - Pin the server's public key or certificate in the client application
  - ~~RSA key exchange~~

Further Reading





The End!

Appendix

C: /dev/urandom (*NIX)

```
 1  int getRandomBytes(uint8_t* randomData, size_t count)
 2  {
 3    FILE* rng = fopen("/dev/urandom", "r");
 4    if (rng != NULL) {
 5      size_t bRead = fread(randomData, 1, count, rng);
 6      fclose(rng);
 7      if(bRead <= 0)
 8        return -1;
 9      else
10        return bRead;
11    } else
12      return -1;
13  }
```

C: libsodium

```
1  #include <sodium.h>
2
3  void getRandomBytes(uint8_t* randomData, size_t count)
4    randombytes_buf(randomData, count);
5  }
```

python: pycryptodome

```
1  from Cryptodome.Random import get_random_bytes
2
3  def getRandomBytes (count):
4
5      data = get_random_bytes(count)
```

java: openjdk 8

```
1  import java.security.SecureRandom;
2
3  public class JdkRng {
4
5    static SecureRandom rng = new SecureRandom();
6
7    public static byte[] getRandomBytes(int count) {
8      byte seed[] = new byte[32];
9      // Get entropy for seed e.g. from /dev/urandom
10     rng.setSeed(seed);
11     byte bytes[] = new byte[count];
12     rng.nextBytes(bytes);
13     return bytes;
14   }
15 }
```

C#: .NET Core 2.1

```
 1  using System;
 2  using System.Security.Cryptography;
 3
 4  public static class DotNetRng {
 5    public static byte[] GetRandomBytes(int count) {
 6      using(var rng = new RNGCryptoServiceProvider()) {
 7        byte[] bytes = new byte[count];
 8        rng.GetBytes(bytes);
 9        return bytes;
10      }
11    }
12  }
```

C: libsodium

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <sodium.h>
4
5  #define DIGEST_LEN 32
6
7  void hash_data(uint8_t* data, size_t count,
8      uint8_t digest[DIGEST_LEN]) {
9
10   crypto_hash_sha256(digest, data, count);
11 }
```

python: pycryptodome

```
 1  from Cryptodome.Hash import SHA256
 2
 3  def hashData (data):
 4      hasher = SHA256.new()
 5      hasher.update(data)
 6      return hasher.digest()
 7
 8  def hashFile (fileName):
 9      hasher = SHA256.new()
10      with open(fileName) as f:
11          for line in f:
12              hasher.update(line.encode("utf-8"))
13      return hasher.digest()
```

java: openjdk 8

```
 1  import java.security.*;
 2
 3  public class JdkHash {
 4
 5    public static byte[] hashData(byte[] data) {
 6      try{
 7        MessageDigest hasher =
 8          MessageDigest.getInstance("SHA-256");
 9        return hasher.digest(data);
10      } catch (NoSuchAlgorithmException e) {
11        return null;
12      }
13    }
14  }
```

C#: .NET Core 2.1

```
1  using System.Security.Cryptography;
2
3  public static class DotNetHash {
4    public static byte[] HashData(byte[] data) {
5      using(var hasher = SHA256.Create()) {
6        return hasher.ComputeHash(data);
7      }
8    }
9
10   public static byte[] HashDataFromStream(Stream s) {
11     using(var hasher = SHA256.Create()) {
12       return hasher.ComputeHash(s);
13     }
14   }
15 }
```

C: libsodium

```c
#include <sodium.h>

#define DIGEST_LEN 32

void hash_password(char pass[], size_t pass_len,
     char hash[128]) {

     int rv = crypto_pwhash_str (hash,
          pass, strlen(pass),
          crypto_pwhash_OPSLIMIT_MODERATE,
          crypto_pwhash_MEMLIMIT_MODERATE);
     /* $argon2id$v=19$m=262144,t=3,
     p=1$0qFiz0g6dUsZe/T24+7IQg$w9oLpTP6C0Lhw8n12[...]
}
```

python: pycryptodome

```
1  from Cryptodome.Protocol.KDF import scrypt
2  from Cryptodome.Random import get_random_bytes
3  import base64
4
5  def hashPassword (password):
6    salt = get_random_bytes(8)
7    # Parameters according to Colin Percival
8    # http://www.tarsnap.com/scrypt/scrypt-slides.pdf
9    # N=16384, r=8, p=1
10   hash = scrypt(password, salt, 32, 16384, 8, 1)
11   hashString = "{0};{1};{2};{3};{4}".format(
12     base64.b64encode(salt), 16384, 8, 1,
13     base64.b64encode(hash))
14   return hashString
```

java: BouncyCastle 1.6.0

```
1  import org.bouncycastle.crypto.generators.SCrypt;
2
3  public class BcPasswordHash {
4    public static String hashPassword(String password) {
5      byte[] salt = JdkRng.getRandomBytes(8);
6      //Parameters acc. to C. Percival: N=16384,r=8,p=1
7      byte[] hash = SCrypt.generate(
8        password.getBytes(StandardCharsets.UTF_8),
9        salt, 16384, 8, 1, 32);
10     Base64.Encoder enc = Base64.getEncoder();
11     return String.format("%s;%d;%d;%d;%s",
12       enc.encodeToString(salt), 16384, 8, 1,
13       enc.encodeToString(hash));
14   }
15 }
```

C#: BouncyCastle 1.8.3

```
1  using Org.BouncyCastle.Crypto.Generators;
2
3  public static class BcPasswordHash {
4    public static string HashPassword(string password) {
5      byte[] salt = DotNetRng.GetRandomBytes(8);
6      //Parameters according to Colin Percival
7      //http://www.tarsnap.com/scrypt/scrypt-slides.pdf
8      //N=16384, r=8, p=1
9      byte[] hash = SCrypt.Generate(
10       Encoding.UTF8.GetBytes(password),
11       salt, 16384, 8, 1, 32);
12     return String.Format("{0};{1};{2};{3};{4}",
13       Convert.ToBase64String(salt), 16384, 8, 1,
14       Convert.ToBase64String(hash));
15   }
16 }
```

C: libsodium

```
 1  #include <sodium.h>
 2
 3  void calculate_tag(uint8_t* key,
 4      uint8_t* data, size_t data_len,
 5      uint8_t* tag, size_t tag_len) {
 6
 7      //key_len==crypto_auth_hmacsha256_KEYBYTES(=32)
 8      // 1 <= tag_len <=32
 9      uint8_t hmac[32];
10      crypto_auth_hmacsha256(hmac, data, data_len,
11          key);
12      memcpy(tag, hmac, tag_len);
13  }
```

python: pycryptodome

```
 1  from Cryptodome.Hash import HMAC, SHA256
 2
 3  def calculateTag (key, data):
 4      hasher = HMAC.new(key, digestmod=SHA256)
 5      hasher.update(data)
 6      return hasher.digest()
 7
 8  def verifyTag (key, data, tag):
 9      hasher = HMAC.new(key, digestmod=SHA256)
10      hasher.update(data)
11      try:
12          hasher.verify(tag)
13          return True
14      except ValueError:
15          return False
```

java: BouncyCastle 1.6.0

```
1  import org.bouncycastle.crypto.digests.*;
2  import org.bouncycastle.crypto.macs.*;
3
4  public class BcMac {
5    public static byte[] calculateTag(byte[] key,
6      byte[] data) {
7    Digest hasher = new SHA256Digest();
8    HMac mac = new HMac(hasher);
9    mac.init(new KeyParameter(key));
10   mac.update(data, 0, data.length);
11   byte[] tag = new byte[32];
12   mac.doFinal(tag, 0);
13   return tag;
14   }
15 }
```

C#: BouncyCastle 1.8.3

```
1  using Org.BouncyCastle.Crypto.Digests;
2  using Org.BouncyCastle.Crypto.Macs;
3
4  public static class BcMac {
5    public static byte[] CalculateTag(byte[] key,
6      byte[] data) {
7    var hasher = new Sha256Digest();
8    HMac mac = new HMac(hasher);
9    mac.Init(new KeyParameter(key));
10   mac.BlockUpdate(data, 0, data.Length);
11   byte[] tag = new byte[32];
12   mac.DoFinal(tag, 0);
13   return tag;
14   }
15 }
```

C#: .NET Core 2.1

```
1  using System.Security.Cryptography;
2
3  public static class NetMac {
4    public static byte[] CalculateTag(byte[] key,
5      byte[] data) {
6      using(var mac = HMAC.Create("HMACSHA256")) {
7        mac.Key = key;
8        return mac.ComputeHash(data);
9      }
10   }
11 }
```

C: libsodium

```
 1  size_t encrypt(uint8_t* key, size_t key_len, uint8_t* data,
 2    size_t data_len, uint8_t* output, size_t output_len) {
 3    //key_len==crypto_aead_chacha20poly1305_IETF_KEYBYTES(=32)
 4    //output_len>=data_len+
 5    //  +crypto_aead_chacha20poly1305_IETF_NPUBBYTES+
 6    //  +crypto_aead_chacha20poly1305_IETF_ABYTES
 7
 8    uint8_t nonce[crypto_aead_chacha20poly1305_IETF_NPUBBYTES];
 9    randombytes_buf(nonce, sizeof(nonce));
10    unsigned long long ciphertext_len=output_len-sizeof(nonce);
11    crypto_aead_chacha20poly1305_ietf_encrypt(
12      output+sizeof(nonce), &ciphertext_len, data, data_len,
13      NULL, 0, NULL, nonce, key);
14    memcpy(output, nonce, sizeof(nonce));
15    return ciphertext_len + sizeof(nonce);
16  }
```

C: libsodium

```
 1  size_t decrypt(uint8_t* key, size_t key_len,
 2    uint8_t* data, size_t data_len,
 3    uint8_t* output, size_t output_len) {
 4
 5    //key_len==crypto_aead_chacha20poly1305_IETF_KEYBYTES(=32)
 6
 7    uint8_t nonce[crypto_aead_chacha20poly1305_IETF_NPUBBYTES];
 8    memcpy(nonce, data, sizeof(nonce));
 9    data += sizeof(nonce);
10    data_len -= sizeof(nonce);
11    int rv = crypto_aead_chacha20poly1305_ietf_decrypt(output,
12      &output_len, NULL, data, data_len, NULL, 0, nonce, key);
13    if(rv < 0)
14      return 0;
15    return output_len;
16  }
```

python: pycryptodome

```
1  from Cryptodome.Cipher import AES
2  from base64 import b64encode, b64decode
3
4  def encrypt (key, data):
5    cipher = AES.new(key, AES.MODE_GCM)
6    #Random nonce used automatically if not specified
7    ciphertext, tag = cipher.encrypt_and_digest(data)
8    return {"nonce": b64encode(cipher.nonce),
9      "ciphertext": b64encode(ciphertext),
10     "tag": b64encode(tag)}
```

python: pycryptodome

```
1  def decrypt (key, enciphered):
2    try:
3      cipher = AES.new(key, AES.MODE_GCM,
4        nonce = b64decode(enciphered["nonce"]))
5      return cipher.decrypt_and_verify(
6        b64decode(enciphered["ciphertext"]),
7        b64decode(enciphered["tag"]))
8    except:
9      return False
```

java: BouncyCastle 1.6.0

```
 1    public static byte[] encrypt(byte[] key, byte[] data) {
 2      GCMBlockCipher gcm = new GCMBlockCipher(new AESEngine());
 3      byte[] nonce = JdkRng.getRandomBytes(12);
 4      AEADParameters param = new AEADParameters(
 5        new KeyParameter(key), 128, nonce);
 6      gcm.init(true, param);
 7      byte[] c = new byte[data.length + 16 + 12];
 8      System.arraycopy(nonce, 0, c, 0, 12);
 9      int enc = gcm.processBytes(data, 0, data.length, c, 12);
10      try{
11        gcm.doFinal(c, enc+12);
12        return c;
13      }catch(Exception e){
14        return null;
15      }
16    }
```

java: BouncyCastle 1.6.0

```
 1    public static byte[] decrypt(byte[] key, byte[] data) {
 2      GCMBlockCipher gcm = new GCMBlockCipher(new AESEngine());
 3      byte[] nonce = new byte[12];
 4      System.arraycopy(data, 0, nonce, 0, 12);
 5      AEADParameters param = new AEADParameters(
 6        new KeyParameter(key), 128, nonce);
 7      gcm.init(false, param);
 8      byte[] plainText = new byte[data.length-12-16];
 9      int dec = gcm.processBytes(data, 12, data.length-12,
10        plainText, 0);
11      try{
12        gcm.doFinal(plainText, dec);
13        return plainText;
14      }catch(Exception e){
15        return null;
16      }
17    }
```

C#: BouncyCastle 1.8.3

```
 1  public static class BcAead {
 2    static UInt32 nonceCounter = 0;
 3    static string nonceLock = "";
 4    public static byte[] Encrypt(byte[] key, byte[] data) {
 5      var gcm = new GcmBlockCipher(new AesEngine());
 6      List<byte> nonce = new List<byte>(
 7        BitConverter.GetBytes(DateTime.Now.ToBinary()));
 8      lock(nonceLock)
 9        nonce.AddRange(BitConverter.GetBytes(nonceCounter++));
10      var param = new AeadParameters(new KeyParameter(key),
11        128, nonce.ToArray());
12      gcm.Init(true, param);
13      byte[] c = new byte[data.Length + 16];
14      int enc = gcm.ProcessBytes(data, 0, data.Length, c, 0);
15      gcm.DoFinal(c, enc);
16      return nonce.Concat(c).ToArray();
17    }
```

C#: BouncyCastle 1.8.3

```
 1    public static byte[] Decrypt(byte[] key, byte[] data) {
 2      var gcm = new GcmBlockCipher(new AesEngine());
 3      var nonce = data.Take(12).ToArray();
 4      var param = new AeadParameters(new KeyParameter(key),
 5        128, nonce);
 6      gcm.Init(false, param);
 7      byte[] plainText = new byte[data.Length-12-16];
 8      int dec = gcm.ProcessBytes(data, 12, data.Length-12,
 9        plainText, 0);
10      gcm.DoFinal(plainText, dec);
11      return plainText;
12    }
13  }
```