Security / Crypto Basics

… for C++ Developers

Topics

- Very quick intro
- Getting Random Data
- How to work with passwords?
- How to encrypt data?
- How to communicate securely?

First rule of Crypto: don’t do it yourself!

- Getting Cryptography right is REALLY hard
    - Use well tested / reviewed implementations!
- Even using standard libs is hard (cf. OpenSSL)
    - But there are nicer alternatives
- Here are some basic guidelines how NOT TO FAIL immediately.

Types of Algorithms - Hashes

Trap Door Functions / “Hash Functions”

- Simple (non-cryptographic) example: f(x) = x % 2
  - Simple to calculate, impossible to reverse
  - CRC / Checksum / parity: validate data, NOT cryptographic!
- Cryptographic Hashes:
  - very easy to calculate hash for an input
  - very (computationally) difficult to calculate the input (“preimage”) of any given hash.
  - very unlikely that two (even slightly) different messages produce the same value (“collision”)
    - should be “not even close”
- Examples: MD5, SHA-1, SHA-2/3 (256,384,512)
  - (SHA-256 [0..128] is also OK!)

recently broken, don’t use (but don’t panic ;))

Types of Algorithms - Symmetric

Most secure (in theory): “One Time Pad” - Problem: len(key) == len(data)

XOR

Key: 43252987948237957298347598734987598274587...

Data: The quick brown fox jumps over the lazy dog ...

Ciphertext: 62496579862987596255760982796740576 9...

Block Cipher

Key: 1234	Key: 1234	Key: 1234	Key: 1234	Key: 1234	Key: 1234
The quick	brown fox	jumps over	the lazy	dog <PAD>	The quick
234345234	102340912	...	...	...	234345234

==

Types of Algorithms

Stream Cipher

XOR

Key: 1234 → RNG(salt: 1234) → 1498759875983789573984759873498572893

Data: The quick brown fox jumps over the lazy dog ...

Ciphertext: 624965798629875962557609827967405769...

Block as Stream Cipher Counter (CTR) or “Chiper Block Chaining” (CBC)

Key: 123|1	Key: 123|2	Key: 123|3
The quick	brown fox	jumps over
234345234	102340912	...

Key: 1234		Key: 123 | i
The quick	i = Hash(ENC-TXT)	brown fox
234345234		020934809

Types of Algorithms - Public Key (RSA or EC)

Private Key + The quick brown fox ... SIG + The quick brown fox ...

\+ Public Key = Verified

Public Key + The quick brown fox ... Ciphertext: 1349130948108092...

\+ Private Key = Plaintext

Encryption Strength

Strength (bits)	Hash	Symmetric	Elliptic Curve (~bits)	RSA Modulus / DH Group
56	MD5	DES-56	112	768
80	~SHA-1	2-DES	160	1024
112	SHA2/3-224	3-DES	224	2048
128	SHA2/3-256	AES-128	256	3072
192	SHA2/3-384	AES-192	384	7680
256	SHA2/3-512	AES-256	512	15360

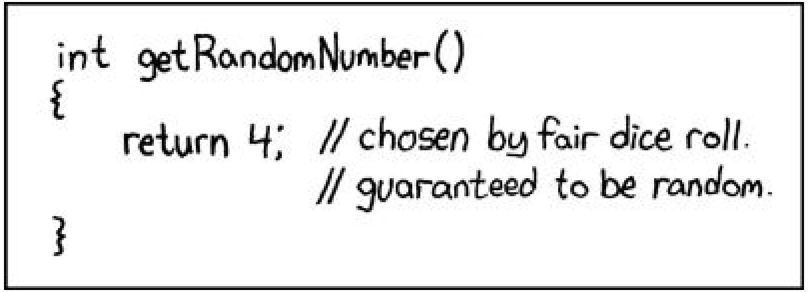
Verordnung Nr. 428/2009 (Dual-Use) 5A002: Symm. 56, EC 112, RSA/DH 512

Crypto Libraries

- “default”: OpenSSL
  - well known / reviewed
  - ugly old API
  - code is a mess, several projects try to fix it up
    - LibreSSL (OpenBSD folks) - close to original API
    - BoringSSL (Google) - more cleanups, diverging API
- Good alternative: Botan
  - nice/safe C++ API
  - SSL/TLS, Crypto Hardware support
  - Recently reviewed / endorsed by BSI
    - 3 year support / bugfix grant
- Crypto++: reasonable C++ API, no SSL

Library Basics: Getting random data

- Good random data essential for secure crypto !
- DON'T use std::random* for cryptography
- Use platform library, OpenSSL by default
  - Other options: read from /dev/random
  - CryptGenRandom on Windows



https://xkcd.com/221/

Example: OpenSSL RAND_bytes

Example: Botan - Crypto/SSL for C++

How to work with passwords?

Hashing for Passwords

- Storing/sending “username:password”
  - Eavesdropping / stealing file
- Storing/sending “hash(password)”
  - attacker does not learn user password (if hash / password are secure)
  - but: sending hash(password) ⇒ effectively your password now!
  - Use challenge-response (see below)
- Challenge-response:
  - server: “challenge”, client: hash(challenge | pw | nonce), nonce
  - Server needs password (or hash-password)
  - Password/hash stolen from server: attacker can impersonate client

Hashing Problems

- Problems: simple Passwords
  - easy to guess
  - pre-compute often used hashes => Rainbowtable
- Rainbowtables
  - use (good random) SALT: hash(salt+password)
  - ⇒ store “username : SALT : hash(salt+password)”
- Computing hash(salt+password) still feasible for known passwords

Better Hashing for Passwords

- Prevent “easy” checking of known passwords: run HASH(salt+PW) X times
  - for large X e.g. PBKDF2
- Problem: GPUs still fast enough
  - ⇒ BCRYPT: combines HASH with mutating MEM table, kills GPU performance
- Problem: ASICs still fast at BCRYPT
  - ⇒ SCRYPT: dynamically growing large MEM table, kills ASIC performance

Example: OpenSSL / Botan Hashing

Storing Passwords

- Never use PLAINTEXT STORAGE!
- Passwd format: $Algorithm$Salt$Hash(Password)

If plaintext PASSWORDS are required (also good for hashes):

- Restrict access to file / verifying process (OS security)
- Separate authentication system
- Encrypt all authentication traffic (e.g also think of NFS, SQL Connections…)
- Encrypt file (key on other medium / TPM) -> safe against discarded disk / stealing file
- Do obfuscate / make access harder, non obvious!

Transmitting Passwords

- Storing hash: protects against storage theft, not eavesdropping!
  - The hash is effectively your new password
- Challenge-response: server stores PW encrypted with e.g. HASH(PBKDF2(PASSWORD)), --> access to server store does not reveal hash
- less secure, simple option: just rely on TLS, send plaintext -> same security (-TLS) for you, but exposes user Password !

“State of the art”: “Secure Remote Pasword” (SRP)

- “DH” for passwords
- Server stores “username: verifier + salt”
  - verifier generated by client, password never transmitted
- Client/server negotiate key using Password(client), Verifier (server)
  - Client → Server: username, random key A
  - Server → Client: salt, random key B
  - If communication works, both generate matching key (may check explicitly or just encrypt)

- Standardized for SSL, available in OpenSSL / Botan
- Only “issue”: stolen verifier can impersonate server
  - So still verify your certs!

How to encrypt data?

Encrypting Data - Use Standard Tools

openssl aes-256-ctr -a -salt -in input.txt -out output.txt.enc

openssl aes-256-ctr -d -a -salt -in input.txt.enc -out output.txt

- Problem: unencrypted data written to disk!

Sqlcipher - SQLite with encryption:

- simple encrypted single file data store, secure “by default”

ZIP / LZMA compressors support encryption

Encrypting Data - Standard Algorithm: AES

Use standard algorithm: AES (perhaps Serpent, Blowfish, Stream Cipher: ChaCha20)

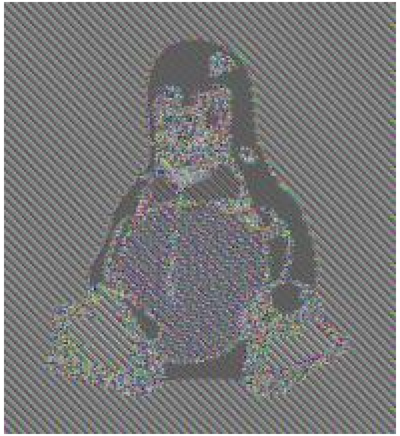
- 128, 192 & 256 are regarded as secure
- NOT DES, RC4

What are block chaining modes?

→ turn block cipher into stream, block key based on position

- CBC: key hash based on previous block (don't use)
- ✓ CTR: key based on "block counter" (parallelizable!)
- ✓ GCM: authenticated encryption (otherwise CTR)
  - Additional "tag" validating data
- ✓ XTS: special disk encryption mode

ECB Mode:



(https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation)

Example: OpenSSL / Botan Encryption

Encrypting Data - Disk Encryption

- Block encryption - device level, independent of filesystem
  - No metadata per block → IV based on Sector / Block offset (“tweak”)
  - CBC not well suited: requires reading in sequence, allows “watermarking” attack
  - Solution: XTS mode - encrypts tweak for each block
    - OpenSSL: different keys for tweak / data → 2x key size!
  - Still: no authentication of data! Attacker can revert or destroy blocks

→ Use XTS, better use existing solution (LUKS, VeraCrypt, BitLocker etc)

- Filesystem encryption: e.g. EncFS, NTFS-EFS
  - Can use metadata (e.g. include GCM tag, like ZFS)
  - Often does not encrypt metadata (filename, change times etc)

How to communicate securely?

Asymmetric Encryption: RSA / EC

Public / private key pair:

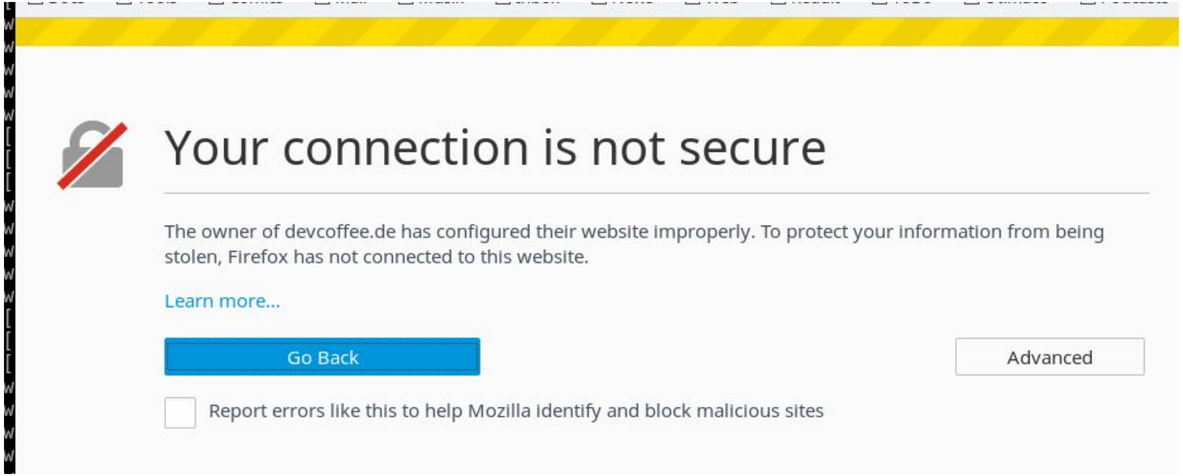
- Private Key encrypts, Public Key decrypts → Signature
- Public Key encrypts, Private Key decrypts → Encryption

Build certificate “chains”: SSL/TLS certificate authorities

- CA certifies intermediate -> intermediate certifies your certificate
- you prove ownership of cert with private key
    - protect your private keys!
    - e.g. again encryption / obfuscation on system

SSL/TLS

- SSL is broken, use TLS 1.2 (1.3)
- if you “Add an exception...”, might as well not encrypt at all:



- validate certificates!
- get CA certified for free: e.g. https://letsencrypt.org/

SSL/TLS - Key Negotiation - Diffie-Hellman

- Negotiate secure key without “exchanging” it
- Use “old” DH od ECDH
- Use “ephemeral” mode (DHE/ECDHE)→ “perfect forward secrecy”
- Generate your own DH parameters:

```
openssl dhparam -outform pem -out dhparam.pem 2048
```

(may take some time, 4096 even more, perhaps pre-compute per release?)

- Problem: man-in-the-middle attacks - relies on CA key verification!

Implementing TLS

- Boost ASIO has good SSL support
  - works around OpenSSL issues
- Botan SSL
- Provide up-to-date CA files or BETTER use OS ones
- Generate (or provide self-generated) DH parameters
- Restrict accepted algorithms
- Implement certificate revocation - way to notify on stolen private keys
  - Note: CRLs not used by browsers, use OSCP

Getting help

If you NEED to DIY, look here:

Books: Practical Cryptography (Ferguson, Schneier), Applied Cryptography (Schneier), Security Engineering (Anderson)

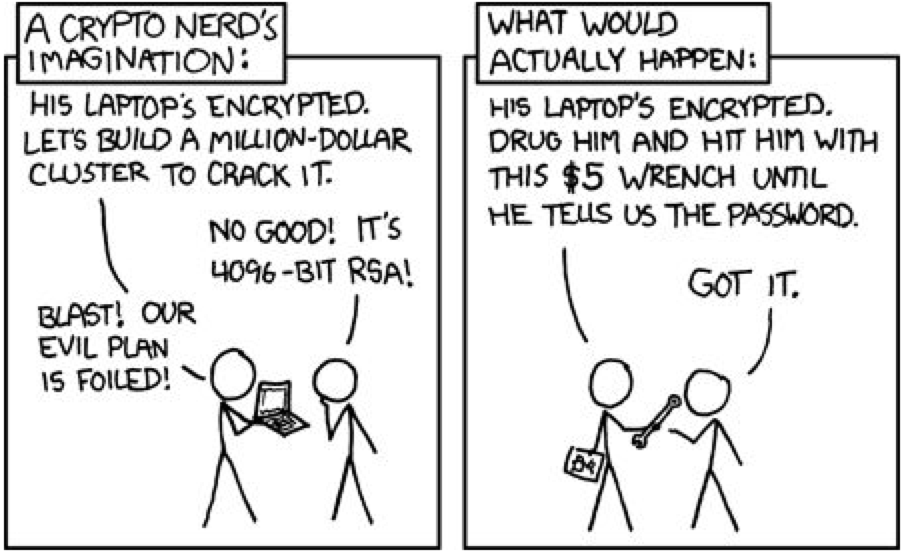
- https://www.securecoding.cert.org
- https://security.stackexchange.com/

Other Crypto APIs:

- Windows: https://msdn.microsoft.com/en-us/library/ms867086.aspx
- Nicer C++ API (buy no SSL) https://www.cryptopp.com

Stick Figure Guide to AES: http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html

Happy (and safe) coding!



https://xkcd.com/538/