Secure Software Engineering

Winterterm 2025/26

Correct Usage of Security Mechanisms & Applied Cryptography

Dr. Christian Tiefenau

Outline today

- Correct Usage of Security Mechanisms
  - Security Code, Do’s & Don’ts
  - Establishing Secure Connections
  - Reliable Input Sanitization

- Applied Cryptography
  - Encryption, Hashing, MACs, Digital Signatures, TLS

- Vulnerability of the day
  - Hardcoded Credentials
  - Unsalted Hashes

Correct Usage of Security Mechanisms

Secure code vs. Security Code

- ***Secure code* refers to a property of your application code as a whole**
  - Software is constructed following a secure development lifecycle
  - Threat/risk analysis, secure design, secure coding practices, security testing

Secure code vs. Security Code

- ***Security code* refers to software components specifically designed to implement security *functionality***
    - Common types of functionality:
        - Libraries for cryptographic operations
        - Libraries for secure connection establishment (e.g. TLS)
        - Libraries for sanitization
        - Frameworks for authentication (e.g. OAuth)
        - Frameworks for access-control

- **Secure code requires secure implementation and secure usage of security code**

General Dos and Don’t for Security Code

- **Unless you are an expert in IT-security, don’t implement such code on your own!**
  - Also third-party code has flaws but it’s much more likely that you will make security critical mistakes on your own.

- **Choose reputable components**
  - Look for libraries/frameworks that have gone through independent security certification
  - Demand access to the source; ideally choose open source projects

- **RTFM**
  - Do your homework, read the provided documentation
  - Do not rely on online sources such as StackOverflow ||[Insert most fancy LLM here]

RTFM

- **On the topic of RTFM…**
  - No one actually does that [1]
  - And no one has been doing it for decades [2]
- **What does that actually mean for us?**
  - When you create something that should be used by others:
    - Make it intuitive, as little overload as possible
    - If you have to: a really short manual which is targeted at your target audience, not at you
    - Give good examples
    - Use sane defaults

[1] David G. Novick and Karen Ward. 2006. Why don't people read the manual? In Proceedings of the 24th annual ACM international conference on Design of communication (SIGDOC '06). Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/1166324.1166329
[2] Alethea L. Blackler, Rafael Gomez, Vesna Popovic, and M. Helen Thompson. 2014. Life Is Too Short to RTFM: How Users Relate to Documentation and Excess Features in Consumer Products. Interacting with Computers 28, 1: 27–46. https://doi.org/10.1093/iwc/iwu023

RTFM

- **What does that actually mean for us? (cont)**
  - Sadly, not many people create intuitive crypto libraries
  - You will still have to read the manual for security relevant libraries
  - You will still have to read the manual for non-security libraries, if they impact your domain logic and can be exploited
  - But you can also do a risk analysis on where you really need to dive deep into the docs and where you only need to skim the docs a bit
    - Have to do encryption? Read the manual.
    - Have to use a library that handles access management? Read the manual.
    - Have to use a library that accesses no private data and does not provide security functionality? Probably don't have to read the manual.

Establishing Secure Connections

- **What properties must a secure connection have?**

- **Encrypted:** sent messages can only be read at communicating endpoints
- **Authenticated:** both endpoints know for sure that they are communicating with the right counterparts
- **Tamper-proof:** Integrity checks assure that messages are received the way they were sent

Establishing Secure Connections

- **How to establish secure connections?**

- **Encryption:** For performance reasons usually a two-step process:
  - Step 1: Use public-key-crypto to exchange a secret session key
    - This secret key must be generated with a secure PRNG!
  - Step 2: Use session key to exchange symmetrically encrypted data
    - Symmetric: same shared key is used for encryption and decryption

- **Authentication:** Typically established by checking cryptographic certificates
  - Certificate chain is validated up to some known “root of trust”
  - Problem: “root of trust” must *really* be trustworthy

- **Integrity:** using hashing / some message integrity code
  - *Sometimes combined with Authentication in Message Authentication Codes (MAC)*

Establishing Secure Connections

- ***Sounds like a lot of work? It is…***
- Lots of opportunities to get things wrong
- Hence better use existing libraries written by experts

- **Best practice: if your processing power allows it, use Transport Layer Security (TLS)**, which is also used in HTTPs
  - TLS establishes encrypted, authenticated and integrity-checked connections
  - Can/must be configured with different algorithms that implement those three steps, so-called *cipher suite*
  - Example: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 defines a key exchange algorithm (Diffie-Hellmann with RSA using Ellyptic Curves), a bulk encryption algorithm (AES, 128 bit in Galois Counter Mode), and a message authentication code (MAC) algorithm (SHA256).

Establishing Secure Connections

- **Java has built-in support for establishing TLS connections**
- **C/C++: use libraries such as OpenSSL or Botan**
- **Other languages? Other libraries…**
- **If you’re deploying a webserver, always use a reverse proxy with automatic SSL renewal (e.g. caddy)!**
  - **Easy, sane (secure) defaults, nice docs, short examples.**

Reliable and correct sanitization of untrusted inputs

- General idea: escape characters such that input cannot become executable

- **Problems:**
  - Must not forget about sanitization in general
  - Must not forget about individual characters to sanitize
  - Must sanitize *the right data* in *the right way*

- **Leave the details to the experts**
  - Again, do not implement sanitization on your own, use libraries (e.g. templating engines or SQL prepared statements) instead
  - Libraries for Java, C/C++, python, rust, …
  - Java: org.apache.commons.lang.StringEscapeUtils
    Also: https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project

On sanitizing outputs and Anonymization

- Sanitization not just useful for inputs, also for outputs
- Example: anonymization
- Beware: in many cases data can be de-anonymized again by correlating different data items
- Hence: Be extra cautious whenever giving out any data, including „anonymized data“
- Field of active research: „differential privacy“, „statistical disclosure control“

Summary: Correct Usage of Security Mechanisms

- Educate yourself about best-practice libraries and frameworks for the language and platform you are using
    - News, CVE database, NOT LLM output
- *Do not* implement security code by yourself
- *Do* consult the appropriate documentation of those libraries and frameworks

Applied Cryptography

Cryptography & You

Never roll your own Crypto

But

Never *not* use Crypto when Needed

(This lecture is about what kind of crypto you should use when)

Basic Cryptographic Concepts

Encryption

Hashing

Certificate

Message Authentication Code

Signature

Encryption

The Basic Problems

- On the internet, there is no security by default
  - Anyone can join
  - Anyone can sniff

- but “distrust everything all the time” is not feasible

- Instead: communicate the right messages with the right entities only

  => confidentiality and authentication

Encryption

- Fundamental Idea:

  Transforming meaningful data into seemingly meaningless gibberish with the possibility of transforming it back

  => Confidentiality

- Modern encryption algorithms require additional secret input, usually called a cryptographic key

Symmetric Encryption

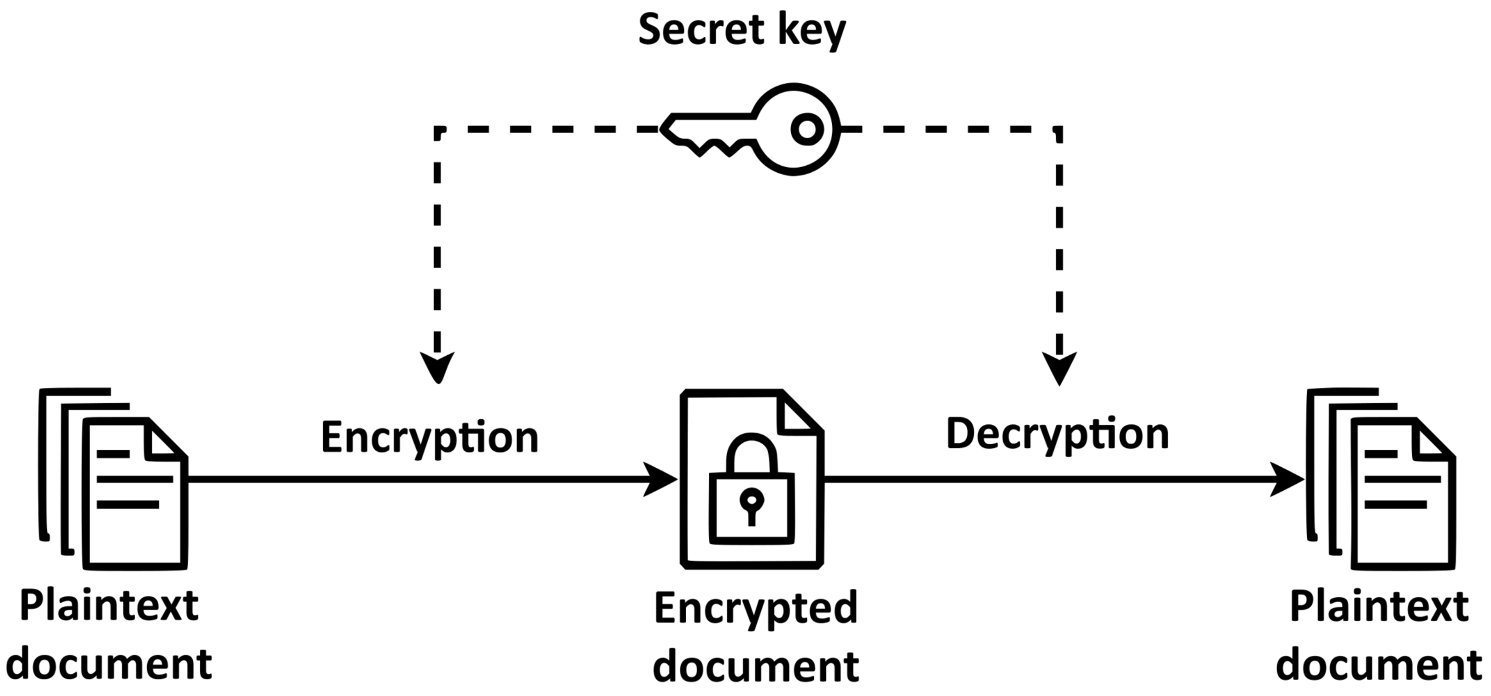
- One key for en- & decryption

  ⇒ Key must be kept secret

- Modern algorithms
  - **AES**
  - Twofish
  - Blowfish
  - …

- AES (Advanced Encryption Standard):
  - A fast, and widely used (standardized form of crypto, established by NIST in 2001)
  - Used for encryption of large data: backups, hard drives, etc.
  - Interestingly *not* mathematically proven to be secure
  - A Block Cipher

Symmetric Encryption
How does it work?



- Only tricky part: The key must be known by the decrypting party (only)
  - Possibilities: Pre-shared keys or secure key-exchange protocols

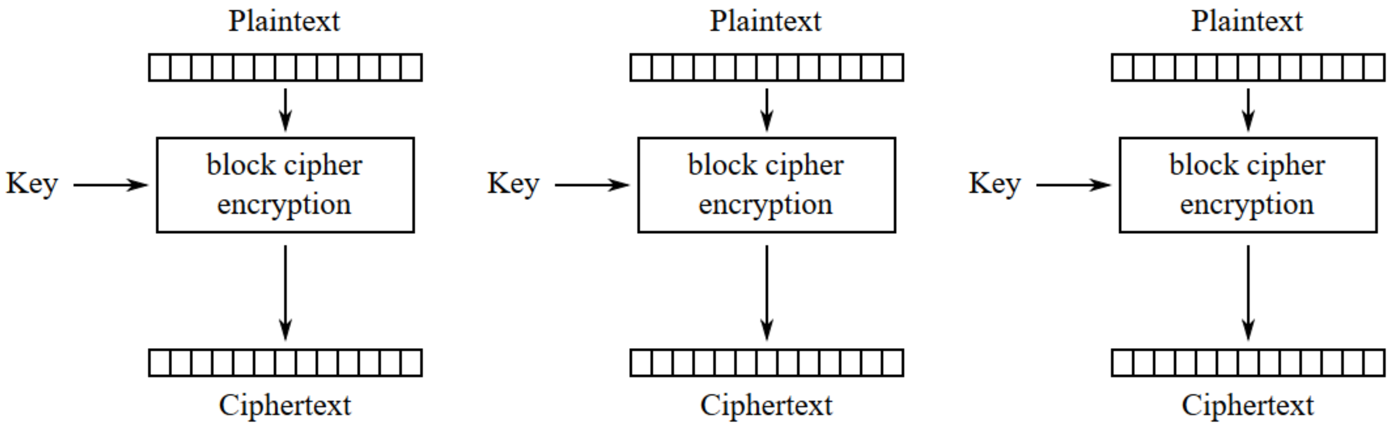
Stream vs. Block Ciphers

- Stream Ciphers
  - Key is transformed into a stream
  - Plaintext converted to ciphertext by xor-ing it with keystream

- Block Ciphers
  - Split plaintexts into blocks of fixed sizes
  - Blockwise encryption
  - Quite efficient, can be parallelized
  - Questions:
    - How to (re)assemble the ciphertext blocks?

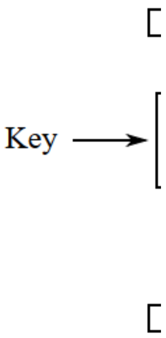
Block Ciphers - Modes of Operation

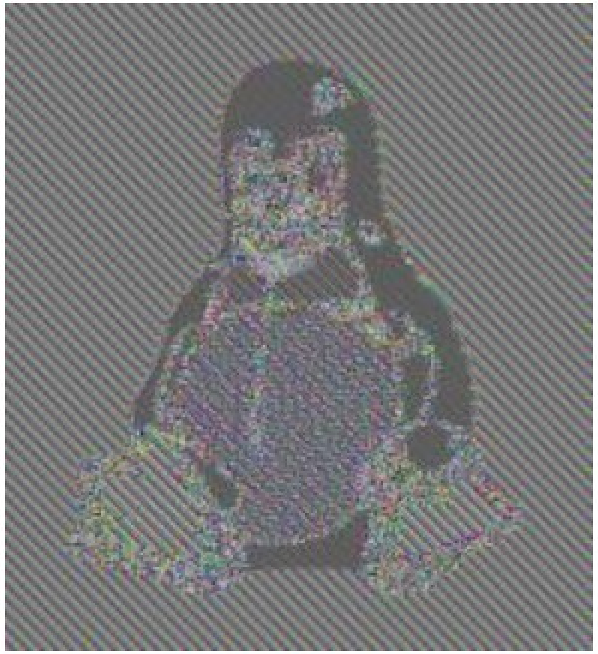
- Mode determines how to evolve key material from one block to the next,
  and how to re-assemble the blocks
- Examples:
  - ECB



Block Ciphers - Modes of Operation

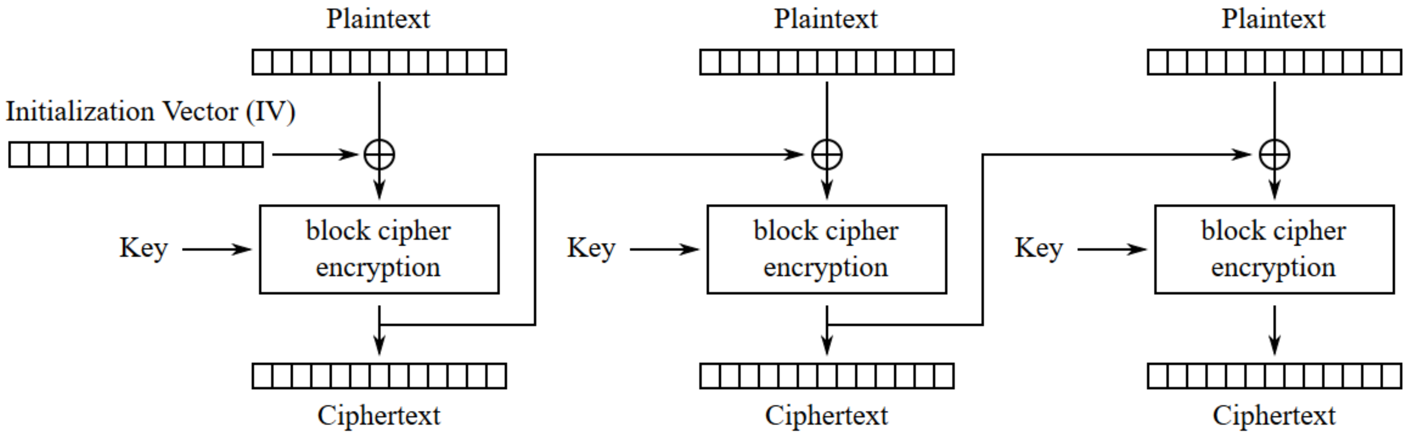
- Mode determines how to evolve key material from one block to the next,
  and how to re-assemble the blocks
- Examples:
  - ECB





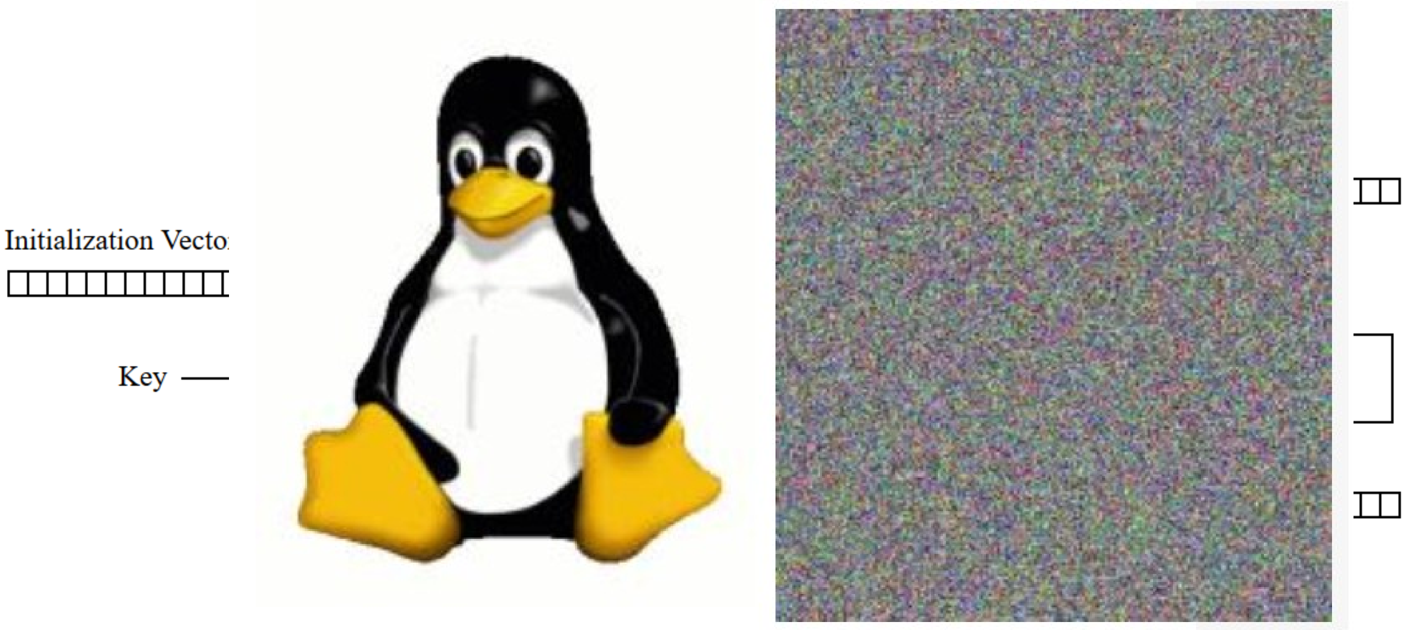
Block Ciphers - Modes of Operation

- Mode determines how to evolve key material from one block to the next,
  and how to re-assemble the blocks
- Examples:
  - ECB
  - CBC



Block Ciphers - Modes of Operation

- Mode determines how to evolve key material from one block to the next,
  and how to re-assemble the blocks
- Examples:
  - ECB
  - CBC



Block Ciphers - Modes of Operation

- Mode determines how to evolve key material from one block to the next,
  and how to re-assemble the blocks
- Examples:
  - ECB
  - CBC
  - CTR
  - GCM
  - …

Assymmetric Encryption

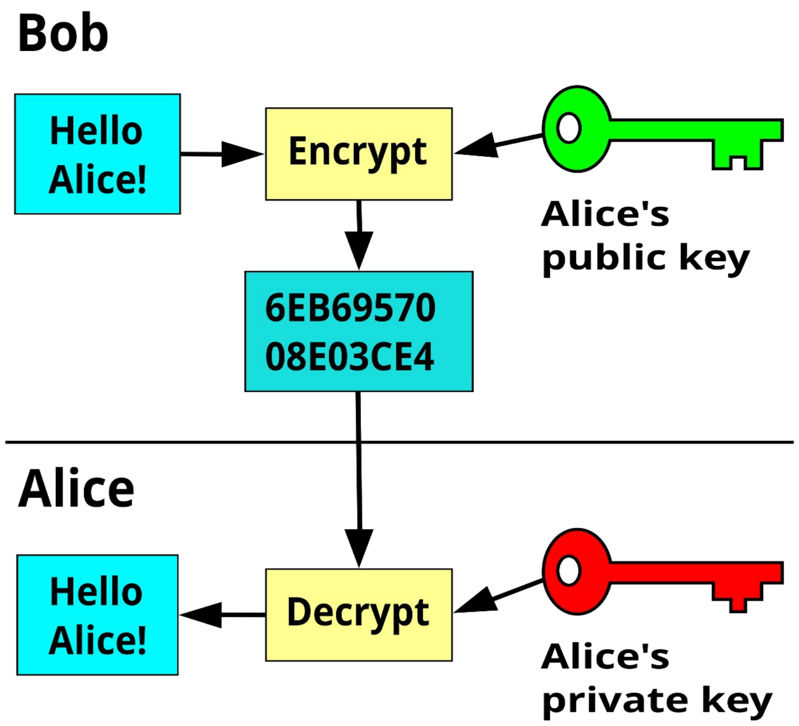
- Also called public-key encryption
- Every party has two keys
  - Public key for others to encrypt data
  - Private key for oneself to decrypt it

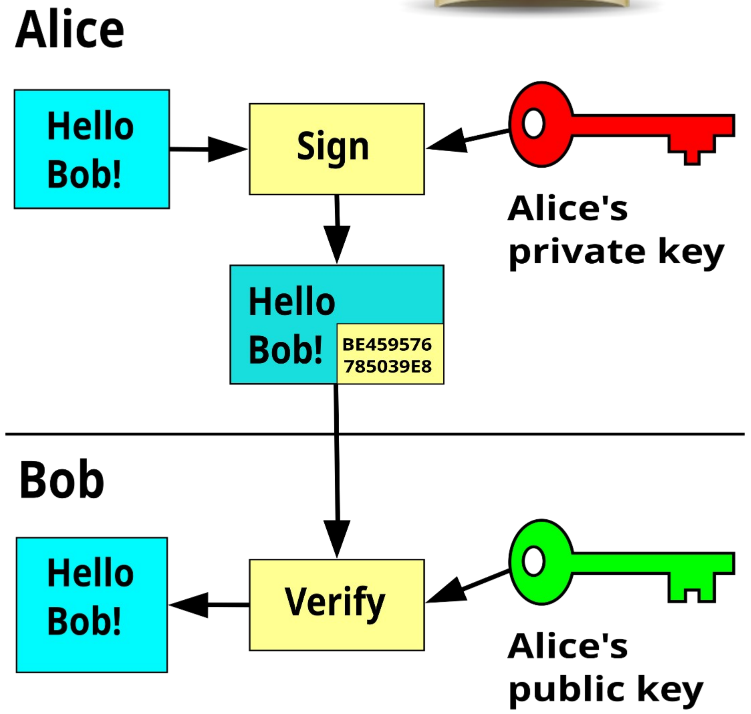
- Solves problem of key exchange, but relatively slow compared to symmetric encryption

- Popular modern algorithm: RSA (>=4096 bit keylength)
  - Factorization of two prime numbers
  - Public/private keys generated from computing two very large prime numbers

- Factoring primes is still hard, which is why RSA has not yet been cracked
- But some of its implementations have!
- Other way: ED25519 (Elliptic Curve)

Public-Key Encryption - How does it work?





- Alice can also authenticate the message by signing it with her private key.
- Signature can then be verified with corresponding public key.
- Tricky Part: How to ensure that key pair belongs to the stated person?

Hybrid Encryption
Public-Key + Symmetric Encryption

- Problem:
  - Public-key encryption rather slow
  - Symmetric encryption requires some sort of key exchange

- Combination
  - Symmetric encryption to encrypt data
  - Public key encryption to encrypt symmetric/session key

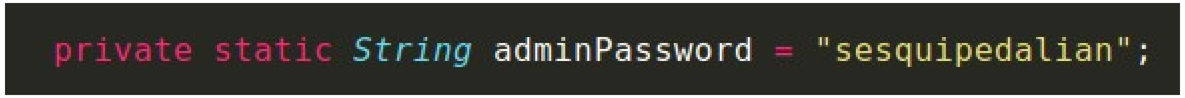
- End of communication – throw away symmetric key

- Often used in higher-level crypto protocols

1st Vulnerability of the Day

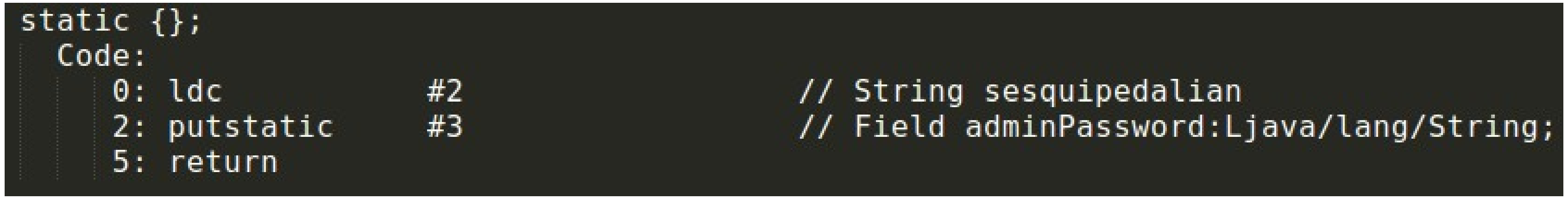
Hardcoded Credentials

Hardcoded Credentials



```
private static String adminPassword = "sesquipedalian";
```

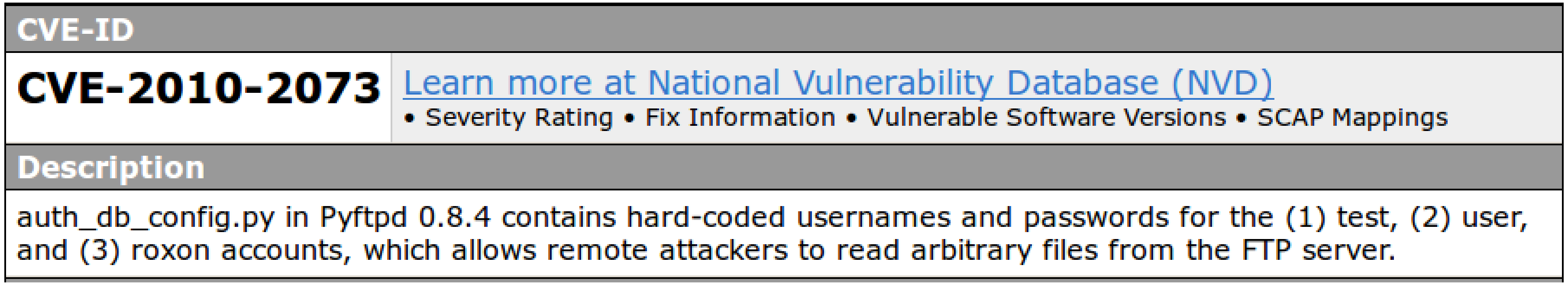
Question: Where is the problem?



```
static {};
  Code:
     0: ldc           #2                  // String sesquipedalian
     2: putstatic     #3                  // Field adminPassword:Ljava/lang/String;
     5: return
```

- Mitigation
  - Never store clear text passwords anywhere, only salted hashes of those!
  - Also store credentials in external file (not the code!), with proper permissions

Real-world Hardcoded Credentials



CVE-ID

CVE-2010-2073 Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

Description

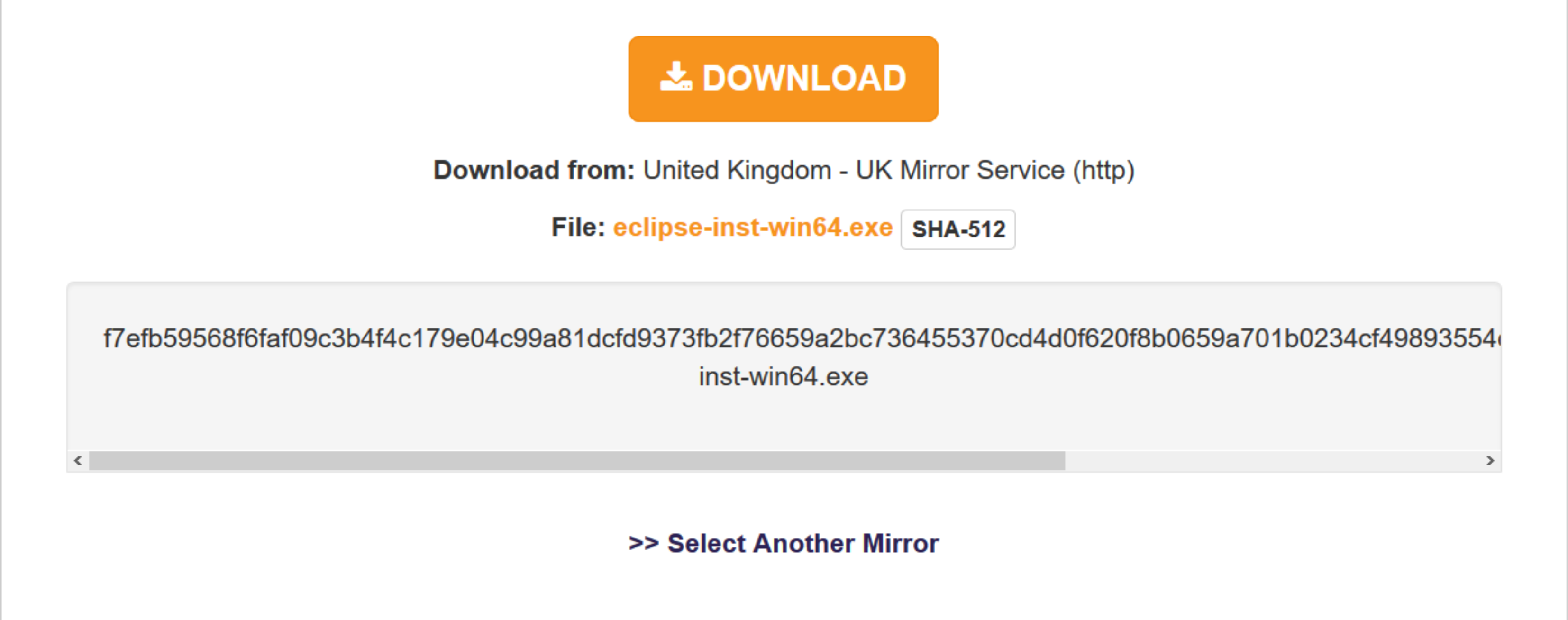
auth_db_config.py in Pyftpd 0.8.4 contains hard-coded usernames and passwords for the (1) test, (2) user, and (3) roxon accounts, which allows remote attackers to read arbitrary files from the FTP server.

- Beware
  - Attackers can reverse engineer your code
  - Obfuscation is not an option (attackers have time and are creative)
  - Similar for license keys, encryption keys…

- Point to take away
  - You cannot keep secrets in your source code!

Hashing

Hashing



Download from: United Kingdom - UK Mirror Service (http)

File: eclipse-inst-win64.exe

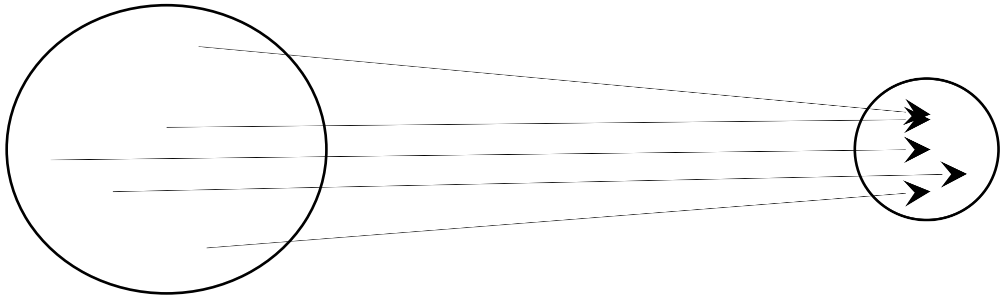
```
f7efb59568f6faf09c3b4f4c179e04c99a81dcfd9373fb2f76659a2bc736455370cd4d0f620f8b0659a701b0234cf49893554
inst-win64.exe
```

>> Select Another Mirror

Hashing

- Idea:

  Transforming a chunk of data into a very large number but no reversal
  => Integrity



- What is this good for? Two reasons:
  - Check integrity of some piece of data (was it manipulated?)
  - Prove possession of a secret without revealing the secret itself
- Things to keep in mind
  - Even the change of a single bit should result in a completely different hash digest
  - Collisions: when a two different chunks of data result in the same hash digest

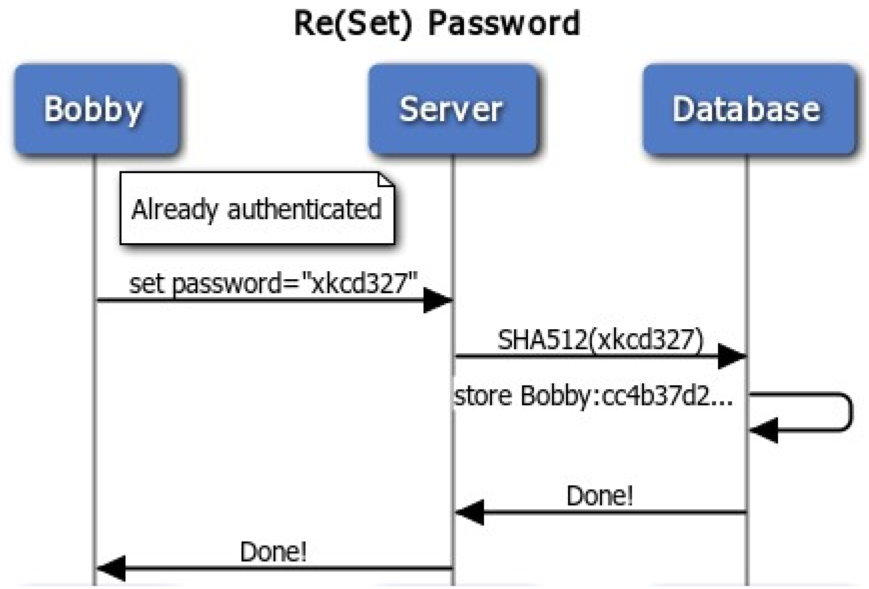
- Modern algorithms: SHA-2 family (SHA-224, SHA-256, SHA-386, SHA-512), SHA-3

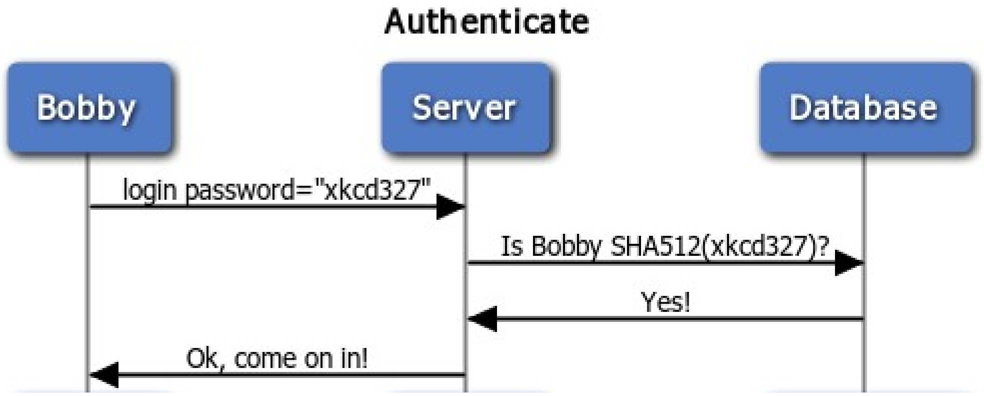
Authentication with Hashes

- Use Case: (re)set password
  - User inputs password
  - Server hashes pw
  - Stores the hash

- Abuse Case: Break-in
  - Attacker steals plaintext passwords from Database
  - Harm done: can authenticate as any user
  - Mitigation: can't reverse the hashes

- Use Case: Authenticate
  - User inputs password
  - Server computes hash
  - Checks the hashes

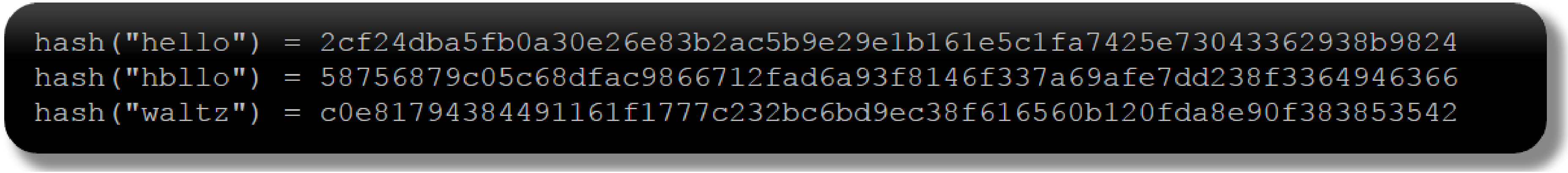




2nd Vulnerability of the Day

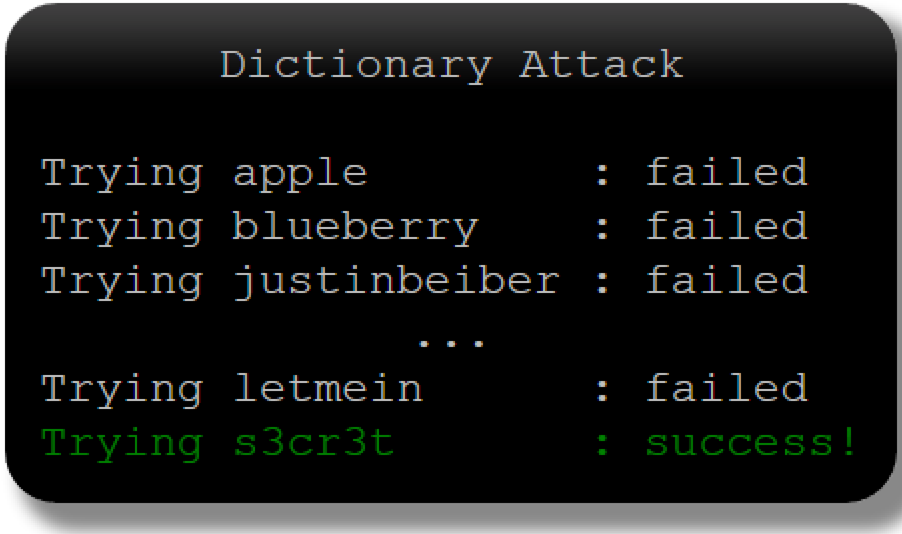
Unsalted Hashes

Vulnerability of the Day –
Abuse Case: Dictionary Attack & Brute Force Attack

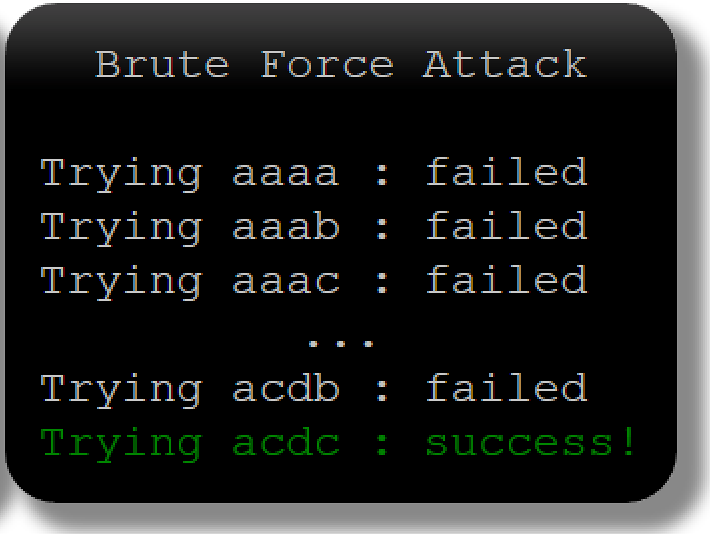


```
hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366
hash("waltz") = c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542
```

- Brute Force Attack: Trying every possible password
- Dictionary Attack: Trying a list of common password
- Rainbow Tables: Precomputed list of hashes

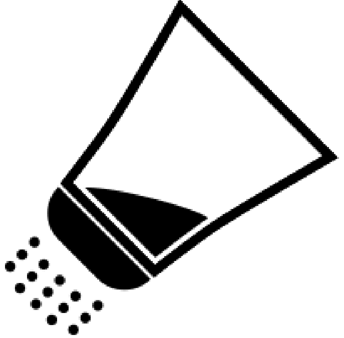


```
         Dictionary Attack

Trying apple         : failed
Trying blueberry     : failed
Trying justinbeiber  : failed
              ...
Trying letmein       : failed
Trying s3cr3t        : success!
```



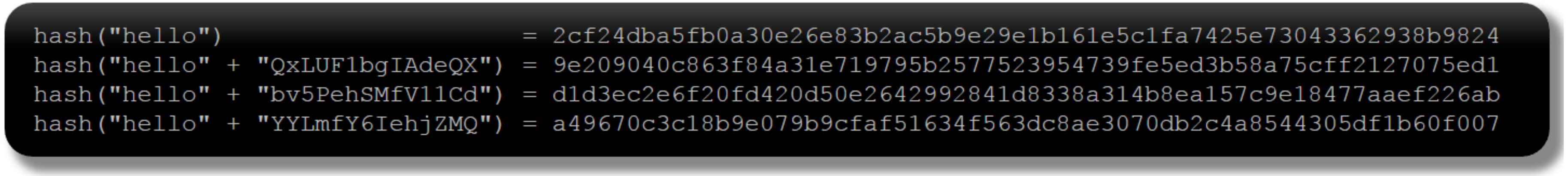
```
   Brute Force Attack

Trying aaaa : failed
Trying aaab : failed
Trying aaac : failed
         ...
Trying acdb : failed
Trying acdc : success!
```

Vulnerability of the Day – Solution: Salting



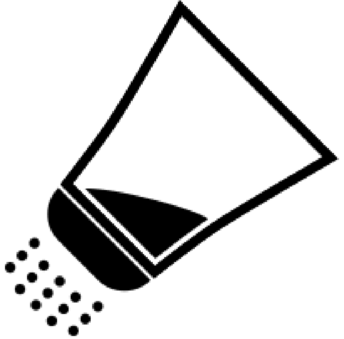
- Salting:

  Random string (=salt) is appended to a password



```
hash("hello")                   = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007
```

How *not* to salt

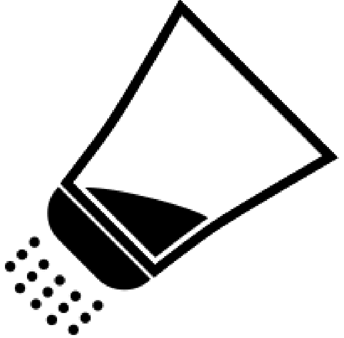


- What is wrong with this way of salting?

```
hash = secureHashAlg( “myCompanySalt‟ + password );
```

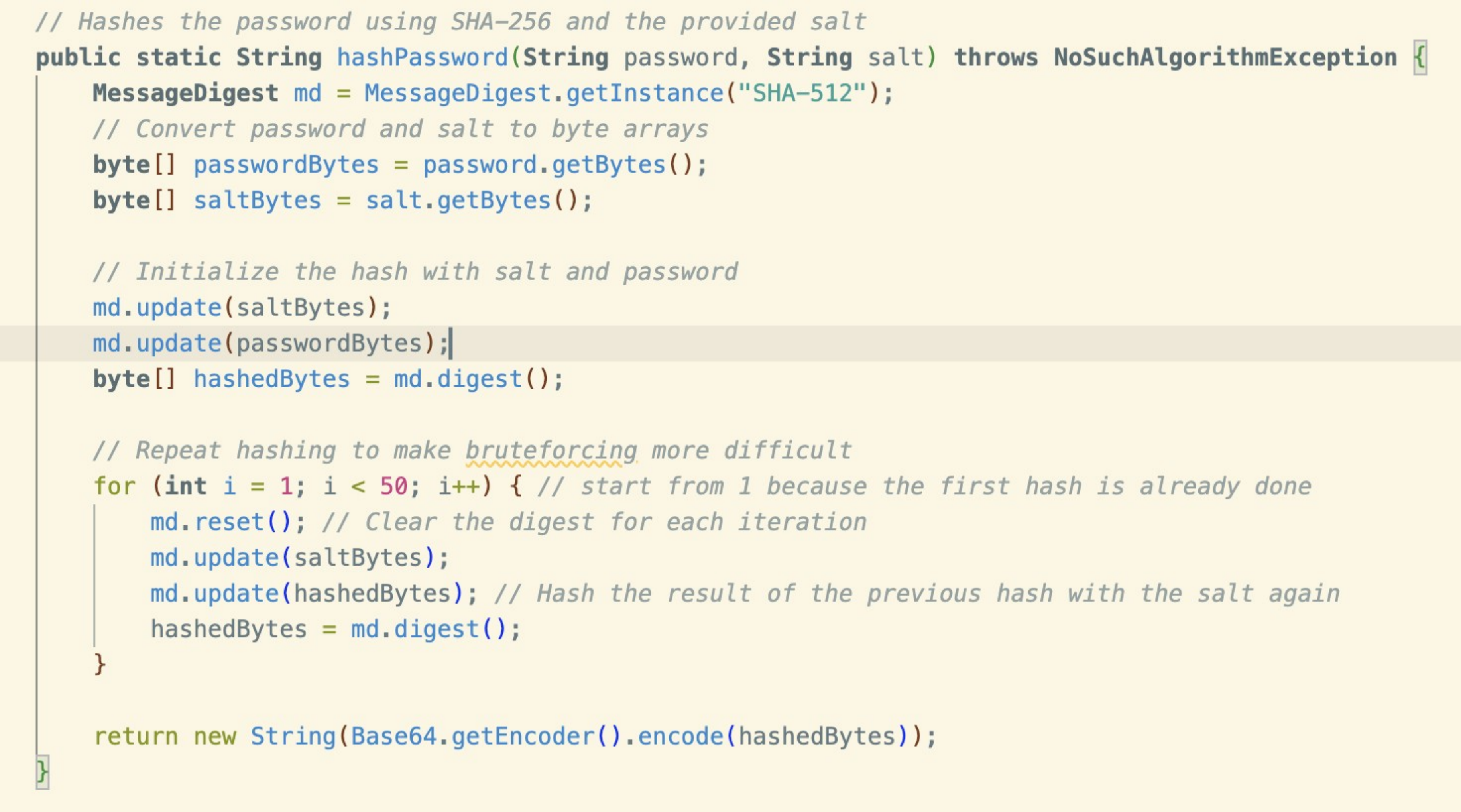
- Problem 1: the salt is global, hence can still pre-compute a rainbow table for the entire company/server/service, just cannot reuse other rainbow tables
- Problem 2: salt is *pre*pended to secret/password
  - Why problematic?
  - Allows for length-extension attacks:
    attacker can save state of hashing algorithm after having hashed the salt, then very fast compute individual hashes password hashes

Salting done right



- Each user/password gets a different salt!
- Salts should be at least 32 bits, should be random
  - Best use a secure random source
- Salts stored in plaintext along with the hashed + salted password

Example Hashing



```java
// Hashes the password using SHA-256 and the provided salt
public static String hashPassword(String password, String salt) throws NoSuchAlgorithmException {
    MessageDigest md = MessageDigest.getInstance("SHA-512");
    // Convert password and salt to byte arrays
    byte[] passwordBytes = password.getBytes();
    byte[] saltBytes = salt.getBytes();

    // Initialize the hash with salt and password
    md.update(saltBytes);
    md.update(passwordBytes);
    byte[] hashedBytes = md.digest();

    // Repeat hashing to make bruteforcing more difficult
    for (int i = 1; i < 50; i++) { // start from 1 because the first hash is already done
        md.reset(); // Clear the digest for each iteration
        md.update(saltBytes);
        md.update(hashedBytes); // Hash the result of the previous hash with the salt again
        hashedBytes = md.digest();
    }

    return new String(Base64.getEncoder().encode(hashedBytes));
}
```

MAC

Message Authentication Codes (MAC)

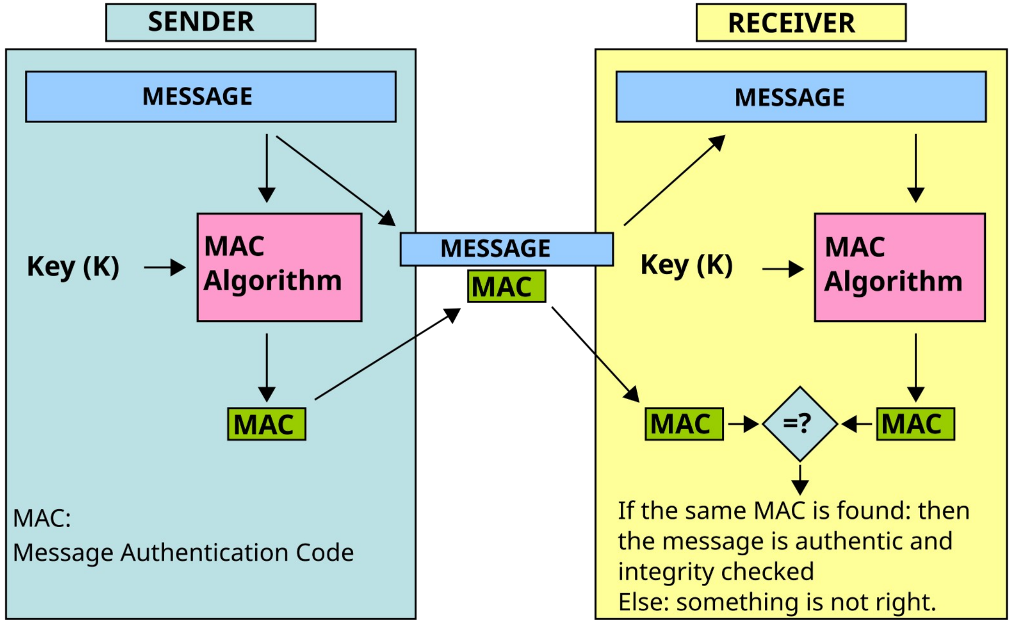
- Computes a hash for a message using a symmetric key

- Provide authentication in addition to the integrity

  ⇒guarantee that the sender sent the message

  ⇒or: guarantee that it was us who wrote a file

MAC
How do they work?



MAC:
Message Authentication Code

If the same MAC is found: then the message is authentic and integrity checked
Else: something is not right.

- Again, the tricky part is how to transmit the key.
- Hence makes more sense in single-party scenarios:
  - *We* write a file to disk, create a MAC, and validate it when reading the file again.
  - In this scenario can safely store MAC unencrypted with the file, as long as the key is kept secret.

Types of MACs

- CMAC
  - Based on Block Ciphers
  - Most often used: MAC using AES in CBC Mode (called CBC-MAC)

- HMAC
  - also called keyed Hashes
  - Based on Hashing Algorithms
  - Key + message are being hashed together

Combinations of Ciphers and MACs

- Encryption provides confidentiality

- MACs provide authentication + integrity

- Combine both to achieve all of three of them

- Also called authenticated encryption

Combinations of Ciphers and MACs

- Three possible ways to combine them:

- **Encrypt-then-MAC:**
  First encrypt plaintext, then MAC the resulting ciphertext, then append it to the ciphertext

- **Encrypt && MAC:**
  Encrypt and MAC the plaintext and append the MAC to the ciphertext

- **MAC-then-Encrypt:**
  MAC the plaintext, then encrypt both the tag and the plaintext

- Encrypt-then-MAC is most secure, as it gives integrity of both the ciphertext and (indirectly) plaintext.

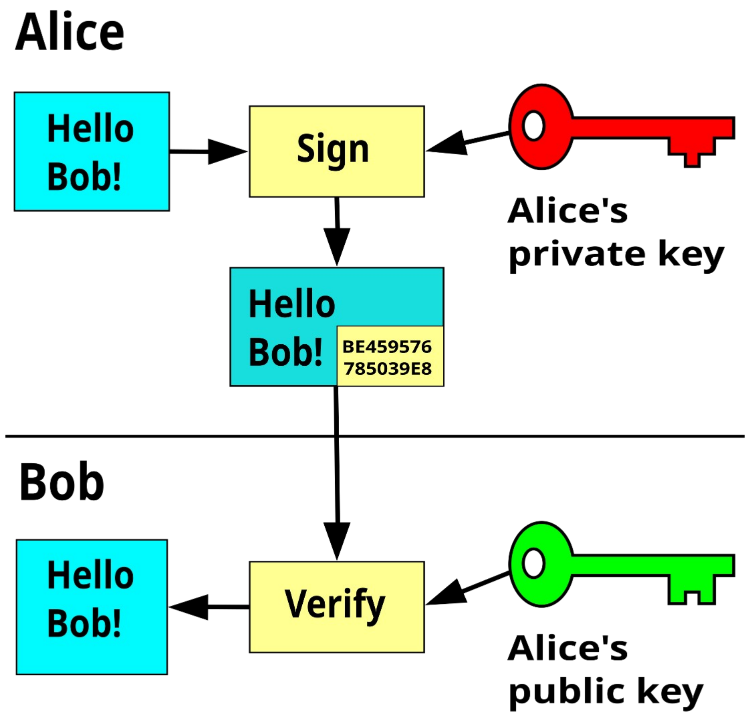
Signatures

Digital Signatures

- Signature created with private key by hashing the message and „encrypting“ the hash
- Signature verified with public key by „decrypting“ the received „encrypted“ message and hashing the original message
- Provide non-repudiation, authentication, and integrity

  => Ensure identity of data and its sender

Digital Signatures
How do they work?



- Tricky Part: How to trust public key?

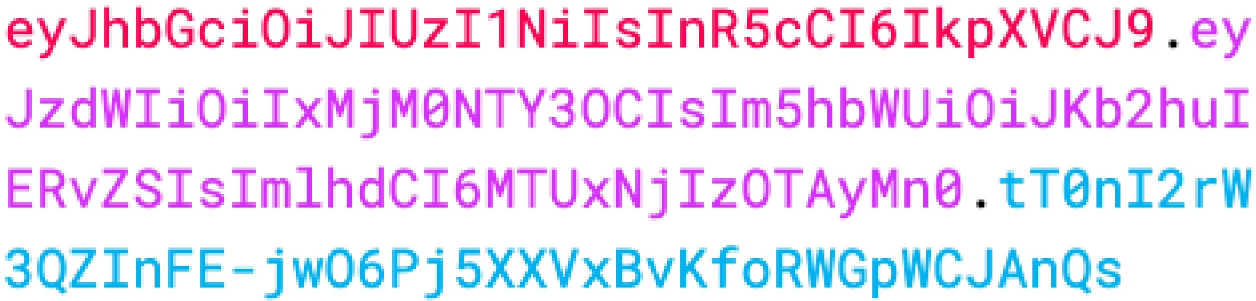
Example JSON Web Token (JWT)

- Open standard for securely transmitting information
- Digitally signed through
  - Shared secret
  - Private/Public key pair
- Used for
  - Authorization
  - Information exchange

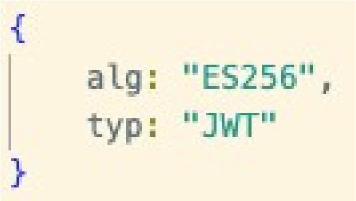
Example JSON Web Token (JWT)

- Structure
  - Header.Payload.Signature
  - Header: Token type, Signing Algorithm (e.g., HMAC SHA256)
  - Payload: Contains claims (statements about the entity, e.g., user),
    - Types: Registered (iss, exp, sub, aud,…), Public claims, Private claims
    - Base64 encoded -> Readable by anyone!

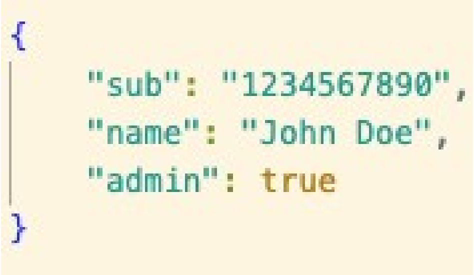
https://jwt.io/



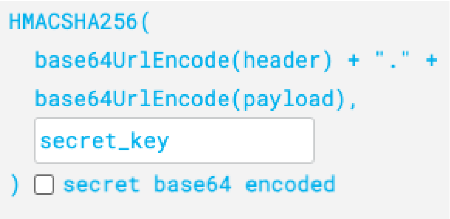
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3OCIsIm5hbWUiOiJKb2huI
ERvZSIsImlhdCI6MTUxNjIzOTAyMn0.tT0nI2rW
3QZInFE-jwO6Pj5XXVxBvKfoRWGpWCJAnQs
```



```
{
    alg: "ES256",
    typ: "JWT"
}
```

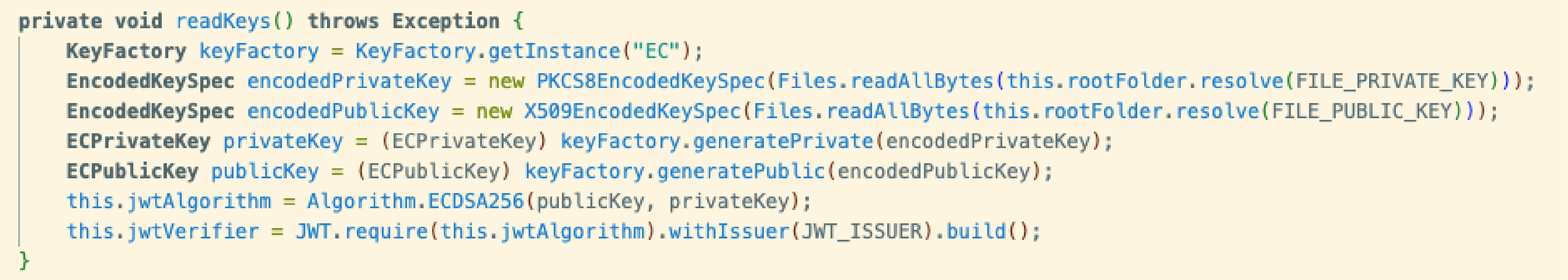


```
{
    "sub": "1234567890",
    "name": "John Doe",
    "admin": true
}
```

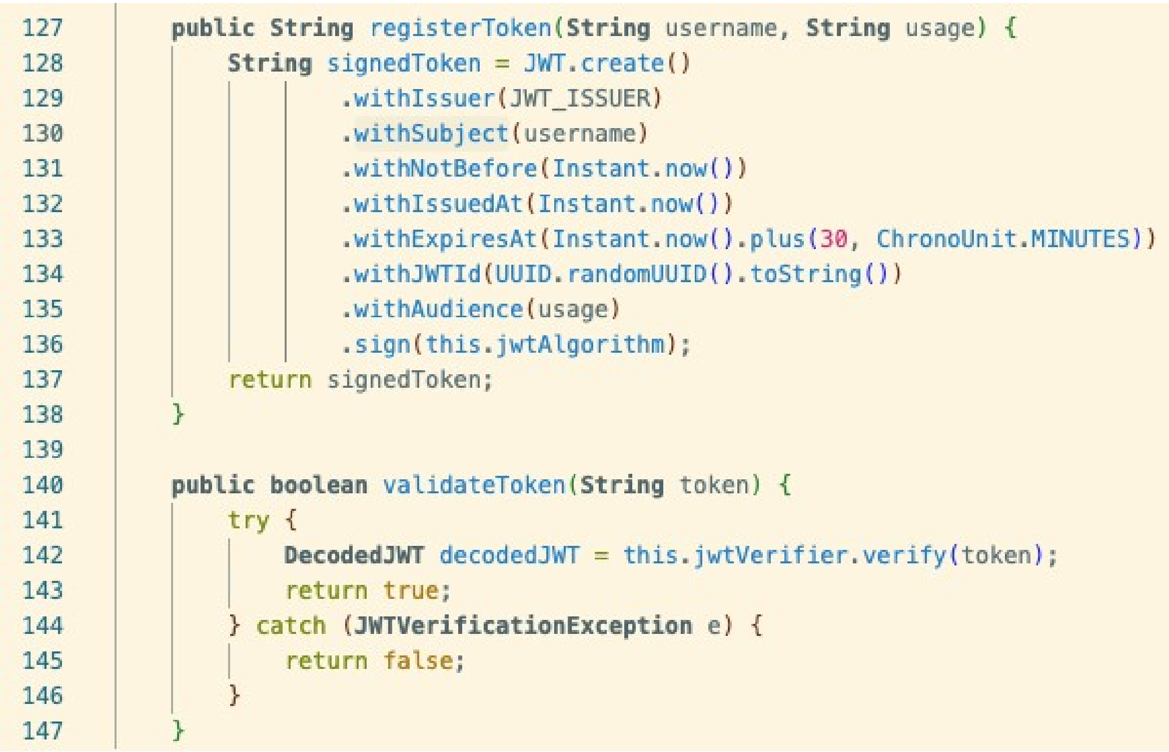


```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret_key
) □ secret base64 encoded
```

Example JSON Web Token (JWT)



```java
private void readKeys() throws Exception {
    KeyFactory keyFactory = KeyFactory.getInstance("EC");
    EncodedKeySpec encodedPrivateKey = new PKCS8EncodedKeySpec(Files.readAllBytes(this.rootFolder.resolve(FILE_PRIVATE_KEY)));
    EncodedKeySpec encodedPublicKey = new X509EncodedKeySpec(Files.readAllBytes(this.rootFolder.resolve(FILE_PUBLIC_KEY)));
    ECPrivateKey privateKey = (ECPrivateKey) keyFactory.generatePrivate(encodedPrivateKey);
    ECPublicKey publicKey = (ECPublicKey) keyFactory.generatePublic(encodedPublicKey);
    this.jwtAlgorithm = Algorithm.ECDSA256(publicKey, privateKey);
    this.jwtVerifier = JWT.require(this.jwtAlgorithm).withIssuer(JWT_ISSUER).build();
}
```



```java
    public String registerToken(String username, String usage) {
        String signedToken = JWT.create()
                .withIssuer(JWT_ISSUER)
                .withSubject(username)
                .withNotBefore(Instant.now())
                .withIssuedAt(Instant.now())
                .withExpiresAt(Instant.now().plus(30, ChronoUnit.MINUTES))
                .withJWTId(UUID.randomUUID().toString())
                .withAudience(usage)
                .sign(this.jwtAlgorithm);
        return signedToken;
    }

    public boolean validateToken(String token) {
        try {
            DecodedJWT decodedJWT = this.jwtVerifier.verify(token);
            return true;
        } catch (JWTVerificationException e) {
            return false;
        }
    }
```

Trusting Keys

- Secret Keys
  - Underlying assumption is that only the involved parties have the key

    => source of trust is secrecy

- Public Keys
  1) Only trust keys you have personally received (is this always viable?)
  2) One signs public keys one trusts, others who trust you can trust keys you trust - Web of Trust
  3) Have trusted locations for public keys e.g. key servers
  4) Central(-ish) authorities sign keys to guarantee their authenticity – guarantees are called certificates

Keeping Up

- Crypto algorithms are constantly changing
  - New crypto protocols, new models
  - Broken crypto algorithms (recent example GnuPG!)

- You will need to keep up with the news on algorithms
  - Organizations: CWE, OWASP, NIST
  - Bloggers & Researchers
    - Bruce Schneier: http://www.schneier.com/
    - Steve Gibson: http://www.grc.com/news.htm
    - Gary McGraw: www.cigital.com, IEEE Privacy & Security

Takeaway:

- Use standardized crypto algorithms instead of inventing your own
- Encryption ensures confidentiality, but usually neither integrity nor authenticity
- MAC ensures integrity and authenticity, but neither confidentiality nor (cryptographic) non-repudiation
- Digital signature ensures integrity, authenticity and (cryptographic) non-repudiation, but no confidentiality
- Choice of appropriate primitives, algorithms and their instantiations is crucial and heavily dependent on the application context

Thank you