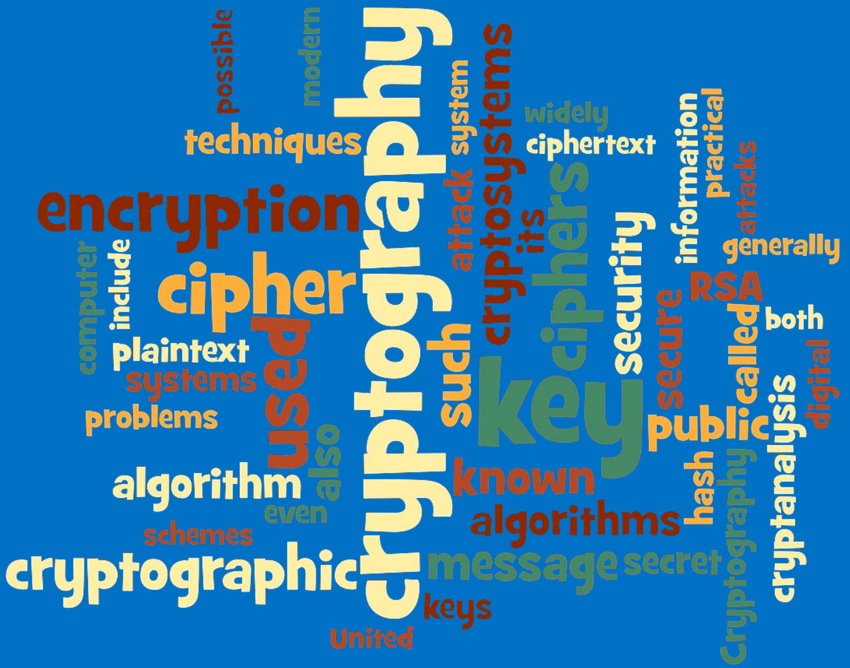
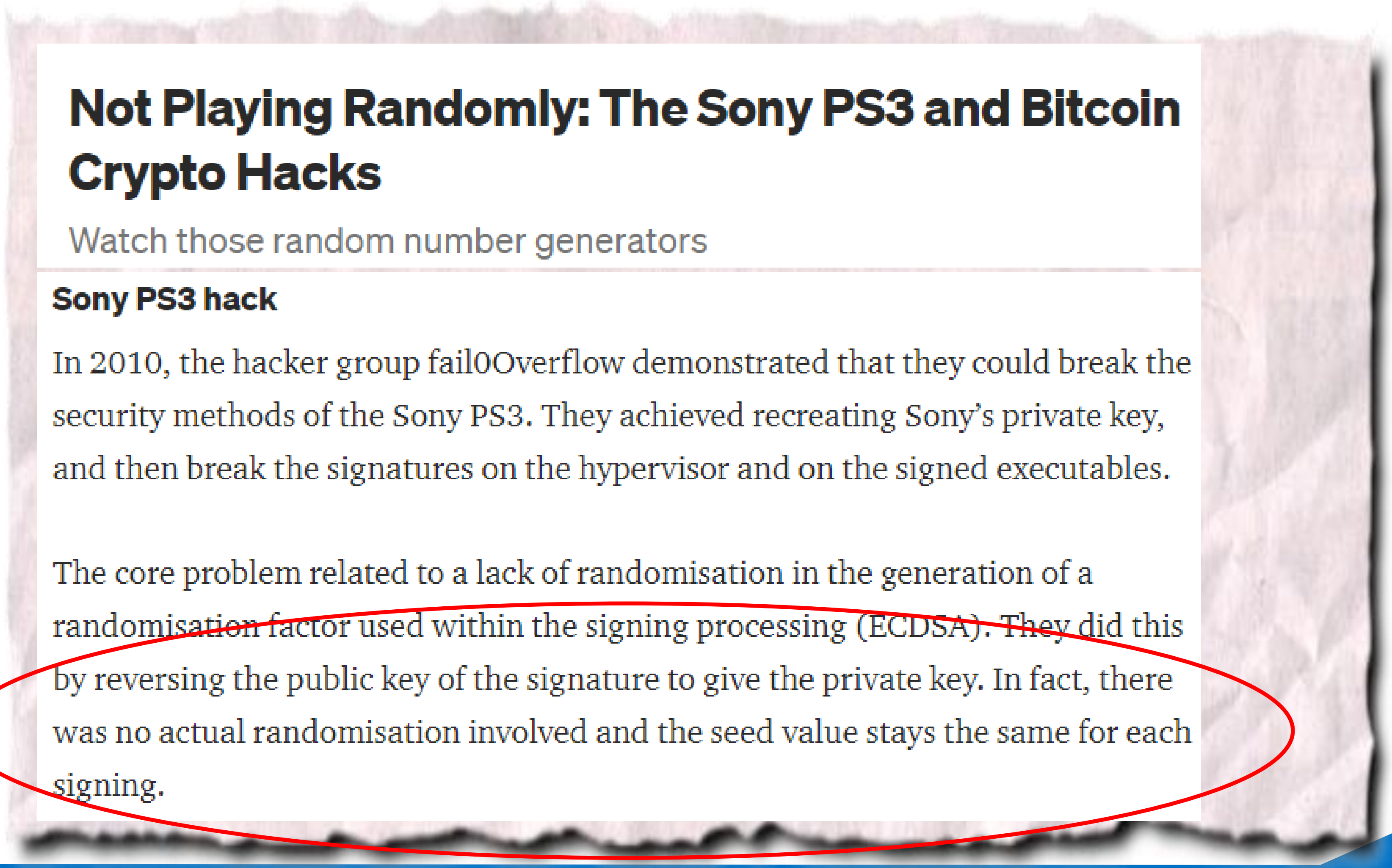
APPLIED CRYPTOGRAPHY EXERCISE



Why?

Many developers are required to design and develop code that implements or uses cryptographic functions

In many cases insufficient practical crypto coding knowledge leads to implementation issues that help hackers to break the system



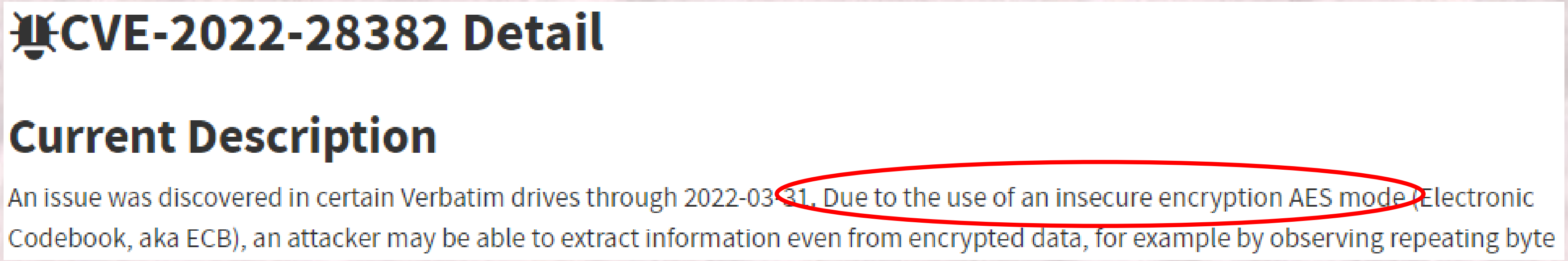
Not Playing Randomly: The Sony PS3 and Bitcoin Crypto Hacks

Watch those random number generators

Sony PS3 hack

In 2010, the hacker group fail0Overflow demonstrated that they could break the security methods of the Sony PS3. They achieved recreating Sony's private key, and then break the signatures on the hypervisor and on the signed executables.

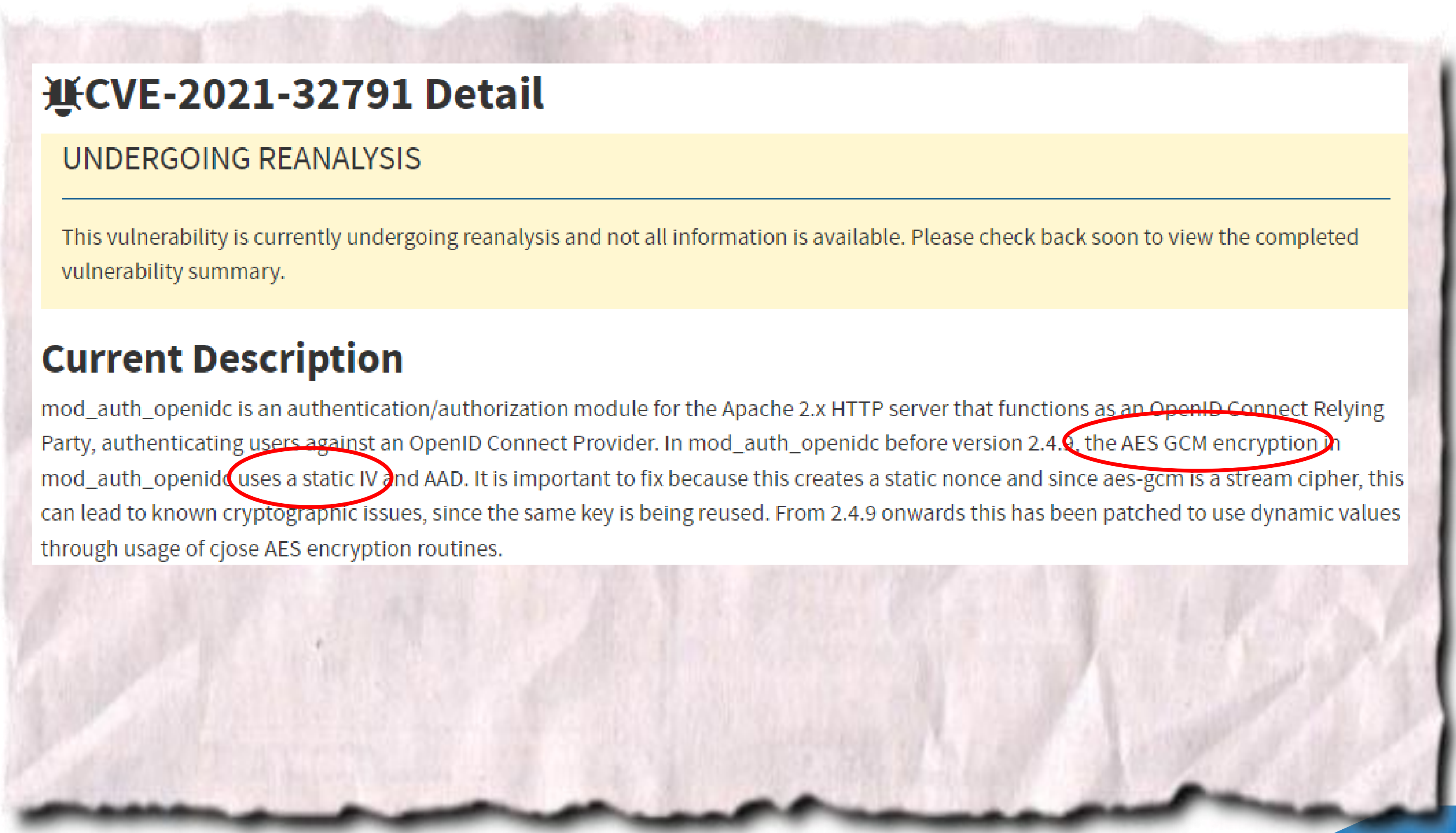
The core problem related to a lack of randomisation in the generation of a randomisation factor used within the signing processing (ECDSA). They did this by reversing the public key of the signature to give the private key. In fact, there was no actual randomisation involved and the seed value stays the same for each signing.



CVE-2022-28382 Detail

Current Description

An issue was discovered in certain Verbatim drives through 2022-03-31. Due to the use of an insecure encryption AES mode (Electronic Codebook, aka ECB), an attacker may be able to extract information even from encrypted data, for example by observing repeating byte



CVE-2021-32791 Detail

UNDERGOING REANALYSIS

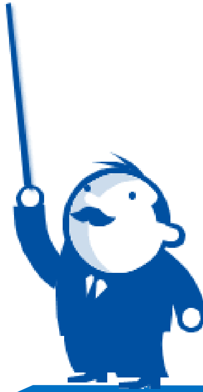
This vulnerability is currently undergoing reanalysis and not all information is available. Please check back soon to view the completed vulnerability summary.

Current Description

mod_auth_openidc is an authentication/authorization module for the Apache 2.x HTTP server that functions as an OpenID Connect Relying Party, authenticating users against an OpenID Connect Provider. In mod_auth_openidc before version 2.4.9, the AES GCM encryption in mod_auth_openidc uses a static IV and AAD. It is important to fix because this creates a static nonce and since aes-gcm is a stream cipher, this can lead to known cryptographic issues, since the same key is being reused. From 2.4.9 onwards this has been patched to use dynamic values through usage of cjose AES encryption routines.

What will you learn in this exercise

- Practice using recommended **cryptographic algorithms**
- Familiarize through practice with popular cryptographic libraries APIs



Course Outline

Disclaimer

The following scenarios are designed to show you basic concepts but is not designed to be used for production as is.

Products should follow guidelines and applicable standards.

CRYPTO RECAP

Cryptographic “toolbox” overview – or what would we need for our exercises

Short recap of crypto “tools”

AES-GCM

SHA

MAC

Digital signatures

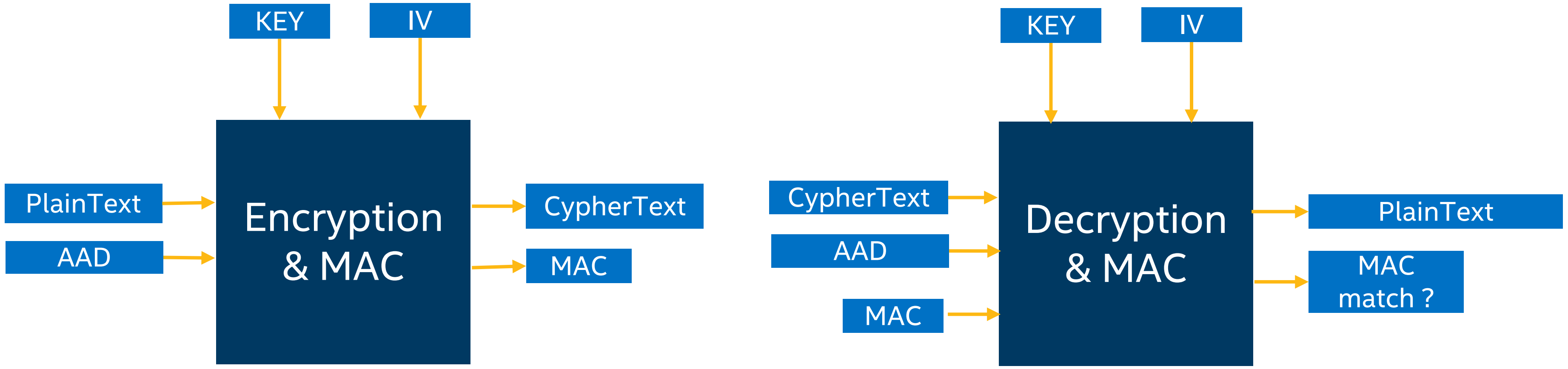
Certificates

KDF

DH

SIGMA

AES-GCM - Galois/Counter Mode



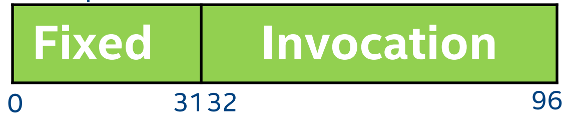
- Combined Encryption and MAC with same operation using the same key - uses CTR mode
- It's a non-deterministic mode - changing IV → different ciphertext and MAC for the same plaintext (IV is not secret)
- Never repeat same IV and Key combination
- Additional Authenticated Data (AAD) is not encrypted but used for MAC generation
- Can provide authentication only by using AAD only → GMAC
- Ciphertext size == plaintext size (no need for padding)

AES-GCM – Uniqueness Requirement on IVs and Keys

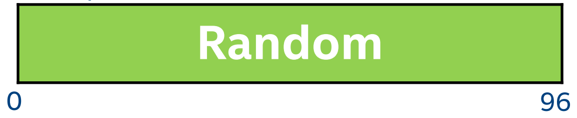
- Prefer 96bit IVs
- IV construction, can be either:
  - **Deterministic construction**
    - IV is a concatenation of 2 fields: Fixed and Invocation
    - Invocation counter: typically, an integer counter or linear feedback shift register
  - **RBG-based construction**
    - IV is a concatenation of
      - Free field – may be empty
      - Random field – at least 96 bits

      Length of these fields must be fixed for the life of the key

Example: Deterministic construction



Example: RBG construction



Strongly recommended AES sizes and modes

- 256 bits for key
- Always prefer authenticated encryption
  - Galois counter mode (GCM) or Counter with CBC-MAC mode (CCM)
  - For Counter mode (CTR) and Cipher block chaining mode (CBC) – use MAC in addition

AES-GCM256 is the recommended choice for cipher block mode by NIST!

SHA allowed usages

Recommended modes

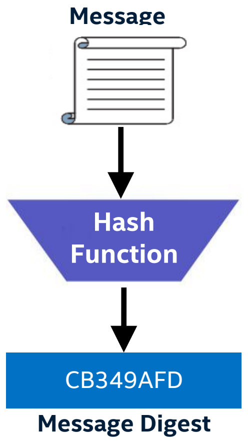
- SHA-2 (SHA2-384, SHA2-512)
- SHA-3 (SHA3-384, SHA3-512)

Should not be used:

- ~~MD2, MD4, MD5, MD6~~
- ~~SHA0, SHA1, SHA2-224, SHA3-224~~

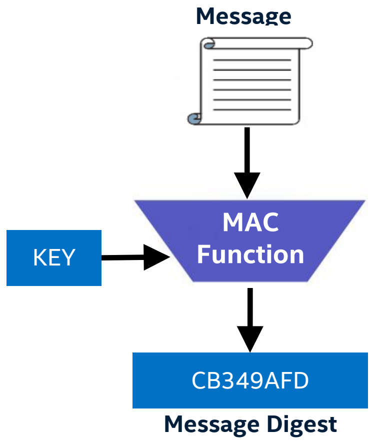
SHA2-256 and SHA3-256 are considered acceptable for:

- HMAC

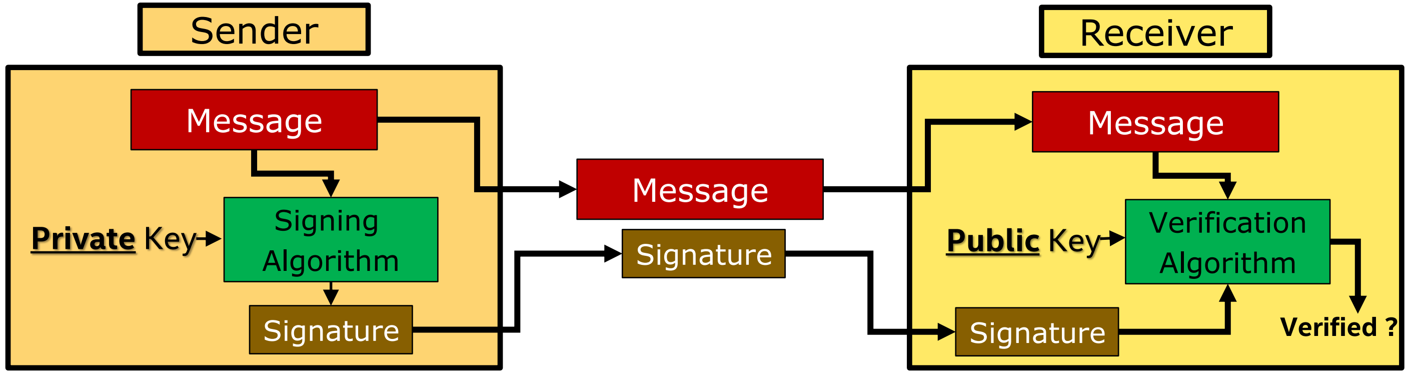


MAC types

MAC type	Core function used	Generated MAC size	Key size [bits]
HMAC	**uses Hash function of a specific size. E.g.: SHA2-256 SHA2-512**	**Based on the hash function size used. E.g. HMAC-SHA2-256 → 256 bits HMAC-SHA2-512 → 512 bits**	**At least: 256 512**
CMAC	uses AES-CBC function of a specific key size. E.g. CMAC256	Irrespective of the AES cipher key size used. Always 128 bits	256
GMAC	uses AES-GCM function of a specific key size. E.g. GMAC256	Irrespective of the AES cipher key size used. Always 128 bits	256
KMAC	Uses SHA3 function of specific size 128, 256, 512 KMAC256 (K, text, L, S)	Based on the hash function size used and indicated by the user, must be > key size	At least: 128 256 512



Digital Signatures

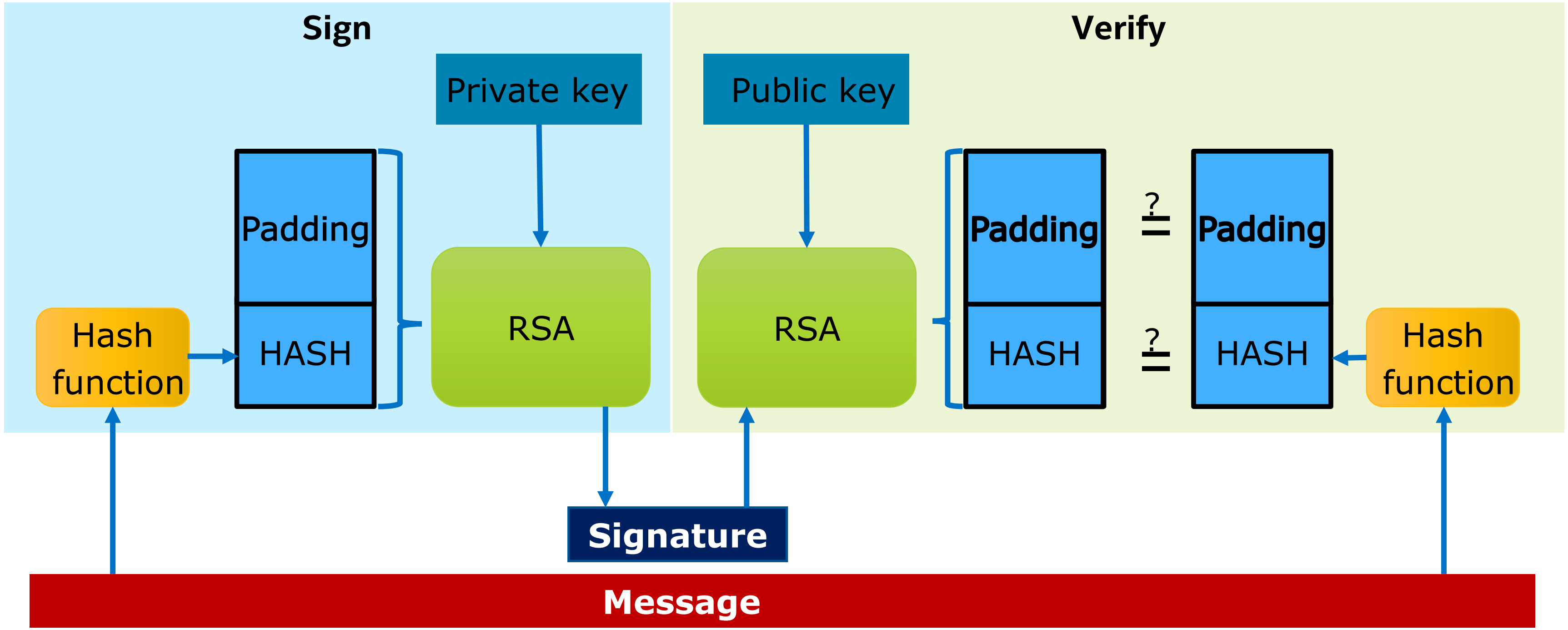


Provides:

- **Authenticity** – The message came from the stated sender
- **Integrity** – The message has not been changed
- **Non-repudiation** - The sender cannot dispute its authorship

A successful authentication guarantees that the message was signed by the owner of the private key that corresponds to the public key found in the certificate.

RSA Signatures



RSA practical aspects

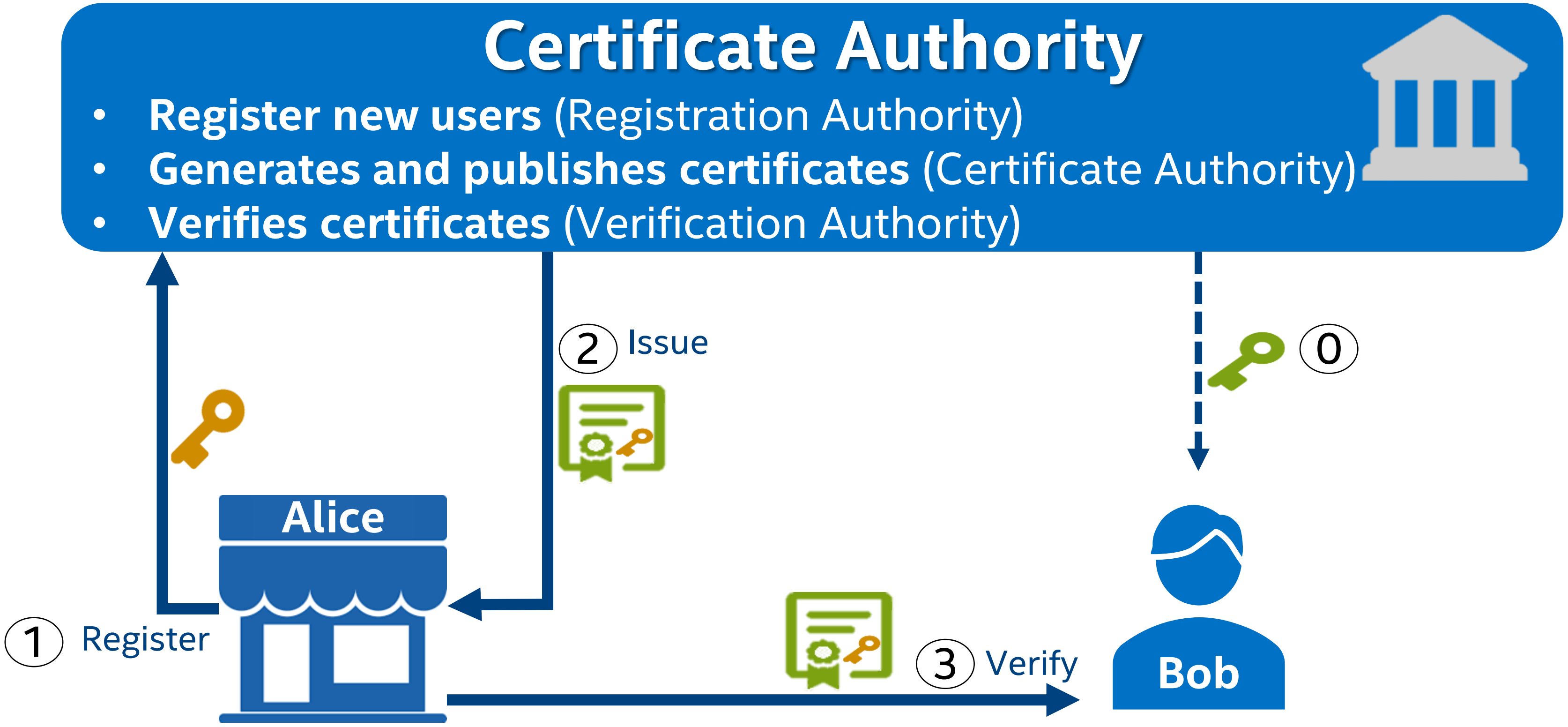
Padding:

- Schoolbook RSA – why padding is important? – Do not use!!!
- PKCS#1_v1.5 (both for encryption and signing) – Do not use for encryption
- PSS (signing) – Can use
- OAEP (encryption) – Can use

Key structure:

- Standard key:
  - Private: (d, N)
  - Public (e, N)
- CRT
  - Private: (d_p, d_q, q^{-1}, N)

PKI


Certificate Authority
• **Register new users** (Registration Authority)
• **Generates and publishes certificates** (Certificate Authority)
• **Verifies certificates** (Verification Authority)
2 Issue
0
Alice
1 Register
3 Verify
Bob

KDF - two phases approach

Phase 1 - Entropy extraction phase

- To address the gaps of the initial secret
- Applied to a weakly random entropy source, together with a salt to generate a highly random output that appears independent from the source and uniformly distributed

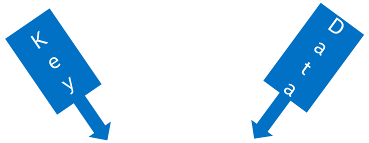
Phase 2 – Key expansion phase

- Generate keys from the Entropy extraction phase output and key context
- Each key is independent from the input and the other generated keys
- Examples: Different key for different purpose, or different key per new version

HKDF - HMAC based Key Derivation Function

$K(1) \,||\, K(2) \,||\, \ldots \,||\, K(t) = \mathrm{HKDF}(\mathrm{XTSalt}; \mathrm{SKM}; \mathrm{CTXinfo}; L)$

- $t = \lceil L/k \rceil$, where k is HMAC key size (== HMAC output)
- XTSalt is optional (either k bits value, or k 0 bits for constrained environments)



Calculation:

PRK = HMAC-SHA(XTSalt, SKM) } Phase 1 - Entropy extraction phase

$$\left.\begin{aligned} &K(1) = \text{HMAC-SHA}(\text{PRK}, \text{CTXinfo} \,||\, 1) \\ &K(i+1) = \text{HMAC-SHA}(\text{PRK}, K(i) \,||\, \text{CTXinfo} \,||\, i+1),\ 1 < i \le t \end{aligned}\right\}$$

Phase 2 – Key expansion phase

Symmetric key generation from a shared secret

Given a shared secret – SKM (Secret Key Material), generate the key using a KDF function

Examples:

K_{enc} = *HKDF(salt, SKM; EncContextInfo; KeyLength)*

K_{MAC} = *HKDF(salt, SKM; MacContextInfo; KeyLength)*

or

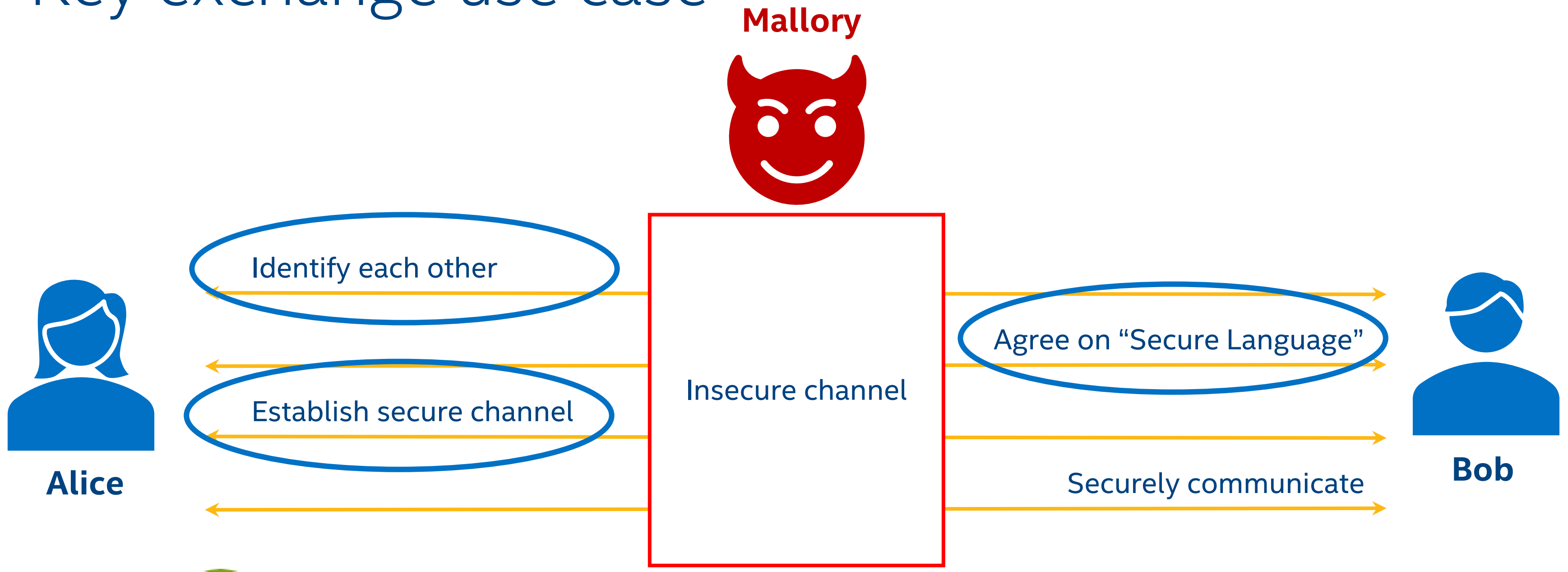
$K_{enc} || K_{MAC}$ = *HKDF(salt, SKM; ContextInfo; KeyLength x 2)*

Prefer the use of multiple independent entropy sources for salt and SKM

Practical notes for HKDF

- If SKM is already a real random, one can skip the extraction phase, but not recommended
- Recommendation is still to use the extraction phase!
- Fill relevant info in CTXInfo
- Length (L) of the desired key can be used within CTXInfo, especially if same inputs are used for different length keys (which is not the common practice)

Key exchange use case



What part of the protocol is key exchange?

Key Exchange Requirements

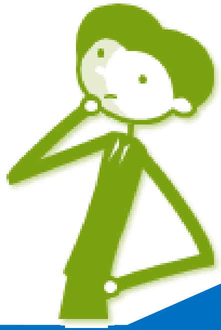
1. Establish a shared secret between two parties
2. Authenticate the other party
3. Forward secrecy

Diffie-Hellman

Digital signatures

Ephemeral keys

What cryptographic methods can you use for each requirement?



Diffie-Hellman principles - mathematical difficulty

Based on Discrete Logarithm Problem mathematical difficulty

Generalized to any finite cyclic group, e.g. 0,1,..., *p*-1 (p is a prime) or points on a discrete Elliptic Curve

Given *p*, *g and y*, *such as* $\boldsymbol{y = g^x \bmod p}$ – find *x*

$$g, g^2, g^3, \ldots, \underbrace{g^{340282366920938463463374607431768211456}}_{\uparrow}, \ldots \bmod p$$

DH Private Key

Diffie-Hellman principles – shared secret

DH shared secret calculation is based on the commutativity property

$$\text{Shared secret} = g^{xy} = (g^x)^y = (g^y)^x$$

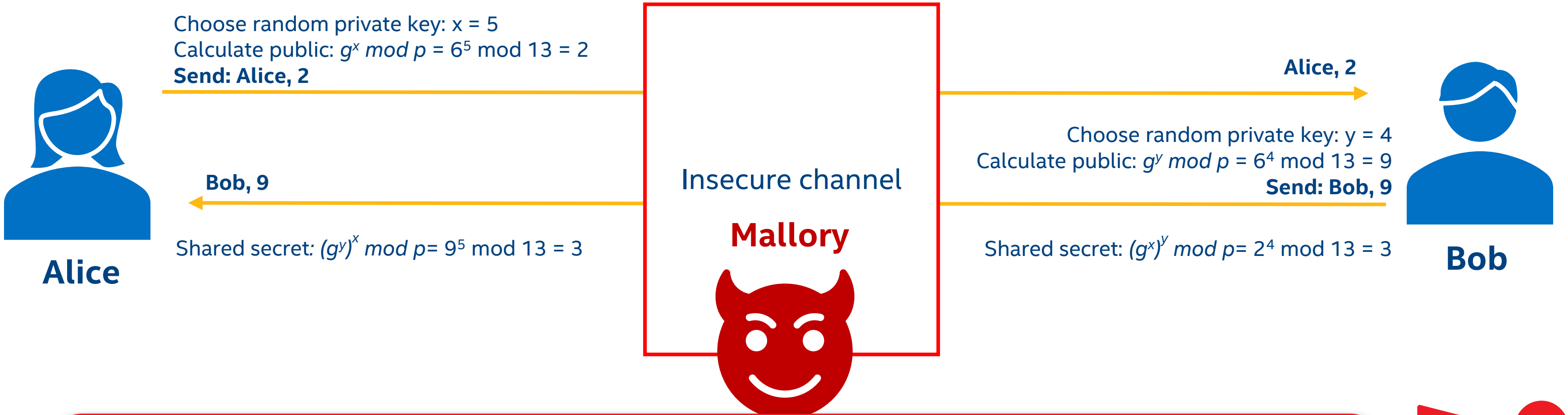
So if each party knows own private key and the other's party public key – it can calculate the shared secret.

Mallory can't calculate the shared secret from g^x and g^y

	Party A knowledge	Party B knowledge	MiM knowldge
Private Key	x	y	-
Public Key	g^x	g^y	-
Other party public key	g^y	g^x	g^y, g^x
Shared Secret	$g^{xy} = (g^y)^x$	$g^{xy} = (g^x)^y$	$g^{xy} = ?$

Diffie-Hellman example

Alice and Bob agreed upfront on group parameters:
$p = 13$ (modulus)
$g = 6$ (generation element)



Choose random private key: x = 5
Calculate public: $g^x \mathit{mod}\ p = 6^5 \bmod 13 = 2$
Send: Alice, 2

Choose random private key: y = 4
Calculate public: $g^y \mathit{mod}\ p = 6^4 \bmod 13 = 9$
Send: Bob, 9

Shared secret: $(g^y)^x \mathit{mod}\ p = 9^5 \bmod 13 = 3$

Shared secret: $(g^x)^y \mathit{mod}\ p = 2^4 \bmod 13 = 3$

Never use the shared secret as a cryptographic key!
Always use a standard Key Derivation Function (KDF) to derive cryptographic keys from a shared secret!

Diffie-Hellman example

Alice and Bob agreed upfront on:
p = 13 (modulus)
g = 6 (generation element)

Choose random private key: x = 5
Calculate public: $g^x \bmod p = 6^5 \bmod 13 = 2$
Send: Alice, 2

Alice, 2

Choose random private key: y = 4
Calculate public: $g^y \bmod p = 6^4 \bmod 13 = 9$
Send: Bob, 9

Bob, 9

Insecure channel

Mallory

Shared secret: $(g^y)^x \bmod p = 9^5 \bmod 13 = 3$

Shared secret: $(g^x)^y \bmod p = 2^4 \bmod 13 = 3$

Alice

Bob

Can we use this protocol for establishing a secure channel? Why?

Diffie-Hellman identity problem

Mallory

Choose random private: x = 5
Calculate public: g^x *mod p* = 6^5 mod 13 = 2
Send: Alice, 2

Bob, 8

Shared secret: $(g^y)^x$ mod p= 8^5 mod 13 = **8**

Alice

Choose random private key:
x' = 2 → $g^{x'}$ *mod p* = 10

Choose random private key:
y' = 3 → $g^{y'}$ *mod p* = 8

Insecure channel

Shared key with Alice:
2^3 mod 13 = 8

Shared key with Bob:
9^2 mod 13 = 3

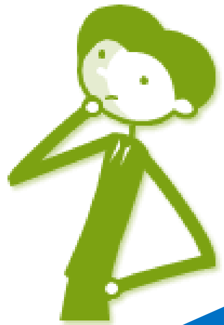
Alice, 10

Choose random private: y = 4
Calculate public: g^y *mod p* = 6^4 mod 13 = 9
Send: Bob, 9

Shared secret: $(g^x)^y$ *mod p* = 10^4 mod 13 = **3**

Bob

How would you solve this Man In the Middle problem?



Diffie-Hellman identity problem solution

The solution has to cryptographically "bind together" the message parts

Seems like a simple task

Several attempts were done

Each had mistakes, eventually leading to SIGn and MAc (SIGMA) definition

SIGMA took the good and learned from the bad of previous protocols

SIGMA

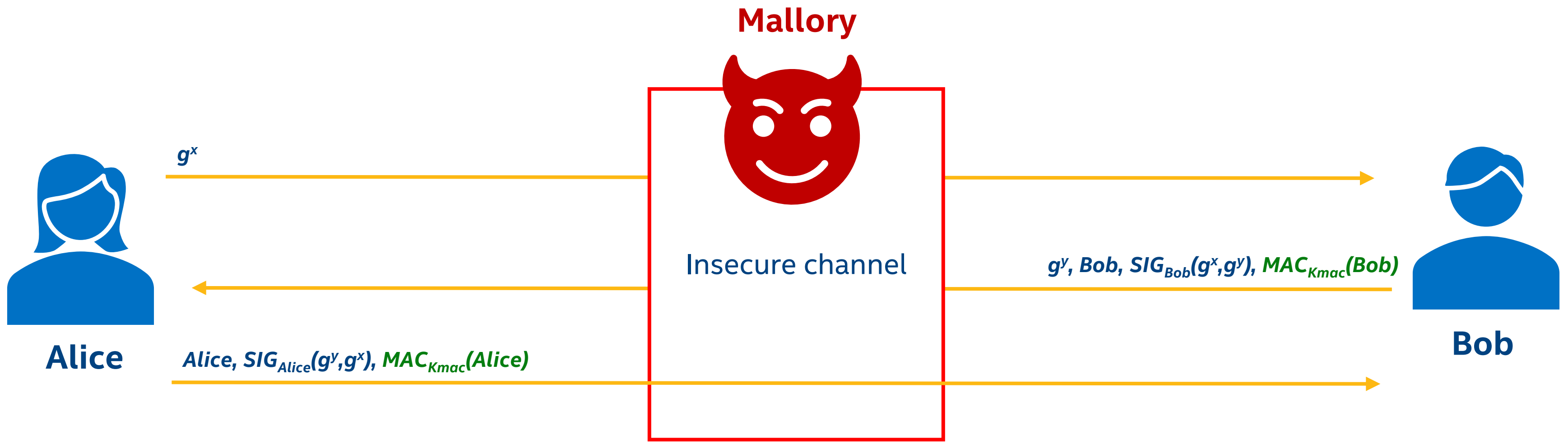
Need to bind the derived keys (K) with the peer identities

SIGn and MAc

- Sign the two ephemeral public keys with own identity
- MAC own identity with a key derived from the shared secret

SIGMA – basic version

$K_{mac} = KDF_{mac}(g^{xy})$

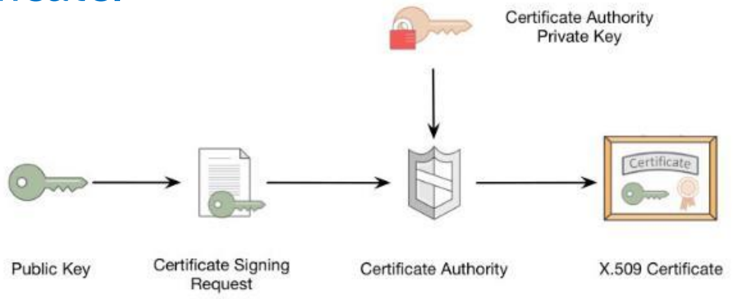

Mallory
Insecure channel
g^x
g^y, Bob, $SIG_{Bob}(g^x,g^y)$, $MAC_{Kmac}(Bob)$
Alice, $SIG_{Alice}(g^y,g^x)$, $MAC_{Kmac}(Alice)$
Alice
Bob

EXERCISE #1 – CREATING DIGITAL CERTIFICATES

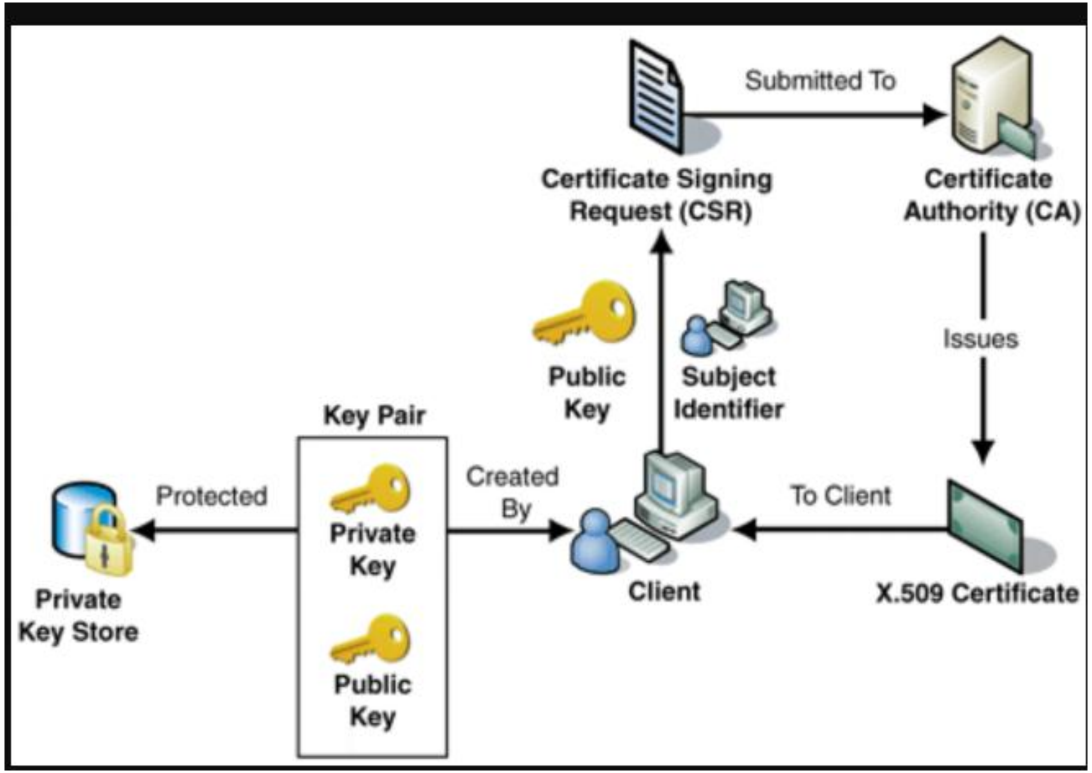
Theory and practice

Certificate Signing Request

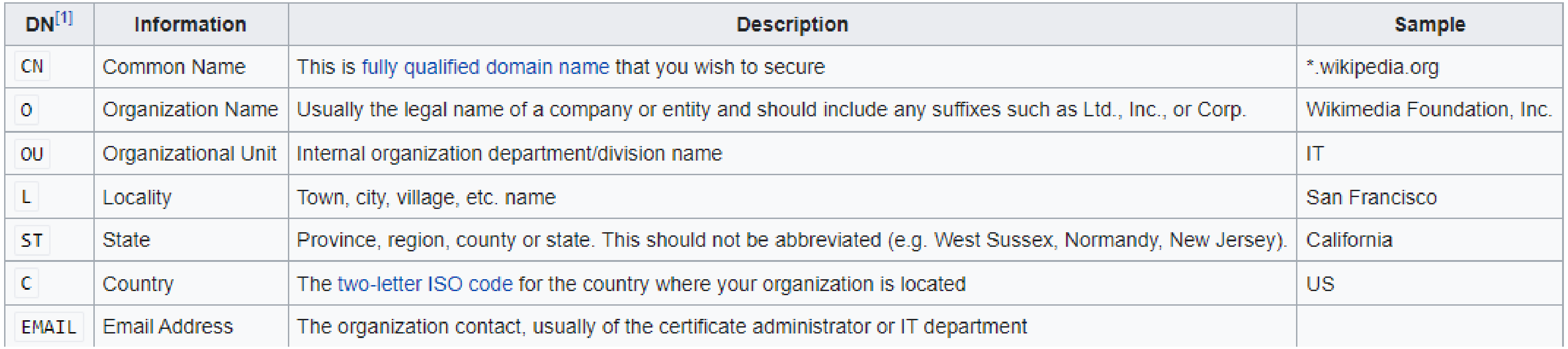
- A certificate signing request (CSR) is a message sent to a certificate authority to request the signing of a public key and associated information
- Most commonly a CSR will be in a PKCS10 format
- The contents of a CSR comprises a public key, as well as a common name, organization, city, state, country, and e-mail
- Not all these fields are required and will vary depending on the assurance level of your certificate.


Certificate Authority Private Key
Certificate
Public Key
Certificate Signing Request
Certificate Authority
X.509 Certificate

Certificate creation flow



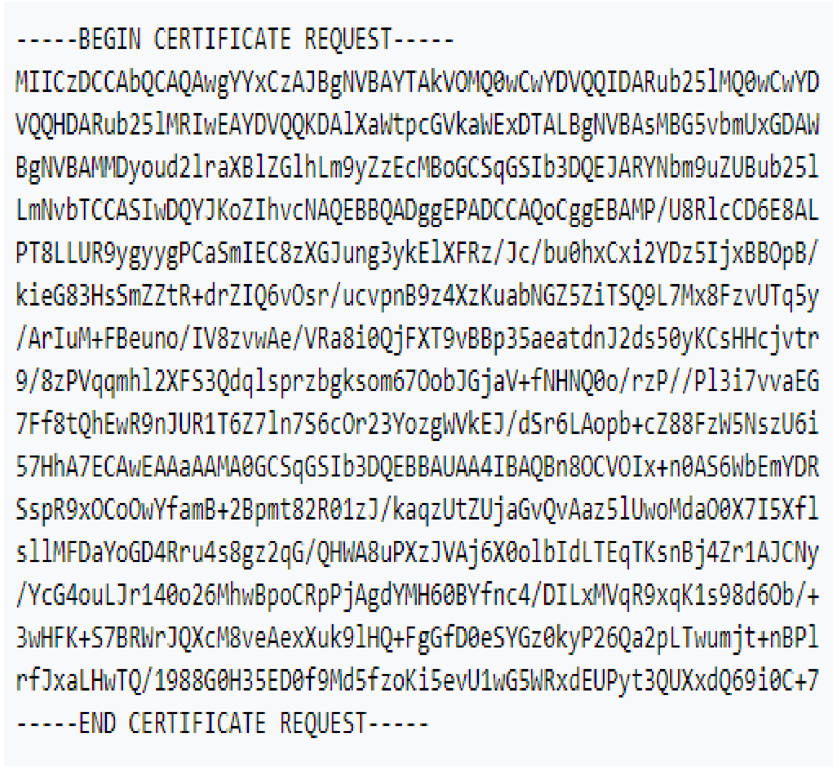
CSR fields



DN[1]	Information	Description	Sample
`CN`	Common Name	This is fully qualified domain name that you wish to secure	*.wikipedia.org
`O`	Organization Name	Usually the legal name of a company or entity and should include any suffixes such as Ltd., Inc., or Corp.	Wikimedia Foundation, Inc.
`OU`	Organizational Unit	Internal organization department/division name	IT
`L`	Locality	Town, city, village, etc. name	San Francisco
`ST`	State	Province, region, county or state. This should not be abbreviated (e.g. West Sussex, Normandy, New Jersey).	California
`C`	Country	The two-letter ISO code for the country where your organization is located	US
`EMAIL`	Email Address	The organization contact, usually of the certificate administrator or IT department	

CSR Format

- The CSR itself is usually created in a Base-64 based PEM format.
- You can open the CSR file using a simple text editor and it will look like the sample below.
- You must include the header and footer (-----BEGIN NEW CERTIFICATE REQUEST-----) when pasting the CSR.



```
-----BEGIN CERTIFICATE REQUEST-----
MIICzDCCAbQCAQAwgYYxCzAJBgNVBAYTAkVOMQ0wCwYDVQQIDARub25lMQ0wCwYD
VQQHDARub25lMRIwEAYDVQQKDAlXaWtpcGVkaWExDTALBgNVBAsMBG5vbmUxGDAW
BgNVBAMMDyoud2lraXBlZGlhLm9yZzEcMBoGCSqGSIb3DQEJARYNbm9uZUBub25l
LmNvbTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMP/U8RlcCD6E8AL
PT8LLUR9ygyygPCaSmIEC8zXGJung3ykElXFRz/Jc/bu0hxCxi2YDz5IjxBBOpB/
kieG83HsSmZZtR+drZIQ6vOsr/ucvpnB9z4XzKuabNGZ5ZiTSQ9L7Mx8FzvUTq5y
/ArIuM+FBeuno/IV8zvwAe/VRa8i0QjFXT9vBBp35aeatdnJ2ds50yKCsHHcjvtr
9/8zPVqqmhl2XFS3QdqlsprzbgksomG7OobJGjaV+fNHNQ0o/rzP//Pl3i7vvaEG
7Ff8tQhEwR9nJUR1T6Z7ln7S6cOr23YozgWVkEJ/dSr6LAopb+cZ88FzW5NszU6i
57HhA7ECAwEAAaAAMA0GCSqGSIb3DQEBBAUAA4IBAQBn8OCVOIx+n0AS6WbEmYDR
SspR9xOCoOwYfamB+2Bpmt82R01zJ/kaqzUtZUjaGvQvAaz5lUwoMdaO0X7I5Xfl
sllMFDaYoGD4Rru4s8gz2qG/QHWA8uPXzJVAj6X0olbIdLTEqTKsnBj4Zr1AJCNy
/YcG4ouLJr140o26MhwBpoCRpPjAgdYMH60BYfnc4/DILxMVqR9xqK1s98d6Ob/+
3wHFK+S7BRWrJQXcM8veAexXuk9lHQ+FgGfD0eSYGz0kyP26Qa2pLTwumjt+nBPl
rfJxaLHwTQ/1988G0H35ED0f9Md5fzoKi5evU1wG5WRxdEUPyt3QUXxdQ69i0C+7
-----END CERTIFICATE REQUEST-----
```

Exercise

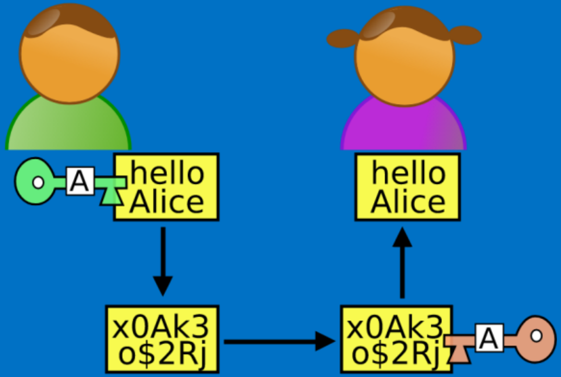
Creating Digital Certificates

1. Create a Self Signed root certificate(rootCA.crt) with RSA key size of 3072 with SHA384 and set serial number 01
2. Generate RSA keypair of size 3072 with SHA384 for “Alice” and sign its CSR with root CA and set serial number 02
3. Generate RSA keypair of size 3072 with SHA384 for “Bob” and sign its CSR with root CA and set serial number 03

```
openssl x509 -text -noout -in cert_file.crt
```

EXERCISE #2 - SECURING A CUSTOM PROTOCOL

Putting your crypto knowledge to practice



Exercise - Securing a custom protocol

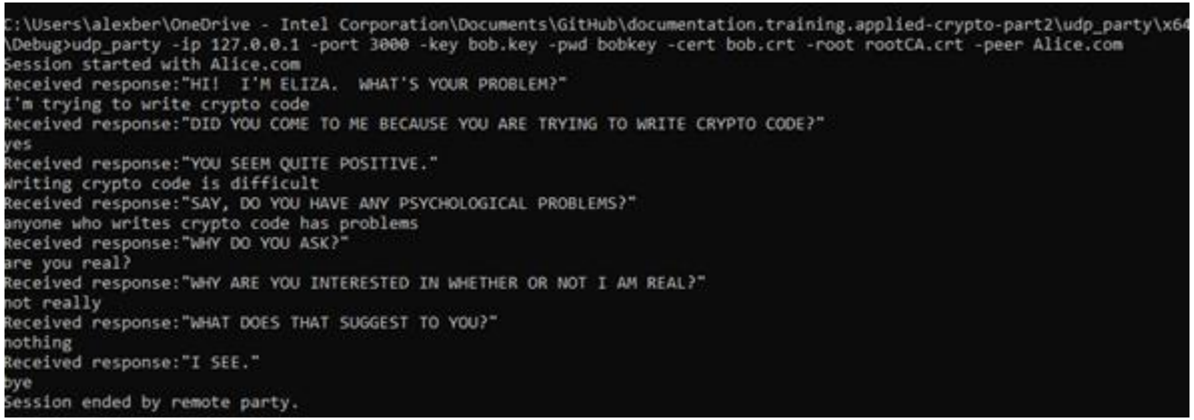
- You will be securing a custom communication protocol between two parties
  - 👤 A patient
  - 👤 An “AI” psychologist - Eliza

- Your job – to secure the traffic

- Two phases
  1. Basic - Implement crypto wrapper and make the crypto unit-test pass
  2. Advanced - Secure the protocol using your crypto wrapper implementation

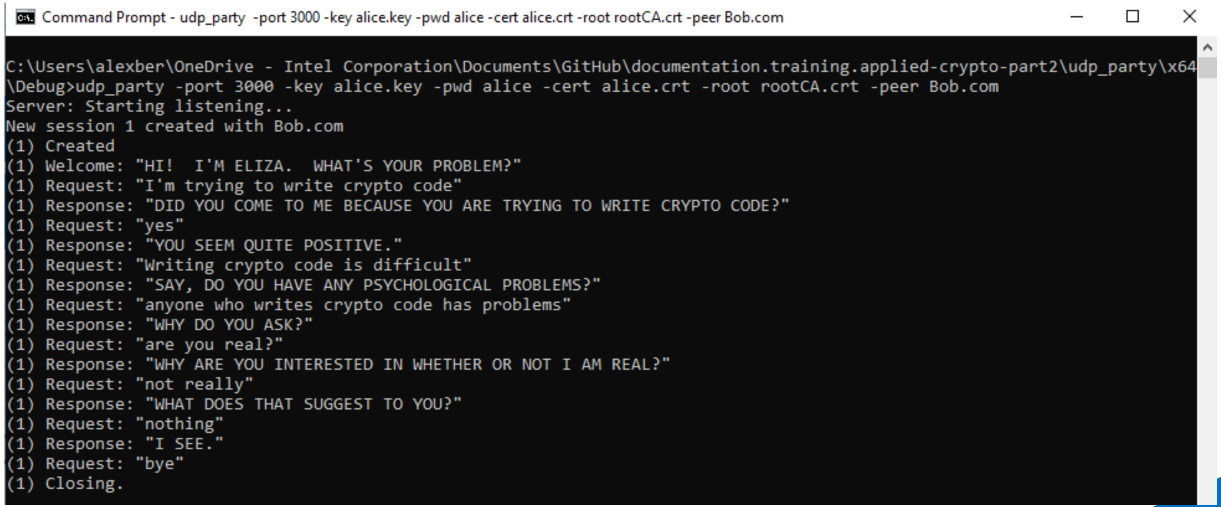
Usage

Client



```
C:\Users\alexber\OneDrive - Intel Corporation\Documents\GitHub\documentation.training.applied-crypto-part2\udp_party\x64
\Debug>udp_party -ip 127.0.0.1 -port 3000 -key bob.key -pwd bobkey -cert bob.crt -root rootCA.crt -peer Alice.com
Session started with Alice.com
Received response:"HI!  I'M ELIZA.  WHAT'S YOUR PROBLEM?"
I'm trying to write crypto code
Received response:"DID YOU COME TO ME BECAUSE YOU ARE TRYING TO WRITE CRYPTO CODE?"
yes
Received response:"YOU SEEM QUITE POSITIVE."
Writing crypto code is difficult
Received response:"SAY, DO YOU HAVE ANY PSYCHOLOGICAL PROBLEMS?"
anyone who writes crypto code has problems
Received response:"WHY DO YOU ASK?"
are you real?
Received response:"WHY ARE YOU INTERESTED IN WHETHER OR NOT I AM REAL?"
not really
Received response:"WHAT DOES THAT SUGGEST TO YOU?"
nothing
Received response:"I SEE."
bye
Session ended by remote party.
```

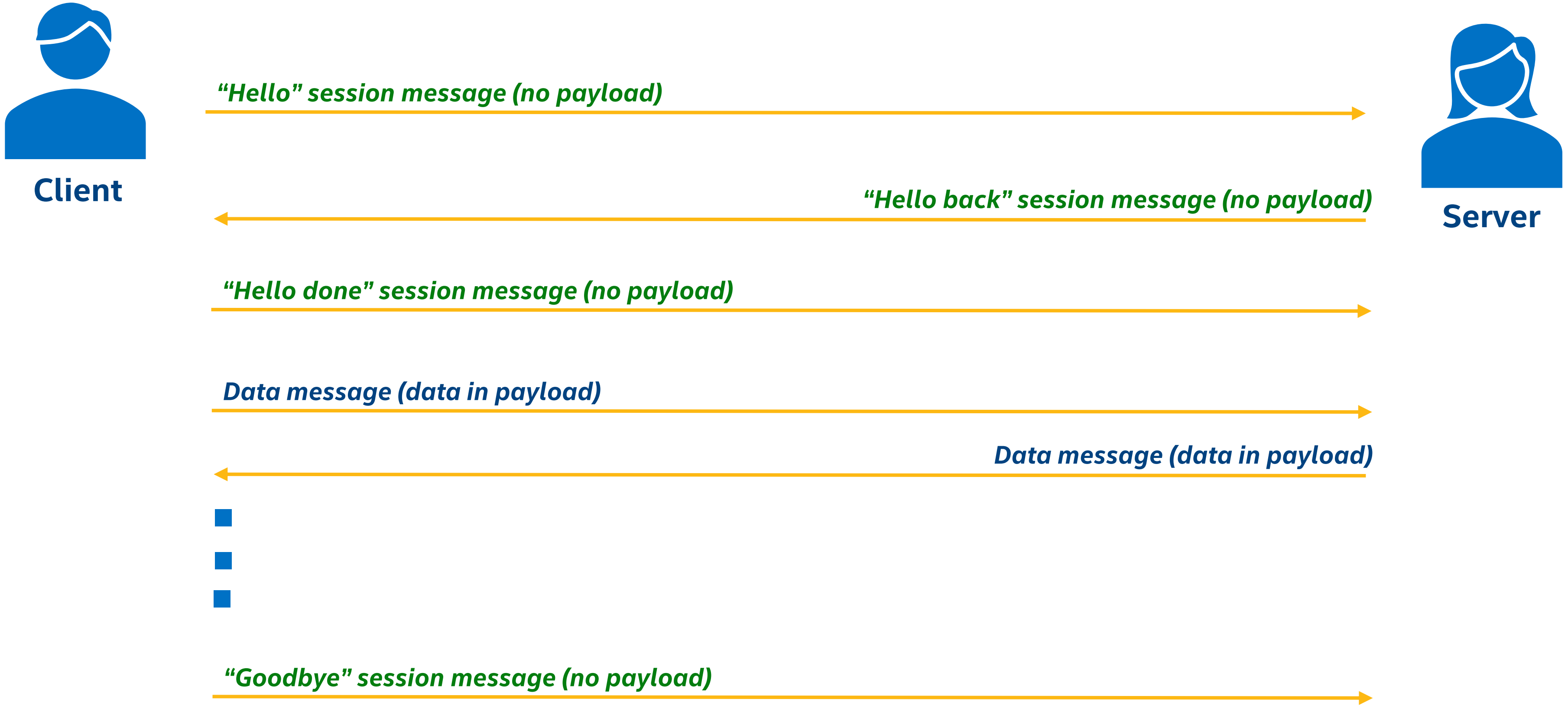
Server



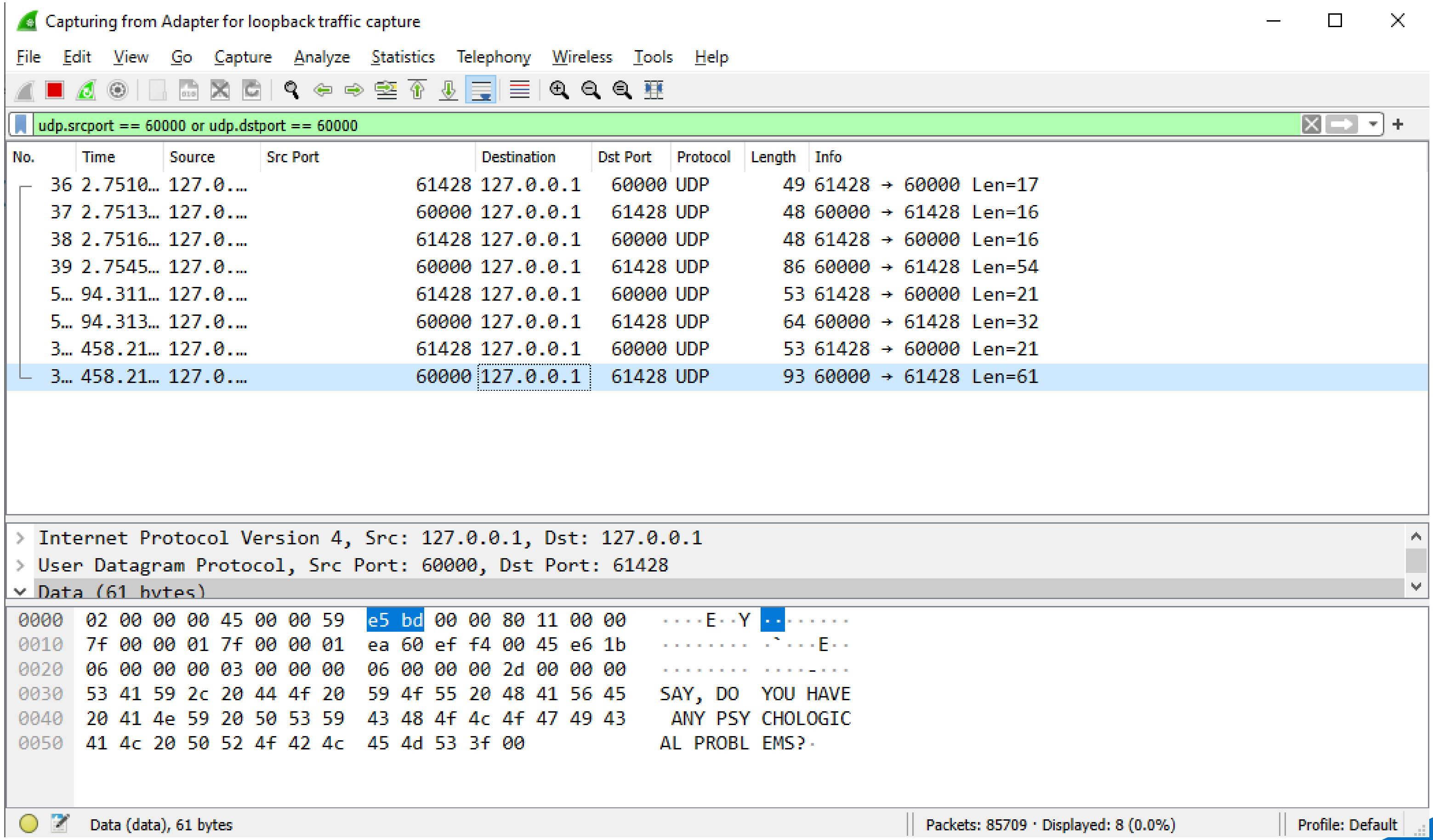
Command Prompt - udp_party -port 3000 -key alice.key -pwd alice -cert alice.crt -root rootCA.crt -peer Bob.com

```
C:\Users\alexber\OneDrive - Intel Corporation\Documents\GitHub\documentation.training.applied-crypto-part2\udp_party\x64
\Debug>udp_party -port 3000 -key alice.key -pwd alice -cert alice.crt -root rootCA.crt -peer Bob.com
Server: Starting listening...
New session 1 created with Bob.com
(1) Created
(1) Welcome: "HI!  I'M ELIZA.  WHAT'S YOUR PROBLEM?"
(1) Request: "I'm trying to write crypto code"
(1) Response: "DID YOU COME TO ME BECAUSE YOU ARE TRYING TO WRITE CRYPTO CODE?"
(1) Request: "yes"
(1) Response: "YOU SEEM QUITE POSITIVE."
(1) Request: "Writing crypto code is difficult"
(1) Response: "SAY, DO YOU HAVE ANY PSYCHOLOGICAL PROBLEMS?"
(1) Request: "anyone who writes crypto code has problems"
(1) Response: "WHY DO YOU ASK?"
(1) Request: "are you real?"
(1) Response: "WHY ARE YOU INTERESTED IN WHETHER OR NOT I AM REAL?"
(1) Request: "not really"
(1) Response: "WHAT DOES THAT SUGGEST TO YOU?"
(1) Request: "nothing"
(1) Response: "I SEE."
(1) Request: "bye"
(1) Closing.
```

Existing protocol details


Client
Server
"Hello" session message (no payload)
"Hello back" session message (no payload)
"Hello done" session message (no payload)
Data message (data in payload)
Data message (data in payload)
"Goodbye" session message (no payload)

Protocol capture



Capturing from Adapter for loopback traffic capture

udp.srcport == 60000 or udp.dstport == 60000

```
No.  Time     Source   Src Port  Destination  Dst Port  Protocol  Length  Info
36   2.7510…  127.0.…  61428     127.0.0.1    60000     UDP       49      61428 → 60000 Len=17
37   2.7513…  127.0.…  60000     127.0.0.1    61428     UDP       48      60000 → 61428 Len=16
38   2.7516…  127.0.…  61428     127.0.0.1    60000     UDP       48      61428 → 60000 Len=16
39   2.7545…  127.0.…  60000     127.0.0.1    61428     UDP       86      60000 → 61428 Len=54
5…   94.311…  127.0.…  61428     127.0.0.1    60000     UDP       53      61428 → 60000 Len=21
5…   94.313…  127.0.…  60000     127.0.0.1    61428     UDP       64      60000 → 61428 Len=32
3…   458.21…  127.0.…  61428     127.0.0.1    60000     UDP       53      61428 → 60000 Len=21
3…   458.21…  127.0.…  60000     127.0.0.1    61428     UDP       93      60000 → 61428 Len=61
```

```
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 60000, Dst Port: 61428
v Data (61 bytes)
```

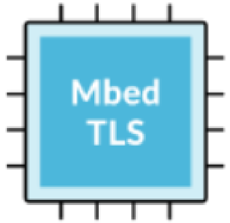
```
0000  02 00 00 00 45 00 00 59  e5 bd 00 00 80 11 00 00   ····E··Y ········
0010  7f 00 00 01 7f 00 00 01  ea 60 ef f4 00 45 e6 1b   ········ ·`···E··
0020  06 00 00 00 03 00 00 00  06 00 00 00 2d 00 00 00   ········ ····-···
0030  53 41 59 2c 20 44 4f 20  59 4f 55 20 48 41 56 45   SAY, DO  YOU HAVE
0040  20 41 4e 59 20 50 53 59  43 48 4f 4c 4f 47 49 43    ANY PSY CHOLOGIC
0050  41 4c 20 50 52 4f 42 4c  45 4d 53 3f 00            AL PROBL EMS?·
```

Data (data), 61 bytes | Packets: 85709 · Displayed: 8 (0.0%) | Profile: Default

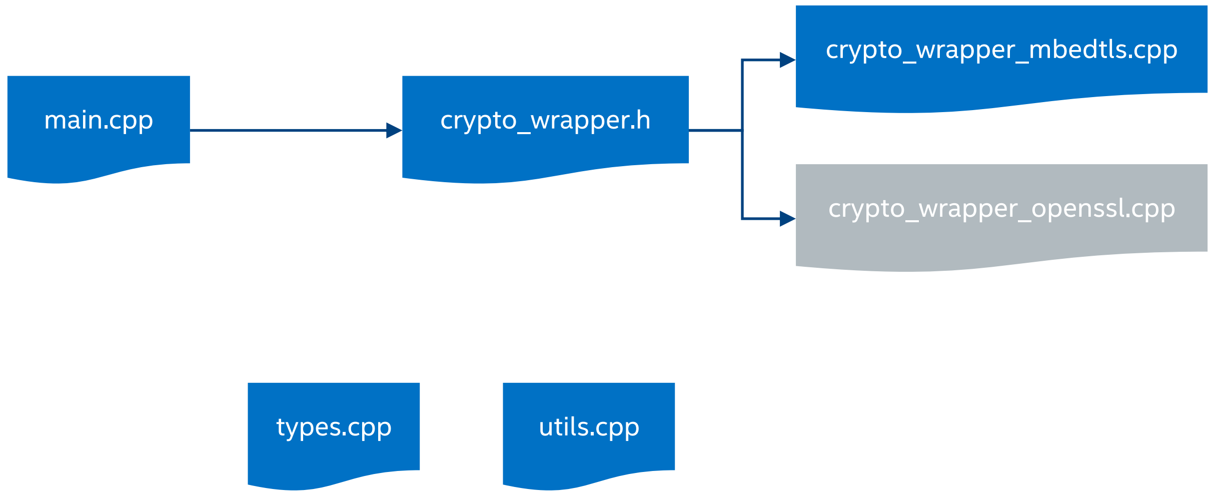
Options

OS

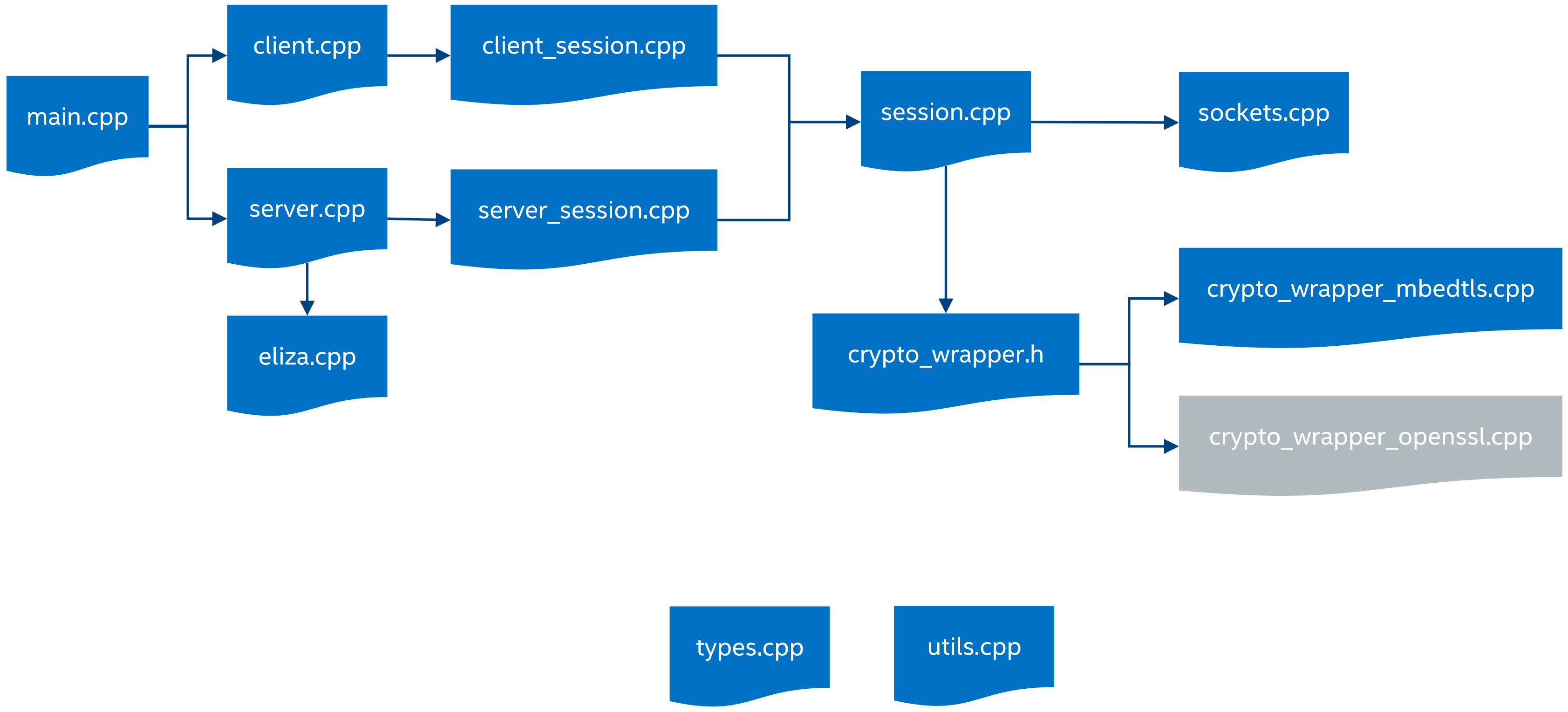
Crypto library



Crypto Unit-test Code Structure

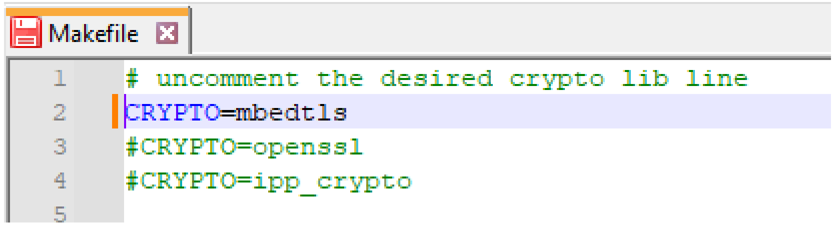


Code Structure



Building

- Linux
  - Modify the Makefile
  - Run `make` or `make clean`



Makefile

```
# uncomment the desired crypto lib line
CRYPTO=mbedtls
#CRYPTO=openssl
#CRYPTO=ipp_crypto
```

Running

- Make sure you have the required key files and certificate files
- For server

```
udp_party -port 60000 -key alice.key -pwd alice -cert alice.crt -
root rootCA.crt -peer Bob.com
```

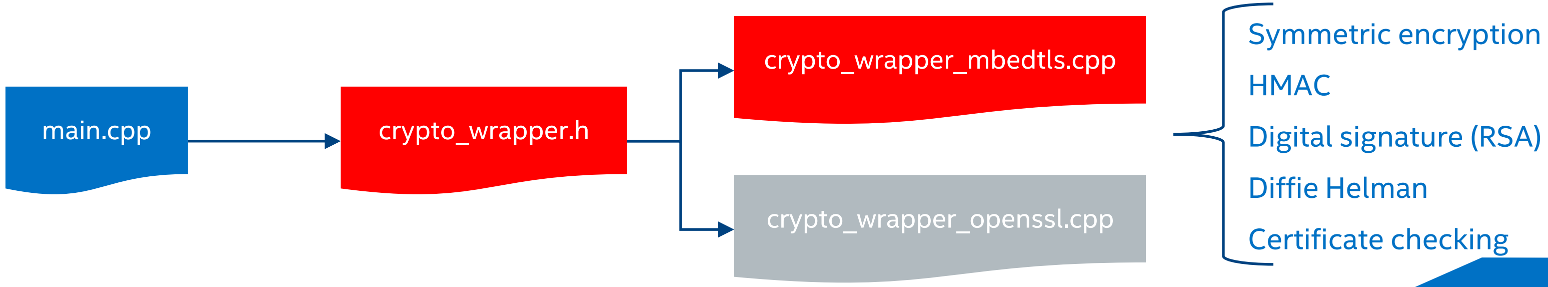
- For client

```
udp_party -ip 127.0.0.1 -port 60000 -key bob.key -pwd bobkey -cert
bob.crt -root rootCA.crt -peer Alice.com
```

Exercise – part I

Implement crypto wrapper and make the crypto unit test pass

1. Install the crypto library
2. Configure your environment to the chosen crypto lib (Makefile)
3. Implement the missing crypto functionality in the chosen wrapper
4. Run the crypto_test project to see all tests pass



Crypto libraries details

mbedTLS

- *sudo apt install libmbedtls-dev*
- Or https://tls.mbed.org/download, configuration – refer to https://github.com/ARMmbed/mbedtls, build - *make*
- udp_party config:
  - Add *CRYPTO=mbedtls* to Makefile

OpenSSL

- Download, config and build:
  - Linux – might be already installed.
  - If not - https://nextgentips.com/2022/03/23/how-to-install-openssl-3-on-ubuntu-20-04/

Usefull APIs

```
mbedtls_md_hmac
mbedtls_hkdf
mbedtls_gcm_setkey
mbedtls_gcm_crypt_and_tag
mbedtls_gcm_auth_decrypt
mbedtls_md
mbedtls_pk_get_type
mbedtls_pk_rsa
mbedtls_rsa_set_padding
mbedtls_rsa_rsassa_pss_sign
mbedtls_md_info_from_type
mbedtls_rsa_rsassa_pss_verify
mbedtls_dhm_set_group
mbedtls_dhm_make_public
mbedtls_dhm_read_public
mbedtls_dhm_calc_secret
mbedtls_x509_crt_verify
```

Usefull resources

https://github.com/Mbed-TLS/mbedtls (refer to Documentation section)

Relevant mbedTLS files

- hkdf.h
- gcm.h
- pk.h
- rsa.h
- entropy.h
- dhm.h
- bignum.h
- md.h
- x509.h
- x509_crt.h

https://cpp.hotexamples.com/

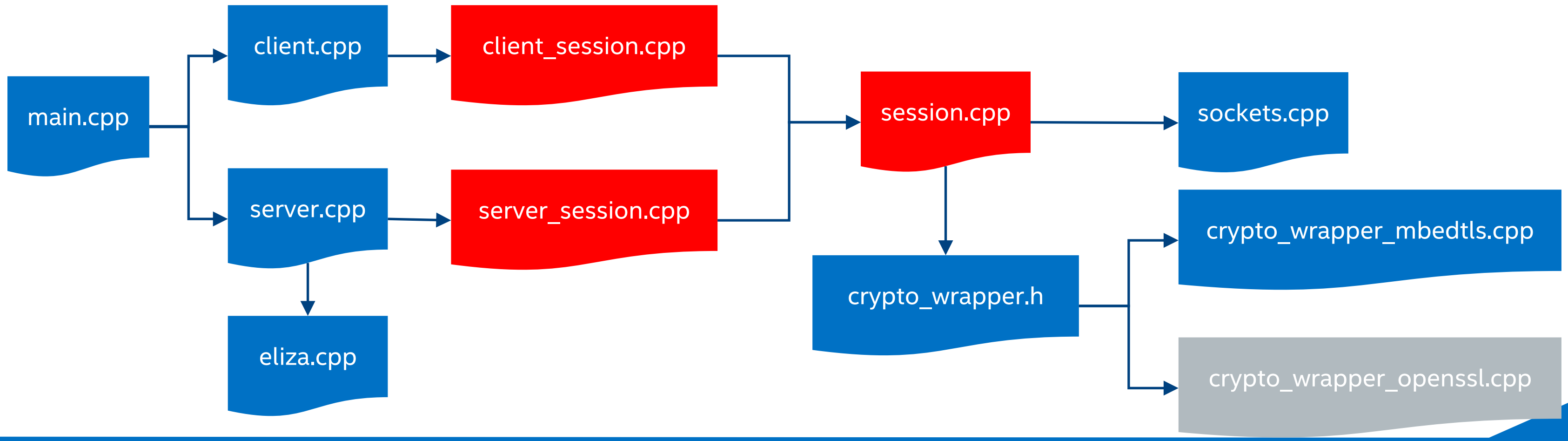
https://github.com/openenclave/openenclave-mbedtls/blob/openenclave-mbedtls-2.16/programs/README.md

Exercise – part II (advanced)

Add encryption and integrity protection over the traffic

Use SIGMA for key exchange

1. Enhance the session classes to use the crypto wrapper for SIGMA and channel protection
2. Make sure the session is working correctly and securely


main.cpp
client.cpp
server.cpp
eliza.cpp
client_session.cpp
server_session.cpp
session.cpp
sockets.cpp
crypto_wrapper.h
crypto_wrapper_mbedtls.cpp
crypto_wrapper_openssl.cpp