Microprocessor Systems Lab

Checkoff and Grade Sheet

Partner 1 Name:______________________________

Partner 2 Name:______________________________

Grade Component	Max.	Points Awarded		TA Init.s	Date
		Partner 1	Partner 2		
Performance Verification: Task 1	10 %				
Task 2	10 %				
Task 3	10 %				
Task 4 [Depth]	10 %				
TA Questioning	10 %				
Documentation and Appearance	50 %				
	Total:				

Report Grader's signature:________________________

Date:________________________

MPS Cryptography Lab

→ Laboratory Goals

By completing this laboratory assignment, you will learn to:

1. Ensure the confidentiality of data by encrypting and decryption it using the Advanced Encryption Standard (AES) in the Counter (CTR) mode of operation,
2. Ensure the confidentiality and authenticity of data by using AES in the Galois Counter (GCM) mode of operation to perform authenticated encryption and decryption,
3. Derive encryption keys from passwords or passphrases using the PBDKF2 key derivation function,
4. Communicate over unsecured channels by using Eliptic Curve Dillie-Hellman (ECDH) key exchange and HMAC-based key derivation (HKDF) to establish shared symmetric encryption keys.

→ Reading and References

R1. mbed TLS: Sections on AES, GCM, ECDH, PBDKF2, and HKDF as described in the notes for each task. Equivalent to reading header files in the project template.

R2. 32f769i_Discovery_Manual.pdf: Pinout diagrams

R3. RM0410-stm32f7_Reference_Manual.pdf: Chapter 22 (RNG)

R4. CryptographyLabTemplate.zip: Project Template for Cryptography Lab

R5. SecureCommunicationPeerDatasheet.pdf: Datasheet and communication protocol for Secure Communication Peer

R6. SecureCommunicationPeerFirmware_v1_0.pdf: Firmware for using Nucleo-F303RE as Secure Communication Peer

MPS Cryptography Lab

Confidentiality and Encryption

Modern digital systems rely heavily on cryptographic techniques to protect against malicious actors (commonly called attackers). The most commonly known technique is encryption, which protects the confidentiality of data by preventing all parties who do not hold the (secret) encryption key from reading the original data. Other techniques such as digital signatures and message authentication codes (MAC) allow recipients to verify the authenticity of the data—that the data was written/prepared by a particular sender. Digital signatures and MACs show that data was written by someone who holds a particular encryption or signing key, and that the data has not changed after they prepared it and generated the signature or MAC.

The Advanced Encryption Standard (AES) is a widely used block cipher which operates to encrypt or decrypt blocks of 16 bytes using an encryption/decryption key of 128, 192, or 256 bits. The simplest approach to encryption with a block cipher is to generate a random key, divide the data into blocks of 16 bytes, and encrypt each block with the same randomly generated key. This is referred to as the Electronic Codebook (ECB) mode, and appears to provide data confidentiality, but generally falls short. Each identical block of unencrypted data, which is also called plaintext, is identical in its encrypted form, as ciphertext. For many datasets (arguably almost all datasets), this doesn't provide adequate protection.

The Counter (CTR) mode of operation attempts to provide better confidentiality by utilizing an Initialization Vector (IV) to ensure that identical blocks of plaintext (within the same message or in separate messages encrypted with the same key) do not produce identical blocks of ciphertext. Counter mode operates by encrypting the IV, then computing the block ciphertext as the exclusive or (XOR) of the block plaintext and the encrypted IV. For each subsequent block, the IV is incremented by 1, then encrypted and XORed with the block plaintext[1].

Mbed TLS implements Counter mode (and many others) for AES and provides simple functions for use by applications to encrypt and decrypt data using AES-CTR. Note that the design of AES-CTR (namely, the property that (A XOR B) XOR B = A) means that encrypting plaintext produces ciphertext, and encrypting the ciphertext recontructs the original plaintext.

⋄ Task 1: Encryption and Decryption with AES-CTR

Generate a random 256-bit key using the STM32F769's True Random Number Generator (RNG). Read a message via USART1 and encrypt the message using AES-CTR. Decrypt the message and show the original plaintext, the ciphertext, and the recovered plaintext. The program must accept messages of variable length (including lengths not evenly divisible by 16), but may impose a reasonable upper bound such as 100 characters. The program should loop to accept additional messages without requiring a system reset.

NOTES:

1. Reading parts of `Middlewares\Third_Party\mbedTLS\include\mbedtls\aes.h` may be helpful. Start by looking at mbedtls_aes_crypt_ctr.
2. The `stream_block` parameter is used only when encryption must be resumed in the future after encrypting some number of bytes which is not divisible by 16. It must be provided to the function, but then can be safely discarded/set to 0.

[1] Wikipedia has some helpful illustrations.

3. Your code should clearly overwrite/discard the original plaintext before decrypting the ciphertext. The memset() function in one way to do this if you are using separate buffers for the plaintext and ciphertext.

4. In this lab, it is acceptable (and sometimes helpful) to display data such as encryption keys and passwords which would generally be considered confidential. Don't use passwords that you hope to keep secret.

Hashing and Authentication

Cryptographic hash functions accept an input of arbitrary length and convert it to an output of fixed length. These functions are designed to have certain key properties, including:

1. One-way: It is not feasible to find the input if the output is known.

2. Collision-resistant: It is not feasible to find multiple inputs which produce the same output.

Consider that any hash function which processes messages (inputs) with length greater than the output length must have at least one collision (two or more inputs which produce the same output). This emphasizes the role that probability plays in cryptography. Almost all claims of confidentiality and authentication are based on keys which are chosen at random from a very large set of possible keys (2^{256} possible keys in task 1) and computing problems which are very hard to solve efficiently. The claims are not that something cannot be done (decrypted, counterfeited, etc.), rather that it is **extremely** unlikely that someone will successfully do it with the technology which is available now or is expected to be available in the short to medium term. This is a big part of the reason that encryption standards must be updated as technology improves: ciphers which were highly secure at the time of their creation now offer negligible protection because of increases in available computing power and the development of new techniques to solve these computing problems efficiently.

Hash functions are frequently used to derive encryption keys from passwords. While a single "application" of a cryptographic hash function is sufficient to produce a fixed length output which could be used as an encryption key, there are better alternatives. Key Derivation Functions are hash functions designed to derive an encryption key from a password, typically in a manner that protects the user by making it harder to guess their password. Functions such as PBKDF2 require that the password be hashed repeatedly so that an attacker cannot guess passwords as quickly. Other functions, such as Argon2, can also be configured to require large amounts of memory which improves their resistance to brute force attacks using GPUs. The central assumption here is that it is far easier to guess a user-chosen password than a 128-256 bit encryption key, so making each password guess slower and more costly improves resistance to brute-force password guessing attacks.

Cryptographic salts are usually chosen at random, and are appended (or prepended, etc.) to passwords or other inputs to key derivation functions. Salts can be chosen such that they are not re-used for multiple users or multiple passwords. This ensures that users will not have identical encryption keys, even if they chose identical passwords. This also provides some protection against a possible rainbow table attack, where an attacker constructs a database of common passwords and hashes them in advance. Even if the attacker manages to include a user's password in the database, the randomly chosen salt ensures that the password corresponds to a unique encryption key.

The Galois-Counter mode (GCM) is a form of authenticated encryption which combines Counter mode with an authentication tag. The tag is generated when the message is encrypted and can be used by a recipient to verify that the party who wrote the message possesses the same encryption key as the recipient (authentication). This also implies that the integrity of the data is preserved, because any changes in the ciphertext result in a change in the expected authentication tag.

◇ Task 2: Password-Based Key Derivation and AES-GCM

Read a password via USART1 and generate a random 128-bit salt. Use the salt and password to derive a 256-bit key using PBKDF2 with an appropriate hash function such as SHA-256. Read a message via USART1 and encrypt it using AES-GCM with an appropriate authentication tag length (16 bytes is fine). Display the ciphertext and authentication tag. Discard the plaintext, password, and encryption key.

Read the password via USART1 again and re-derive the encryption key. Attempt to decrypt the ciphertext. If the decryption and authentication succeeds, display the recovered plaintext. Verify that the authentication fails for incorrect passwords.

NOTES:

1. Reading parts of `pkcs5.h`, `md.h`, and `gcm.h` may be helpful.

Asymmetric Cryptography

Symmetric ciphers, such as AES are limited by the fact that the party encrypting a message must possess the same key as the party decrypting the message. If the parties have access to a secure channel, such as a private conversation away from eavesdroppers and monitoring devices, it's easy to agree upon a secret key which can be used to encrypt and decrypt future messages. In many cases, such as communication between a server and any one of the billions of internet capable devices, no such channel exists.

Asymmetric cryptography solves the key exchange/agreement problem through use of multiple keys per party (generally one public and one private) to support the establishment of a shared secret using an unsecured channel.

Some implmentations, such as RSA [2], allow anyone to use a public key to encrypt a message which can only be decrypted with the corresponding private key. This message is limited by the size of the public key used.

Diffie-Hellman key exchange (and Elliptic Curve Diffie-Hellman key exchange) also relies on the use of two keys, one private and one public. Rather than encrypting a message with the recipient's public key, in DH or ECDH the sender combines their own private key with the recipient's public key to produce a shared secret, which is then used to derive a symmetric encryption key. The recipient can reproduce this symmetric key using their private key and the sender's public key, but no one else can derive the symmetric key unless they have a copy of the sender or recipient's private key.

For DH and ECDH it's clear that the system still relies on symmetric cryptography—DH and ECDH serve only to establish a shared symmetric key. In practice, RSA is also used to establish a shared symmetric key rather than encrypting entire messages because RSA is far more computationally expensive than a well-designed symmetric cipher.

[2] See RFC 3447

Mbed TLS provides functions to generate an ECDH key pair (public and private key), to export the public key in a usable format, to import another party's public key, and to derive the corresponding shared secret (often called the pre-master secret). The resulting shared secret is not necessarily uniformly random, so HKDF (HMAC-based Key Derivation Function) is used with a salt to derive a suitable shared symmetric encryption key.

◇ Task 3: Public Key Exchange

Generate an ECDH keypair using Curve 25519 and transmit your public key to the Secure Communication Peer via USART6. The required format is described in R5.

The Secure Communication Peer will respond by transmitting its own public key.

Derive the shared pre-master secret and compute the shared encryption key using HKDF with the SHA-256 hash function. The salt used with HKDF is the XOR of the salts provided in the public key messages.

Choose a number (random is fine) and use the Secure Communication Peer to determine whether the number is even or odd. Try numbers of different lengths.

NOTES:

1. Reading parts of `ecdh.h` may be helpful.
2. The public key, shared pre-master secret, and shared encryption key should all be 32 bytes long. Depending on the mbedTLS function used, the public key may be written to the buffer you provide as a length byte (0x20) followed by the 32 byte public key.
3. The Secure Communication Peer does not pause between transmitting the length bytes and transmitting the rest of the message, so you may need to use an interrupt-based UART implementation to ensure you don't miss any incoming data.

◇ Task 4: [Depth] Multi-Party Key Exchange and Authenticated Encryption

Transmit your ECDH public key (again using Curve 25519) to the Secure Communication Peer. Specify a number of recipients greater than 1. Your implementation should support values between 1 and 10.

Multiple peers will respond in turn by transmitting their own public keys. The number of peers that respond is determined by the number of recipients you specified, which cannot exceed 10. Derive shared encryption keys for each peer.

Transmit data to be stored by multiple peers using the format described in R5. This data can be random or can be provided via user input. Test your implementation with multiple data lengths.

Request that the peers return a copy of your previously stored data. At least one peer will respond. Verify that the returned data matches the original data. If the original data is not successfully recovered, try again.

NOTES:

MPS Cryptography Lab

1. You might not have time to derive the shared encryption key for one peer before the next peer begins transmitting its public key.

2. In a distributed network, it's unwise to assume that the other nodes are trustworthy. You're encouraged to encrypt and tag your data with a key known only to your node such that the peer nodes only receive ciphertext and supporting metadata (nonce, auth tag). This also allows you to delete your data after transmission without sacrificing the ability to verify the integrity of the returned data.