Chapter 3: Rust's Cryptographic Strengths and Vulnerabilities

Introduction

In this chapter, our exploration into Rust's cryptographic landscape reveals a canvas of libraries and tools designed to strengthen digital communications. At the forefront stands the `rustls` library, a robust Rust implementation of TLS, which serves as the cornerstone for securing SSL/TLS communications. Configuring a secure server involves creating an instance of `ServerConfig` and tailoring it to specific needs, such as cipher suite preferences and key management. Rust's type safety and expressive syntax contribute to a more resilient implementation, mitigating common pitfalls associated with memory safety.

Beyond the SSL/TLS domain, Rust empowers us to tackle the complexities of mutual authentication using the `openssl` library. This involves crafting a secure handshake process where both client and server authenticate each other's identity through X.509 certificates. Configuring an SSL acceptor involves loading server certificates and private keys while establishing trust relationships through CA certificates. Rust's emphasis on ownership and lifetimes ensures that cryptographic keys and sensitive data are managed securely throughout the authentication process, reducing the risk of memory leaks and unauthorized access.

As we navigate Rust's cryptographic terrain, symmetric-key cryptography emerges as a pivotal aspect, facilitated by libraries like `crypto`. Our exploration takes us into the domain of the Advanced Encryption Standard (AES), where data encryption and decryption become intricate dances with key sizes and block modes. Rust's focus on zero-cost abstractions and performance ensures that cryptographic operations are executed efficiently, catering to the demands of secure data transmission. Leveraging Rust's concurrency model, we can parallelize cryptographic tasks, enhancing throughput without compromising security.

1. Cryptography in Rust

Before kicking off our exploration of cryptographic operations in Rust, it's essential to immerse ourselves in the fundamental concepts that make this complex field. We'll traverse these concepts meticulously to ensure a robust comprehension of the cryptographic landscape.

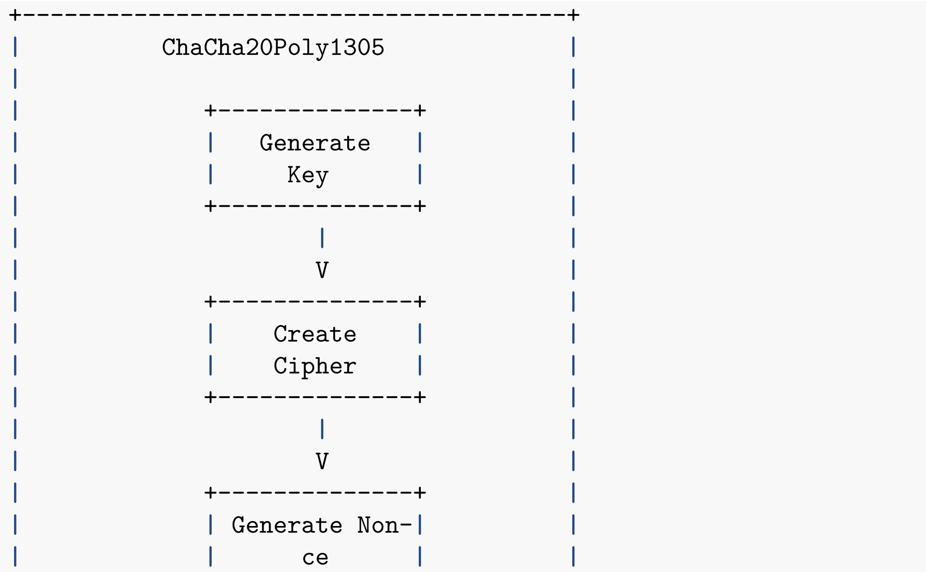
1.1 Encryption and Decryption

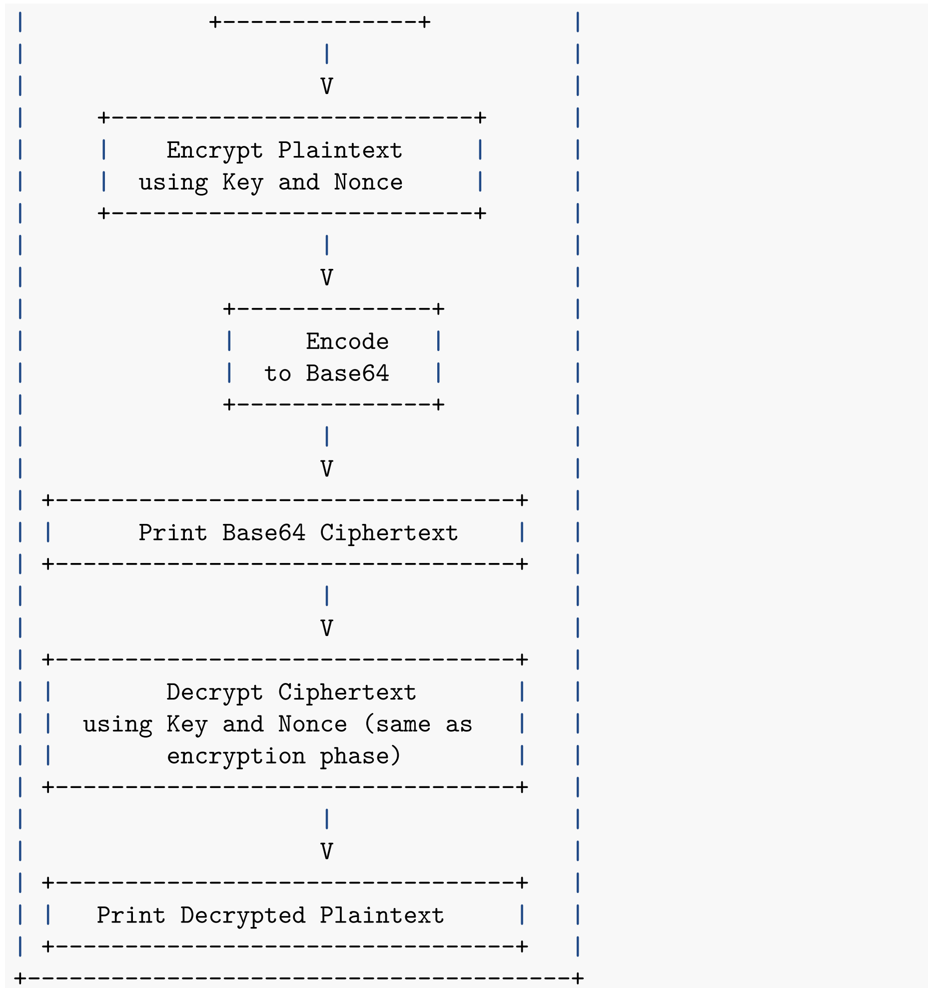
Encryption, a part of cryptography, surpasses mere confidentiality maintenance. It represents a dual-purpose functionality, allowing data scrambling and subsequent unscrambling. At its core, encryption involves a cryptographic function, that takes both data and a key as inputs, yielding either ciphertext or the original

cleartext. Symmetric algorithms work with a single key for both encryption and decryption, while asymmetric counterparts work with different keys.

At the heart of our cryptographic exploration lies ChaCha20Poly1305 using the `chacha20poly1305` crate, a cipher notable for its simplicity and speed. The underlying ChaCha20 stream cipher, employing a blend of add, rotate, and XOR instructions (ARX), works seamlessly with the straightforward Poly1305 hash function. While not universally endorsed by standards bodies like NIST, ChaCha20Poly1305 is widely adopted, being mandatory in the Transport Layer Security (TLS) protocol.

```rust
use chacha20poly1305::{aead::{Aead, KeyInit, OsRng}, ChaCha20Poly1305, Nonce, AeadCore};
use base64::{Engine as _, engine::{self, general_purpose}, alphabet};
{
    let key = ChaCha20Poly1305::generate_key(&mut OsRng);
    let cipher = ChaCha20Poly1305::new(&key);
    let nonce = ChaCha20Poly1305::generate_nonce(&mut OsRng);
    let plaintext = "Hello, Rust Cryptography!";

    // Encryption
    let ciphertext = cipher.encrypt(&nonce, plaintext.as_bytes())?;
    let b64_ciphertext = general_purpose::STANDARD.encode(&ciphertext);
    println!("Base64 Cipher Text: {}", b64_ciphertext);

    // Decryption
    let decrypted_text = cipher.decrypt(&nonce, &*ciphertext)?;
    println!("Decrypted Text: {}", String::from_utf8_lossy(&decrypted_text));
}
```





Encryption works like a protector for information moving from one place to another and a keeper of important data stored away. It waits to be unlocked for later use or carefully watched for any signs of deceitful actions.

```
:dep chacha20poly1305 = {version="0.10.1"}
```

```
:dep base64 = {version="0.21.5"}
```

```
use chacha20poly1305::{aead::{Aead, KeyInit, OsRng}, ChaCha20Poly1305, Nonce, AeadCore};
use base64::{Engine as _, engine::{self, general_purpose}, alphabet};
{
    let key = ChaCha20Poly1305::generate_key(&mut OsRng);
    let cipher = ChaCha20Poly1305::new(&key);
    let nonce = ChaCha20Poly1305::generate_nonce(&mut OsRng);
```

```
    let plaintext = "Hello, Rust Cryptography!";

    // Encryption
    let ciphertext = cipher.encrypt(&nonce, plaintext.as_bytes())?;
    let b64_ciphertext = general_purpose::STANDARD.encode(&ciphertext);
    println!("Base64 Cipher Text: {}", b64_ciphertext);

    // Decryption
    let decrypted_text = cipher.decrypt(&nonce, &*ciphertext)?;
    println!("Decrypted Text: {}", String::from_utf8_lossy(&decrypted_text));
}
```

```
Base64 Cipher Text: lOE/Ps3i63+RU8iZcoGYABZOpUqDHqMoVGpF4jFk7w97FZPOXm7Ka4Y=
Decrypted Text: Hello, Rust Cryptography!
```

```
()
```

Note that the coded message is printed as a string encoded in base64, and the decoded message is displayed as a standard UTF-8 string.

1.2 Hashing

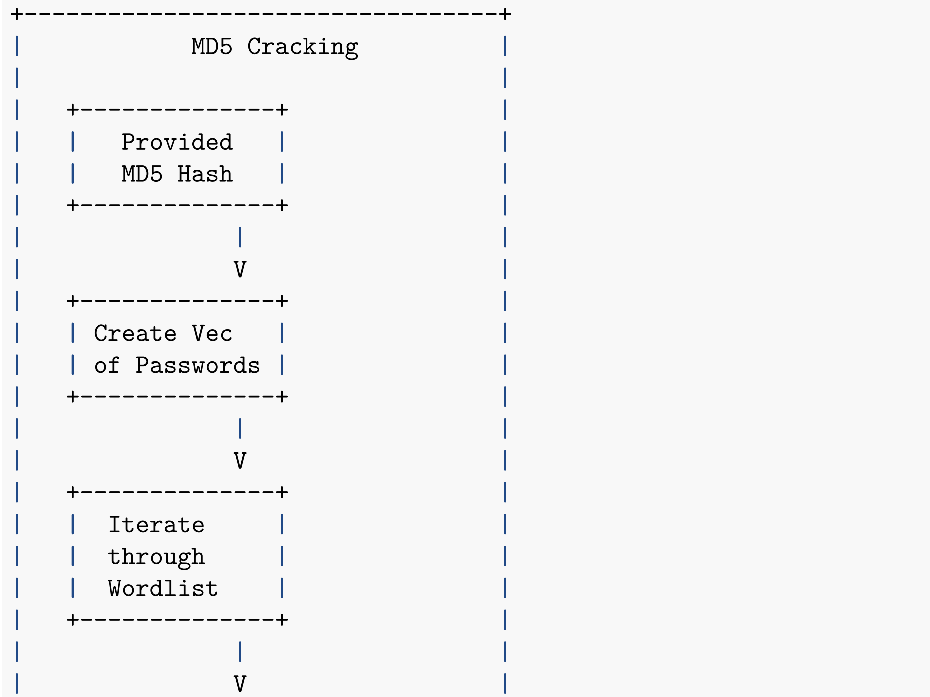
Hashing stands as a fundamental cryptographic process, operating as a unidirectional function meticulously crafted to produce a fixed-length and inherently unique output, all delegation upon a variable-length input. Its complex elegance lies in the irreversible nature of the transformation, blocking any possibility of figuring out the original input from the resultant hash value. This cryptographic technique finds its power in various scenarios, particularly those where the preservation of the original cleartext source becomes unnecessary for subsequent processing or to guarantee the integrity of data. An exemplary manifestation of secure practices within the domain of information protection is the storage of hashed passwords, a practice ideally complemented by the introduction of salt - an additional layer of randomized data. This cryptographic seasoning enhances the unpredictability of the hash values, strengthening security and aligning with best practices in safeguarding sensitive information.

In our exploration of hashing within the Rust programming language, we delve into two illustrative examples. The first demonstrates the cracking of MD5 hashes through an offline dictionary attack using the `md-5` crate, employing Rust's capabilities to generate and compare hash values systematically. The second example showcases the implementation of bcrypt using the `bcrypt` crate, an advanced algorithm enhancing the security of sensitive data such as passwords. Rust's versatility shines as it seamlessly integrates these cryptographic techniques,

emphasizing its adaptability in addressing diverse security challenges.

1.2.1 Cracking MD5 Hashes Let's examine the code for cracking MD5 hashes in Rust:

```
use md5::{Md5, Digest};
use hex_literal::hex;

let md5_hash = hex!("517e90f2a52e730701a5d7ec89ef0f40");

let wordlist = vec!["p@ssword231", "12345656789", "password", "Password", "Mahmoud123"];

for password in wordlist {
    let mut md5_hasher = Md5::new();
    md5_hasher.update(password);
    let md5_generated = md5_hasher.finalize();
    let md5_hex_string: String = md5_generated.iter().map(|byte| format!("{:02x}", byte)).c

    println!("[INFO] Trying hash {:?}", md5_hex_string);
    if md5_generated[..] == md5_hash {
        println!("[INFO] Password found (MD5): {}", password);
    }
}
```



```
+----------------------------------+
|            MD5 Cracking          |
|                                  |
|   +--------------+               |
|   |   Provided   |               |
|   |   MD5 Hash   |               |
|   +--------------+               |
|               |                  |
|               V                  |
|   +--------------+               |
|   | Create Vec   |               |
|   | of Passwords |               |
|   +--------------+               |
|               |                  |
|               V                  |
|   +--------------+               |
|   |  Iterate     |               |
|   |  through     |               |
|   |  Wordlist    |               |
|   +--------------+               |
|               |                  |
|               V                  |
```

```
|   +--------------+                |
|   |   MD5 Hash   |                |
|   |  Calculation |                |
|   +--------------+                |
|               |                   |
|               V                   |
|   +--------------+                |
|   | Convert to   |                |
|   | Hex String   |                |
|   +--------------+                |
|               |                   |
|               V                   |
|   +--------------+                |
|   |   Print MD5  |                |
|   |  Hash (Info) |                |
|   +--------------+                |
|               |                   |
|               V                   |
|   +--------------+                |
|   | Compare with |                |
|   | Provided Hash|                |
|   +--------------+                |
|               |                   |
|               V                   |
|   +--------------+                |
|   |   Print      |                |
|   | Password if  |                |
|   | Matched      |                |
|   +--------------+                |
+-----------------------------------+
```

In this Rust example, we start by defining variable `md5_hash` that hold the target hash value. This hash were obtained post-exploitation, and the goal is to discover the cleartext passwords that produced it. The code reads from a vector of strings, generating MD5 hashes for each password and comparing them with the target hash.

```
:dep md-5 = {version="0.10.6"}
```

```
:dep hex-literal = {version="0.4.1"}
```

```
use md5::{Md5, Digest};
use hex_literal::hex;

let md5_hash = hex!("517e90f2a52e730701a5d7ec89ef0f40");
```

```rust
let wordlist = vec!["p@ssword231", "12345656789", "password", "Password", "Mahmoud123"];

for password in wordlist {
    let mut md5_hasher = Md5::new();
    md5_hasher.update(password);
    let md5_generated = md5_hasher.finalize();
    let md5_hex_string: String = md5_generated.iter().map(|byte| format!("{:02x}", byte)).c

    println!("[INFO] Trying hash {:?}", md5_hex_string);
    if md5_generated[..] == md5_hash {
        println!("[INFO] Password found (MD5): {}", password);
    }
}
```

```
[INFO] Trying hash "ab2d83f72c018c04bda54551f7963e30"
[INFO] Trying hash "576333879d388b3537287481aa07f7c6"
[INFO] Trying hash "5f4dcc3b5aa765d61d8327deb882cf99"
[INFO] Trying hash "dc647eb65e6711e155375218212b3964"
[INFO] Trying hash "517e90f2a52e730701a5d7ec89ef0f40"
[INFO] Password found (MD5): Mahmoud123
```

```
()
```

1.2.2 Implementing bcrypt Now, let's explore how to use bcrypt to encrypt and authenticate passwords in Rust:

```rust
use bcrypt::{DEFAULT_COST, hash, verify};

let stored_hash = "$2b$12$gPxgyRNV5G/DTaADM4rnuu3LcEbQeVdqhdNaKobgmdiNeyNRmV2me";

let password = "p@ssword123";

// Hash the password using bcrypt
let hashed = hash(password, DEFAULT_COST)?;

println!("Generated Hash: {}", hashed);

// Verify the hashed password against the stored hash
if verify(password, &stored_hash)? {
    println!("[INFO] Authentication successful");
} else {
    println!("[INFO] Authentication failed");
```

```
}
```

```
+-------------------------------------+
|               bcrypt                |
|                                     |
|     +------------------------+      |
|     |     Hash Password      |      |
|     |                        |      |
|     |  +------------------+  |      |
|     |  | Cost Generation|   |      |
|     |  +------------------+  |      |
|     |           |            |      |
|     |           V            |      |
|     |  +------------------+  |      |
|     |  |   Hash Function |  |      |
|     |  | (bcrypt algorithm)||      |
|     |  +------------------+  |      |
|     |           |            |      |
|     |           V            |      |
|     |  +------------------+  |      |
|     |  |    Salt + Cost  |  |      |
|     |  +------------------+  |      |
|     |           |            |      |
|     |           V            |      |
|     |  +------------------+  |      |
|     |  |  Generate Hash  |  |      |
|     |  +------------------+  |      |
|     |           |            |      |
|     |           V            |      |
| +--------------------------+|      |
| |      Print Generated     ||      |
| |        Hash Value        ||      |
| +--------------------------+|      |
|           |                 |      |
|           V                 |      |
| +--------------------------+|      |
| |  Verify Hashed Password  ||      |
| +--------------------------+|      |
|           |                 |      |
|           V                 |      |
| +--------------------------+|      |
| | Print Authentication Info||      |
| +--------------------------+|      |
+-------------------------------------+
```

In this Rust example, we utilize the `bcrypt` crate to implement bcrypt hashing

and authentication. The `hash` function generates a bcrypt hash from a cleartext password, and `verify` is used to compare the generated hash with a stored hash to authenticate the password. The cost factor ensures the algorithm's resource-intensive nature, enhancing security against brute-force attacks.

```
:dep bcrypt = {version="0.15.0"}
```

```
use bcrypt::{DEFAULT_COST, hash, verify};

let stored_hash = "$2b$12$gPxgyRNV5G/DTaADM4rnuu3LcEbQeVdqhdNaKobgmdiNeyNRmV2me";

let password = "p@ssword123";

// Hash the password using bcrypt
let hashed = hash(password, DEFAULT_COST)?;

println!("Generated Hash: {}", hashed);

// Verify the hashed password against the stored hash
if verify(password, &stored_hash)? {
    println!("[INFO] Authentication successful");
} else {
    println!("[INFO] Authentication failed");
}
```

```
Generated Hash: $2b$12$TZj8KWVRZ2YarPsY8LdhA.hX3ezZIVjrgWA8y91EDypHVmTq/x.ES
[INFO] Authentication successful
```

```
()
```

These Rust examples provide insights into the practical application of hashing and bcrypt in real-world scenarios. As we delve into Rust's cryptographic features, a deeper understanding of these concepts will empower your cryptographic journey.

1.3 Message Authentication

When we exchange messages, we want to be certain that the data hasn't been altered during transmission by someone unauthorized. Additionally, we need to confirm that the message is genuinely from an authorized sender and not a forgery by another entity. To tackle these concerns, we can utilize the `ring` crate, which provides robust cryptographic functionalities. Specifically, we will employ the HMAC (Hash-based Message Authentication Code) algorithm, a widely accepted standard for ensuring message integrity and source authenticity.

The HMAC algorithm involves a hashing function and a shared secret key known only to authorized parties. Attempting to forge a valid HMAC without possessing this shared secret becomes highly improbable.

Implementing HMAC in Rust is straightforward with the `ring` crate. Let's explore an example that demonstrates how to achieve message authentication securely.

```
use ring::hmac;
use ring::rand::{SecureRandom, SystemRandom};
use ring::error::Unspecified;
use ring::constant_time;
use hex;

const KEY_SIZE: usize = ring::digest::SHA256_OUTPUT_LEN;
const MESSAGE: &str = "Attach at 12:30";

fn generate_key(rng: &SystemRandom) -> Result<hmac::Key, Unspecified> {
    let mut key_value = [0u8; KEY_SIZE];
    rng.fill(&mut key_value)?;
    Ok(hmac::Key::new(hmac::HMAC_SHA256, &key_value))
}

fn generate_hmac(key: &hmac::Key, message: &[u8]) -> hmac::Tag {
    hmac::sign(key, message)
}

fn verify_hmac(key: &hmac::Key, message: &[u8], received_tag: &[u8]) -> Result<(), Unspecif
    let calculated_tag = generate_hmac(key, message);
    constant_time::verify_slices_are_equal(calculated_tag.as_ref(), received_tag)
}

let rng = SystemRandom::new();

// Sender side
let key = generate_key(&rng).expect("Failed to generate key");
let tag = generate_hmac(&key, MESSAGE.as_bytes());

// Simulate transmission (In a real implementation, this would be sent over the network)

// Receiver side
let received_tag_hex = "69d2c7b6fbbfcaeb72a3172f4662601d1f16acfb46339639ac8c10c8da64631d";
let received_tag = hex::decode(received_tag_hex).expect("Failed to decode received tag");

match verify_hmac(&key, MESSAGE.as_bytes(), &received_tag) {
    Ok(()) => println!("[INFO] Message is authentic"),
    Err(_) => println!("[INFO] Message may be tampered"),
```

```
}
```

```
+-----------------------+
|        Sender         |
+-----------------------+
         |
         V
+-----------------------+
|    Generate Key       |
|   using SystemRandom  |
+-----------------------+
         |
         V
+-----------------------+
|   Generate HMAC       |
|   using Key and       |
|   Message             |
+-----------------------+
         |
         V
+-----------------------+
|    Transmit Message   |
|   and HMAC (simulate  |
|   network transfer)   |
+-----------------------+
         |
         V
+-----------------------+
|       Receiver        |
+-----------------------+
         |
         V
+-----------------------+
|   Decode Received     |
|   HMAC from Hex       |
+-----------------------+
         |
         V
+-----------------------+
|   Verify HMAC         |
|   using Key and       |
|   Received HMAC       |
+-----------------------+
         |
         V
```

```
+----------------------+
|   Print Result       |
|  (Authentic/Tampered)|
+----------------------+
```

In this Rust example, the `key` variable represents the shared secret key. In a real-world scenario, this key would be securely managed and shared between authorized endpoints.

The `verify_hmac` function takes a key, a message, and the received HMAC as parameters. It calculates the HMAC using the `ring` crate and compares it in constant time to mitigate timing attacks. The subsequent statements simulate the reception of a message, decoding the received HMAC from a hex string.

By employing Rust's `ring` crate, we ensure a secure and efficient implementation of HMAC for message authentication, addressing concerns related to data tampering and source legitimacy. This example simplifies the process for clarity, focusing solely on HMAC functionality without incorporating network communication aspects.

```
:dep ring = {version = "0.17.7"}
```

```
:dep hex = {version = "0.4.3"}
```

```
use ring::hmac;
use ring::rand::{SecureRandom, SystemRandom};
use ring::error::Unspecified;
use ring::constant_time;
use hex;

const KEY_SIZE: usize = ring::digest::SHA256_OUTPUT_LEN;
const MESSAGE: &str = "Attach at 12:30";

fn generate_key(rng: &SystemRandom) -> Result<hmac::Key, Unspecified> {
    let mut key_value = [0u8; KEY_SIZE];
    rng.fill(&mut key_value)?;
    Ok(hmac::Key::new(hmac::HMAC_SHA256, &key_value))
}

fn generate_hmac(key: &hmac::Key, message: &[u8]) -> hmac::Tag {
    hmac::sign(key, message)
}

fn verify_hmac(key: &hmac::Key, message: &[u8], received_tag: &[u8]) -> Result<(), Unspecif
    let calculated_tag = generate_hmac(key, message);
    constant_time::verify_slices_are_equal(calculated_tag.as_ref(), received_tag)
}
```

```
let rng = SystemRandom::new();

// Sender side
let key = generate_key(&rng).expect("Failed to generate key");
let tag = generate_hmac(&key, MESSAGE.as_bytes());

// Simulate transmission (In a real implementation, this would be sent over the network)

// Receiver side
let received_tag_hex = "69d2c7b6fbbfcaeb72a3172f4662601d1f16acfb46339639ac8c10c8da64631d";
let received_tag = hex::decode(received_tag_hex).expect("Failed to decode received tag");

match verify_hmac(&key, MESSAGE.as_bytes(), &received_tag) {
    Ok(()) => println!("[INFO] Message is authentic"),
    Err(_) => println!("[INFO] Message may be tampered"),
}
```

```
[INFO] Message may be tampered
```

```
()
```

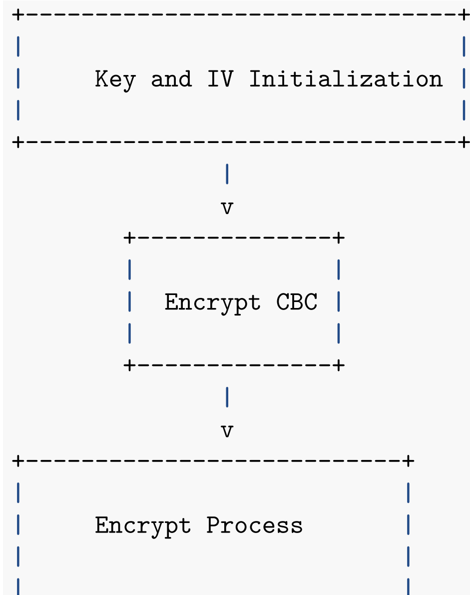
1.4 Symmetric Encryption

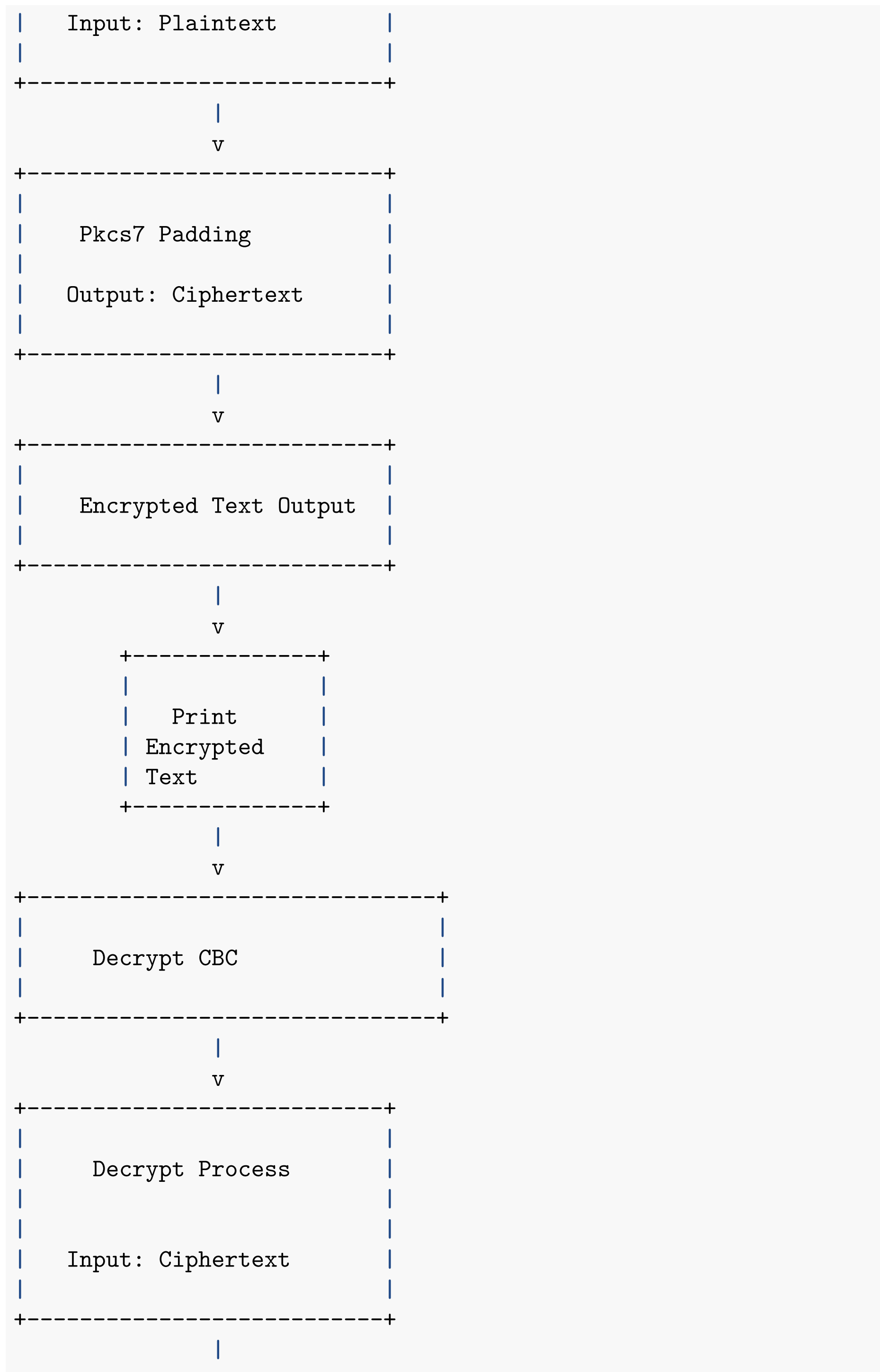
In the realm of Rust development, we embark on a journey into the world of encryption, focusing our attention on the foundational concept of symmetric-key encryption. This cryptographic approach employs a single secret key for both the encryption and decryption processes. Rust, with its robust ecosystem, facilitates the implementation of symmetric cryptography by supporting a variety of common algorithms within its default or extended packages.

Let's delve into a practical example within the Rust paradigm. Picture a scenario where a breach has occurred in an organization, granting access to an e-commerce web server and its backend database housing encrypted financial transactions. The encryption algorithm in play is the Advanced Encryption Standard (AES), specifically operating in Cipher Block Chaining (CBC) mode. The following Rust code snippet illustrates two functions responsible for encrypting and decrypting credit card information encrypted using AES in CBC mode.

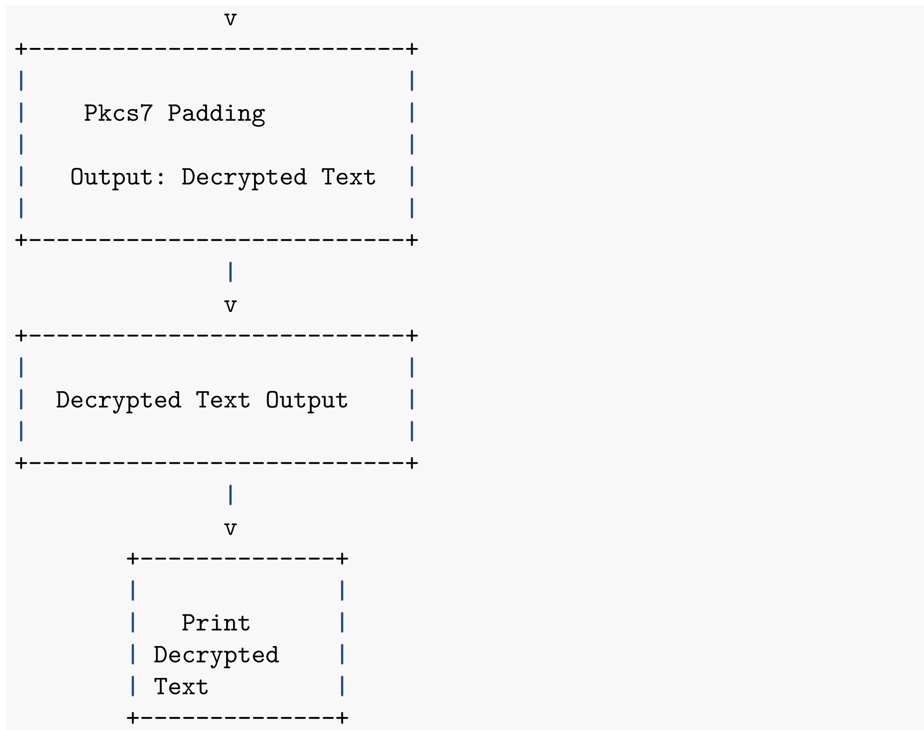
```
use aes::cipher::{block_padding::Pkcs7, BlockDecryptMut, BlockEncryptMut, KeyIvInit};
use cbc::{Encryptor, Decryptor};
use hex_literal::hex;

type Aes128CbcEnc = Encryptor<aes::Aes128>;
```

```rust
type Aes128CbcDec = Decryptor<aes::Aes128>;

fn encrypt_cbc(key: &[u8], iv: &[u8], plaintext: &[u8]) -> Vec<u8> {
    let mut buf = Vec::from(plaintext);
    let cipher = Aes128CbcEnc::new(key.into(), iv.into());
    cipher.encrypt_padded_vec_mut::<Pkcs7>(&mut buf)
}

fn decrypt_cbc(key: &[u8], iv: &[u8], ciphertext: &[u8]) -> Vec<u8> {
    let mut buf = Vec::from(ciphertext);
    let cipher = Aes128CbcDec::new(key.into(), iv.into());
    cipher.decrypt_padded_vec_mut::<Pkcs7>(&mut buf).unwrap()
}

let key = [0x42; 16];
let iv = [0x24; 16];
let plaintext = *b"Hello, World!";

// Encrypt
let ciphertext = encrypt_cbc(&key, &iv, &plaintext);
println!("Encrypted Text: {:?}\n", String::from_utf8_lossy(&ciphertext));

// Decrypt
let decrypted_text = decrypt_cbc(&key, &iv, &ciphertext);
println!("Decrypted Text: {:?}", String::from_utf8_lossy(&decrypted_text));
```





```
|   Input: Plaintext         |
|                            |
+----------------------------+
               |
               v
+----------------------------+
|                            |
|    Pkcs7 Padding           |
|                            |
|   Output: Ciphertext       |
|                            |
+----------------------------+
               |
               v
+----------------------------+
|                            |
|    Encrypted Text Output   |
|                            |
+----------------------------+
               |
               v
        +---------------+
        |              |
        |   Print      |
        | Encrypted    |
        | Text         |
        +---------------+
               |
               v
+--------------------------------+
|                                |
|     Decrypt CBC                |
|                                |
+--------------------------------+
               |
               v
+----------------------------+
|                            |
|     Decrypt Process        |
|                            |
|                            |
|   Input: Ciphertext        |
|                            |
+----------------------------+
               |
```



This code snippet features two functions: `encrypt_cbc` for encryption and `decrypt` for decryption. The code validates the integrity of the ciphertext and proceeds with the decryption process, considering the nuances of CBC mode and ensuring proper padding removal.

In Rust, as in any language, such cryptographic implementations demand attention to detail. The `decrypt_padded_vec_mut` function assumes certain conditions that warrant explicit validation in a real-world scenario. The `decrypt` function, the heart of the operation, orchestrates the decryption process, ensuring adherence to AES specifications in CBC mode.

Understanding symmetric-key encryption proves invaluable in penetration testing scenarios, where knowledge of algorithms and modes can enhance success. Rust, with its syntax and constructs, empowers developers to navigate the intricacies of encryption securely. Symmetric-key encryption, while efficient, poses key management challenges, a characteristic where we must exercise attention in key distribution and security protocols. Asymmetric cryptography, a topic yet to be explored in this context, stands as a potential solution to the key distribution difficulty, offering enhanced security measures against unauthorized access.

```
:dep aes = {version="0.8.3"}
```

```
:dep cbc = {version="0.1.2", features=["alloc"]}
```

```
use aes::cipher::{block_padding::Pkcs7, BlockDecryptMut, BlockEncryptMut, KeyIvInit};
use cbc::{Encryptor, Decryptor};
use hex_literal::hex;

type Aes128CbcEnc = Encryptor<aes::Aes128>;
type Aes128CbcDec = Decryptor<aes::Aes128>;

fn encrypt_cbc(key: &[u8], iv: &[u8], plaintext: &[u8]) -> Vec<u8> {
    let mut buf = Vec::from(plaintext);
    let cipher = Aes128CbcEnc::new(key.into(), iv.into());
    cipher.encrypt_padded_vec_mut::<Pkcs7>(&mut buf)
}

fn decrypt_cbc(key: &[u8], iv: &[u8], ciphertext: &[u8]) -> Vec<u8> {
    let mut buf = Vec::from(ciphertext);
    let cipher = Aes128CbcDec::new(key.into(), iv.into());
    cipher.decrypt_padded_vec_mut::<Pkcs7>(&mut buf).unwrap()
}

let key = [0x42; 16];
let iv = [0x24; 16];
let plaintext = *b"Hello, World!";

// Encrypt
let ciphertext = encrypt_cbc(&key, &iv, &plaintext);
println!("Encrypted Text: {:?}\n", String::from_utf8_lossy(&ciphertext));

// Decrypt
let decrypted_text = decrypt_cbc(&key, &iv, &ciphertext);
println!("Decrypted Text: {:?}", String::from_utf8_lossy(&decrypted_text));
```

```
Encrypted Text: "??A?\u{1f}?_?)?R\u{5}P???"

Decrypted Text: "Hello, World!"
```

1.5 Asymmetric Encryption

In the world of Rust programming, we delve into the world of asymmetric cryptography; A domain that offers solutions to the challenges posed by symmetric-key encryption. Unlike its counterpart, asymmetric cryptography employs two interconnected yet distinct keys: one accessible to the public, the other safeguarded privately. The essence lies in the fact that data encrypted with the private key is decipherable solely by the public key, and vice versa. This nature of the private key ensures the confidentiality of data encrypted with the public key.

Additionally, the private key can authenticate a user by enabling them to sign messages, decryptable only by the public key.

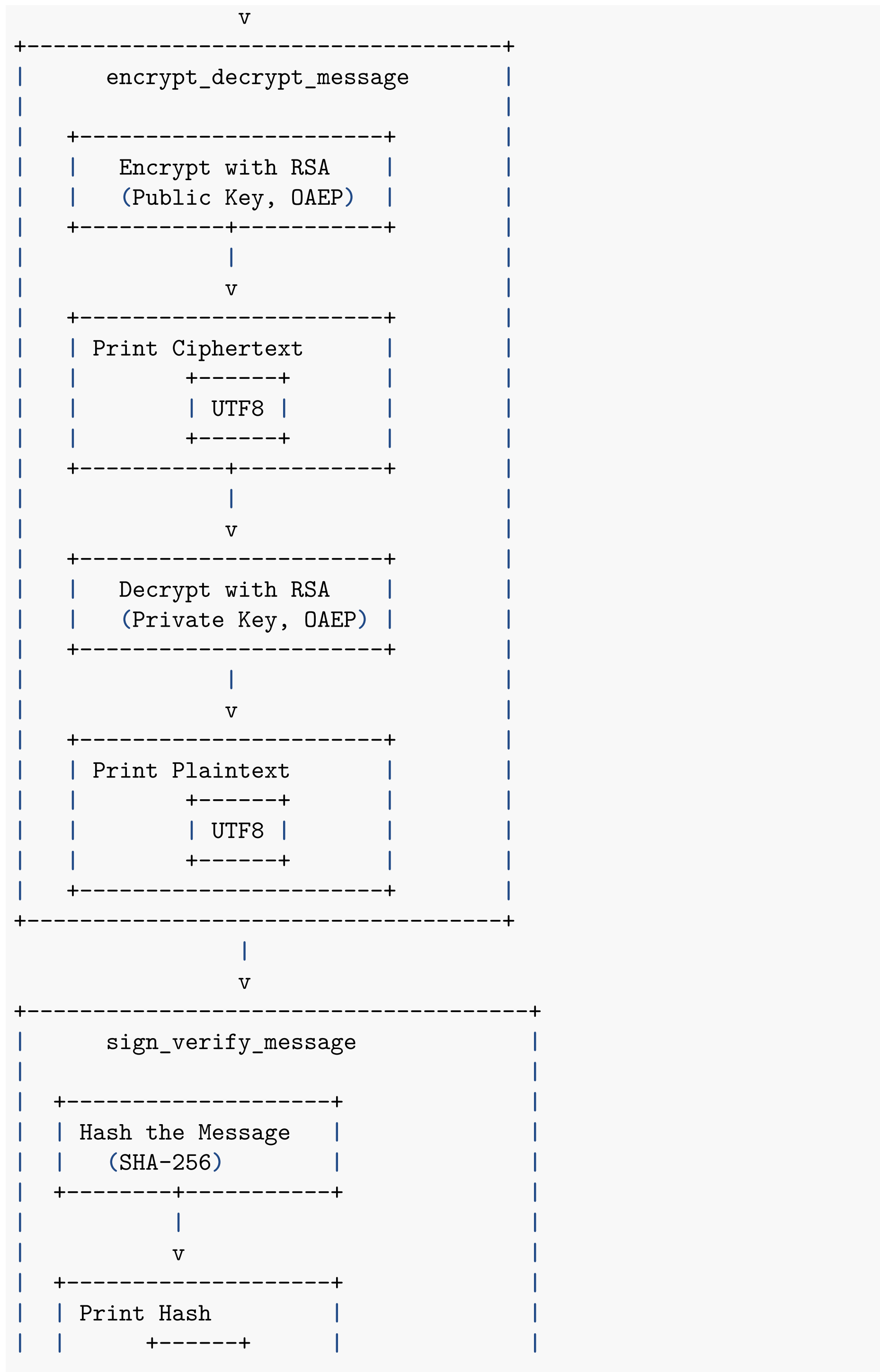
Now, one might consider the necessity of symmetric-key cryptography given the guarantees provided by public-key encryption. The answer lies in speed; public-key cryptography tends to be slower than its symmetric counterpart. To strike a balance, organizations often adopt a hybrid approach, utilizing asymmetric cryptography for initial communication negotiations and subsequently establishing an encrypted channel for the exchange of a smaller symmetric key, known as a session key.

Let's delve into typical use cases of public-key cryptography in Rust, beginning with encryption and signature validation. In the provided Rust code snippet, we observe the implementation of asymmetric encryption and the validation of digital signatures. The main function encompasses key pair generation, encryption, decryption, and signature processes. It's essential to note that while this example is comprehensive, it simplifies the complexities inherent in a practical implementation, which would typically include key exchange between remote nodes.

```
use rsa::{Pkcs1v15Encrypt, RsaPrivateKey, RsaPublicKey};
use rsa::pkcs1v15::{SigningKey, VerifyingKey};
use rsa::signature::{Signer, Verifier};
use rand::thread_rng;
use rsa::sha2::{Digest, Sha256};

fn generate_key_pair() -> (RsaPrivateKey, RsaPublicKey) {
    let mut rng = thread_rng();
    let bits = 2048;
    let private_key = RsaPrivateKey::new(&mut rng, bits).expect("Failed to generate private
    let public_key = RsaPublicKey::from(&private_key);
    (private_key, public_key)
}

fn encrypt_decrypt_message(public_key: &RsaPublicKey, private_key: &RsaPrivateKey, message:
    // Encrypt with OAEP padding
    let enc_data = public_key
        .encrypt(&mut thread_rng(), Pkcs1v15Encrypt, message)
        .expect("Encryption failed");
    println!("Ciphertext: {:?}\n", String::from_utf8_lossy(&enc_data));

    // Decrypt with OAEP padding
    let dec_data = private_key
        .decrypt(Pkcs1v15Encrypt, &enc_data)
        .expect("Decryption failed");
    println!("Plaintext: {:?}\n", String::from_utf8_lossy(&dec_data));
}
```

```
fn sign_verify_message(private_key: &RsaPrivateKey, public_key: &RsaPublicKey, message: &[u8
    // Hash the message using SHA-256
    let hash = Sha256::digest(message);

    // Sign the hash with PKCS#1 v1.5 padding
    let signing_key = SigningKey::<Sha256>::new(private_key.clone());
    let signature = signing_key.sign(&hash);
    println!("Signature: {:?} \n", signature);

    // Verify
    let verifying_key = signing_key.verifying_key();
    verifying_key.verify(&hash, &signature).expect("failed to verify");
    println!("Signature verified");
}

let (private_key, public_key) = generate_key_pair();
let message = b"A super duper secret message!";

encrypt_decrypt_message(&public_key, &private_key, message);
sign_verify_message(&private_key, &public_key, message);
```

```
+---------------------------------+
|         generate_key_pair       |
|                                 |
|            +------+             |
|            | RNG  |             |
|            +--+---+             |
|               |                 |
|               v                 |
|            +------+             |
|            |RSAKey|             |
|            |Gen.  |             |
|            +---+--+             |
|                |                |
|                v                |
|            +------+             |
|            |RSAKey|             |
|            |Pub.  |             |
|            +------+             |
|                |                |
|                v                |
|           (private_key,         |
|            public_key)          |
+---------------------------------+
                |
```



```
                 v
+-------------------------------------+
|      encrypt_decrypt_message        |
|                                     |
|   +------------------------+        |
|   |   Encrypt with RSA     |        |
|   |   (Public Key, OAEP)   |        |
|   +-----------+------------+        |
|               |                     |
|               v                     |
|   +------------------------+        |
|   | Print Ciphertext       |        |
|   |        +------+        |        |
|   |        | UTF8 |        |        |
|   |        +------+        |        |
|   +-----------+------------+        |
|               |                     |
|               v                     |
|   +------------------------+        |
|   |   Decrypt with RSA     |        |
|   |   (Private Key, OAEP)  |        |
|   +------------------------+        |
|               |                     |
|               v                     |
|   +------------------------+        |
|   | Print Plaintext        |        |
|   |        +------+        |        |
|   |        | UTF8 |        |        |
|   |        +------+        |        |
|   +------------------------+        |
+-------------------------------------+
                 |
                 v
+---------------------------------------+
|      sign_verify_message              |
|                                       |
|  +---------------------+              |
|  | Hash the Message    |              |
|  |   (SHA-256)         |              |
|  +--------+------------+              |
|           |                           |
|           v                           |
|  +---------------------+              |
|  | Print Hash          |              |
|  |      +------+       |              |
```

```
|   |       | UTF8 |       |              |
|   |       +------+       |              |
|   +--------------------+              |
|             |                          |
|             v                          |
|   +--------------------+              |
|   | Sign Hash with RSA |              |
|   |  (Private Key,     |              |
|   |   PKCS\#1 v1.5)    |              |
|   +---------+----------+              |
|             |                          |
|             v                          |
|   +--------------------+              |
|   | Print Signature    |              |
|   |       +------+     |              |
|   |       |UTF8HEX|    |              |
|   |       +------+     |              |
|   +--------------------+              |
|             |                          |
|             v                          |
|   +--------------------+              |
|   | Verify Signature   |              |
|   |    (Public Key,    |              |
|   |     PKCS\#1 v1.5)  |              |
|   +--------------------+              |
|             |                          |
|             v                          |
|   +--------------------+              |
|   | Print "Signature   |              |
|   | Verified"          |              |
|   +--------------------+              |
+----------------------------------------+
```

This Rust program illustrates two fundamental functions in public-key cryptography - encryption/decryption and message signing. The program begins by generating a key pair, followed by operations such as encrypting a message with the public key, decrypting it with the private key, and verifying the message signature using the public key. This example serves as an introduction to the essential concepts of asymmetric cryptography within the Rust programming language.

```
:dep rsa = {version="0.9.6", features=["sha2"]}
```

```
:dep rand = {version="0.8.5"}
```

```
use rsa::{Pkcs1v15Encrypt, RsaPrivateKey, RsaPublicKey};
use rsa::pkcs1v15::{SigningKey, VerifyingKey};
use rsa::signature::{Signer, Verifier, Keypair};
use rand::thread_rng;
use rsa::sha2::{Digest, Sha256};

fn generate_key_pair() -> (RsaPrivateKey, RsaPublicKey) {
    let mut rng = thread_rng();
    let bits = 2048;
    let private_key = RsaPrivateKey::new(&mut rng, bits).expect("Failed to generate private
    let public_key = RsaPublicKey::from(&private_key);
    (private_key, public_key)
}

fn encrypt_decrypt_message(public_key: &RsaPublicKey, private_key: &RsaPrivateKey, message:
    // Encrypt with OAEP padding
    let enc_data = public_key
        .encrypt(&mut thread_rng(), Pkcs1v15Encrypt, message)
        .expect("Encryption failed");
    println!("Ciphertext: {:?}\n", String::from_utf8_lossy(&enc_data));

    // Decrypt with OAEP padding
    let dec_data = private_key
        .decrypt(Pkcs1v15Encrypt, &enc_data)
        .expect("Decryption failed");
    println!("Plaintext: {:?}\n", String::from_utf8_lossy(&dec_data));
}

fn sign_verify_message(private_key: &RsaPrivateKey, public_key: &RsaPublicKey, message: &[u8
    // Hash the message using SHA-256
    let hash = Sha256::digest(message);

    // Sign the hash with PKCS#1 v1.5 padding
    let signing_key = SigningKey::<Sha256>::new(private_key.clone());
    let signature = signing_key.sign(&hash);
    println!("Signature: {:?} \n", signature);

    // Verify
    let verifying_key = signing_key.verifying_key();
    verifying_key.verify(&hash, &signature).expect("failed to verify");
    println!("Signature verified");
}

let (private_key, public_key) = generate_key_pair();
let message = b"A super duper secret message!";
```

```
encrypt_decrypt_message(&public_key, &private_key, message);
sign_verify_message(&private_key, &public_key, message);
```

```
Ciphertext: "????=e?)?\u{1f}\u{15}y?Qj??-????_-??V??i?\u{3}Q?9g\u{16}????R?\u{13} ?]\u{6}o?

Plaintext: "A super duper secret message!"

Signature: Signature("08237351B51C72F47641724FAB834BA9FAE5BEC23C62D2A69AE1BAFAEE4EB038D44ED

Signature verified
```