

Secure File and Message Encryption/Decryption with PHP

Overview

This web-based cryptography demonstration is a secure tool for encrypting and decrypting both text messages and files. It combines two powerful cryptographic methods:

1. **Diffie-Hellman Key Exchange:** A secure way for two parties to share a secret key without sending it over the network. This shared key is used for encryption.
2. **AES-256-CBC (Advanced Encryption Standard):** A strong encryption algorithm that protects messages and files from unauthorized access.

Key Features

1. Message Encryption

You can type a message into the application, and it will encrypt it using the shared key. Encrypted messages can be sent securely to another party. The recipient can then use the application to decrypt the message, restoring it to its original form.

2. File Encryption

The application also allows you to upload files, such as documents, for encryption. Once encrypted, the file becomes unreadable without the correct key. Decrypting the file will return it to its original state.

3. Secure Key Sharing Between Two Parties

The Diffie-Hellman Key Exchange ensures that two users can share a secure key even if they are communicating over an insecure channel. This shared key is used for encryption and decryption, ensuring secure communication.

How It Works

1. **Key Exchange:** The two parties use the Diffie-Hellman algorithm to generate a shared key without revealing it over the network.
2. **Encryption:** The shared key is used with AES-256-CBC to encrypt messages or files.
3. **Decryption:** The same shared key is used to decrypt the encrypted content, making it readable again.

This process ensures that all communication and file sharing between the parties are secure and private, protecting sensitive information from potential threats.

This document will guide you through a PHP-based implementation for secure communication, involving Diffie-Hellman key exchange for secure key generation, and AES (Advanced Encryption Standard) for encryption and decryption of messages and files. The solution is designed to run in a local environment using XAMPP (Starting the Apache) and Visual Studio Code. The code is structured in such a way that it provides a practical demonstration of how cryptographic protocols work in real-world applications.

We will break down the PHP code, discussing its purpose, functionality, and how it fits into the overall system, step by step. The environment setup, configuration files, and how to run the system will also be explained.

Step-by-Step Breakdown of the Process

1. Accessing the Application

To run this application I have created the php file inside encryption_decryption directory, place the encryption_decryption directory (that contains the php file) inside the htdocs directory of your XAMPP installation and run the Xampp (start the Apache). Once the file is in place, open your web browser and navigate to http://localhost/encryption_decryption/dh_aes.php. This will bring up a simple web form where you can choose different actions like encrypting or decrypting messages and files.

2. Diffie-Hellman Key Exchange Process

The heart of this demonstration is the Diffie-Hellman (DH) key exchange algorithm. This is a cryptographic method used to securely share a secret key between two parties over an insecure channel. Here's how it works:

- **Private and Public Keys:** Each party generates a private key, which is a random number. From this private key, a public key is computed using the Diffie-Hellman algorithm. The public keys are exchanged between the two parties.
- **Shared Secret:** Both parties use their private keys and the received public keys to compute a shared secret. This shared secret is the same for both parties and is used to derive a secret key for encryption.

The program uses the Party class to simulate two parties, each generating their own private key and computing the shared secret. The shared secret is then hashed using SHA-256 to form a secure encryption key. This key will be used to encrypt or decrypt messages and files.

```

12 <?php
13 // Constants for Diffie-Hellman algorithm
14 // A large prime number (DH_PRIME) used as part of the Diffie-Hellman algorithm.
15 // This prime ensures secure exchange of keys between parties.
16 const DH_PRIME = '26959946667150639794667015087019630673637144422540572481103610249215' .
17 '862404159721685259687786139792977216328806797416773759226260202991' .
18 '0089912213596423414883009694977329268040609468028130167643337792914' .
19 '32893441050911660774194075867720991474836581319810300767321046996567' .
20 '45894096775906117876994783423200847665569424640676370666546365917169' .
21 '86636126815380807940256022072559125059334804366032422325864875631266' .
22 '04228254701566090700716245984701338977994718494815488227615651981412' .
23 '00034194003322961484801923976024994946501752483598389004593236842278';
24 // Generator value for Diffie-Hellman algorithm
25 const DH_GENERATOR = 5; // A common generator value used in many DH implementations
26

```

The code starts by defining two constants:

- **DH_PRIME**: A large prime number that is used as part of the Diffie-Hellman algorithm. This prime number ensures the security of the key exchange.
- **DH_GENERATOR**: A fixed integer value used in the algorithm to generate public keys.
- These values are important because Diffie-Hellman relies on modular exponentiation, where public keys are generated using the formula:

publicKey = (generator^{privateKey}) % prime.

- The **Party** class simulates two parties involved in the Diffie-Hellman key exchange.

```

04 class Party
05 {
06     private $privateKey; // Private key for the party (secret)
07     private $publicKey; // Public key generated from private key
08     private $sharedSecret; // Shared secret computed with another party (result of Diffie-Hellman exchange)
09
10     public function __construct()
11     {
12         // Generate a random private key within the range of 1 to the square root of the prime (DH_PRIME).
13         $this->privateKey = random_int(1, (int)bcsqrt(DH_PRIME, 0));
14     }
15
16     public function generatePublicKey()
17     {
18         $this->publicKey = bcpowmod(DH_GENERATOR, $this->privateKey, DH_PRIME);
19         return $this->publicKey; // Return the public key for sharing
20     }
21
22     public function computeSharedSecret($otherPublicKey)
23     {
24         $this->sharedSecret = bcpowmod($otherPublicKey, $this->privateKey, DH_PRIME);
25     }
26     public function getSharedSecretKey()
27     {
28         return hash('sha256', $this->sharedSecret, true); // Return the shared secret key
29     }
30     public static function exchangeKeys($partyA, $partyB)
31     {
32         $partyAPublic = $partyA->generatePublicKey(); // Generate public key for Party A
33         $partyBPublic = $partyB->generatePublicKey(); // Generate public key for Party B
34
35         // Simulate the exchange of public keys: each party computes the shared secret using the other party's public key
36         $partyA->computeSharedSecret($partyBPublic); // Party A computes shared secret using Party B's public key
37         $partyB->computeSharedSecret($partyAPublic); // Party B computes shared secret using Party A's public key
38     }
39 }

```

The **Party** class represents one of the two participants in the Diffie-Hellman key exchange. It stores a private key, public key, and shared secret. The private key is randomly generated, and the public key is derived from it using the Diffie-Hellman formula. The **generatePublicKey** method computes the public key for sharing, while the **computeSharedSecret** method calculates the shared secret using the other party's public key. This shared secret, identical for both parties, is hashed with SHA-256 to create a secure encryption key. The **exchangeKeys** method simulates public key exchange, enabling both parties to compute the shared secret securely.

3. AES Encryption and Decryption

Once the shared secret key is established using Diffie-Hellman, it is used with AES encryption for securing the data. AES is a symmetric-key encryption algorithm that uses a fixed-length key (256 bits in this case) to encrypt and decrypt data. In this demonstration, AES operates in CBC (Cipher Block Chaining) mode, which provides enhanced security by using an initialization vector (IV) for each operation.

- **Encrypting Messages:** A user can enter a plaintext message into the web form, and the system will encrypt it using the shared secret key and AES-256-CBC encryption. The IV is randomly generated for each encryption to make it more secure.
- **Decrypting Messages:** The encrypted message can be entered into the form, and the system will use the same shared secret key to decrypt it back to its original form.
- **Encrypting Files:** A user can upload a file to be encrypted. The system reads the file in chunks, encrypts it using AES, and saves the encrypted file with the IV prepended to it.
- **Decrypting Files:** Similarly, an encrypted file can be uploaded to the system, and it will be decrypted using the shared secret key.

```
104 class AES
105 {
106     public static function encryptMessage($key, $message)
107     {
108         $iv = random_bytes(16); // Generate a random 16-byte IV (Initialization Vector)
109         $ciphertext = openssl_encrypt($message, 'aes-256-cbc', $key, OPENSSL_RAW_DATA, $iv); // Encrypt the message
110         return base64_encode($iv . $ciphertext); // Return the encrypted message along with the IV (Base64 encoded)
111     }
112     public static function decryptMessage($key, $encryptedMessage)
113     {
114         $data = base64_decode($encryptedMessage); // Decode the encrypted message from Base64 format
115         $iv = substr($data, 0, 16); // Extract the IV from the beginning of the data
116         $ciphertext = substr($data, 16); // Extract the ciphertext (the actual encrypted data)
117         return openssl_decrypt($ciphertext, 'aes-256-cbc', $key, OPENSSL_RAW_DATA, $iv);
118     }
119     public static function encryptFile($key, $filePath, $outputPath)
120     {
121         $iv = random_bytes(16); // Generate a random 16-byte IV
122         $inputHandle = fopen($filePath, 'rb'); // Open the input file for reading
123         $outputHandle = fopen($outputPath, 'wb'); // Open the output file for writing
124         fwrite($outputHandle, $iv); // Write the IV to the beginning of the output file
125         while (!feof($inputHandle)) {
126             $data = fread($inputHandle, 8192); // Read 8192 bytes from the input file
127             $ciphertext = openssl_encrypt($data, 'aes-256-cbc', $key, OPENSSL_RAW_DATA, $iv); // Encrypt the data
128             fwrite($outputHandle, $ciphertext); // Write the encrypted chunk to the output file
129         }
130         fclose($inputHandle); // Close the input file
131         fclose($outputHandle); // Close the output file
132     }
133     public static function decryptFile($key, $filePath, $outputPath)
134     {
135         $inputHandle = fopen($filePath, 'rb'); // Open the input file for reading
136         $iv = fread($inputHandle, 16); // Read the IV from the beginning of the file
137         $outputHandle = fopen($outputPath, 'wb'); // Open the output file for writing
138         while (!feof($inputHandle)) {
139             $data = fread($inputHandle, 8192); // Read 8192 bytes from the input file
140             $plaintext = openssl_decrypt($data, 'aes-256-cbc', $key, OPENSSL_RAW_DATA, $iv); // Decrypt the data
141             fwrite($outputHandle, $plaintext); // Write the decrypted chunk to the output file
142         }
143         fclose($inputHandle); // Close the input file
144         fclose($outputHandle); // Close the output file
145     }
146 }
```

The AES class provides methods to encrypt and decrypt messages and files using the AES-256-CBC algorithm. It ensures data privacy by using a random 16-byte Initialization Vector (IV) for Cipher Block Chaining (CBC) mode. The encryptMessage method encrypts a message with the provided key and IV, returning a Base64-encoded result. The decryptMessage method extracts the

IV and ciphertext from the Base64-encoded input, decrypting it with the key and IV. For files, encryptFile reads and encrypts file chunks, writing the IV at the start of the output file, while decryptFile reads the IV from the file and decrypts it in chunks. This process secures both messages and files during storage or transmission.

User Interaction

The user interacts with the web interface by selecting one of the following actions from a simple form:

- **Encrypt a Message:** The user enters a message, which is then encrypted using the shared secret key and displayed on the webpage.
- **Decrypt a Message:** The user enters an encrypted message, which is decrypted using the same shared secret key.
- **Encrypt a File:** The user uploads a file, and the system encrypts the file using AES with the shared secret key.
- **Decrypt a File:** The user uploads an encrypted file, and the system decrypts it back to its original content.

For each action, the appropriate response (encrypted message or decrypted file) is displayed, making the encryption and decryption process seamless for the user.

```
63 if ($SERVER['REQUEST_METHOD'] === 'POST') {
64     $action = $_POST['action'] ?? ''; // Get the selected action from the form
65     $inputMessage = $_POST['inputMessage'] ?? ''; // Get the input message from the form
66     $inputFile = $_FILES['inputFile']['tmp_name'] ?? ''; // Get the input file from the form
67     $outputFile = $_POST['outputFile'] ?? ''; // Get the output file name from the form
68     try {
69         $party = new Party(); // Create a new party for the user
70         $party->generatePublicKey(); // Normally exchanged with another party
71         $sharedKey = $party->getSharedSecretKey(); // Generate the shared key from the private/public key exchange
72         switch ($action) {
73             case 'encryptMessage': // Encrypt a message
74                 if (empty($inputMessage)) {
75                     throw new Exception("Message is required for encryption."); // Error message if no message is provided
76                 }
77                 $encryptedMessage = AES::encryptMessage($sharedKey, $inputMessage); // Encrypt the message
78                 echo "<span class='success'>Encrypted Message: $encryptedMessage</span>"; // Display the encrypted message
79                 break; // End the case
80             case 'decryptMessage': // Decrypt a message
81                 if (empty($inputMessage)) {
82                     throw new Exception("Message is required for decryption."); // Error message if no message is provided
83                 }
84                 $decryptedMessage = AES::decryptMessage($sharedKey, $inputMessage); // Decrypt the message
85                 echo "<span class='success'>Decrypted Message: $decryptedMessage</span>"; // Display the decrypted message
86                 break;
87             case 'encryptFile': // Encrypt a file
88                 if (empty($inputFile) || empty($outputFile)) {
89                     throw new Exception("Input file and output file name are required for file encryption."); // Error message if no file is provided
90                 }
91                 AES::encryptFile($sharedKey, $inputFile, $outputFile); // Encrypt the file
92                 echo "<span class='success'>File successfully encrypted to: $outputFile</span>"; // Display success message
93                 break;
94             case 'decryptFile': // Decrypt a file
95                 if (empty($inputFile) || empty($outputFile)) {
96                     throw new Exception("Input file and output file name are required for file decryption."); // Error message if no file is provided
97                 }
98                 AES::decryptFile($sharedKey, $inputFile, $outputFile); // Decrypt the file
99                 echo "<span class='success'>File successfully decrypted to: $outputFile</span>"; // Display success message
100                break;
101            default: // Invalid action selected
102                throw new Exception("Invalid action selected."); // Error message for invalid action
103        }
104    } catch (Exception $e) {
```

This code handles form submissions in a PHP script, specifically when the form is submitted via a POST request. First, it checks if the request method is 'POST', which means the form has been

submitted. It then retrieves the form inputs, such as the action (e.g., encrypting or decrypting a message or file), the input message, the uploaded input file, and the output file name. The code creates a new instance of the Party class, which generates a public key and computes a shared secret key using the Diffie-Hellman method. Based on the selected action, the script will either encrypt or decrypt a message or file. If the action is to encrypt or decrypt a message, it checks if the message is provided. If the action is to encrypt or decrypt a file, it checks if both the input file and output file name are given. If any required input is missing, it throws an error. When the action is successfully completed, it outputs a success message with the result (such as the encrypted/decrypted message or file). If an invalid action is selected or an error occurs, an error message is displayed. The try-catch block ensures that any errors are caught and displayed without crashing the script. This code manages user inputs, processes encryption/decryption tasks, and handles errors, providing clear feedback to the user.

Step-by-Step Guide for Encrypting and Decrypting with Diffie-Hellman and AES

To use the form for encryption and decryption on the `dh_aes.php` page after running Apache in XAMPP, let's walk through the process in detail step by step. The form is designed to allow you to encrypt and decrypt both messages and files using a combination of the Diffie-Hellman algorithm for secure key exchange and AES (Advanced Encryption Standard) for encryption and decryption. Below is a breakdown of how a user would interact with the form and how it functions in the background.

Step-by-Step Explanation of the Form

1. Opening the Page in the Browser

Once you have the Apache server up and running on XAMPP, you can begin using the application by navigating to the appropriate page in your browser. Follow these steps:

1. Start Apache in XAMPP

- Launch **XAMPP Control Panel** on your computer.
- Click on the **Start** button next to **Apache** to initiate the web server.

2. Navigate to the Application Page

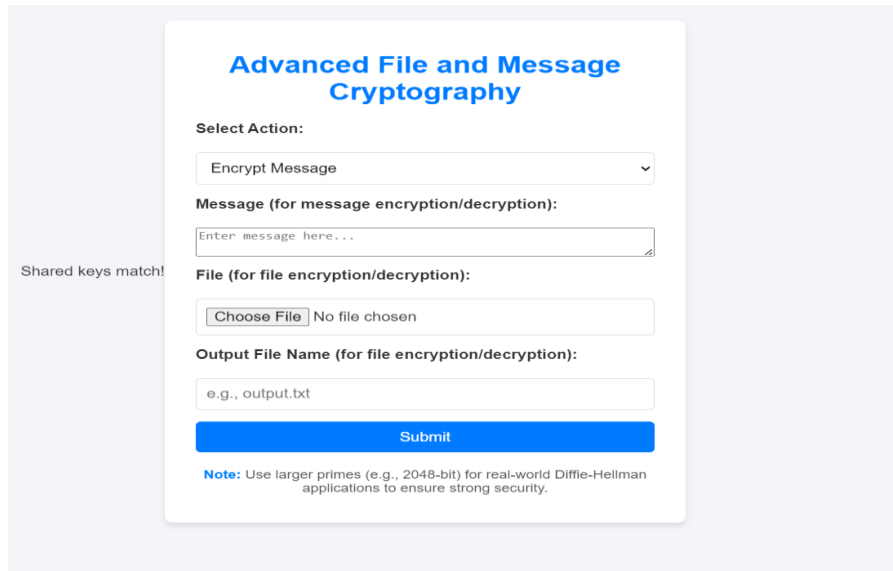
- After Apache is running, open your preferred web browser (e.g., Google Chrome, Firefox, etc.).
- In the address bar, type the following URL and hit **Enter**:

`http://localhost/encryption_decryption/dh_aes.php`

This URL points to the `dh_aes.php` file located inside the **encryption_decryption** folder, which should be placed within the **htdocs** directory of your XAMPP installation.

Once the page is loaded, you will be greeted with a **well-designed form** that serves as the main interface for encrypting and decrypting messages and files. The form is designed to be user-friendly, allowing you to select various actions with ease (As it is shown in the pic). You'll find that the form is divided into several key sections, as depicted in the image below:

- **Action Dropdown:** This allows you to choose what you want to do, whether it's encrypting or decrypting a message or a file.
- **Message Field:** This area allows you to input a message that you wish to encrypt or decrypt.
- **File Input Field:** This is where you can select a file to encrypt or decrypt.
- **Output File Name:** Here, you can provide a name for the output file after encrypting or decrypting a file.
- **Submit Button:** Once you fill in the necessary information, this button will submit the form and initiate the action.



The screenshot shows a web form titled "Advanced File and Message Cryptography". On the left side, there is a status message: "Shared keys match!". The form itself contains the following sections:

- Select Action:** A dropdown menu with "Encrypt Message" selected.
- Message (for message encryption/decryption):** A text input field with the placeholder "Enter message here...".
- File (for file encryption/decryption):** A file selection area with a "Choose File" button and the text "No file chosen".
- Output File Name (for file encryption/decryption):** A text input field with the placeholder "e.g., output.txt".
- Submit:** A prominent blue button.
- Note:** A small text note at the bottom states: "Note: Use larger primes (e.g., 2048-bit) for real-world Diffie-Hellman applications to ensure strong security."

2. Action Selection (Encrypt/Decrypt)

The first thing you need to do is decide what action you want to perform. The form provides a dropdown menu (select) with the following options:

- **Encrypt Message:** Encrypt a plaintext message.
- **Decrypt Message:** Decrypt an encrypted message.
- **Encrypt File:** Encrypt a file using AES encryption.
- **Decrypt File:** Decrypt a previously encrypted file.

Example: If you want to encrypt a message, you would select the "Encrypt Message" option.

Advanced File and Message Cryptography

Select Action:

Encrypt Message

Encrypt Message

Decrypt Message

Encrypt File

Decrypt File

File (for file encryption/decryption):

Choose File No file chosen

Output File Name (for file encryption/decryption):

e.g., output.txt

Submit

Note: Use larger primes (e.g., 2048-bit) for real-world Diffie-Hellman applications to ensure strong security.

Shared keys match!

3. Message Field for Encryption/Decryption

Depending on the action you select, you will either need to enter a message or use the file upload option. If you choose to encrypt or decrypt a **message**, there is a field labeled **Message (for message encryption/decryption)** where you need to input your text.

- **For Encryption:** If you select the **Encrypt Message** option and enter a message like:
 - **Example Message:** "Hello, this is a test message."

Advanced File and Message Cryptography

Select Action:

Encrypt Message

Message (for message encryption/decryption):

Hello, this is a test message.

File (for file encryption/decryption):

Choose File No file chosen

Output File Name (for file encryption/decryption):

e.g., output.txt

Submit

Note: Use larger primes (e.g., 2048-bit) for real-world Diffie-Hellman applications to ensure strong security.

Shared keys match!

The result after clicking on submit:

Shared keys match! Encrypted Message: TXu+Wfbi+mSgYoJdTE5A7bzdTXCdarrUB/N4e/TGG7pPuhvSUYtoq1kuzCqXTCQo

- **For Decryption:** If you select the **Decrypt Message** option, you would enter an encrypted message that was previously encrypted using AES. For example, an encrypted message might look something like:
- **Example Encrypted Message :**
"TXu+Wfbi+mSgYoJdTE5A7bzdTXCdarrUB/N4e/TGG7pPuhvSUYtoq1kuzCqXTCQo"

The screenshot shows a web application titled "Advanced File and Message Cryptography". It has a "Select Action:" dropdown menu with "Decrypt Message" selected. Below this is a text input field for the "Message (for message encryption/decryption):" containing the encrypted message "TXu+Wfbi+mSgYoJdTE5A7bzdTXCdarrUB/N4e/TGG7pPuhvSUYtoq1kuzCqXTCQo". There is also a "File (for file encryption/decryption):" section with a "Choose File" button and the text "No file chosen". Below that is an "Output File Name (for file encryption/decryption):" input field with the placeholder "e.g., output.txt". A blue "Submit" button is at the bottom. A note at the bottom states: "Note: Use larger primes (e.g., 2048-bit) for real-world Diffie-Hellman applications to ensure strong security."

After clicking on submit:

Shared keys match! Decrypted Message: Hello, this is a test message.

4. File Upload for Encryption/Decryption

If you want to encrypt or decrypt a file instead of a message, you need to upload the file through the **File (for file encryption/decryption)** input field. You can choose a file from your local machine to upload.

- **For Encrypting a File:** If you select the **Encrypt File** action, browse and choose a file to upload (e.g., a text file). After selecting the file, you'll need to specify an output file name in the **Output File Name** field.

- **For Decrypting a File:** If you select the **Decrypt File** action, you will also need to upload the encrypted file and specify a name for the output decrypted file.

Example: Let's say you want to encrypt a file called message (this example file is already in . After selecting the file, you would specify an output file name like encrypted_message (you can give it any name).

Advanced File and Message Cryptography

Select Action:

Encrypt File

Message (for message encryption/decryption):

Enter message here...

File (for file encryption/decryption):

Choose File message.txt

Output File Name (for file encryption/decryption):

encrypted_message.txt

Submit

Note: Use larger primes (e.g., 2048-bit) for real-world Diffie-Hellman applications to ensure strong security.

Shared keys match!

After clicking on submit:

Shared keys match!File successfully encrypted to: encrypted_message

You will find the doc/file automatically in your folder in xampp.

5. Submitting the Form

Once you've selected the action and filled out the relevant fields (Message or File, Output File Name), you can click the **Submit** button to initiate the process. After submitting, the form will handle the backend processing based on the Diffie-Hellman key exchange and AES encryption algorithms.

6. How the Encryption/Decryption Works in the Background

When the form is submitted, the PHP code first uses the **Diffie-Hellman** algorithm to securely exchange keys between two parties. Here's a quick breakdown of the key generation process:

- **Diffie-Hellman Key Exchange:** This is a cryptographic protocol that allows two parties to securely exchange keys over an insecure communication channel. Each party generates a public/private key pair, and the public keys are exchanged. Then, each party computes the shared secret using the other party's public key and its own private key. This shared secret is then used to derive an AES key for encryption and decryption.
- **AES Encryption:** Once the shared secret is computed, it is hashed to create a secure encryption key. This key is used with the AES algorithm (AES-256-CBC in this case) to encrypt or decrypt the message or file.

7. Encryption Process for Messages and Files

- **Encrypting a Message:** After you submit a message for encryption, the system encrypts it using the shared secret derived from Diffie-Hellman. The message is encrypted using the AES algorithm, and the result is returned to you as a base64-encoded string.
- **Encrypting a File:** When you upload a file for encryption, the system reads the file in chunks (to avoid memory issues), encrypts each chunk using AES, and then stores the encrypted file. The Initialization Vector (IV) used for AES encryption is prepended to the encrypted file to ensure that decryption is possible later.

8. Decryption Process for Messages and Files

- **Decrypting a Message:** If you have an encrypted message and select the **Decrypt Message** action, the system uses the same Diffie-Hellman-derived key to decrypt the message. The decrypted message will then be displayed.
- **Example:** If you input the encrypted message:
“ALAQ8DI3JZvqtrNvKF4pQ0jnz4FTeBf5VM2vct1+oDM=”, the system would output the original message: “Hello”
- **Decrypting a File:** For file decryption, the system reads the encrypted file, extracts the IV, and then decrypts the content in chunks using the AES algorithm. The decrypted file is then saved and can be downloaded with the specified output name.
 - **Example:** You upload an encrypted file called encrypted_message, and after decryption, the system generates a decrypted file decrypted_message.

Advanced File and Message Cryptography

Select Action:

Decrypt File

Message (for message encryption/decryption):

Enter message here...

File (for file encryption/decryption):

Choose File encrypted_message

Output File Name (for file encryption/decryption):

decrypted_message

Submit

Note: Use larger primes (e.g., 2048-bit) for real-world Diffie-Hellman applications to ensure strong security.

Shared keys match!

After clicking on submit:

Shared keys match!File successfully decrypted to: decrypted_message

9. Handling Errors and Displaying Messages

The form also handles errors such as missing fields (e.g., if you try to encrypt a message but don't provide any message) and displays success or error messages to the user. For instance:

- **Success Message:**

"Encrypted Message: U2FsdGVkX19iqzGh5wNxqlID3bdUM7ywW9obkPdkpq4="

- **Error Message:** "Error: Message is required for encryption."

These messages are displayed directly under the form, letting you know whether your encryption or decryption process was successful or if there was an issue.

Conclusion:

By following these steps, the user is able to securely encrypt and decrypt both messages and files using AES encryption and Diffie-Hellman for key exchange. The user-friendly interface ensures that even someone without extensive knowledge of encryption algorithms can easily use the system for their cryptographic needs. You simply need to select your action, provide your input (message or file), and the system takes care of the rest, using secure algorithms to process your data.

Example Scenario:

Let's say you are working on a sensitive project and need to send a file securely to a colleague. You upload your file (e.g., `project_notes`) and select the "Encrypt File" action. The system encrypts the file, and you get an output file `encrypted_project_notes`. You send this file to your colleague.

Later, your colleague uses the same system to decrypt the file by selecting the "Decrypt File" action, uploads the encrypted file, and enters an output name for the decrypted file. The system decrypts the file and returns the original file, ensuring that the file remains secure during the transfer process.

Now let's say that you want to change and test the key:

To test the form and change the key during testing, here are some practical methods and scenarios to try out. This will help you understand how changes in keys affect the encryption and decryption process.

Ways to Change and Test the Key:

Here are three ways to change and test the key, along with instructions on how to validate the changes:

Modify the Diffie-Hellman Key Exchange Inputs (Prime or Generator)

- **What to do:** Change the `DH_PRIME` and `DH_GENERATOR` values in the script.
- **Why this works:** Since the key is generated using the formula $(\text{generator}^{\text{privateKey}}) \% \text{prime}$, changing the prime or generator will change the public keys and, consequently, the shared secret.
- **How to test:**
 1. Change the prime to a smaller prime (only for testing). For example:
 2. `const DH_PRIME = '23'; // Smaller prime for testing purposes`
 3. Change the generator to a new value. For example:
 4. `const DH_GENERATOR = 2; // New generator`

What to check:

1. Submit a message for encryption.
2. Copy the encrypted message.
3. Change the prime or generator values.
4. Reload the page and try to decrypt the encrypted message.
5. Expected result: Decryption will fail because the shared key has changed.

Notes:

- Why this works: Changing the prime or generator changes the structure of how public keys are derived, affecting the shared secret.
- Security Note: In production, use large primes (2048+ bits) and safe primes for security.

Use Fixed Private Keys (Instead of Random Ones)

- What to do: Fix the private key instead of generating it randomly.
- Why this works: By fixing the private key, you can manually control the key to see its impact on the public and shared keys.
- How to test:
 1. Change the constructor in the Party class to use a hardcoded private key like this:
 2.

```
public function __construct(){ $this->privateKey = 10; // Hardcoded private key (instead of random)}
```

What to check:

1. Use the form to encrypt a message.
2. Decrypt the message and ensure it works as expected.
3. Change the hardcoded private key to a new value.
4. Try to decrypt the previously encrypted message.
5. Expected result: Decryption will fail because the shared key has changed.

Notes:

- Why this works: The shared key is derived from the private keys. By changing the private key, you alter the shared key and encryption key.
- How to undo: Replace the line back to the original:
- ```
$this->privateKey = random_int(1, (int)bcsqrt(DH_PRIME, 0));
```

### Key Takeaways

- The key in Diffie-Hellman is **determined by the prime, generator, and private key**. Changing any of these affects the shared key.
- By changing these inputs, you can simulate changes to the shared key, which you can then test by encrypting and decrypting messages and files.
- **Pro Tip:** If you want to reset everything, change back the **private key** to random generation and revert the **shared secret** calculation.