

Security Engineering - Lab 2

Harris A. Ransom

April 21, 2025

Introduction

In order to ensure the confidentiality, integrity, and availability of information (particularly financial information), it is important to understand how to choose and implement appropriate cryptographic technologies and controls. This lab report will demonstrate how cryptographic techniques such as encryption, hashing, key exchange, and digital signatures can work together to create a secure financial transaction system.

Note: All source files associated with this lab can be found [in this GitHub repository](#).

Step 1: Get Your Crypto Toolkit (Docker Image)

```
harris@gaming-pc: ~/Lab02$ docker pull dlambros/labs:lab2
permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.45/images/create?fromImage=dlambros%2Flabs&tag=lab2": dial unix /var/run/docker.sock: connect: permission denied

harris@gaming-pc: ~/Lab02$ sudo !!
[sudo] password for harris:
lab2: Pulling from dlambros/labs
f18232174bc9: Pull complete
22aa3859edea: Pull complete
bd91b261bd2: Pull complete
ad1850f0a8bc: Pull complete
aa97cc186eb3: Pull complete
747a4aa00c18: Pull complete
22fe7f75568a: Pull complete
59eceb8b9082: Pull complete
bb64cca7ce0e: Pull complete
4f4fb700ef54: Pull complete
Digest: sha256:33e500150102b123fb2f992b5840f6e28d244a7bf9e1c205a2359c205fc40f05
Status: Downloaded newer image for dlambros/labs:lab2
docker.io/dlambros/labs:lab2

harris@gaming-pc: ~/Lab02$ sudo docker images
REPOSITORY          TAG          IMAGE ID      CREATED      SIZE
dlambros/labs        lab2         a24414fbbd77 17 hours ago 185MB
dlambros/labs        lab1         043596fdeff2 8 weeks ago  11.1MB
tf-sensor-augment    latest       5ed7064fec2e 3 months ago 4.05GB
<none>               <none>       5c819e3dba02 3 months ago 8.04GB
<none>               <none>       7f42bc7cc6be 3 months ago 7.96GB
tensorflow/tensorflow latest-gpu    df6d9691dd54 5 months ago 7.36GB
isg                  latest       e5a994f63fc6 22 months ago 3.61GB
ubuntu              22.04       1f6ddc1b2547 22 months ago 77.8MB
<none>               <none>       a385e3d1f422 2 years ago  920MB
oscarakaelvis/evil-winrm latest       d7e6d6dcefa9 2 years ago  917MB
ruby                 latest       5bfd2dfe01e7 2 years ago  892MB
tensorflow/tensorflow 1.15.2-gpu  182b13b4a8e6 5 years ago  3.56GB

harris@gaming-pc: ~/Lab02$ sudo docker run -it dlambros/labs:lab2
~/lab_files $ whoami
student
~/lab_files $
```

Figure 1: Getting the Docker Image

Step 2: Cracking the Crapple Pay Code

1. **Frequency analysis** is a useful technique to break common classical ciphers, especially against **substitution ciphers**. In a substitution cipher, a mapping

is created between a plaintext alphabet letter and a ciphertext alphabet letter (e.g. in a Caesar cipher, A is mapped to D, B is mapped to E, etc.) However, this strategy doesn't affect the relative *frequency* of the letters in the plaintext and ciphertext. This makes it possible to compare the apparent frequency of letters in the ciphertext and try to match them up to the frequencies of letters in typical English text. For example, "E" is the most commonly used letter in English text, so if we see that the ciphertext from a substitution cipher has "U" as the most common letter, we can guess that U maps to E in the substitution cipher.

Frequency analysis can also be used to perform fingerprinting of ciphers. For example, if the observed ciphertext frequencies approximately match the shape of frequencies in English plaintext, then it's a safe bet to guess that a substitution cipher was used. However, if the ciphertext frequencies appear very different when compared to English plaintext frequencies (e.g. uniform frequency distribution), then something other than a substitution cipher was probably used to encrypt the message.

2. From the provided ciphertext in *crapple_pay_code.txt*, we can compare the relative frequencies of the ciphertext characters against English text (see Fig. 2).

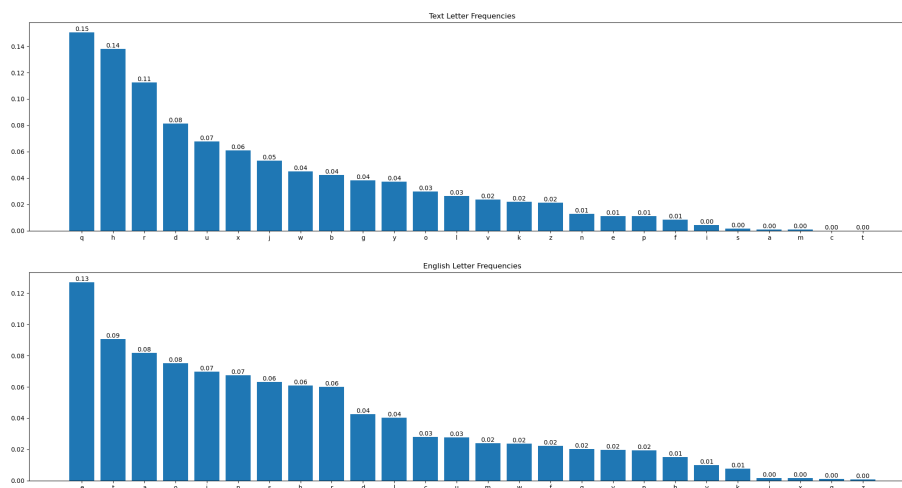


Figure 2: Relative Character Frequencies

3. The Caesar cipher is not secure for real-world use due to how it shifts every character modulo 26 (i.e. there are only 26 possible characters in its alphabet). Therefore, the shift value only has 26 unique values, making it easy to brute-force the rotation and find the plaintext (see Fig. 3).

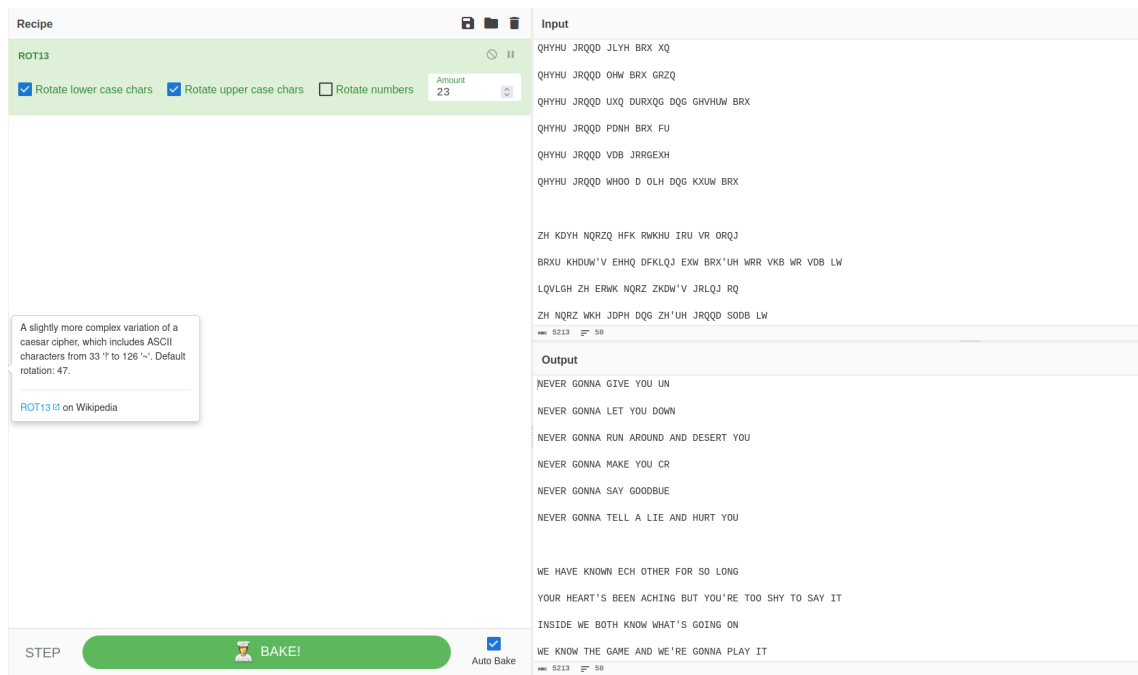


Figure 3: Decrypted Caesar Cipher Text

Step 3: Securing the Transaction Channel

1. When securing a communication channel, it is usually acceptable to use *symmetric encryption* due to its easier setup and lower complexity. Symmetric cryptosystems (e.g. AES) are usually faster and have lower computational overhead compared to asymmetric cryptosystems (e.g. RSA). It is also easier to share the single key between the two communicating parties via a key-exchange algorithm (e.g. Diffie-Hellman). The ease of setup combined with the speed and low computational overhead make symmetric cryptosystems a suitable choice for data-in-transit.
2. A **salt** is a piece of random data that is used as an additional input to a one-

```

~/lab_files $ cat payment_request.txt
Sender: Alice
Recipient: Bob's Widgets
Amount: 99.50
Current: USD
Account Number: 1234567890
Transaction ID: TXN-20250317-1615
Timestamp: 2025-03-17T16:15:00Z
Description: Purchase of Super Widget Pro~/lab_files $ openssl rand -out secret.key 32
~/lab_files $ ls -al
total 28
drwxr-xr-x 1 student student 4096 Apr  8 00:48 .
drwxr-xr-x 1 student student 4096 Apr  8 00:36 ..
-rw-rw-r-- 1 root root 1505 Mar 17 20:39 crapple_pay_code.txt
-rw-rw-r-- 1 root root 200 Mar 17 20:27 payment_request.txt
-rw-rw-r-- 1 root root 252 Apr  7 07:01 sample_passwords.txt
-rw-r--r-- 1 student student 32 Apr  8 00:48 secret.key
~/lab_files $ openssl enc -aes-256-cbc -salt -pbkdf2 -in payment_request.txt -out payment_request.enc -pass file:secretkey
Can't open file secretkey
Error getting password
288B2D3264760000:error:80000002:system library:BIO_new_file:No such file or directory:crypto/bio/bss_file.c:67:calling fopen(secretkey, r)
288B2D3264760000:error:10000000:BIO routines:BIO_new_file:no such file:crypto/bio/bss_file.c:75:
~/lab_files $ openssl enc -aes-256-cbc -salt -pbkdf2 -in payment_request.txt -out payment_request.enc -pass file:secretkey
~/lab_files $ ls -al
total 32
drwxr-xr-x 1 student student 4096 Apr  8 00:49 .
drwxr-xr-x 1 student student 4096 Apr  8 00:36 ..
-rw-rw-r-- 1 root root 1505 Mar 17 20:39 crapple_pay_code.txt
-rw-r--r-- 1 student student 224 Apr  8 00:49 payment_request.enc
-rw-rw-r-- 1 root root 200 Mar 17 20:27 payment_request.txt
-rw-rw-r-- 1 root root 252 Apr  7 07:01 sample_passwords.txt
-rw-r--r-- 1 student student 32 Apr  8 00:48 secret.key
~/lab_files $ cat payment_request.enc
Salted__aib$HhZ]sUQ-
00<7C )966s8Sm5CKK%
~{3VI mmgG_W J)K+ 5
Gf 2<,gB0z?WT--(
pA~/lab_files $
~/lab_files $ openssl enc -d -aes-256-cbc -salt -pbkdf2 -in payment_request.enc -out payment_request_decrypted.txt -pass file:secretkey
~/lab_files $ ls -al
total 36
drwxr-xr-x 1 student student 4096 Apr  8 00:50 .
drwxr-xr-x 1 student student 4096 Apr  8 00:36 ..
-rw-rw-r-- 1 root root 1505 Mar 17 20:39 crapple_pay_code.txt
-rw-r--r-- 1 student student 224 Apr  8 00:49 payment_request.enc
-rw-rw-r-- 1 root root 200 Mar 17 20:27 payment_request.txt
-rw-r--r-- 1 student student 200 Apr  8 00:50 payment_request_decrypted.txt
-rw-rw-r-- 1 root root 252 Apr  7 07:01 sample_passwords.txt
-rw-r--r-- 1 student student 32 Apr  8 00:48 secret.key
~/lab_files $ diff payment_request.txt payment_request_decrypted.txt

```

Figure 4: AES Channel Encryption

way function (e.g. encryption, hashing). This salt provides a unique input for the password-based key derivation function (PBKDF2), which takes the input password and the salt value and derives a secret key for use in the encryption process. Using this salt value combined with PBKDF2 helps prevent dictionary or brute-force attacks, as the secret key that is derived from the input password and salt value will be unique and therefore harder to crack.

3. Cipher block chaining (CBC) mode is a mode of operation for block ciphers. In this mode, the ciphertext from a prior block is added to the next plaintext block and fed into the input for the encryption mechanism. This effectively "chains" the blocks together as their plaintexts and ciphertexts are inherently linked. The primary drawbacks of this mode, however, are that any errors in an earlier step will propagate through to subsequent steps and the operation is difficult to parallelize. CBC mode mitigates some vulnerabilities of modes like electronic codebook (ECB) where the same plaintext input will always produce

the same ciphertext output. This can leak information about the plaintext in a known-ciphertext scenario, as an attacker can compare the ciphertexts to determine if the inputs were identical. CBC mitigates this by chaining the inputs and ciphertexts, meaning the output ciphertext blocks will not always be identical for the same given input.

- Both symmetric and public-key cryptosystems rely on secure key exchange to make encryption/decryption and digital signatures possible. Cryptographic key management involves the generation, exchange, storage, and destruction/replacement of cryptographic keys. In real-world systems that utilize Public Key Infrastructure (PKI) such as SSH and HTTPS, both public and private keys (and certificates) for all parties involved need to be securely managed. This can involve (but is certainly not limited to) file permission auditing, secure key exchange protocols, secure key storage, and integration with hardware security modules (HSMs). Secure key exchange can also be seen in communication protocols such as the Four-Way Handshake in IEEE 802.11 WPA2 to establish a secured network connection. Key management is a key pillar (pun intended) of any cryptographic security control, as it enables the shared secrets that cryptographic protocols rely on to be securely exchanged and used such that they remain secret.

Step 4: Verifying Transaction Integrity

```
~/lab_files $ sha256sum payment_request.txt
950401d29f96feebddc7d6228a51bf649eda947aa913c84adac9b795be4ae57 payment_request.txt
~/lab_files $ openssl genrsa -out private.pem 2048
~/lab_files $ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
~/lab_files $ ls -al
total 44
drwxr-xr-x 1 student student 4096 Apr  8 00:51 .
drwxr-sr-x 1 student student 4096 Apr  8 00:36 ..
-rw-rw-r-- 1 root root 1505 Mar 17 20:39 crapple_pay_code.txt
-rw-rw-r-- 1 student student 224 Apr  8 00:49 payment_request.enc
-rw-rw-r-- 1 root root 200 Mar 17 20:27 payment_request.txt
-rw-rw-r-- 1 student student 200 Apr  8 00:50 payment_request_decrypted.txt
-rw----- 1 student student 1704 Apr  8 00:51 private.pem
-rw-rw-r-- 1 student student 451 Apr  8 00:51 public.pem
-rw-rw-r-- 1 root root 252 Apr  7 07:01 sample_passwords.txt
-rw-rw-r-- 1 student student 32 Apr  8 00:48 secret.key
~/lab_files $ openssl dgst -sha256 -sign private.pem payment_request.txt | base64 > signature.sig
~/lab_files $ base64 -d signature.sig | openssl dgst -sha256 -verify public.pem -signature /dev/stdin payment_request.txt
Verified OK
```

Figure 5: Transaction Integrity Operations

- Hashing enables us to validate the integrity of a piece of data due to the properties of cryptographic hashing algorithms. A cryptographic hash provides

a unique one-way mapping of the input data. Therefore, if we want to determine if a piece of data (e.g. configuration file) has been modified, we can take an initial hash of the file and compare it to the current hash. If the hashes match, then we can be sure that the data has not changed (otherwise, the hashes would not match due to the one-way mapping). This protects the integrity of the data by providing an easy way to check if the data has been modified.

2. Digital signatures provide authenticity by ensuring that a message has actually been sent by the sender who claims to have sent it. Through public-key cryptography, a sender can sign (i.e. encrypt with a private key) a message (typically the hash of a piece of information) and send it to a verifier. The verifier can then take the signed hash and decrypt it with the sender's public key to verify it. If the signature is valid (i.e. the signed message decrypts properly) and the hash matches, the verifier can assume that the message was indeed sent by the sender and has not been modified. This guarantees the authenticity of the sender to the recipient.
3. It is more efficient to sign the hash of a message rather than the entire message, as signing just the hash accomplishes similar security goals with lower computational overhead. It is computationally expensive to encrypt/decrypt information, so depending on the size of the message it may be infeasible/unscalable to sign the entire message. We can instead compute the hash of the message, which is less computationally intensive than encrypting the entire message. This hash is also (usually) smaller than the entire message. When we send both the message and the signed hash, this allows the recipient to verify the authenticity of the message via the digital signature and the integrity of the message via the hash. Thus, it is generally more efficient to sign the hash of a message rather than signing the entire message.
4. In a digital signature, the private key of the signer is used to encrypt (sign) the message. Then, any other verifier that receives the digital signature can verify it by decrypting the signature using the signer's public key. This works because public-key cryptosystems like RSA are reciprocal (i.e. it still functions no matter what type of key is used).

Step 5: Protecting User Credentials

1. The use of salts in storing password hashes protects the hashes from being cracked via rainbow tables. A *rainbow table* is a table of precomputed hashes

```

harris gaming-pc ~/Lab02 nano salt_passwords.sh
harris gaming-pc ~/Lab02 cat salt_passwords.sh
#!/bin/sh

cat sample_passwords.txt | while read line
do
    SALT=$(openssl rand -base64 16)
    HASH=$(echo -n "$line$SALT" | openssl dgst -sha256 -binary | base64)
    echo "$SALT,$HASH" >> hashed_passwords.csv
done

harris gaming-pc ~/Lab02 nano sample_passwords.txt
harris gaming-pc ~/Lab02 chmod +x salt_passwords.sh
harris gaming-pc ~/Lab02 ./salt_passwords.sh
VkfodEgyg/cfLSCVE+PiYg==,NsQRIERaj3ly2yWanYmR6xo0nVI0Fb0texC6/MWg8-
aaYD9wRZnZA60jYs/0lNqaw==,WUDXyElc5ZqQ4L6J7RlssJ/EwFcZe2GmG/gMX/flV8Q-
8b6KhUUAxnfB556v5jnBQ==,uvRPmOYgletko9P3ZzDc1JQW0L4a1RPGx/HpugBvasU-
j3P5EwYecwZQHmVCFt5yDw==,ZA7NlnokDQ1BL/0mSjxolZXXY+2BYQ019ydsPhc6W8-
ovd0vE0hpX/hTi+f/Fx8UQ==,MKHR2uW50LK4z63GA1051bH/mh0BYy28wNirEE3T41Y-
lRju0H0HnIugFLD1Tpaecw==,Aoz6Vg6uZrGT7cpx8BTpyB3YuCheUpXDGMSNEklf2wg-
SoPrAiHcd1ZKhnnoo3Hk8Q==,SX97gtvFJ7RmutV3/bgRcrdbue1zjdh0kTrNz4JpU-
AhoP3vJF9w75vVvGEMF+7g==,rFPuNw4BuoGmKjgmDLPvEJ+AV63Bbzgf0bxbadyn+g-
wzv4l+EweELTdchTa7W69g==,ZYfzA8KL9FDsfg+gBu13GZF7vD01b5Kf501LwJ9n2E-
pj009+J2GAM6cnPD2tj01A==,AVkPZFuoBqUBhLer1/SC8jBYswRq8b9dfuA6myndmI-
YrCKI6IgaQ10sqWAF4o0A==,KVRbJvEB5lt0zAUrwXILQGGwYf+0/KocfQ0atx8daE0-
6Zs4o51GngtXN060n/ny4g==,zq72L5V6yV5P1VAH0gbVxLE3HSfr7Qlfva1hrZ4WqU-
mS1He6ASw9fvkH0nNVH3N0==,D4408qN0y82M1B086yE4PdJjWwDVOAL+dTGGrn3Bjr8-
rmZtwnC6B4mL36g17FMGbQ==,l/KdUr4gJGid4ggZ+1Nh0+JaAPnAC1Gi0VljwC0m48-
HivUCHEv9uu/AuAtrtnwWw==,q6FFAK6aqCGx7Ur7+pCz0EdK1Y012mHz5M9sAwzHeUE-
lPwA7MY6ff3ap5fLNsC0ZQ==,EZ5IBTeR7LMgu40wU8kjobRMF9RUMAmrYfL9XX0RjB0-
Ntp8IeeauQIaoPrw0M9hPA==,h5SR0h4PrIQWdCGYl3sJHFUzYlaxFJJyDdBwzyHhc-
sIa6L575zc0WByUykglsXQ==,2KYLWu3u7lrLEnZ+tk9sJfeJ70gURd7G6TqBwdh5wc-
rWujz0kChf5Cw9car9N0ng==,aIG4mT2qdQpzn7+pgm10h0TutA0Cm07IevIDVsGFM-
1cVlFRFRZEuFXqmBjth0A==,HVloy50cFnV00qv1JEZFaV0i1laH0suM6o6IlkEZTKs=

harris gaming-pc ~/Lab02 ls
CaesarCrack.py      decrypted_caesar_cipher.txt  hashed_passwords.csv  salt_passwords.sh  'Security Engineering - Lab 2.pdf'
crapple_pay_code.txt  DiffieHellman.py           Lab02Screenshots     sample_passwords.txt

```

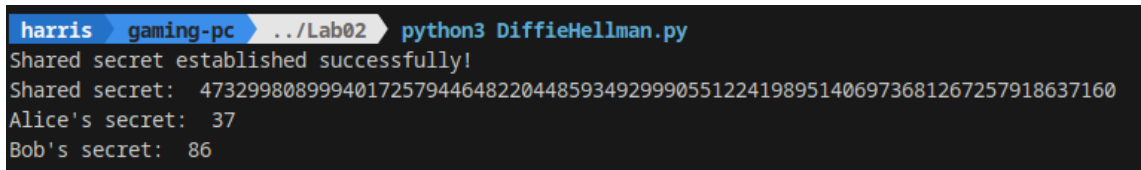
Figure 6: Protecting User Credentials

of common passwords, which can be used in a dictionary attack to attempt to find the matching hash (and by extension a matching password) for a given hash in a password table. Salting limits the effectiveness of these rainbow tables by introducing a random value that is used as an additional input into the hash function. This extra random value grows the effective size requirement of the rainbow table, making them infeasible for a suitably long salt.

2. If a password salt is reused between password hashes in a table, it negates the security benefits of adding a salt by making it easy for an attacker to generate a rainbow table. All the attacker then has to do is take into account the salt value used in the password table to re-generate the rainbow table. If instead a unique salt is used for each password, an attacker cannot simply re-generate the rainbow table as this would require a separate rainbow table for each salt.
3. SHA-256 is preferred over MD5 and SHA-1 since both of the latter hashing algorithms have been broken. For a hashing algorithm to be considered "broken," it means that the algorithm has been shown to lack collision resistance (i.e. it is computationally feasible to find two data inputs that produce the same output). Collision resistance is important for cryptographic hashing al-

gorithms since it prevents a fake piece of input data from being substituted in place of the real original data.

Step 6: Secure Key Exchange



```
harris gaming-pc ../Lab02 python3 DiffieHellman.py
Shared secret established successfully!
Shared secret: 47329980899940172579446482204485934929990551224198951406973681267257918637160
Alice's secret: 37
Bob's secret: 86
```

Figure 7: Diffie-Hellman Key Exchange

1. Diffie-Hellman Key Exchange Protocol steps:

- (a) Alice and Bob agree publicly on a shared modulus p and base (generator). The base must be a primitive root modulo p since it needs to be able to generate all possible powers within the set of integers mod p .
- (b) Alice randomly selects a secret integer a and computes $A = g^a \mod p$.
- (c) Bob randomly selects a secret integer b and computes $B = g^b \mod p$.
- (d) Alice and Bob publicly exchange their computed values A and B . They can do this without compromising their secret values a and b due to the computational hardness of the discrete log problem (especially if p is sufficiently large).
- (e) Alice computes $K_A = B^a \mod p$ and Bob computes $K_B = A^b \mod p$. We know that K_A and K_B are equivalent since $K_A = B^a = (g^b)^a = (g^a)^b = A^b = K_B \mod p$. Therefore, Alice and Bob have successfully exchanged a shared secret value K without compromising their secret values a and b .

2. As described previously, the shared secret can be derived by Alice and Bob separately since they both know a.) their own secret value and b.) the public calculated value of the other party. From these values, we are able to show that K_A and K_B are equivalent: $K_A = B^a = (g^b)^a = (g^a)^b = A^b = K_B \mod p$. Therefore, Alice and Bob can derive a common shared secret from their own secret value and the public values provided by the other party.

3. The *generator* is the exponential base agreed upon by Alice and Bob for all of the computations in the Diffie-Hellman algorithm. From an abstract algebra perspective, it is the generating element of the cyclic group modulo p . This implies that the generator must be a primitive root modulo p for it to generate the group. If the generator isn't suitably chosen (i.e. it isn't a primitive root modulo p), then the possible space of values able to be created when performing exponentiation with the secret values from Alice and Bob is reduced. This smaller space makes the discrete log problem easier, and thus makes the protocol more susceptible to cryptanalysis.
4. *Forward secrecy*, in the context of key agreement, means that prior communication sessions will not be compromised by future compromises of long-term secret keys. This is typically implemented by generating unique session keys for each communication session. This per-session key generation is facilitated by key exchange protocols like the Diffie-Hellman algorithm due to its speed and reliability.
5. The standard Diffie-Hellman algorithm is susceptible to a man-in-the-middle (MITM) attack since it does not authenticate the parties involved before performing the key exchange. If a malicious eavesdropper Eve is present and can read/write the communication channel between Alice and Bob, it is possible for Eve to have Alice and Bob establish shared secrets with herself instead of with each other. When Alice and Bob exchange their public exponents A and B , Eve can intercept them and instead send her own public exponents $C = g^c \bmod p$ to Bob and $D = g^d \bmod p$ to Alice. Alice will then compute $K_1 = g^{ad} \bmod p$ and Bob will compute $K_2 = g^{bc} \bmod p$. This means that, whenever Alice or Bob want to send encrypted information to each other using K_1 and K_2 , Eve has the ability to decrypt and re-encrypt the information as she has knowledge of A, B, C, D, p , and g . Thus, Eve has effectively established two separate keys with Alice and Bob and placed herself in the middle of their communication.

To prevent this attack, Alice and Bob could implement some kind of authentication or PKI system to ensure that their messages are being exchanged with each other and not with an eavesdropper. Timestamps/nonces could also be used to provide authenticity and non-repudiation in the key exchange. The timestamp value would help prevent older keys from prior sessions from being used in a current communication session.

Step 7: Preventing Replay Attacks

1. Timestamps and *nonces* (an arbitrary one-time use number) can be used to ensure the "freshness" of a message to prevent replay attacks. By appending a timestamp/nonce to end of a message and hashing it, the receiving party can verify that a message was sent within a specified window of time. If the message was not sent within this time window (e.g. a prior message is being replayed in a later session), then the receiver can disregard the message.
2. One way to incorporate timestamps/nonces into Crapple Pay would be to append them to the end of authentication messages or session key exchanges before hashing. This implementation would allow the recipient of the message to verify that the nonce hasn't been changed and that the message is "fresh" and can be used for authentication. Alternatively, Crapple Pay could be set up to use a Kerberos or RADIUS server for authenticating users to payment services, as both of these protocols incorporate timestamps into their authentication systems.
3. Timestamping would require time synchronization between the communicating parties, which may be difficult to do across timezones and for remote employees. If nonces are used instead of timestamps, it may be difficult to implement a secure and fault-tolerant random number generation system. This may require the use of a hardware security module (HSM) or a trusted platform module (TPM) to perform secure hardware random number generation (HRNG). A hybrid scheme of timestamping and nonces could be implemented, where timestamps are used for authentication and nonces are used for key exchange, but this inherits both the time synchronization and random number generation challenges.