Lab 2 – Securing Crapple Pay – Building a Trustworthy Transaction System

Employee: You (New Hire)
Supervisor: Seymour Bugs (Security Guru)

Introduction (Seymour Bugs):

Alright, newbie, welcome to week 2! This week, we're tackling something near and dear to Crapple's heart (and its bank account): Crapple Pay. It's the future of finance, the pinnacle of payment processing… and a juicy target for every digital bandit out there. Your mission, should you choose to accept it, is to bolster the security of Crapple Pay. We're talking encryption, hashing, digital signatures – the whole nine yards. We need to make sure every transaction is secure, authentic, and tamper-proof. Get ready to build a fortress around our digital dollars!

Your Mission (Lab 2):

You will design and implement security measures for a simplified version of Crapple Pay. This involves choosing and applying appropriate cryptographic techniques to ensure the confidentiality, integrity, and authenticity of transactions.

Step 1: Get Your Crypto Toolkit (Docker Image):

I've prepared a special package for you: a Docker image pre-loaded with a lean, mean Linux distribution. Think of it as your own personal sandbox (where you can play with fire, within reason).

1. Open your terminal (or command prompt, if you're still living in the dark ages).
2. Type the following incantation: `docker pull dlambros/labs:lab2`. This downloads the image to your machine. Don't worry, it won't bite... unless you try to run it on a potato.
3. You can verify your image is downloaded by running `docker images`
4. Type: `docker run -it dlambros/labs:lab2`

Step 2: Cracking the Crapple Pay Code:

We intercepted a suspicious message related to Crapple Pay. We suspect it's encrypted using a simple Caesar cipher. Your first task is to crack the code!

Inside the container, you'll find a file named `crapple_pay_code.txt`. It's encrypted using a Caesar cipher. Use a Python script to perform frequency analysis on the ciphertext and decrypt the message.

Analysis

1. Explain how frequency analysis works and why it's effective against Caesar ciphers.

2. Compare the letter frequencies in the ciphertext to the known frequencies of letters in English text (you can find these online).

3. Discuss the limitations of the Caesar cipher and why it's not secure for real-world applications. *(Include the Python script, the decrypted message, and a graph comparing the frequency distributions in your report)*

Step 3: Securing the Transaction Channel:

A Crapple Pay user wants to send a payment request. This request contains sensitive information that must be protected during transmission. We'll use AES symmetric encryption to secure the channel.

1. Use `cat payment_request.txt` to view the contents of the payment request.
2. Use `openssl rand -out secret.key 32` to generate a 256-bit random key and save it to `secret.key`.
3. Encrypt `payment_request.txt` using AES-256 in CBC mode with a salt:

   ```
   openssl enc -aes-256-cbc -salt -pbkdf2 -in payment_request.txt -out payment_request.enc -pass
   file:secret.key
   ```

4. Verifying encryption. Use `cat payment_request.enc` to view the encrypted file. It should appear as binary gibberish.
5. Decrypt the file

   ```
   openssl enc -d -aes-256-cbc -salt -pbkdf2 -in payment_request.enc -out
   payment_request_decrypted.txt -pass file:secret.key
   ```

6. Compare with original

   ```
   diff payment_request.txt payment_request_decrypted.txt
   ```

Analysis

1. Explain why symmetric encryption is suitable for securing the transaction channel (speed, efficiency, etc).

2. Discuss the importance of the salt and how it enhances security.

3. Explain the role of the CBC mode and how it prevents certain attacks.

4. Emphasize the importance of secure key management in real-world applications.

Step 4: Verifying Transaction Integrity:

Crapple Pay needs to ensure that the payment request hasn't been tampered with and verify the request's origin.

1. Calculate the SHA-256 hash of the `payment_request.txt` file.
2. Use `openssl genrsa -out private.pem 2048` to generate a 2048-bit RSA private key and save it to `private.pem`.
3. Use `openssl rsa -in private.pem -out public.pem -outform PEM -pubout` to generate the corresponding public key and save it to `public.pem`.
4. Use the following `openssl` command to sign the hash of the payment request using the private key:

   ```
   openssl dgst -sha256 -sign private.pem payment_request.txt | base64 > signature.sig
   ```

5. Verify the signature using the public key:

   ```
   base64 -d signature.sig | openssl dgst -sha256 -verify public.pem -signature /dev/stdin
   payment_request.txt
   ```

Analysis:

1. Explain how hashing provides integrity.
2. Explain how digital signatures provide authenticity.
3. Explain why signing the hash is more efficient and secure than signing the entire message.
4. Discuss the role of the private and public keys in digital signatures.

Step 5: Protecting User Credentials

Crapple Pay stores user passwords (hashed). We need to secure these hashes. Use openssl to generate SHA-256 hashes of sample passwords seen in file `sample_passwords.txt`.

1. View the contents of sample passwords file.
2. For each password, generate a unique random salt. A salt is a random string that is concatenated with the password before hashing. Use `openssl rand -base64 16` to generate a 16-byte (128-bit) salt in Base64 encoding.
3. Concatenate each password with its corresponding salt. Hash the concatenated string using SHA-256. Use `openssl dgst -sha256 -binary | base64` to hash the string and encode the binary output in Base64. Store the salt and the hashed password in a secure format (.csv).

Example hashing

```
SALT=$(openssl rand -base64 16)
HASH=$(echo -n "password123$SALT" | openssl dgst -sha256 -binary | base64)
echo "$SALT,$HASH" >> hashed_passwords.csv
```

Analysis:

1. Explain why salting is crucial for password security. How does it protect against rainbow table attacks?

2. Why is it essential to use a unique salt for each password?

3. Explain why SHA-256 is preferred over MD5 and SHA-1 for password hashing. Discess the weaknesses of MD5 and SHA-1.

Step 6: Secure Key Exchange

Before a user can securely communicate with Crapple Pay, they need to establish a shared secret key over an insecure channel. We will simulate the Diffie-Hellman key exchange process.

Write a Python script with

1. A large prime number stored as a variable
2. Two secret values:
    a. **a** (Alice's secret): Choose a small random odd integer between 3 and 101 (inclusive).
    b. **b** (Bob's secret): Choose a small random even integer between 2 and 100 (inclusive).
3. A generator **g**. For simplicity, you can use a value **g=2**.
4. Perform modular exponentiation
    a. $A = g^a \bmod p$
    b. $B = g^b \bmod p$
5. Calculate shared secret:
    a. $s = B^a \bmod p$
    b. $s = A^b \bmod p$
6. Verify both secrets are the same

Analysis

1. Explain each step of the Diffie-Hellman key exchange.

2. Explain how the shared secret is derived.

3. What is the role of the generator and why is it important to choose a suitable generator?

4. Explain how Diffie-Hellman contributes to forward secrecy.

5. Discuss how Diffie-Hellman is vulnerable to a man-in-the-middle attack. What additional security measures are needed to prevent this attack?

Step 7: Preventing Replay Attacks

Discuss how timestamps and nonces can prevent replay attacks.

1. Explain how timestamps and nonces can be used.

2. Discuss how you would incorporate them into Crapple Pay.

3. Discuss the challenges of implementing these mechanisms.

Step 8: Report Time (aka "Impress Seymour"):

Now, put it all together in a professional-looking report. Remember, presentation matters! Briefly explain the purpose of the lab. Describe the steps you took in the lab. Explain why you used them. Answer the analysis questions thoughtfully summarizing the key takeaways.