DA563A

Introduction to Computer Security

Project 1: A Toy Cryptosystem

Zandra Hedlund

VT-24

Table of Contents

Introduction

A symmetric cryptosystem is a type of encryption scheme where the same key is used for both encryption and decryption of data. This means that both the sender and the receiver must have access to the same secret key in order to communicate securely. Popular symmetric encryption algorithms include AES (Advanced Encryption Standard), DES (Data Encryption Standard), and 3DES (Triple DES).

How symmetric cryptosystems typically works

1. **Key Generation**. The first step is to generate a secret key. This key should be known only to the sender and the intended recipient. The key should be kept confidential and securely transmitted between the parties.
2. **Encryption**. To encrypt a message or data, the sender applies an encryption algorithm along with the secret key. This algorithm transforms the plaintext (original message) into ciphertext (encrypted message). The resulting ciphertext can only be decrypted back into plaintext using the same secret key.
3. **Decryption**. The recipient, who possesses the same secret key, uses the decryption algorithm along with the secret key to transform the received ciphertext back into plaintext. This allows them to recover the original message.

Advantages of symmetric cryptosystems

- **Efficiency**. Symmetric encryption is generally faster and more efficient compared to asymmetric encryption (where different keys are used for encryption and decryption).
- **Scalability**. Symmetric encryption is well-suited for encrypting large amounts of data efficiently.
- **Security**. When implemented properly and with a strong key, symmetric encryption can provide a high level of security.

Challenges with symmetric cryptosystems

- **Key Distribution**. Since the same key is used for both encryption and decryption, securely distributing the key to all parties involved can be a challenge, especially in large-scale systems.
- **Key Management**. Managing and securely storing the secret keys is crucial for maintaining the security of the system. If a key is compromised, all communications encrypted with that key become vulnerable.
- **Limited to Two Parties**. Symmetric encryption is typically limited to scenarios where only two parties need to communicate securely. Establishing secure communication among multiple parties often requires more complex protocols.

Despite these challenges, symmetric cryptosystems remain widely used in various applications, ranging from securing communication channels to protecting data stored on devices.

Program

I've written a simple Python program that implements a toy symmetric cryptosystem and a brute-force decryption attack. This program serves as a demonstration of the vulnerability of the symmetric cryptosystem to brute-force attacks and highlights the importance of using strong encryption algorithms and key management practices in real-world scenarios.

Toy symmetric cryptosystem

- Keys are 16-bit values.
- Messages are strings with an even number of characters, where each character is an 8-bit value using ASCII coding. A blank space (ASCII 32) is added at the end of an odd-length string.
- The encryption of a message *M* of length *n* (in bytes) is given by the following equation, where the key *K* is repeated *n*/2 times:

$$E_K(M) = M \oplus (K \| K \| \cdots)$$

- The decryption algorithm for a ciphertext *C* is the same as the encryption algorithm:

$$D_K(C) = C \oplus (K \parallel K \parallel \cdots)$$

Test cases and English detection

- The program is tested on messages that are automatically generated, by randomly selecting *n* words from a text file with English words. A new message is generated every time the program runs.
- The text file with English words, named "english_words.txt", is a copy of the text file that is usually found at /usr/share/dict/words, in UNIX-like systems.
- Since the test cases are generated from the text file, we already know that they are in English, but to include a language check, the decrypted message is checked for English ASCII characters.

Running the program and output

It's a simple program that doesn't have a graphical user interface. The number of words in the message is hard coded into the main function and to run the program, you have to use the terminal or run it through an IDE, like VS code.

When running the program, it encrypts the message and decrypts it using the original decryption method. It also attempts a brute-force decryption attack on the encrypted message. The brute-force decrypted message is then compared to the message that was decrypted using the original decryption method.

The program prints the original message, the encrypted message, the decrypted message using the original method, and the decrypted message obtained from the brute-force attack, along with the brute-force decryption key. It also prints a statement saying whether the brute-force decryption attempt was successful or not.

Brute-force decryption attack

The brute-force decryption attack is performed by trying all possible keys (2^16) to decrypt the given ciphertext. The attacks are not 100 % successful with the current implementation, but the message is decrypted successfully >90 % of the times the program runs. The brute-force function is very simple and, with a few adjustments, it would probably be possible to get a 100 % success rate.

Functions

1. **generate_random_key()**. This function generates a random 16-bit key for encryption.
2. **pad_message().** This function pads the input message with a blank character if its length is odd. This ensures that the message length is even, as required by the encryption algorithm.
3. **encrypt().** This function encrypts a message using the toy symmetric cryptosystem. It XORs each character of the message with a corresponding character from a repeated key derived from the generated key.
4. **decrypt()**. This function decrypts a ciphertext using the toy symmetric cryptosystem. It performs the reverse operation of encryption by XORing each character of the ciphertext with the repeated key derived from the encryption key.
5. **generate_random_english_message()**. This function generates a random English text message by selecting a specified number of words randomly from a word list. These words are joined together as a string.
6. **brute_force_decrypt()**. This function attempts a brute-force decryption attack by trying all possible keys (2^16) to decrypt the given ciphertext. It decrypts the ciphertext using each key and checks if the decrypted message appears to be in English, using the is_english() function.
7. **is_english()** is a function that determines whether a given text appears to be English by checking if all characters belong to a set of English characters (letters, digits, punctuation marks, and space).

8. **main()** is the entry point of the program. It reads a word list from the file "english_words.txt", and generates a random English text message using the generate_random_english_message() function. It encrypts using encrypt(), decrypts using decrypt() and brute_force_decrypt() and prints the output.

Program code

```
#!/usr/bin/env python3
import random
import string

# Function to generate a random 16-bit key
def generate_random_key():
    return random.randint(0, 2**16 - 1)

# Function to pad a message with a blank character if its length is odd
def pad_message(message):
    if len(message) % 2 != 0:
        message += " "
    return message

# Function to encrypt a message using the toy symmetric cryptosystem
def encrypt(message, key):
    repeated_key = (key.to_bytes(2, 'big') * (len(message) // 2))
    ciphertext = ""
    for char, key_char in zip(message, repeated_key):
        encrypted_char = chr(ord(char) ^ key_char)
        ciphertext += encrypted_char
    return ciphertext

# Function to decrypt a ciphertext using the toy symmetric cryptosystem
def decrypt(ciphertext, key):
    repeated_key = (key.to_bytes(2, 'big') * (len(ciphertext) // 2))
    decrypted = ""
    for char, key_char in zip(ciphertext, repeated_key):
        decrypted_char = chr(ord(char) ^ key_char)
        decrypted += decrypted_char
    return decrypted
```

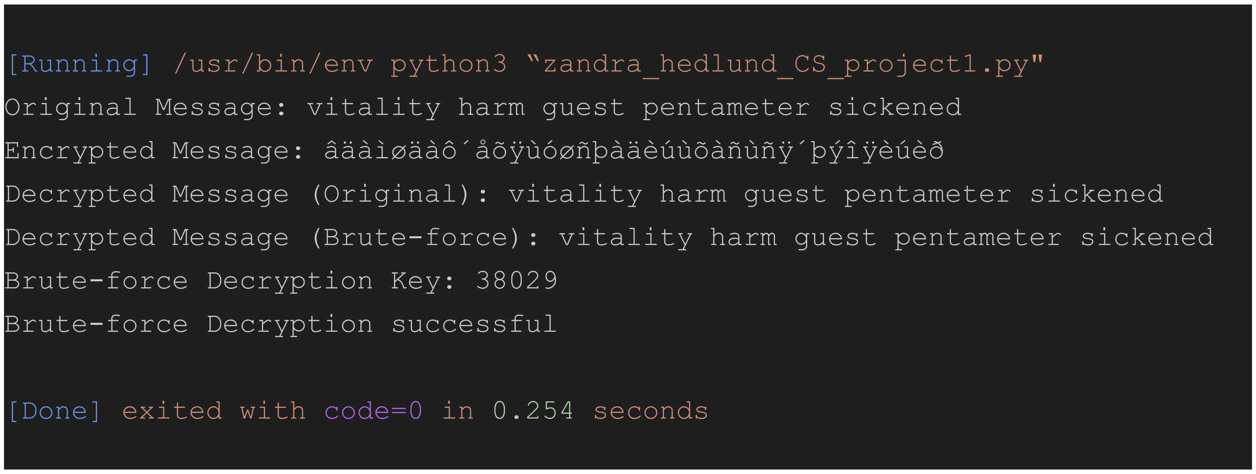
```python
# Function to generate random English text messages
def generate_random_english_message(word_list, num_words):
    filtered_words = [word for word in word_list if word.isalpha()]
    selected_words = random.sample(filtered_words, num_words)
    return ' '.join(selected_words)

# Brute-force decryption attack
def brute_force_decrypt(ciphertext):
    for key in range(2**16):
        decrypted = decrypt(ciphertext, key)
        if is_english(decrypted):
            return decrypted, key  # Return the decrypted message and the
key
    return None, None  # Return None if decryption fails

# Function to determine if text is English
def is_english(text):
    english_chars =
set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ")
    for char in text:
        if char not in english_chars:
            return False
    return True

# Test the cryptosystem
def main():
    with open("english_words.txt", "r") as f:
        word_list = [word.strip() for word in f]

    message = generate_random_english_message(word_list, 5)  # Generate a
random English text message
    message = pad_message(message)
    key = generate_random_key()
    ciphertext = encrypt(message, key)
    print("Original Message:", message)
    print("Encrypted Message:", ciphertext)
```

```
# Decrypt using the original decryption method
   decrypted = decrypt(ciphertext, key)
   print("Decrypted Message (Original):", decrypted)

   # Attempt brute-force decryption attack
   brute_decrypted, brute_key = brute_force_decrypt(ciphertext)
   print("Decrypted Message (Brute-force):", brute_decrypted)
   print("Brute-force Decryption Key:", brute_key)
   if decrypted.lower() == brute_decrypted.lower():
       print("Brute-force Decryption successful")
   else:
       print("Brute-force Decryption failed")

if __name__ == "__main__":
   main()
```

Example output

This example shows the output when generating a message with five words. The first example shows a brute-force attack that failed to decrypt a message and one attack that was successful.

```
[Running] /usr/bin/env python3 “zandra_hedlund_CS_project1.py"
Original Message: scalpel goldenrod Tbilisi loped sophistry
Encrypted Message: ñÎãÁòÈîåÂîÉçÃðÂæÖÏëÁëÞëîÂòÈæñÂòÅëÞößû
Decrypted Message (Original): scalpel goldenrod Tbilisi loped sophistry
Decrypted Message (Brute-force): pcblseo doodfnqog Wbjljsj ooseg poshjswrz
Brute-force Decryption Key: 33197
Brute-force Decryption failed

[Done] exited with code=0 in 0.497 seconds
```



```
[Running] /usr/bin/env python3 "zandra_hedlund_CS_project1.py"
Original Message: vitality harm guest pentameter sickened
Encrypted Message: âäàìøäàô´åõÿùóøñþàäèúùõàñùñÿ´þýîÿèúèð
Decrypted Message (Original): vitality harm guest pentameter sickened
Decrypted Message (Brute-force): vitality harm guest pentameter sickened
Brute-force Decryption Key: 38029
Brute-force Decryption successful

[Done] exited with code=0 in 0.254 seconds
```

Discussion

Even for me, as an inexperienced programmer, it was fairly easy to implement a successful brute-force decryption attack on this system. It only takes a few lines of code. This clearly shows the vulnerability of a symmetric cryptosystem and why it's not a secure option for encrypting important data in real-world scenarios.

A 16-bit key symmetric cryptosystem suffers from limitations due to its small key size. This leads to vulnerabilities and compromises in security. Firstly, the key space is limited to 65,536 possible keys, which makes it susceptible to brute-force attacks. This means an adversary can feasibly try all possible keys to decrypt the ciphertext.

To enhance the security of the cryptosystem, it's essential to address the limitations imposed by the small key size. One approach is to increase the key size significantly. By using longer keys, such as 128 bits or higher, the key space expands exponentially, making brute-force attacks impractical.

Additionally, employing stronger cryptographic algorithms and protocols can increase security. Modern symmetric encryption algorithms like AES (Advanced Encryption Standard) offer robust security features and larger key sizes, making them more resistant to attacks.

Furthermore, implementing additional security measures, such as key management practices and secure communication protocols, can further enhance the overall security posture of the cryptosystem.

Conclusion

While a 16-bit key symmetric cryptosystem offers simplicity and efficiency, its inherent limitations in key size pose significant security risks, including vulnerability to brute-force and cryptanalysis attacks. To address these challenges and enhance security, transitioning to longer key sizes, adopting stronger cryptographic algorithms, and implementing robust security measures are crucial steps. By prioritizing these enhancements, organizations can better safeguard sensitive data and mitigate potential security threats in their communication and data storage systems.

References

- M. Goodrich, R. Tamassia. *Computer Security*, 1st edition.
- W. Stallings, L. Brown. *Computer Security: principles and practice*, 3rd edition.