CWE-327: Use of a Broken or Risky Cryptographic Algorithm

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL

Name: SHUMBUL ARIFA - 181CO152

Course: NETWORK SECURITY (CS463)

Submitted to: Mahendra Pratap Singh

Introduction

We live so much of our lives today on the internet. Whether it's for storing our personal information, finding entertainment, making purchases, or doing our jobs, our society relies increasingly on an online presence. This increased dependence on the internet means that information security is more important than ever. Users need to know that their sensitive data is kept confidential, unmodified, and readily available to authorized readers. Through the years we've used many different kinds of encryption algorithms to be able to protect our data, and as the years have gone on some of these algorithms have become easier and easier to crack, whereas others have maintained their strength.

Regardless of the algorithm that's used to encrypt data, everything is always subject to brute force checking. If somebody's trying every possible key to try to find a way to access data, they can certainly go through the brute force method. But, of course, keys can be made so large that it would be functionally impossible to be able to try every possible key in the limited amount of time that we have.

Encryption algorithms like PGP, AES, etc. are considered strong, whereas encryption algorithms like DES (56-bit keys), and SHA-1 (160-bit keys) are considered to be weak. There are ways to create stronger keys using weak keys by performing multiple processes on the same key. For instance, one could take and hash a password, and then hash the hash of that password, and so on. This is called key stretching, or key strengthening, and the only way one could then brute force that particular key is to reverse each one of those hashes to get back to the original key.

Use of a Broken or Risky Cryptographic Algorithm

Using broken or weak cryptographic algorithms can leave data vulnerable to being decrypted. Many cryptographic algorithms provided by cryptography libraries are known to be weak, or flawed. Using such an algorithm means that an attacker may be able to easily decrypt the encrypted data. It may result in the disclosure of sensitive information. The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected.

Encryption algorithms such as DES and hashing algorithms such as SHA-1 are considered to be weak. The Data Encryption Standard's (DES) 56-bit key is no longer considered adequate in the face of modern cryptanalytic techniques and supercomputing power.

SHA-1 (Secure Hash Algorithm 1) is a cryptographically broken but still widely used hash function that takes an input and produces a 160-bit (20-byte) hash value known as a message digest – typically rendered as a hexadecimal number, 40 digits long. Since 2005, SHA-1 has not been considered secure against well-funded opponents.

Weak encryption algorithms and hashing functions are used today for a number of reasons, but they should not be used to guarantee the confidentiality of the data they protect.

Consequences of using broken/risky cryptographic algorithms

1. **Confidentiality:** The confidentiality of sensitive data may be compromised by the use of a broken or risky cryptographic algorithm.
2. **Integrity:** The integrity of sensitive data may be compromised by the use of a broken or risky cryptographic algorithm.
3. **Accountability:** Any accountability to message content preserved by cryptography may be subject to attack.

Technical Impact: Broken Cryptography can result in unauthorised access and retrieval of sensitive information of the users.

Business Impact: Broken Cryptography can cause a number of business impacts, like:

- Privacy Violations
- Information Theft
- Code Theft
- Intellectual Property Theft
- Reputational Damage

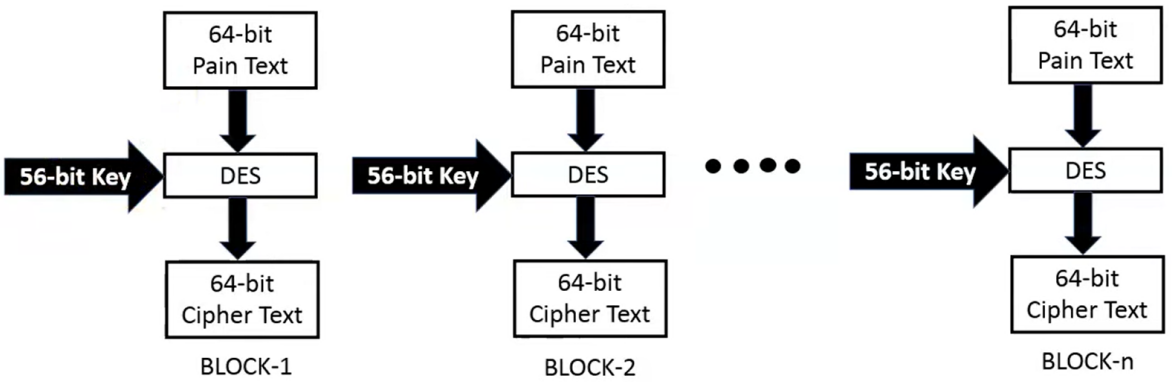
Data Encryption Standard (DES)

DES is a symmetric block cipher developed by IBM in the early 1970s. It is based on the Feistel block cipher. The algorithm uses a 56-bit key to encipher/decipher a 64-bit block of data. The key is always presented as a 64-bit block, every 8th bit of which is ignored. However, it is usual to set each 8th bit so that each group of 8 bits has an odd number of bits set to 1.

DES is the most widely used symmetric algorithm in the world, despite claims that the key length is too short. Ever since DES was first announced, controversy has raged about whether 56 bits is long enough to guarantee security. As the key length is 56 bits, there are 2^{56} possible keys. Trying all 2^{56} possible keys is not hard these days, it just takes few hours to crack DES using todays computers. AES is a replacement to DES.

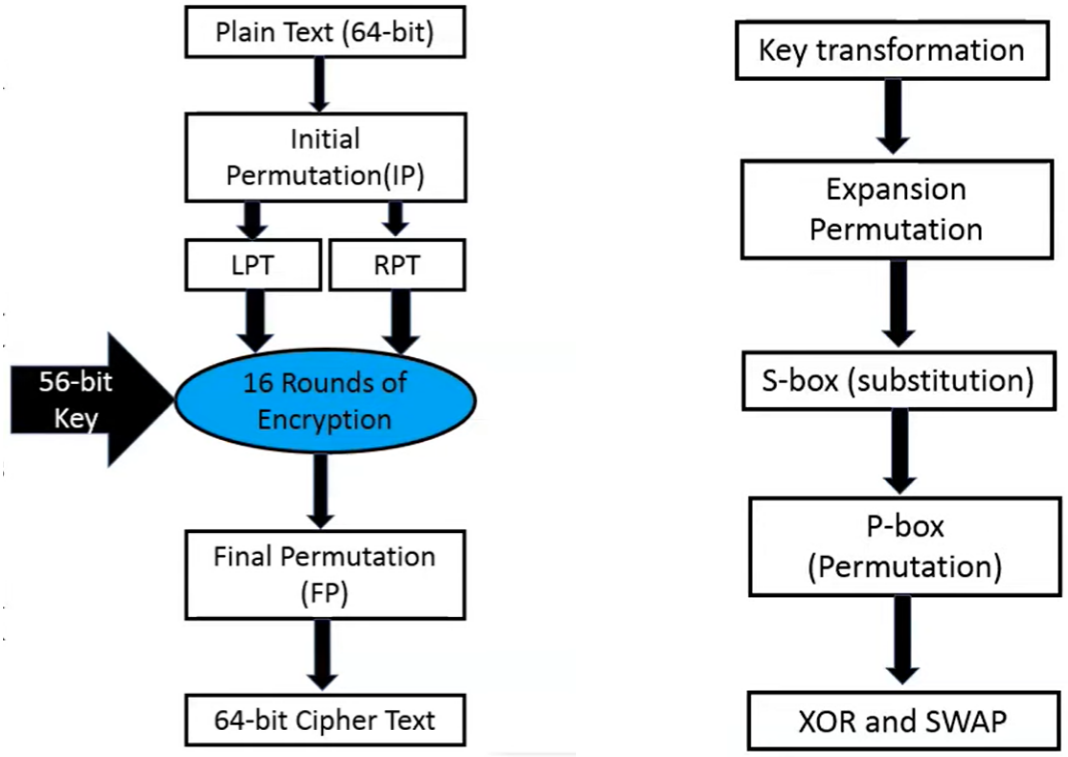
DES PROCESS

64-bit plaintext and 56-bit key is passed to DES as inputs. DES outputs the encrypted 64-bit ciphertext. Originally we use a 64-bit key, and discard every 8th bit of the original key.



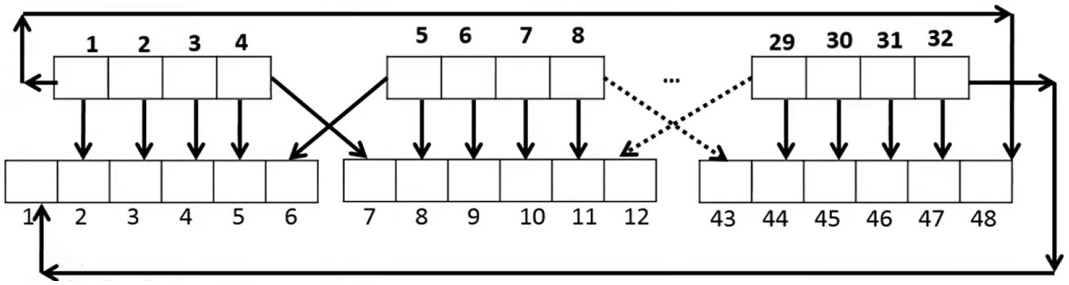
STEPS OF DES ALGORITHM

1. The process begins with the 64-bit plain text block getting handed over to an initial permutation (IP) function.
2. The initial permutation (IP) is then performed on the plain text.
3. Next, the initial permutation (IP) creates two halves of the permuted block, referred to as Left Plain Text (LPT) and Right Plain Text (RPT).
4. Each LPT and RPT goes through 16 rounds of the encryption process.
5. Finally, the LPT and RPT are rejoined, and a Final Permutation (FP) is performed on the newly combined block.
6. The result of this process produces the desired 64-bit ciphertext.

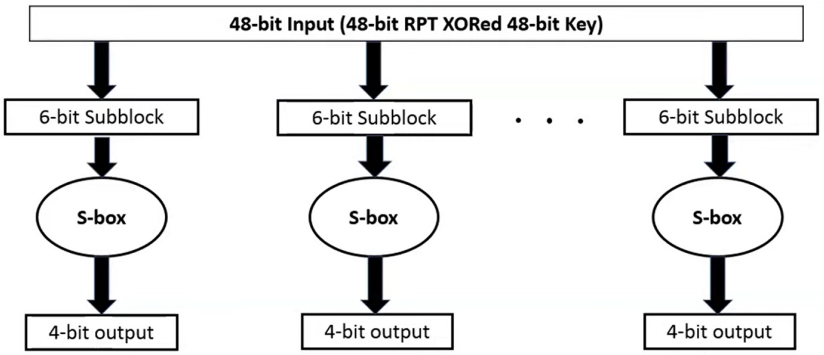


Steps involved in each round of encryption

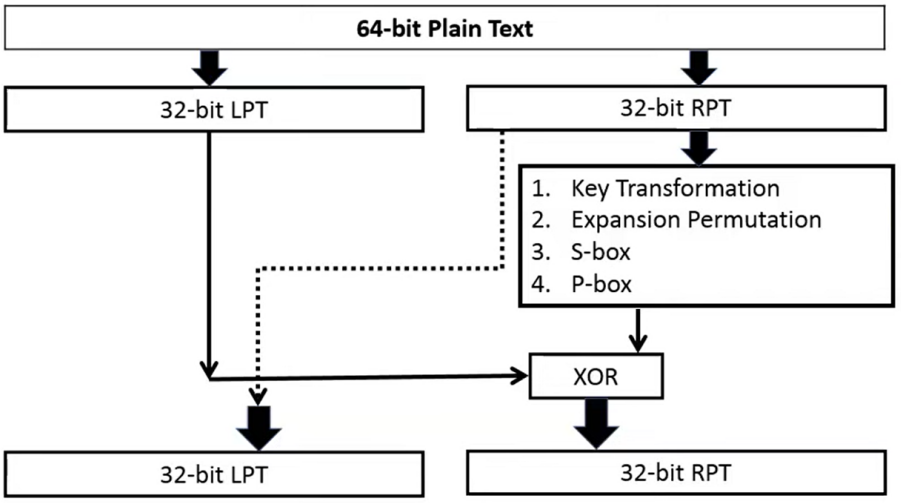
Expansion Permutation



Working of S-box:



XOR and SWAP



DES code in Python (Click here for the complete code)

1. 64-bit plain text and 64-bit key is inputted

```
pt = "123456ABCD132536"

key = "AABB09182736CCDD"


# Key generation

# --hex to binary

key = hex2bin(key)

print("Key: ", key, "\nSize of key (in bits): ", len(key))
```

Output:

Key: 1010101010111011000010010001100000100111001101101100110011011101

Size of key (in bits): 64

2. 64-bit key is compressed to 56-bit key by discarding every 8th bit of the key

```
# getting 56 bit key from 64 bit using the parity bits

key = permute(key, keyp, 56)

print("Key: ", key, "\nSize of key (in bits): ", len(key))
```

Output:

Key: 11000011110000000011001110100011001111110000110011111010

Size of key (in bits): 56

3. Splitting keys in 2 parts, and storing keys for each of the 16 rounds of encryption.

```
# Splitting
left = key[0:28] # rkb for RoundKeys in binary
right = key[28:56] # rk for RoundKeys in hexadecimal


rkb = []
rk = []
for i in range(0, 16):
  # Shifting the bits by nth shifts by checking from shift table
  left = shift_left(left, shift_table[i])
  right = shift_left(right, shift_table[i])


  # Combination of left and right string
  combine_str = left + right


  # Compression of key from 56 to 48 bits
  round_key = permute(combine_str, key_comp, 48)


  rkb.append(round_key)
  rk.append(bin2hex(round_key))
```

4. Encryption

```
# Encrypt function
def encrypt(pt, rkb, rk):
  pt = hex2bin(pt)


  # Initial Permutation
```

```
  pt = permute(pt, initial_perm, 64)
  print("After initial permutation", bin2hex(pt))

  # Splitting
  left = pt[0:32]
  right = pt[32:64]
  for i in range(0, 16):
    # Expansion D-box: Expanding the 32 bits data into 48 bits
    right_expanded = permute(right, exp_d, 48)

    # XOR RoundKey[i] and right_expanded
    xor_x = xor(right_expanded, rkb[i])

     # S-boxes: substituting the value from s-box table by calculating
row and column
    sbox_str = ""
    for j in range(0, 8):
      row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
       col = bin2dec(int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j *
6 + 3] + xor_x[j * 6 + 4]))
      val = sbox[j][row][col]
      sbox_str = sbox_str + dec2bin(val)

    # Straight D-box: After substituting rearranging the bits
    sbox_str = permute(sbox_str, per, 32)

    # XOR left and sbox_str
    result = xor(left, sbox_str)
    left = result
```

```
    # Swapper
    if(i != 15):
      left, right = right, left
     print("Round ", i + 1, " ", bin2hex(left), " ", bin2hex(right), "
", rk[i])

  # Combination
  combine = left + right

  # Final permutation: final rearranging of bits to get cipher text
  cipher_text = permute(combine, final_perm, 64)
  return cipher_text

print("Encryption")
cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ",cipher_text)
```

Output:

Encryption

After initial permutation 14A7D67818CA18AD

Round 1 18CA18AD 5A78E394 194CD072DE8C

Round 2 5A78E394 4A1210F6 4568581ABCCE

Round 3 4A1210F6 B8089591 06EDA4ACF5B5

Round 4 B8089591 236779C2 DA2D032B6EE3

Round 5 236779C2 A15A4B87 69A629FEC913

Round 6 A15A4B87 2E8F9C65 C1948E87475E

Round 7 2E8F9C65 A9FC20A3 708AD2DDB3C0

Round 8 A9FC20A3 308BEE97 34F822F0C66D

Round 9 308BEE97 10AF9D37 84BB4473DCCC

Round 10 10AF9D37 6CA6CB20 02765708B5BF

Round 11 6CA6CB20 FF3C485F 6D5560AF7CA5

Round 12 FF3C485F 22A5963B C2C1E96A4BF3

Round 13 22A5963B 387CCDAA 99C31397C91F

Round 14 387CCDAA BD2DD2AB 251B8BC717D0

Round 15 BD2DD2AB CF26B472 3330C5D9A36D

Round 16 19BA9212 CF26B472 181C5D75C66D

Cipher Text : C0B7A8D05F3A829C

5. Decryption:

```
print("Decryption")

rkb_rev = rkb[::-1]

rk_rev = rk[::-1]

text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))

print("Plain Text : ",text)
```

Output:

Decryption

After initial permutation 19BA9212CF26B472

Round 1 CF26B472 BD2DD2AB 181C5D75C66D

Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D

Round 3 387CCDAA 22A5963B 251B8BC717D0

Round 4 22A5963B FF3C485F 99C31397C91F

Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3

Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5

Round 7 10AF9D37 308BEE97 02765708B5BF

Round 8 308BEE97 A9FC20A3 84BB4473DCCC

Round 9 A9FC20A3 2E8F9C65 34F822F0C66D

Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0

Round 11 A15A4B87 236779C2 C1948E87475E

Round 12 236779C2 B8089591 69A629FEC913

Round 13 B8089591 4A1210F6 DA2D032B6EE3

Round 14 4A1210F6 5A78E394 06EDA4ACF5B5

Round 15 5A78E394 18CA18AD 4568581ABCCE

Round 16 14A7D678 18CA18AD 194CD072DE8C

Plain Text : 123456ABCD132536

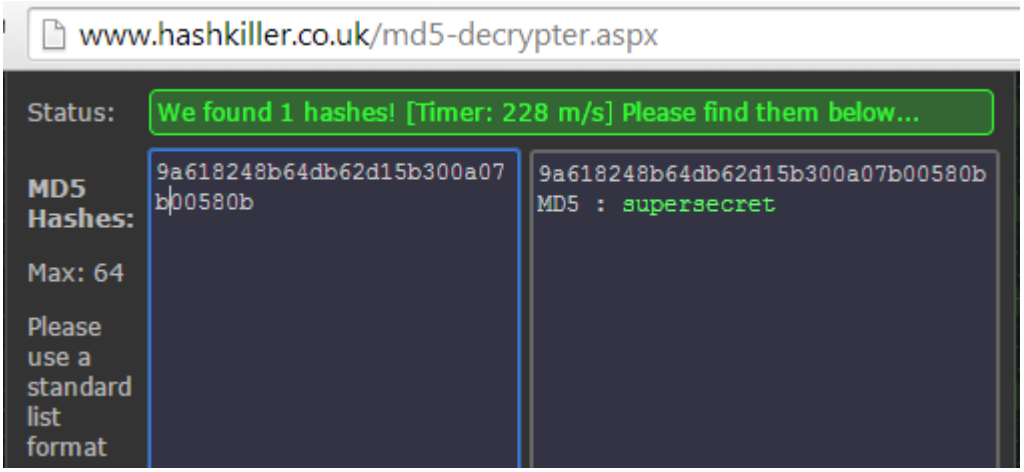
SHA-1

A 160-bit hash is generated by SHA-1. Every message is hashed down to a 160-bit integer in this way. There are an infinite number of collisions because there are an infinite number of messages that hash to each possible value. However, as there are so many possible hashes, the odds of finding one by chance are extremely small (one in 280, to be exact). One would find one pair that hashed to the same value if you hashed 280 random messages. This is the brute-force method of detecting collisions, which is entirely dependent on the length of the hash value. Breaking the hash function allows you to find collisions much faster.

Three Chinese cryptographers demonstrated that SHA-1 is not collision-free in a study published by Bruce Schneier on Security in 2005. That is, they discovered a method for detecting collisions that is faster than brute force. They were able to find collisions in SHA-1 in 2^{69} calculations, about 2,000 times faster than brute force.

The SHAttered attack was announced in 2007 by the Centrum Wiskunde & Informatica (CWI) and Google, in which they created two different PDF files with the identical SHA-1 hash in around $2^{63.1}$ SHA-1 evaluations. This attack is about 100,000 times faster than brute-forcing an SHA-1 collision. Although, the equivalent processing power of 6,500 years of single-CPU computations and 110 years of single-GPU computations was required for the attack.

Now a days, cracking the hashes involved in cryptographic algorithms like SHA-1 and MD5 has become much more easier. An attacker can use an online service to perform a rainbow table attack as shown below.



As we can see in the above figure, the hash is cracked.

Solution

Attempting to create non-standard and non-tested algorithms, using weak algorithms, or applying algorithms incorrectly will pose a high weakness to data that is meant to be secure. Use cryptographically stronger options:

- Instead of DES or TripleDES encryption, use AES encryption.
- Instead of SHA1 or RIPEMD160 hashing functions, use the ones in the SHA-2 family (e.g. SHA512, SHA384, SHA256)