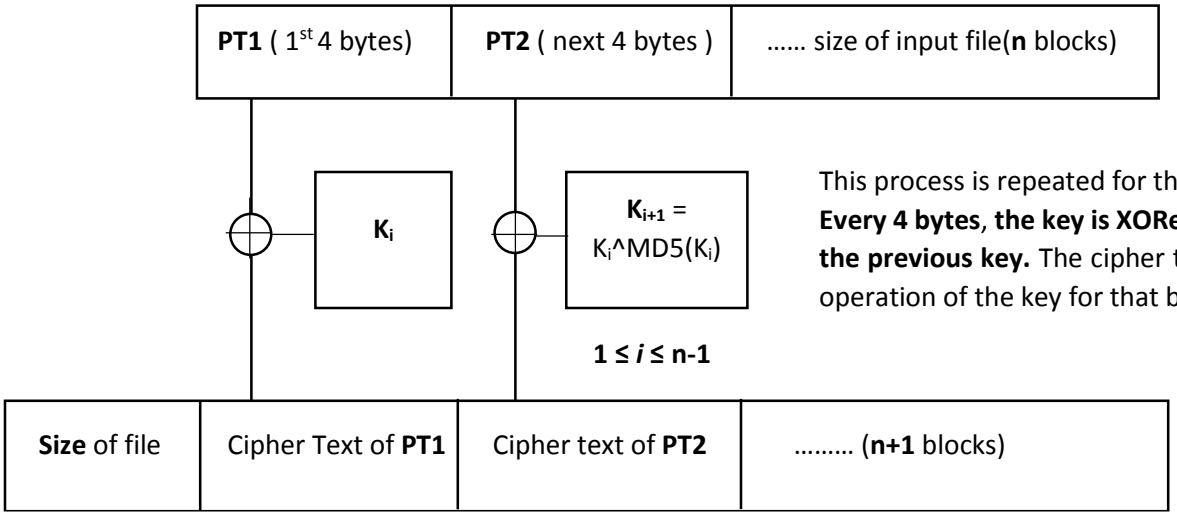
CSE 539: HOMEWORK 2 (due 2/18/16) **ANAND GANESH – 1209408847**

KARTHIK SRINIVAASAN AYENGAR DEVANATHAN- 1209320095

1. The encryption algorithm is diagrammatically depicted below:



This process is repeated for the length of the input file. **Every 4 bytes**, **the key is XORed with the MD5-Hash of the previous key.** The cipher text block is then an XOR operation of the key for that block and the plain text.

……The cipher text is padded and then has an extra 4 bytes (size)

The **size** of the input file is stored as the **first 4 bytes** of the cipher text and is not encrypted. The **second 4 bytes** of the cipher text contains the encrypted text corresponding to the **first 4 bytes of the plain text**. The **third 4 bytes** of the cipher text, of the **second 4 bytes of plain text** and so on.

DOES THE ALGORITHM CHEAT?

An encryption algorithm is supposed to ensure that the cipher text, even if stolen by an attacker who has knowledge of how the algorithm works, should make the key computation infeasible. This is **NOT** the case with the above algorithm as it has loopholes which make this very much feasible. It **CHEATS** because the weakness is not very apparent (cipher text is not readable) and despite the fact that it uses **MD5** to induce confusion and randomness, an attacker can exploit the loophole to decrypt information easily.

2. The common headers present in the file types **.png** and **.pdf** would serve as key dictionary words (%PDF-1.x and %PNG) that can be searched for, in the beginning of the decrypted cipher texts. The key that returns these values as the plain text would be the secret key.

- **a. PDF file key**: 0x1739d398
- **b. PNG file key**: 0xc9034bf4

In the case of **text** files the decrypted files are searched for the one key that generated a file comprising of ASCII characters only (hex range **0x00** – **0x7f**) as they represent printable and controlling ASCII characters. This enabled us to find the key:

- **c. TEXT file key:** 0x98d63c96

3. The following can be stated as the weaknesses in the cipher design:

- **a.** The algorithm is vulnerable to **STEREOTYPED BEGINNINGS attack** for certain file types. When the value of the plain text and cipher text is known, the encrypted file can be decrypted by **BRUTE FORCE** as it is only a 32 bit key.
- **b. The second block in cipher text** is a simple XOR function of the key and 1st block of the plain text, therefore the KEY can be easily derived from the cipher text by reversing the XOR operation.

c. **The SIZE of the file** is present in the cipher text unencrypted and this knowledge could potentially be of use to the attacker as he/she now knows the exact length of the plain text which could help in **KNOWN-PLAIN TEXT attacks.**
d. Since the size of the file is the first block in all the encrypted messages the attacker can **MODIFY** the size (as it is just plain text) thereby changing the length of the original message.
e. The **key** generated from the dev/urandom system entropy generator has weaknesses (doesn't guarantee high levels of randomness) **when run on machines without a solid disk drive**, like live CDs and routers with diskless storage. Their boot state is predictable and hence reduces the pool of entropy values.

4. **PDF and PNG:** By observing weakness **'b'** from the above list, we can retrieve the key for **.pdf** and **.png** files from the encrypted file as the second block of the cipher text (which corresponds to the common beginnings of different types of files) is also just a simple XOR function of the key and the plain text. By knowing both the plain text and the cipher text we can find the key by doing this:

KEY = 1ST 4 BYTES OF PLAIN TEXT $\oplus$ 2ND 4 BYTES OF CIPHER TEXT

TEXT FILE: In the case of the text file we can reduce the search space of the keys. Since the second 4 bytes of the encrypted file are simply XORed with the 'key', and being a text file we assume that the range of the input lies between **0x00** and **0x7f**, we can generate the list of possible keys by XORing all the possible ASCII values and the second 4 bytes of the input. This will give us a **reduced search space** for the possible keys.

We can then perform an efficient brute force on these keys to find the key that generates only ASCII values throughout.

PSEUDO CODE FOR BRUTE FORCE:

1. Skip the first 4 bytes of encrypted text.
2. Read the second 4 bytes
3. For all values of key = 0x00000000 to 0xffffffff
    a. Start decrypt(key)
    b. For each set of decrypted bytes obtained, check
        i. For pdf if they equal "%PDF" (hex equivalent)
        ii. For png if they equal "%PNG" (hex equivalent)
    c. If they equal, continue decrypting with that key to get the full message. Set key_found = 1
    d. If they don't, abort brute forcing with that key
4. Return key if key_found= 1;

PSEUDO CODE FOR BRUTE FORCE(TEXT FILE):

1. Same as above algorithm
2. Same as above
3. For all the values of the key=0x00000000 to 0xffffffff
    a. Start decrypt(key)
    b. For each set of decrypted bytes, check whether all the bytes are within the ASCII range (0x00 to 0x7f)
    c. Continue with decryption if b holds good, abort if not.
4. If decryption completes fully to size of input file, return key