Information Security Exercises

Chapter 2: Set two: Encryption Part I. Deadline: September 30, 9:00 AM

Notes:

- Some programming languages in some operating systems (e.g., Windows and C/C++) make a distinction between 'text' and 'binary' files. With text files physical line delimiters (e.g., carriage return/linefeed (CRLF) combinations) may then automatically be replaced by single characters (e.g., newlines). Encrypted files normally do not follow such 'text file' conventions. Consequently, they should normally not be processed as 'text files' but as 'binary files'.

Exercise 5. Purpose: learn to work with a simple Feistel cipher (see also the advanced exercise about the Feistel cipher)

Define and implement a simple Feistel cipher in a little program.

The user of the program provides a (textual) password or passphrase.

The password/passphrase is passed to the SHA256 program to compute a 64 byte hash value and that hash value is then passed again to the SHA256 program, producing an additional set of 64 bytes.

E.g, for the password 'Feistel' (without the quotes) the sha256sum is:

```
184b4d16bbe3200c5a5f500cc09efa68cddd42cbda27c1e49fa7a0f2e2735007
```

Presenting the above string to sha256sum you get

```
bd11fd28eabd0b87f2ff4595a50041bfb882bbf8ae058ea5d677c7da07d43786
```

The sha256sum's bytes are used in the key-schedule, described below.

The characters of the sha256sum must be interpreted as hexadecimal values. The lowest byte of the first sha256sum equals 0x18, followed by 0x4b, 0x4d, 0x16, etc, until finally 0x07. Having used the characters of the sha256sum the bytes those of the second sha256sum follow: 0xbd, 0x11, 0xfd, etc., until finally 0x86. In total we thus have 64 bytes.

Implement the feistel function as a function processing blocks of 8 bytes. Each block is split into a left half (LH) and right half (RH).

As usual, RH becomes the LH of the next round, and LH xor f(key) becomes the RH of the next round.

Although the ECB block cipher should be avoided, it is used in this exercise to encrypt subsequent blocks of plain text. This exercise concentrates on the Feistel method, and once that's available it can be used in combination with other block cipher methods as well.

Key Schedule

To fully complete this method an additional manipulation would be used in which the RH and key are manipulated (see also 3DES). Eventually RH2 becomes

```
LH1 ^ F(RH, key)
```

where the key schedule is a separate operation, and the RH may also be manipulated, combining them to `F(RH, key)`.

This part (the key schedule handling) is not further elaborated in this exercise. In this exercise the key schedule simply consists of repeated sha256sum computations.

The key schedule is implemented in f(key), returning the key for a particular round.

For this exercise use 16 rounds. The keys for each of the rounds consist of four bytes. Using the above example: the key for the first round is 0x18 (first byte of the key), followed by 0x4b, 0x4d, 0x16, (fourth byte). Then, the key for the second round consists of the bytes 0xbb 0xe3 0x20 0x0c, etc. etc.

Since blocks consist of 8 bytes, we need keys of 4 bytes for each of the rounds. The two sha256sums provide just that: 16 keys of 4 bytes each, in total 64 bytes.

Submit your implementation. You may assume that the size of the file to encrypt is a multiple of 8 bytes.

Exercise 6. Purpose: learn to work with a superincreasing knapsack.

Send the TAs the public key of your superincreasing knapsack (consisting of at least 16 blank delimited unsigned values, the first value representing the value

having index 0, and so that one's the smallest.)

Notes:

- This is different from what Mark Stamp's doing: in his calculations the first value represents index 7;
- It's also different in that blocks of at least two characters are encrypted, rather than just characters. If you correctly design your program then you can encrypt blocks of any size, making it increasingly harder to break the encryption using brute force.
- Unless your program can handle integral values of unlimited size, make sure your encrypted values fit in your integral type.
- Implementation suggestion: if your block-size equals `N`, then when encrypting store `N` subsequent chars in a variable of your integral type, shifting the accumulated value over 8 bits before adding the next character.

  When decrypting recompute this variable through knapsack decryption, and then subsequently store the lowest 8 bits in a character array, and shift the decrypted value 8 bits to the right until the decrypted value has decayed to 0.

  Next, display the characters in the array in reversed order. Checking for the value 0 doesn't work if you intend to encrypt binary files. For this exercise you may assume that no 0-valued bytes were encrypted.
- Don't wait too long before sending a teaching assitant your public key. He must have received your public key at the latest one *full* day before this exercise's deadline or much earlier if you want to have the opportunity of resubmitting this exercise if not rated fully OK.

Having received your knapsack he'll send you some encrypted text using your knapsack.

The encrypted text he'll send you consists of a series of blank delimited numbers: it's the knapsack-encrypted text that you should decrypt using your private key. The exercise is completed when you submit (in your answer, on paper) the matching decrypted text.

With respect to the notes: what if you *do* want to allow 0 bytes? What's an easy modification of the current algorithm (describe it, no need to implement it)

Exercise 7. Purpose: describe the steps to perform when using the CBC block cipher mode

Describe the steps to perform when encrypting a file using the CBC block mode. You may use a 'pseudo' programming language, so no working program is required. E.g. to read the next plain text block write something like `read plainText`.

Next describe in a similar way the steps to perform to decrypt the encrypted file again.

You may use the notation `E(K, T)` to indicate the encryption of the text `T` using key `K` and `D(K, C)` to indicate the decryption of the code block `C` using the key `K`. Further specification of the encryption/decryption algorithm is not required.

Exercise 8. Purpose of this exercise: learn to work with a one-time pad

When using a Vernam cipher, the plain text `informationsecurity` may be encrypted using the key `vlaksjdhfgqodzmxncb`.

Note: use the xor-operator to en/decrypt; use the ASCII-character set, so don't design your own character set as Mark Stamp does in his book.

Xoring characters may easily result in non-displayable characters. If the encrypted message or key contains at least one non-displayable character, display the result as a series of ASCII character numbers.

If a key or encrypted message merely contains displayable characters then submit them as text-strings.

Assume we're using an alternative key resulting in the same encrypted text. The alternative key is:

```
tlftrffwmixor|{xbch
```

What is the resulting alternative original text?

Exercise 9. Purpose: learn to implement a program using a simple Feistel cipher.

See the basic feistel exercise for a description of most of the program's characteristics.

Augment the basic feistel implementation in such a way that padding using the *'n' bytes of value 'n'* padding method is used.

Show that you can encrypt and decrypt a text; submit your implementation.

Encryption:

The text to *encrypt* is made available on Nestor as the file `ns.spy.txt`. (The file contains some ascii chars > 127, which show up weirdly in html displays. You can ignore the fact that these characters are weirdly displayed. Just process them with your Feistel cipher).

Make sure you download it correctly: its sha256sum is

```
341e680120ae200cfa87c026997e6cf3febb919ccefe5ee599f958b30cbb70a8
```

The passphrase you must use for encryption is `Angela vs. the NSA`.

Send the encrypted file as an attachment to the Teaching Assistants. The TAs will check the sha256 sum of the received attachment: it must be

```
a1de533ae72b5907cad02b6ff951cdd363c8d7628ca1e4d22480a49488125c7c
```

Decryption:

The text to *decrypt* is the file `feistel.enc`

Make sure you download it correctly: its sha256sum is

```
0b97568930a743a5fba685a6049e0ffc9630fb8ea6dd918adb24dfb0547bf716
```

The passphrase that was used for encryption was `The Feistel Cipher`.

Exercise 10. Purpose: learn to implement a stream cipher

Implement the RC4 stream cipher. To prevent the published weakness of this stream cipher ignore the first 256 encrypted characters (e.g., call `keyStreamByte` 256 times before you actually start encrypting/decrypting.

The RC4 stream cipher is implemented as a filter: because of the xor operation the same algorithm can be used for both encrypting and decrypting.

Block ciphers tend to be preferred over stream ciphers. Why?

Submit your answer to the above question, submit your program and decrypt and submit the text in the file `rc4.enc`, using the key `2019`. Make sure you correctly download `rc4.enc`. Its sha256sum equals

```
f5b35ef47f806814d62b0f90e7b314ffa9b75e2f45d7fbc8bcfc23371b4186d1
```

Note: the program used for creating rc4.enc also ignored the first 256 `keyStreamByte` calls.