CPS 713 Applied Cryptography, Lab 4

Due date: Friday December 1st, 2017 @ 9 AM via D2L

1 Background

Diffie Hellman key exchange is a specific method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols. It allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

2 Lab Tasks

2.1 Diffie Hellman key exchange

Incomplete C code of Diffie Hellman key exchange protocol can be downloaded from the lab's page. After unzipping, you'll find the source code for a client (*dhe.c*) and a server ($dhe_server.c$), along with a Makefile and fixed Diffie-Hellman p and g parameters in the files *dhparam.pem*. Please make sure to edit the paths in the Makefile according to your installation paths. You are to use OpenSSL library to complete the missing parts for key exchange between a client and a server. The client and server have a similar structure. Each of them should build a public key, then send it to the other party, receive the public key from the other party and finally compute the secret key. Your task is to complete the missing parts. For this, consult the openssl documentation here. Since they are similar, focus only on one of them and then do similarly on the other one.

2.2 Merkle's puzzles

Consider the following scenario:
Alice would like to share a secret with Bob. However, she knows that Eve is listening on the communication channel. So, Alice cannot explicitly share the secret with Bob. Hence, she and Bob establish a small game which helps them think about the same secret, without revealing it to Eve. The game goes like this:
Alice sends a list of puzzles to Bob. A puzzle goes as follows:
$Puzzle_i = S - AES(k = 0^{14}||i, plaintext = Puzzle||i||secret_i$
where $secret_i$ represents the i-th secret Alice generated (represents 8 randomly generated bytes) and which should be figured out by Bob.

A random puzzle is chosen by Bob. Bob knows the first 14 bytes of the key are 0, so he tries to brute-force the rest of the key until he finds a plain text starting with "Puzzle". This way, he knows Alice's secret (the last part of the plain text). Bob sends the index of the puzzle that he solved, in order to let Alice know which secret they agreed on. In the end, both Alice and Bob share the same secret, and Eve has no clue about it.
Both Bob and Eve would have to brute-force keys to find the correct puzzle. In your own words, describe the advantages of using Merkle's puzzles key exchange protocol for the legal user (Bob) over the attacker (Eve) in terms of computational costs.

Your are to implement this mechanism in C. You must use a block cipher to encrypt/decrypt your messages. For this, you have two options:

- If you were successful in implementing Light-DES in the previous lab, you may reuse it here.
- Otherwise, you may use built in functions from any C crypto library, such as OpenSSL.

The main function of your program, and required functions can be found in the next page.

```
1
2
3  /* ************************** Functions *************************** */
4
5
6  /* ========================= Encryption ========================= */
7  /*
8    Encrypt a message m with a key k in ECB mode using Light-DES as follows:
9    c = Light-DES(k, m)
10
11   Args:
12     m should be a bytestring multiple of 16 bytes (i.e. a sequence of characters
       such as 'Hello...' or '\x02\x04...')
13     k should be a bytestring of length exactly 16 bytes.
14
15   Return:
16     The bytestring ciphertext c */
17
18 /* ========================= Decryption ========================= */
19
20 /* Decrypt a ciphertext c with a key k in ECB mode using Light-DES as follows:
21  m = Light-DES(k, c)
22
23   Args:
24     c should be a bytestring multiple of 16 bytes (i.e. a sequence of characters
       such as 'Hello...' or '\x02\x04...')
25     k should be a bytestring of length exactly 16 bytes.
26
27   Return:
28     The bytestring message m.    */
29
30
31 /* ======================= Generating Puzzles ======================= */
32 /* gen_puzzles()
33     This is Alice. She generates 2^16 random keys and 2^16 puzzles.
34     A puzzle has the following formula:
35     puzzle[i] = aes_enc(key = 0..0 + i, plaintext ="Puzzle" + chr(i) + chr(j) +
       alice_keys[i])
36     This function shall fill in the alice_keys list and shall return a list of 2^16
        puzzles. */
37
38
39 /* ======================= Solving a Puzzle ======================= */
40 /* solve_puzzle(puzzles)
41
42     This is Bob's function. He tries to solve one random puzzle. His purpose is to
       solve one random puzzle
43     offered by Alice.
44
45     This function shall fill in the bob_key list with the secret discovered by Bob.
46     The function shall return the index of the chosen puzzle. */
47
48
49
50 int main(int argc, char const *argv[])
51 {
52
53   char alice_keys = [];
54     char bob_key = [];
55
56
57
58
59   /* Alice generates some puzzles */
60   puzzles = gen_puzzles();
61
62   /* Bob solves one random puzzle and discovers the secret */
63   x = solve_puzzle(puzzles);
64
65   printf("Bob's Secret key %s \n", bob_key[0]);
66
67   printf("Alice's secret key: %s \n", alice_keys[x]);
68
```

```
69    if bob_key[0] == alice_keys[x]:
70        printf( "Puzzle Solved! \n") ;
71      else
72        printf("Puzzle Not Solved :( \n")
73
74    return 0;
75  }
```

3 Deliverables

Your submission should include all source files used for both tasks, the binary generated (if any), sample files, and clear documentation on what is included and how the code can be executed in addition to answers to the question in section 2.2. For your code development, you have to use C. You can assume that the plaintext is already in a binary form.