Project for ECE 525

Description: Add a Mechanism to Generate the HELP Parameters can carry out Token Authentication.

1) This project adds to the code you created in the previous labs.

2) You have assigned constant values to the following parameters in your existing code:
- LFSR_seed_low
- LFSR_seed_high
- Reference mean
- Reference range
- Modulus
- Margin

The HELP algorithm generates values for these parameters randomly for each authentication, and from the Protocol screencasts on HELP, both the verifier (server) and token (FPGA) generate nonces which are XOR'ed, and then the XOR'ed nonce is used to select these parameters. This is accomplished on BOTH the token and verifier using the RANDOM device that exists under Linux. Use the following code to open up the RANDOM device in both the token_regeneration.c file in the SDK directory and verifier_regeneration.c file in PROTOCOL directory.

```
int RANDOM;

if ( (RANDOM = open("/dev/urandom", O_RDONLY)) == -1 )
   { printf("ERROR: Could not open /dev/urandom\n"); fflush(stdout); exit(EXIT_FAILURE); }
printf("Successfully open '/dev/urandom'\n");
```

Following the open calls, get nonces on both the token and verifier using:

```
if ( read(RANDOM, token_n1, 8) == -1 )
   { printf("ERROR: Read /dev/urandom failed!\n"); fflush(stdout); exit(EXIT_FAILURE); }
```

Note use *verifier_n2* as the name of the server nonce variable.

The *token_n1* and *verifier_n2* nonces should be defined as

```
unsigned char token_n1[8];  // In token_regeneration.c
unsigned char verifier_n2[8];  // In token_regeneration.c

unsigned char verifier_n2[8];  // In verifier_regeneration.c
```

3) The verifier should transmit its 8 bytes of the nonce to the token. The token receives the *verifier_n2* nonce and computes an *XOR_nonce* that is a bitwise XOR of the 8 bytes from each of the *token_n1* and *verifier_n2* nonces.

4) The *XOR_nonce* is then sent back to the verifier (all 8 bytes).

5) Both the token and verifier will use the following code to compute the parameters. Here, we define a function *SelectParams* to encapsulate these operations so that you can include this same function in both the token_regeneration code in the SDK directory and the verifier_regeneration code in PROTOCOL directory code.

```
void SelectParams(int nonce_len_bytes, unsigned char nonce_bytes[nonce_len_bytes],
   unsigned int *LFSR_seed_low_ptr, unsigned int *LFSR_seed_high_ptr, float *new_mean_ptr,
   float *new_range_ptr, unsigned short *Modulus_ptr, unsigned short *Margin_ptr)
   {
```

```
float mean_lower = -10.0;
float range_lower = 150.0;

*LFSR_seed_low_ptr = (0x000007FF & ((nonce_bytes[1] << 8) + nonce_bytes[0]));
*LFSR_seed_high_ptr = (0x000007FF & ((nonce_bytes[3] << 8) + nonce_bytes[2]));

*new_mean_ptr = ((unsigned)(0x3F & nonce_bytes[4])) + mean_lower;
*new_range_ptr = (0x1F & nonce_bytes[5]) + range_lower;

if ( (0x01 & nonce_bytes[6]) == 0 )
   *Margin_ptr = 2;
else
   *Margin_ptr = 3;

if ( (0x01 & nonce_bytes[6]) == 0 )
   *Modulus_ptr = 12 + ((unsigned)(0x07 & nonce_bytes[7]) << 1);
else
   *Modulus_ptr = 18 + ((unsigned)(0x03 & nonce_bytes[7]) << 1);

return;
}
```

6) You should ensure that *SelectParams* is called at some point before these parameters are used by the functions you wrote in previous labs *ComputePNDiffs*, *TVComp*, *ComputePNDco* and *ComputeModulus*. As mentioned, use the LFSR_seed_low as the seed in *ComputePNDco* operation.

7) At this point, you should ensure that both the token_regeneration and verifier_regeneration are generating the exact same values for the six HELP parameters.

- LFSR_seed_low
- LFSR_seed_high
- Reference mean
- Reference range
- Modulus
- Margin

8) Transmit to the verifier the SHD and SBS bitstrings that you generated on the token, using calls similar to these for the SHD. In token_regeneration.c, a call similar to:

```
if ( SockSendB(SHD, MAX_PNDIFFS/8, verifier_socket_desc) < 0 )
   { printf("ERROR: Token helper data send failed\n"); fflush(stdout); exit(EXIT_FAILURE); }
```

In verifier_regeneration.c, a call similar to:

```
if ( (token_SHD_num_bytes = SockGetB(token_SHD, MAX_PNDIFFS/8, token_socket_desc)) < 0 )
   { printf("ERROR: Get token’s helper data failed!\n"); fflush(stdout); exit(EXIT_FAILURE); }
```

Do the same for SBS noting that the SBS bitstring is smaller than MAX_PNDIFFS/8

9) At this point, you are done with the token_regeneration.c code. The rest of work you need to do involves modifications to verifier_regeneration.c. The goal is to determine if you can find a match to the token_SBS that was just received in the data stored in the secure database. This data has been read in by verifier_regeneration.c.

In order to accomplish this, you need to create a loop that carries out an exhaustive search of the database. During each iteration, you will process the enrollment data for one chip. The enrollment data read in by verifier_regeneration is stored in a PNR and PNF array that is also a 2-D array BUT REDEFINES the dimensions as chip_num and PN_num. The routine *ReadDatabasePNsEx-*

act computes the averages for you. So you can access the data for any given instance by using PNR[chip_num][PN_num] and PNF[chip_num][PN_num].

Inside the loop, you need to call the same sequence of routines that you used in token_regeneration.c to compute the SHD and SBS. You can use those same routines in the verifier_regeneration.c code with two exceptions as noted below. In fact, the arrays fPND, fPNDc, fPNDco and fmodPNDco are already defined for you.
Exceptions:

- You need to specify the arguments to *ComputePNDiffs* as PNR[chip_num] and PNF[chip_num] in verifier_regeneration.c.
- You MUST use the helper data transmitted from the token (token_SHD) when constructing the verifier_SBS. Use the token_SHD to determine which bits are to be stored in the verifier_SBS. You function call should look like:
```
num_strong_bits = ConstructVerifierSBS(MAX_PNDIFFS, fmodPNDco, verifier_SBS, token_SHD, Modulus);
```
  Note there are no Margin parameters to this function.

Also, you need to store the SHD and SBS in verifier_regeneration.c into the verifier_SHD and verifier_SBS arrays, which are already defined for you.

10) Your routine *ConstructVerifierSBS* returns the number of strong bits in the token_SHD bitstring (which is just the number of 1's in this bitstring). You should then construct a loop that reads and compares each bit from the token_SBS received from the token and the verifier_SBS bitstring that you just constructed for the current chip. Count the number of bits that do NOT match (mismatches) and generate a line as follows:
```
chip x   Number of strong bits (from token SHD) yyy   Number of mismatches zzz
```

You should generate 19 lines in this format, one for each HELP instance that you will include your final report.

11) The instance of HELP that is programmed onto the Zybo board is the last one programmed when you ran enrollment in lab2. You can reprogram the board with any of the 19 HELP instances by running the *program_FPGA.elf* program that I have provided in the BITSTREAMS directory. First ensure that you have transferred all of the *.bin files to /tmp on the Zybo board, as you did in lab2 before running this command. Download *program_FPGA.elf* from UNM learn, transfer it to your Zybo board and run it as follows:
```
program_FPGA.elf 10
```
This programs the FPGA with the 10th bitstream from the /tmp directory.

NOTE: Do not attempt to program the FPGA using xsdk or vivado on the UNM server. I have not provided the .bit files that you would need for this, nor do you have permission to the usb programming ports. Please use the *program_FPGA.elf* process instead.

You should find that there are zero or nearly 0 mismatches in your mismatch results for the *chip x* that corresponds to the instance you have programmed onto the Zybo board using *program_FPGA.elf x*.

12) Run your code as usual by running the updated version of verifier_regeneration.c as follow:

```
    ./verifier_regeneration 192.168.1.20 KG_FU_TVChar_NumSeeds_15_optimalKEK_TVN_0.60_WID_1.10 MasterDB
```

And then run token_regeneration on the Zybo board

```
    /token_regeneration.elf 192.168.1.20
```

Replace the .1. with the ZYBO board you have been assigned.

13) Create a final report that includes a description of the work you have done. Include a copy of your *ConstructVerifierSBS* routine and the results of at least 5 trial runs, each of which would include 19 lines as the one shown above. You should pick 5 different instances of HELP (any 5) and re-program the FPGA before each of the 5 runs with a different instance.

14) 10 POINTS EXTRA CREDIT: Implement the Dual Helper Data scheme described in the HELP protocol screencasts. If you do, you would replace *ConstructVerifierSBS* with *ConstructVerifierSBS_DHD*, and provide a copy of this routine in your final report.