Project-2, EE 5010-01

Members: Dilip Pandit, Zachary Tovar

Data Generation and Encryption:

- For this project, we will use data for students which needs to be encoded and transmitted. The data will include sensitive information such as student name, grades, SSN and account information as seen in Figure 1. Both the original and encrypted data are stored and retrieved from a file in the project directory.

```
Name:    Dolores
Age:     18
Grade:   13
GPA:     2.75
Email:   dolo1@school.edu
Phone:   3031231234
SSN:     1234
Balance: 45.82
Courses:
Parents:

Name:    Teddy
Age:     19
Grade:   14
GPA:     3.42
Email:   tedd5@school.edu
Phone:   3031234321
SSN:     4321
Balance: 14.42
Courses:
Parents:

Name:    Bernard
Age:     20
Grade:   14
GPA:     3.9
Email:   bern6@school.edu
Phone:   3037654321
SSN:     6543
Balance: 0
Courses:
Parents:
```

Figure 1: Unencrypted Student Data from file

```
Student 1:
*&.qkkkk$'$9.8A
,.qkkkkkzsA9*/.qkkkzxA
qkkkkkye|~A&*"'qkkk/$'$z8(#$$'e./>A#$%.qkkkx{xzyxzyxAqkkkkkzyxA      *'*%(.qk~esyA$>98.8qkA*9.%?8qkA

Student 2:
*&.qkkkk.//2A
,.qkkkkkzrA9*/.qkkkzA
qkkkkkxeyA&*"'qkkk?.//~8(#$$'e./>A#$%.qkkkx{xzyxxyzAqkkkkkxyzA      *'*%(.qkzeyA$>98.8qkA*9.%?8qkA

Student 3:
*&.qkkkk       .9%*9/A
,.qkkkkky{A9*/.qkkkzA
qkkkkkxerA&*"'qkkk).9%}8(#$$'e./>A#$%.qkkkx{x|}~xyzAqkkkkk}~xA      *'*%(.qk{A$>98.8qkA*9.%?8qkA
```

Figure 2: XOR Encrypted Student Data

Server Side

- The student data is read from a class. It is then XOR encoded with encryption key of ‘K’. All of this is accomplished in the server-side function “first.cpp”.
- The data before and after encryption are shown in Figures 1 and 2.
- The key for Cyclic Redundancy check is $X^3 + 1$ which in binary is 1001.
- A modulo-2 division is performed on the binary string of the XOR encoded string. This is accomplished in server-side function “crc.cpp”.
- After obtaining the remainder, the remainder is appended to the end of the binary string (for the first student data, it was 010).
- In the main function “sock_server.cpp”, socket programming is done. Port number 444 has been reserved for the communication. The server will indefinitely wait for a client connection. Once a client is found, it will transfer the first student data using send() function.
- The whole server-side program is done in iterations of 3 (number of students). After every iteration WSAcleanup is performed to send the next data over.
- Figure 3 shows the server side for the first student data. As we can see, at the very end of CRC encoded binary string, the remainder 010 is appended.
- The short-coming in our project is that, automatically the data is not sent sequentially. Example: after the first student data is received by the client, the system pause at server side is cleared to send the second data. Then, the system pause at client side is cleared to receive the second data.

```
The unencoded message without CRC is
Name:    Dolores
Age:     18
Grade:   13
GPA:     2.75
Email:   dolo1@school.edu
Phone:   3031231234
SSN:     1234
Balance: 45.82
Courses:
Parents:


The encoded message without CRC is
*&.qkkkk$'$9.8A
,.qkkkkkzsA9*/.qkkkzxA
kkkkkye|~A&*"'qkkk/$'$z8(#$$'e./>Akkkx{xzyxzyxAqkkkkkzyxA        *'*%(.qk~esy$>98.8qkA.%?8qkA

The remainder is:       010

The encoded message with CRC is
0000010100101010001001100010111001110001011010110110101101101011011010110000111100100100001001110010010000111001001011110
0011100001000000100001010001011000010111001110001011010110110101101101011011010110111101001110011010000001000011100
0011100100101010001011110010111001110001011010110110101101101011011110100111100001000000100001100000110110000101001110001
0110101101101011011010110110101101101011011110010110010101111100011111001000001000011100010011000101010001000100100111
0111000101101011011010110110101100101111001001000010011100100100011110100000101100111000000101000000100011001001000010010
0010011101100101001011100010111100111110010000010001101100100011001001000010010100101110011100010110101101101011011010110
0111100001111011011110000111101001111001011110000111101001111001011110000111111010000010001100000011000000000101011100001
0110101101101011011010110110101101101011011110100111100101111000011111101000000100001001001010100010011100101010001001010
0010100000101110011100010110101101111111011111100110010101110011011110010100000100001000000100100001111100011100100111000
0010111000111000001110001011010110100000100011011001010100011100100101110001001010011111100111000011100010110101101000001
010

       SERVER: WAITING for incoming connection...
```

Figure 3: Server Side Terminal Window

Client side:

- After running the executable for client-side program, a connection is established with the host. The recv() function will take the data and store it in a character buffer message.
- The received message is first sent to “crc_rev.cpp” function to perform a CRC decoding of the data with the key “1001”. If the remainder is zero, the data is not tampered and is successfully received as seen in the message in Figure 4. Otherwise, an error message will pop up saying there was a transmission error.
- The decoded binary string is recovered by eliminating its remainder. Then, the binary string is converted to character string using ASCII values. The resulting encoded string is finally XORed with the key “K” to retrieve the original student data.
- Figure 4 shows the message received on the client side.

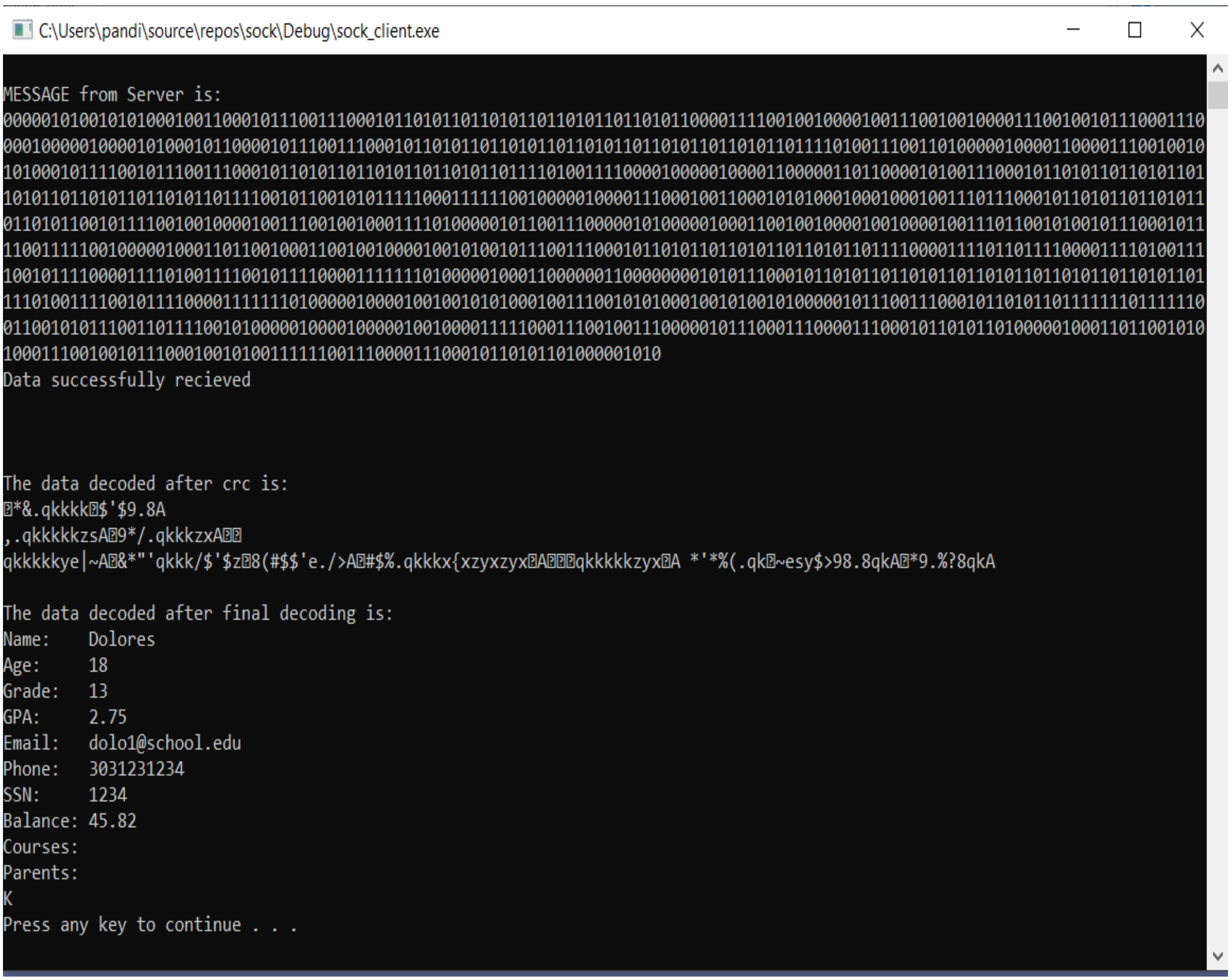

Figure 4: Receiver Side Terminal Window

Data tampering:

- To illustrate data tampering, we will consider a scenario where the third student's data is lost/tampered intentionally during data transmission.
- In the client side, after receiving the message we will edit some binary bits such that the remainder of CRC key division will not be zero. The client program should be able to detect the tampering and send a message to resend the data.
- Figure 5 is the data for student "Bernard" which is successfully sent from the server. However, before the message is received in the "sock_client.cpp", a FOR loop is implemented to change some bits in the MESSAGE buffer.
- The resulting message does not yield a 0 remainder and error message is sent indicating data needs to be re sent which is seen in Figure 6.

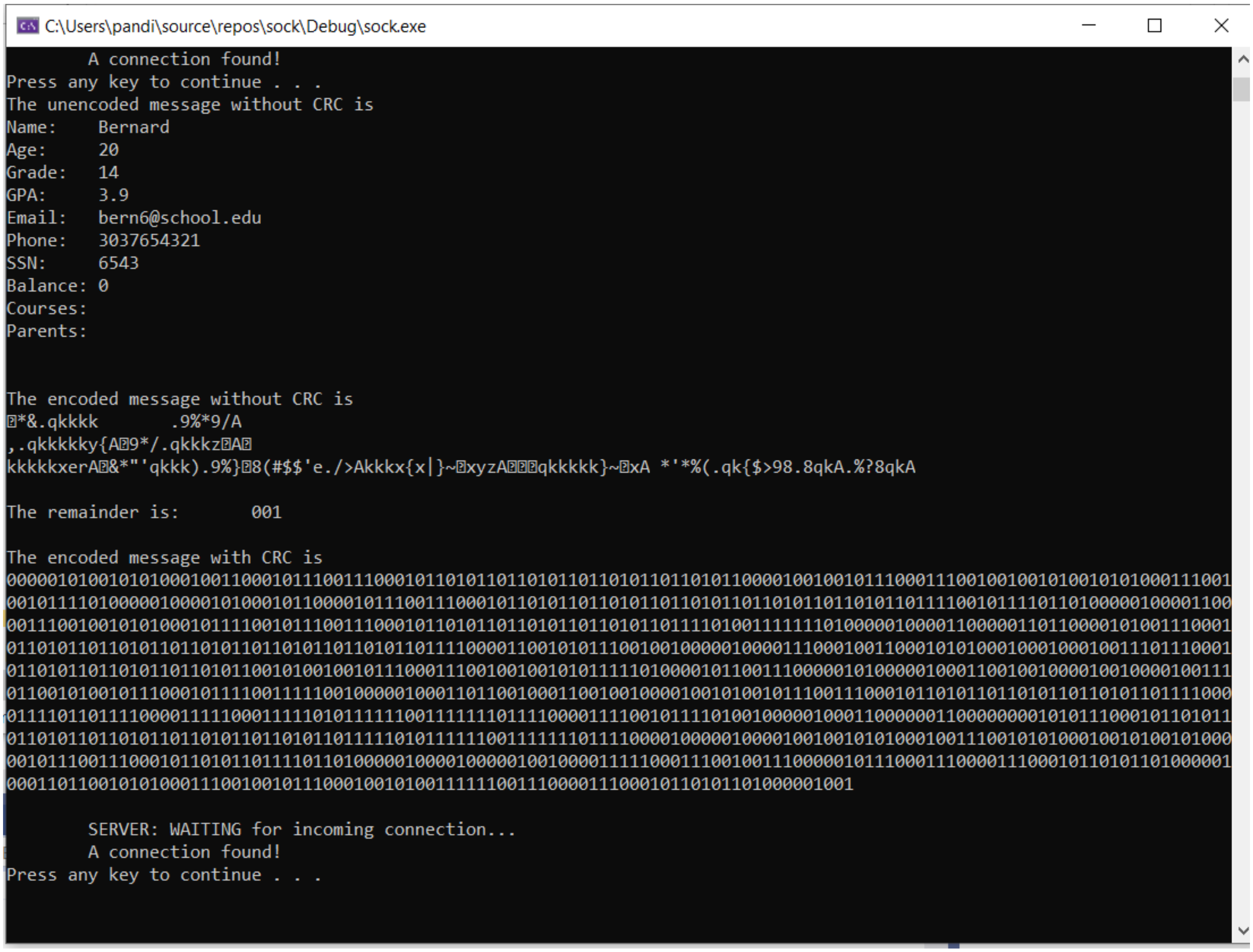

Figure 5: Student 3 data successfully sent from Server end

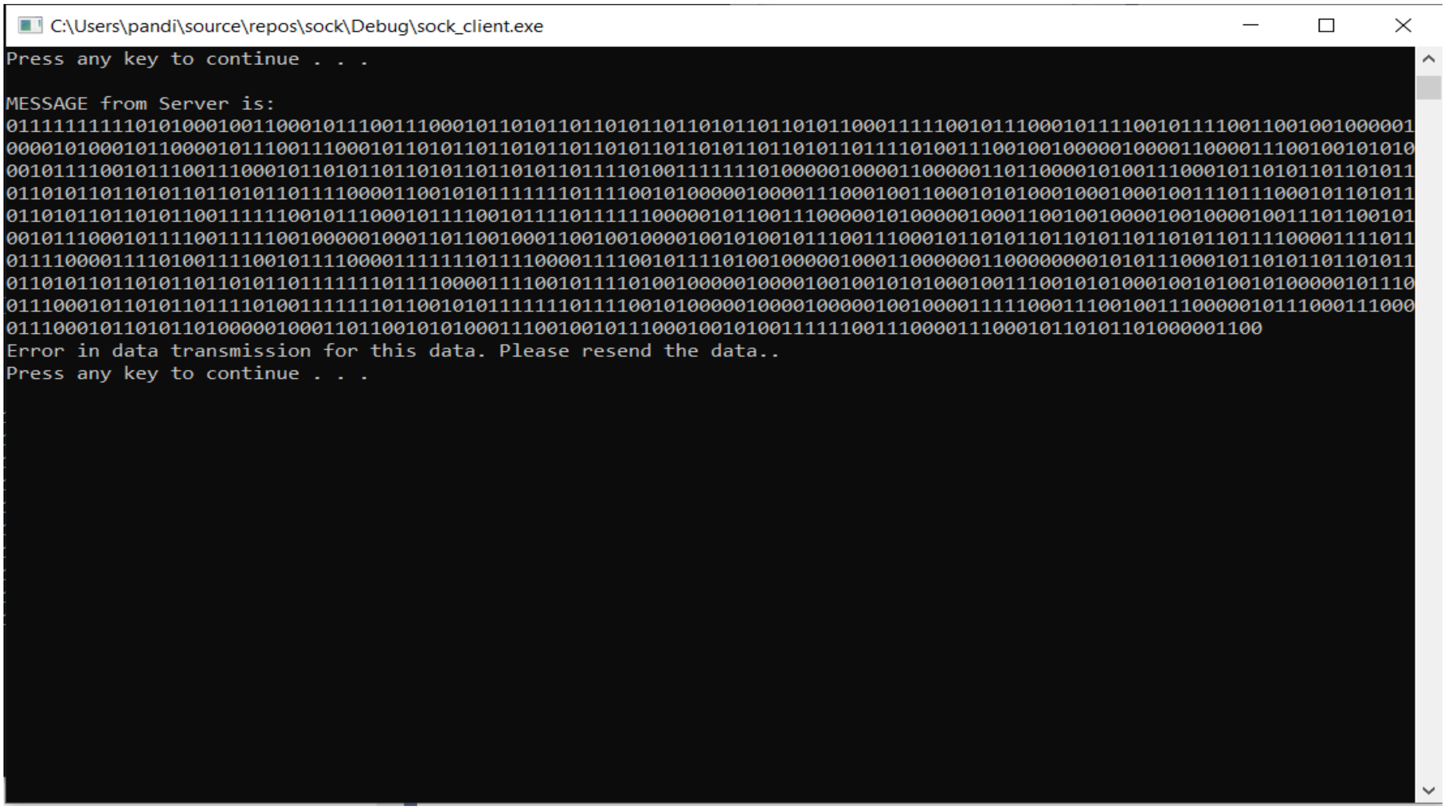

Figure 6: Student 3 data error in Client end

IDA pro analysis of the software:

In this section, the disassembly is done on the executable of the server file. We will attempt to reverse engineer the keys for XOR encoding and Cyclic Redundancy Check. We will assume to have some information about the program.

- First in the IDA disassembly, we check for the imports of the data. We can see two interesting imports related to file handling and socket. There are multiple imports such as fread, fwrite, fclose related to file handling. We know that running the executable we obtain encrypted and decrypted student information on the code directory.
- Another import is related to network. There are imports of WSAstartup, WSAcleanup, listen, bind, socket, etc which indicate the program will listen for upcoming connection.
- From the imports, we went to the subroutines where they were called. It was difficult to decode anything from file imports. However, it was seen that the all the network functions were called from a single subroutine sub_4236C0. From Figure 7, we can see the calls the subroutine makes to WSAstartup, creates a socket, and uses INETaddress. Furthermore, to prove that it was main function, at the start of the function the stack content is compared with 3 which represents number of student data for which the server code is iterated. It can be seen in Figure 8.
- Inside the main function, strings were found that represented the message displayed on screen. First, we checked the section for CRC. Before displaying the binary string, a call was made to sub_411636. Using the X-ref, the subroutine sub_41A230 called the function where an offset value of 1001 is pushed as seen in Figure 9. Location loc_41A501 of the subroutine has a long

arithmetic operation which hints at modulo-2 division and remainder calculation. Thus, we can conclude that sub_41A230 represents the “crc.cpp” where CRC was performed.

- Next, we need to find the key for the XOR encoding. A search is made for all possible XOR opcodes and the initializing operations are discarded. Still, there are numerous XOR operations between registers, registers and memory locations. So, it is hard to find the key using this approach.
- Again, we will reverse our way from the main function. Since we have the subroutine for CRC, we know that the XOR encoding is done prior to the CRC. So, we will monitor calls made before the CRC call. One of the calls to the subroutines, sub_411726 made a jump to a location in the subroutine sub_417F90. The subroutine was a hub for many subroutines but interestingly, it was a hub for subroutines which used the file imports. Searching for XOR operations in subroutines that called sub_417F90 from sub_411726, the subroutine sub_41ECD0 was most susceptible. It XORed the content obtained from sub_417F90 with a variable var_15 as seen in Figure 10. Moving up to see what was loaded in var_15, Figure 11 should a value of 4Bh was loaded. It is the ASCII of ‘K’ which is the key for XOR encoding in our case.

```
.text:0042396C                 call    ds:WSAStartup
.text:00423972                 cmp     esi, esp
.text:00423974                 call    sub_4116CC
.text:00423979                 mov     [ebp+var_C0], eax
.text:0042397F                 mov     [ebp+addrlen], 10h
.text:00423989                 mov     esi, esp
.text:0042398B                 push    0               ; protocol
.text:0042398D                 push    1               ; type
.text:0042398F                 push    2               ; af
.text:00423991                 call    ds:socket
.text:00423997                 cmp     esi, esp
.text:00423999                 call    sub_4116CC
.text:0042399E                 mov     [ebp+var_2A0], eax
.text:004239A4                 mov     esi, esp
.text:004239A6                 push    offset cp       ; "127.0.0.1"
.text:004239AB                 call    ds:inet_addr
.text:004239B1                 cmp     esi, esp
.text:004239B3                 call    sub_4116CC
```

Figure 7: Disassembly of the main function

```
.text:0042371C loc_42371C:                             ; CODE XREF: sub_4236C0+51↑j
.text:0042371C                 cmp     [ebp+var_18], 3
.text:00423720                 jg      loc_423B3D
.text:00423726                 lea     ecx, [ebp+var_3C]
.text:00423729                 call    sub_41119A
.text:0042372E                 mov     [ebp+var_4], 0
.text:00423735                 lea     ecx, [ebp+var_60]
.text:00423738                 call    sub_41119A
.text:0042373D                 mov     byte ptr [ebp+var_4], 1
.text:00423741                 push    offset unk_42DEB5
.text:00423746                 lea     ecx, [ebp+var_84]
.text:0042374C                 call    sub_4110D7
```

Figure 8: Main function iteration detection

```
loc_41A396:                                     ; CODE XREF: sub_41A230+F6↑j
                push    offset a1001    ; "1001"
                lea     ecx, [ebp+var_78]
                call    sub_41112C
                lea     ecx, [ebp+var_30]
                call    sub_4111D6
                mov     [ebp+var_FC], eax
                lea     ecx, [ebp+var_78]
                call    sub_4111D6
                mov     [ebp+var_108], eax
                lea     eax, [ebp+var_30]
                push    eax
```

Figure 9: Key value for Cyclic Redundancy Check

```
call    sub_411726
movsx   ebx, byte ptr [eax]
movsx   ecx, [ebp+var_15]
xor     ebx, ecx
```

Figure 10: Call to subroutine before XOR operation

```
.text:0041ED2B                 mov     [ebp+var_15], 4Bh
.text:0041ED2F                 lea     eax, [ebp+arg_4]
.text:0041ED32                 push    eax
.text:0041ED33                 lea     ecx, [ebp+var_3C]
.text:0041ED36                 call    sub_4116E5
```

Figure 11: Key for XOR operation