

Q1 Team Name
0 Points

Group Name

team_ethereum

Q2 Commands
5 Points

List all the commands in sequence used from the start screen of this level to the end of the level

go-> wave->dive->go->read->password->c->usojbmmmtel

Q3 Cryptosystem
5 Points

What cryptosystem was used at this level?

The crypto system used at this level is **EAEAE cipher** which is a substitution-affine block cipher with **5 layers**. It is a variant of AES cipher. Here, E is a element wise exponential vector of **8** bytes and A is a linear transformation matrix over F_{128} .

Q4 Analysis
80 Points

Knowing which cryptosystem has been used at this level, give a detailed description of the cryptanalysis used to figure out the password.

EXPLANATION:

1. We found ourselves in front of a passage leading to a deep underground well, so we decided to proceed by entering the command **"go."**

2. However, on the next screen, we found ourselves in a free fall with nothing to grab onto. We attempted to use the **"go"** command again, but

unfortunately, we perished.

3. Undeterred, we decided to retry from the beginning and used the command **"wave"** instead of "go" on the second screen, and this time we were successful in reaching the third screen.

4. On the third screen, we found ourselves swimming in both directions but could not find anything of interest. In an attempt to investigate further, we used the **"dive"** command.

5. This led us to the next screen, where we discovered a well-lit passage in the wall. We proceeded by using the **"go"** command.

6. The subsequent screen had a glass panel, which prompted us to use the **"read"** command. Unfortunately, we encountered an issue referred to as the **"EAEAE problem."**

7. Upon further investigation, we discovered that the problem involved input with a block size of 8 bytes, represented as an 8×1 vector over the finite field F_{128} .

8. The EAEAE is a weak form of SASAS attack. SASAS and EAEAE attacks are types of differential cryptanalysis attacks that take advantage of weaknesses in block ciphers. These attacks involve examining the variations between plaintexts and their corresponding ciphertexts generated by the encryption process in order to extract information about the encryption key or to break the encryption itself. The EAEAE encryption scheme is a block cipher that operates on a fixed-size block of 8 bytes. Each byte is an element of the finite field F_{128} , which consists of 128 elements represented as 16-byte vectors over the binary field $GF(2)$. The field F_{128} is defined by the irreducible polynomial $x^{128} + x^7 + x^2 + x + 1$.

9. When plaintext is encrypted using EAEAE, the resulting ciphertext consists of 8 bytes, each of which is an element of F_{128} . However, when we analyzed several ciphertexts generated by EAEAE, we noticed that the output contained only 16 different letters ranging from **'f' to 'u'**. This suggested that the encryption scheme had a weakness that allowed the ciphertext to be represented using a smaller set of symbols than expected.

10. To further investigate this observation, we decided to represent each of the 16 letters by 4 bits, with 'f' represented as **0000** and 'u' as **1111**. This

allowed us to represent each byte of the ciphertext as a pair of letters, or 8 bits. By doing so, we noticed that the most significant bit (MSB) of each byte was always 0. This was expected since each byte is an element of F_{128} , and the field has a maximum value of 127, which can be represented using 7 bits. Therefore, we concluded that the MSB of each byte must always be 0.

11. Based on this observation, we further concluded that the possible letter pairs in the ciphertext ranged from 'ff' to 'mu', since these are the only possible combinations of two letters that can be represented using 8 bits and have the MSB set to 0. This analysis highlights a weakness in the EAEAE encryption scheme that allows the ciphertext to be represented using a smaller set of symbols than expected, which can make the ciphertext vulnerable to attacks.

12. We discovered that any inputs of the form "gf" or "hf" resulted in the same output as inputs of type "g" or "h" respectively. That indicates that "f" was used as padding.

13. The EAEAE cipher system was analyzed using a specific input format: C^7P , where the output showed a pattern of having only the last byte of the output varying while all other bytes remained constant. Further experimentation with different input formats, such as C^6PC^1 , revealed that changing the i^{th} byte of the input caused all subsequent bytes of the output to change. Based on this observation, it was hypothesized that the transformation matrix used in the cipher system is a lower triangular matrix.

CRYPTANALYSIS

Observations:

By entering several plain texts, we saw that

1. If input plain text is ffffffffffffffff then output is also ffffffffffffffff.
2. If first i bytes of plaintext is f's then first i bytes of ciphertext is also f's.
3. On changing i^{th} byte of plaintext then the output ciphertext also changes from the i^{th} byte. Let plaintext P be p_0, p_1, \dots, p_7 where p_i is 1 byte. Then if we change input from $p_0, p_1, \dots, p_k, p_{k+1}, p_{k+1}, \dots, p_7$ to $p_0, p_1, \dots, p_k, p_{k+1}, \dots, p_7$ then resulting ciphertexts differ after k^{th} byte. As was mentioned above, matrix A appears to be a lower triangular matrix.

Computation of transformation matrix A and E

The matrix \mathbf{A} is of dimension 8×8 and \mathbf{E} is of dimension 8×1 .

Let $a_{i,j} \in \mathbf{A}$; where i is row index and j is column index and let $e_i \in \mathbf{E}$

To generate the plaintext set for use in the attack, we utilized the program **"plaintextgenerator.py"**. The plaintexts were generated using the formula $C^{i-1} * P * C^{8-i}$, where $C = ff$ and P is a value from the range of $[ff, mu]$, while i is a value from the range of $[1, 8]$. By using this formula, we generated 8 sets of plaintexts, with each set containing **128 plaintexts** that differed only at the i^{th} byte value. The resulting plaintexts were stored in the plaintexts.txt file.

To obtain the corresponding ciphertext for each plaintext in **plaintexts.txt**, we used a python script called **"server.py"**. This script utilized the pexpect library to establish a connection with the game server and simulate human-like behavior by inputting commands in a specific order. After establishing a connection with the server, the plaintexts were passed to the script to obtain the corresponding ciphertexts. The obtained ciphertexts were then stored in the ciphertexts.txt file for further analysis.

\mathbf{A} is lower triangular matrix we know that and

$$C = (A * (A * (P)^E)^E)^E$$

here P is plaintext and C is ciphertext.

Find the potential diagonal components of matrix A now, and then we'll use the brute force approach to find the components of E . The observation mentioned above suggested that matrix A is a lower triangular matrix.

The encryption process of the game involves several operations that are performed over the field F_{128} , including *exponentiation, linear transformation, exponentiation, linear transformation, exponential*. The modulo used for the exponentiation operations is $x^7 + x + 1$, which is an irreducible polynomial over the field F_2 .

In the field F_{128} , addition is performed as the XOR operation on integers. An attacker aims to determine the diagonal elements of matrix \mathbf{A} and the elements of matrix \mathbf{E} . To achieve this, the attacker systematically tests the values of $[0-127]$ for matrix \mathbf{A} and $[1-126]$ for matrix \mathbf{E} by encrypting plaintext-ciphertext pairs and comparing the results. The attacker's objective is to identify plaintexts that correspond to the given ciphertexts.

In the iteration process, an attacker can save the matrices \mathbf{A} and \mathbf{E} along with

their corresponding ciphertexts generated from plaintexts. By doing so, the attacker can determine the specific values used in the encryption algorithm for **A** and **E**. This knowledge may be exploited by the attacker to uncover vulnerabilities in the algorithm.

i^{th} Byte	Possible values of $a_{i,i}$	Possible values of e_i
0	[73, 84, 20]	[18, 21, 88]
1	[29, 52, 70]	[6, 7, 114]
2	[119, 43, 5]	[2, 38, 87]
3	[37, 126, 12]	[22, 37, 68]
4	[109, 112, 67]	[59, 90, 105]
5	[11, 106, 70]	[40, 89, 125]
6	[27, 14]	[20, 108]
7	[38, 8, 124]	[23, 48, 56]

The next step is to identify the non-diagonal elements of matrix A and remove certain pairs of $(a_{i,i}, e_i)$. We will perform an iterative search through the plaintext-ciphertext pairs with $(a_{i,i}, e_i)$ to determine the values that satisfy the equation: $C = (A * (A * (P)^E)^E)^E$.

i^{th} Byte	values of $a_{i,i}$	values of e_i
0	84	21
1	70	114
2	43	38
3	12	68
4	112	90
5	11	40
6	27	20
7	38	23

In order to determine the value of $a_{i,j}$, it is necessary to have knowledge of all the elements in the set.

$$Z_{i,j} = \{ a_{n,m} \mid n > m, j \leq n, m \leq i \} \cap \{ a_{n,n} \mid j \leq r \}$$

The final **Linear Transformation** Matrix **A** is :

$$\begin{bmatrix} 84 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 113 & 70 & 0 & 0 & 0 & 0 & 0 & 0 \\ 18 & 30 & 43 & 0 & 0 & 0 & 0 & 0 \\ 105 & 22 & 12 & 12 & 0 & 0 & 0 & 0 \\ 111 & 35 & 0 & 111 & 112 & 0 & 0 & 0 \\ 24 & 39 & 25 & 47 & 98 & 11 & 0 & 0 \\ 9 & 122 & 8 & 103 & 22 & 94 & 27 & 0 \\ 76 & 6 & 87 & 26 & 13 & 67 & 11 & 38 \end{bmatrix}$$

Then Final **Exponent Vector** E is :

$$E = [21, 114, 38, 68, 90, 40, 20, 23]$$

Decryption Of The Password:

In order to decrypt the password "**decrypt.py**" file has been used. To decrypt a password that has been encrypted with a transformation matrix A and exponent vector E , the encryption process needs to be reversed. This is done by dividing the encrypted password into blocks of 8 bytes and performing the inverse transformation on each block. Specifically, for each block of ciphertext, the inverse of matrix A , raised to the power of the corresponding element in the exponent vector E , needs to be computed. This is equivalent to raising the matrix A to the power of $-E_i$. Then, the resulting matrix is multiplied by the 8-byte block of ciphertext.

The outcome of this operation is an 8-byte block that corresponds to a portion of the original password. This process is repeated for each block of ciphertext until all blocks have been decrypted. After each block is decrypted, they are concatenated together to form the original plaintext password. It is important to note that the inverse of matrix A must exist for this decryption process to work.

$$E^{-1}(A^{-1}(E^{-1}(A^{-1}(E^{-1}(p)))))$$

Encrypted password :

"lhkhkumjkujmhmmfltlrkukfmml"

Encrypted Block 1 = "lhkhkumjkujmhmmfltlrkukfmml"

Encrypted Block 2 = "mmfltlrkukfmml"

Decrypted Block1 ASCII = [117, 115, 111, 106, 98, 109, 109, 116]

Decrypted password 1 = "usojbmmt"

Decrypted Block1 ASCII = [101, 108, 48, 48, 48, 48, 48, 48]

Decrypted Password 2 = "e1000000"

Decrypted Password :**” usojbmmtel000000”****'000000'** was assumed to be padding at the end and tried**” usojbmmtel”** as password for the level and thereby, successfully cleared the corresponding level.

Q5 Password

10 Points

What was the password used to clear this level?

usojbmmtel

Q6 Code

0 Points

Please add your code here. It is MANDATORY.

▼ modern_5_.zip

 [Download](#)

1 | Binary file hidden. You can download it using the button above.

▼ Assignment_5.ipynb

 [Download](#)**Plain-text Generation**

```

In [ ]: a = ['f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u']
        text = []
        z = 8
        for x in range(z):
            temporary = []
            for i in range(8):
                for j in range(16):
                    x1 = a[i] + a[j]
                    x2 = 'f'*x + 'f'*(7-x)
                    sting = x1 + x2
                    temporary.append(sting)
            text.append(temporary)

        file = open('plaintexts.txt','w')
        for i in text:
            sting = ''.join(i) + '\n'
            file.write(sting)
        file.close()

```

Server code to generate Output-Input pair

This pexpect module is not compatible with windows. So, run this notebook in mac or linux environment

```
In [ ]: import pexpect

child = pexpect.spawn('/usr/bin/ssh
student@172.27.26.188')
def calls():

    child.expect('student@172.27.26.188\'s password:')
    child.sendline('cs641') # Entering the password

    child.expect('Enter your group name: ', timeout=1)
    child.sendline("team_ethereum") # Entering the
group name

    child.expect('Enter password: ', timeout=1) # Waiting
for the password prompt
    child.sendline("team_ethereum") # Entering the
password

    child.expect('\r\n\r\n\r\nYou have solved 4 levels so
far.\r\nLevel you want to start at: ',
                timeout=1)
    child.sendline("5")

    child.expect('.')
    child.sendline("go")

    child.expect('.')
    child.sendline("wave")

    child.expect('.')
    child.sendline("dive")

    child.expect('.')
    child.sendline("go")

    child.expect('.')
    child.sendline("read")

    child.expect('.')
    child.sendline("password")

    child.expect('.')
    child.sendline("c")

    child.expect('.')

calls()
```



```

f = open("plaintexts.txt", 'r')
f1 = open("ciphertexts1.txt", 'w')

for line in f.readlines():
    li = line.split()

    for l in li:
        child.sendline(l)

        f1.write(s)
        f1.write(" ")
        child.expect("Slowly, a new text starts*")
        s = str(child.before)[48:64]

        f1.write(s)
        f1.write(" ")
        child.sendline("c")
        child.expect("The text in the screen vanishes!")
        f1.write("\n")

    child.sendline("ffffffffffffffmu")
    s = str(child.before)[48:64]
    f1.write(s)
    f1.write(" ")

child.close()

f.close()
f1.close()

```

Analysis

```

In [8]: import numpy as np
        from pyfinite import ffield
        import galois

        F = ffield.FField(7, gen=0x83, useLUT=-1)

        def expo(base,power):
            ans = base
            for i in range(1,power):
                ans = F.Multiply(ans,base)
            return ans

        def Linear(linmat,msg):
            ans = [0]*8
            for i in range(8):
                temp = []
                mul = [F.Multiply(linmat[i][j],msg[i]) for j in range(8)]
                for k in range(8):
                    temp.append(np.bitwise_xor(ans[k],mul[k]))
                ans = temp

```

```
return ans
```

```
def decode_block(cipher):
    plain=""
    for i in range(0,len(cipher),2):
        plain+=chr(16*(ord(cipher[i:i+2][0]) - ord('f')) + ord(cipher[i:i+2][1]) - ord('f'))
    return plain
```

```
PosExpo = [[] for i in range(8)]
posDiaVals=[[] for i in range(8)] for j in range(8)]
input_file = open('plaintexts.txt','r')
output_file = open('ciphertxts.txt','r')
input = (input_file.readlines()[0]).strip().split(' ')
output = output_file.readlines()
```

```
in_string = []
for msg in input:
    in_string.append(decode_block(msg)[0])
```

```
out_string = []
for i in range(len(output)):
    x = []
    for msg in output[i].strip().split(' '):
        x.append(decode_block(msg)[i])
    out_string.append(x)
```

```
for k in range(8):
    for i in range(1, 127):
        for j in range(1, 128):
            flag = True
            for m in range(128):
                if(ord(out_string[k][m]) !=
expo(F.Multiply(expo(F.Multiply(expo(ord(in_string[m]), i), j
                    flag = False
                    break
            if(flag):
                PosExpo[k].append(i)
                posDiaVals[k][k].append(j)
print("Possible diagonal values: \n")
print(posDiaVals)
print("\n\nPossible exponents: \n")
print(PosExpo)
```

```
out_string = []
for i in range(len(output)-1):
    x = []
    for msg in output[i].strip().split(' '):
        x.append(decode_block(msg)[i+1])
    out_string.append(x)
```

```
for ind in range(7):
    for i in range(1, 128):
```

```

for p1, e1 in zip(PosExpo[ind+1], posDiaVals[ind+1][ind+
for p2, e2 in zip(PosExpo[ind], posDiaVals[ind][ind]):
    for k in range(128):
        flag = True
        x1 = F.Multiply(expo(F.Multiply(expo(ord(in_strin
e2), p2), i)
        x2 = F.Multiply(expo(F.Multiply(expo(ord(in_strin
i), p1), e1)
        c1 = np.bitwise_xor(x1,x2)
        if(ord(out_string[ind][k]) != expo(c1,p1)):
            flag = False
            break
    if flag:
        PosExpo[ind+1] = [p1]
        posDiaVals[ind+1][ind+1] = [e1]
        PosExpo[ind] = [p2]
        posDiaVals[ind][ind] = [e2]
        posDiaVals[ind][ind+1] = [i]

print("\n\n*****")
print("Linear Transformation Matrix A values: \n")
print(posDiaVals)
print("\n\nExponent Vector E values : \n")
print(PosExpo)

```

```

def EAEAE(msg, lin_mat, exp_mat):
    msg = [ord(m) for m in msg]
    res = [expo(msg[i], exp_mat[i]) for i in range(8)]
    res = Linear(lin_mat, res)
    res = [expo(res[i], exp_mat[i]) for i in range(8)]
    res = Linear(lin_mat, res)
    res = [expo(res[i], exp_mat[i]) for i in range(8)]
    return res

```

```

input_file = open('plaintexts.txt','r')
output_file = open('ciphertxts.txt','r')
input = input_file.readlines()
output = output_file.readlines()

```

```

in_string = []
for i in range(len(input)):
    x = []
    for msg in input[i].strip().split(' '):
        x.append(decode_block(msg))
    in_string.append(x)

```

```

out_string = []
for i in range(len(output)):
    x = []
    for msg in output[i].strip().split(' '):
        x.append(decode_block(msg))
    out_string.append(x)

```

```

for index in range(0,6):
    offset = index + 2

exp_list = [e[0] for e in PosExpo]
lin_tran_lst = np.zeros((8,8),int)

for i in range(8):
    for j in range(8):
        if(len(posDiaVals[i][j]) != 0):
            lin_tran_lst[i][j] = posDiaVals[i][j][0]
        else:
            lin_tran_lst[i][j] = 0

for index in range(8):
    if(index > (7-offset)):
        continue

for i in range(127):
    lin_tran_lst[index][index+offset] = i+1
    flag = True
    for inps, outs in zip(in_string[index], out_string[index]):
        x1 = EA_EA(inps, lin_tran_lst, exp_list)[index+offset]
        x2 = outs[index+offset]
        if x1 != ord(x2):
            flag = False
            break
    if flag == True:
        posDiaVals[index][index+offset] = [i+1]

A = np.zeros((8,8),dtype='int')

for i in range(0,8):
    for j in range(0,8):
        if len(posDiaVals[j][i]) != 0:
            A[i][j] = posDiaVals[j][i][0]

E = exp_list

print('\n\nLinear Transformation Matrix : \n',A)
print('\n\n')
print('Exponent Vector : \n',E)

Einverse = np.zeros((128, 128), dtype = int)

for base in range(0,128):
    temp = 1
    for power in range(1,127):
        result = F.Multiply(temp, base)
        Einverse[power][result] = base
        temp = result

GF = galois.GF(2**7)
A = GF(A)
invA = np.linalg.inv(A)

```

```

password = "lhkhkumjkujmhhirmmfltlrkukfmmll"
GF = galois.GF(2**7)

def Einverse(block, E):
    return [Einverse[E[i]][block[i]] for i in range(8) ]

def Ainv(block, A):
    block = GF(block)
    A = GF(A)
    return np.matmul(A,block)

decrypted_password = ""
for i in range(0,2):
    elements = password[16*i:16*(i+1)]
    currentBlock = []
    for j in range(0,15,2):
        currentBlock+=[(ord(elements[j]) - ord('f'))*16 +
(ord(elements[j+1]) - ord('f'))]
    EAEAE = Einverse(Ainv(Einverse(Ainv(Einverse(currentBlock, E), inv
invA),E)
    for ch in EAEAE:
        decrypted_password += chr(ch)

print("\n\nPassword is",decrypted_password)

```

Possible diagonal values:

```
[[[73, 84, 20], [], [], [], [], [], [], []], [[], [29, 52, 70], [], [], [], [
```

Possible exponents:

```
[[18, 21, 88], [6, 7, 114], [2, 38, 87], [22, 37, 68], [59, 90, 105
```

=====

Linear Transformation Matrix A values:

```
[[[84], [113], [], [], [], [], [], []], [[], [70], [30], [], [], [], [], []],
```

Exponent Vector E values :

```
[[21], [114], [38], [68], [90], [40], [20], [23]]
```

Linear Transformation Matrix :


```
[[ 84  0  0  0  0  0  0  0]
 [113 70  0  0  0  0  0  0]
 [ 18 30 43  0  0  0  0  0]
 [105 22 12 12  0  0  0  0]
 [111 35  0 111 112  0  0  0]
 [ 24 39 25 47 98 11  0  0]
```

```
[ 9 122 8 103 22 94 27 0]
[ 76 3 87 26 13 67 11 38]]
```

Exponent Vector :
[21, 114, 38, 68, 90, 40, 20, 23]

Password is usojbmmmtel000000

In []:

Assignment 5	● Graded
Group	
DIVYESH DEVANGKUMAR TRIPATHI	
ALLAN ROBEY	
AVNISH TRIPATHI	
 View or edit group	
Total Points	
100 / 100 pts	
Question 1	
Team Name	0 / 0 pts
Question 2	
Commands	5 / 5 pts
Question 3	
Cryptosystem	5 / 5 pts
Question 4	
Analysis	80 / 80 pts
Question 5	
Password	10 / 10 pts
Question 6	
Code	0 / 0 pts