Challenge #1 - scrypt

You were able to dump the Administrator scrypt hash from a database. The password can be found among the top 1000 weakest passwords.

a. Recover the cleartext password.

b. Implement the solution in python programming language.

Administrator:8:8:1:c2VjcmV0X3NhbHQ:GydhHNhVQP4zvPGf2I/kgzQ6onwF4/+mxWKOmcY+BWA

I need to iterate through the passwords in the top 1000 weakest passwords (for example the rockyou.txt) and check if a password's hash matches with the hash which was dumped from the database.

If I understand correctly, the salt for the hashing is c2VjcmV0X3NhbHQ and the scrypt hash is c2VjcmV0X3NhbHQ:GydhHNhVQP4zvPGf2I/kgzQ6onwF4/+mxWKOmcY+BWA.

I need to use the given salt to make the hash function deterministic and to get the same hash for the correct password.

Other parameters: N = 8, r = 8, p = 1, dkLen = 43.

```
import pyscrypt

salt = "c2VjcmV0X3NhbHQ"
scryptHash = b"GydhHNhVQP4zvPGf2I/kgzQ6onwF4/+mxWKOmcY+BWA".hex() #length = 43

with open("password1000.txt", "r") as f:
    password1000 = f.read().split("\n")

def recoverPassword(passwordList, salt, correctHash):

    for password in passwordList:
        passwordHash = pyscrypt.hash(password.encode(), salt.encode(), 8, 8, 1, 43).hex()
        if passwordHash == correctHash:
            return password

recoverPassword(password1000, salt, scryptHash)
```

Doesn't work!

Challenge #2 - MD5 cracking

MD5 hash from database, the clear text is a Hungarian vehicle registration plate ("ABC-012").

Recover the cleartext using hashcat and implement the solution in any chosen programming language.

Soulition with hashcat:

```
/kripto/gy$ hashcat -a 3 -m 0 493ff01f9fd0ccf331f070aebfab3534 ?1?1?1?2?3?3?3 -1
 ?u -2 - -3 ?d --potfile-disable
hashcat (v6.2.5) starting
```

```
493ff01f9fd0ccf331f070aebfab3534:RCX-789

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: 493ff01f9fd0ccf331f070aebfab3534
Time.Started.....: Tue May 31 16:05:56 2022 (0 secs)
Time.Estimated...: Tue May 31 16:05:56 2022 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Mask.......: ?1?1?1?2?3?3?3 [7]
Guess.Charset....: -1 ?u, -2 -, -3 ?d, -4 Undefined
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:   586.1 MH/s (0.14ms) @ Accel:64 Loops:128 Thr:256 Vec:1
Recovered........: 1/1 (100.00%) Digests
Progress.........: 16640000/17576000 (94.67%)
Rejected.........: 0/16640000 (0.00%)
Restore.Point....: 0/1000 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:16512-16640 Iteration:0-128
Candidate.Engine.: Device Generator
Candidates.#1....: BYQ-199 -> XUO-594
Hardware.Mon.#1..: Temp: 57c Util: 98% Core:1567MHz Mem:6000MHz Bus:8

Started: Tue May 31 16:05:53 2022
```

Own solution in python3

```
import string
import hashlib

MD5 = "493ff01f9fd0ccf331f070aebfab3534"

def crackMD5_plate(hashedPassword):

    for c1 in string.ascii_uppercase:
        for c2 in string.ascii_uppercase:
            for c3 in string.ascii_uppercase:
                for d1 in string.digits:
                    for d2 in string.digits:
                        for d3 in string.digits:
                            s = f"{c1}{c2}{c3}-{d1}{d2}{d3}"
                            if hashlib.md5(s.encode()).hexdigest() == hashedPassword:
                                return s

password = crackMD5_plate(MD5)
print(password)
print(f"{password} hash == {MD5} : {hashlib.md5(password.encode()).hexdigest() == MD5}")
```

```
RCX-789
RCX-789 hash == 493ff01f9fd0ccf331f070aebfab3534 : True
```

Challenge #3 - Wrong implementation

a. Find a password for which the algorithm returns “Access GRANTED”.

b. Modify the code to make it secure. HINT: https://link.springer.com/chapter/10.1007/978-3-030-68884-4_8

```
<?php
$ELTE_hashed_password=
"0e000000000000000000000000000000000000000000000000000000000000";
$mypassword = readline("Please enter your password: ");
if ($ELTE_hashed_password == hash('sha256',$mypassword))
{echo "Access GRANTED\n";}
else {echo "Access DENIED\n";}
?>
```

```
import hashlib
hashlib.sha256(b"34250003024812").hexdigest()
```

```
'0e462890320380659161396210390858837734138209919207062996950513 32'
```

The core issue here is that the implementation uses the type unsafe com-parison variant of the language. The ELTE_hashed_password and sha256("34250003024812") digests have the same form - starting with zeroes followed by an e and then decimal digits only - and would be interpreted as integer 0, so in linenumber 5 the if condition evaluates to true.

a)

sha256(34250003024812) -> 0e462890320380659161396210390858837734138209919207062996950513 32

Correct password = 34250003024812 which is a magic hash.

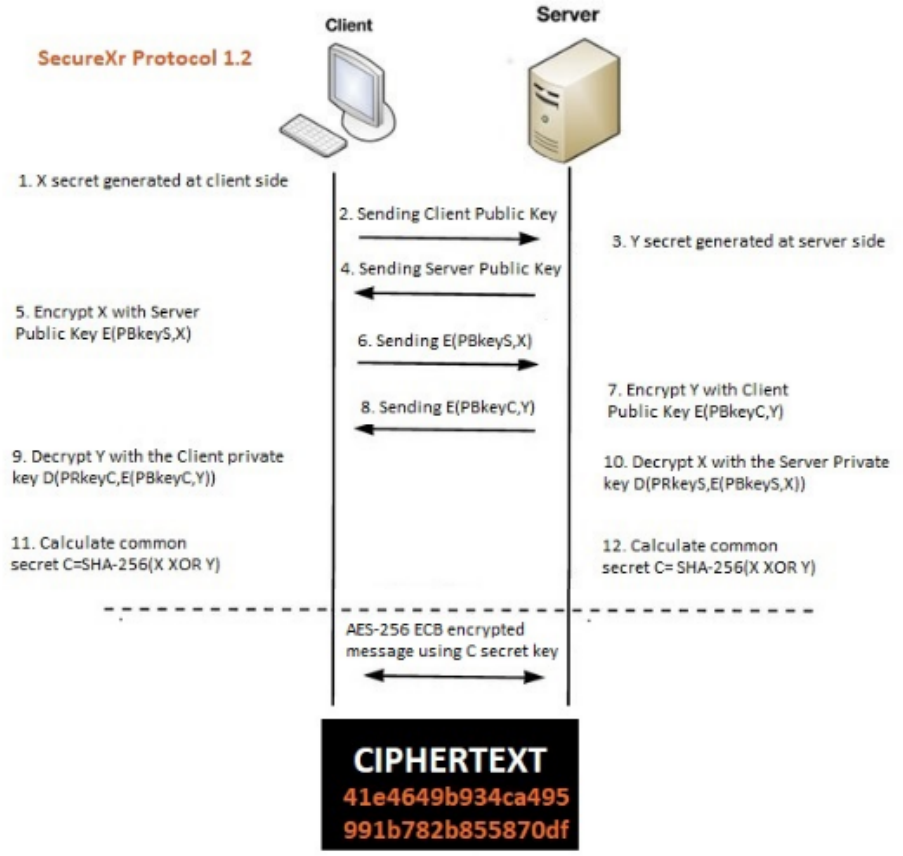
b)

The code above uses loose comparison (==) to test a given password. It should use a strict comparison (===) to avoid the magic hash attack:

```
<?php
$ELTE_hashed_password=
"0e000000000000000000000000000000000000000000000000000000000000";
$mypassword = readline("Please enter your password: ");
if ($ELTE_hashed_password === hash('sha256',$mypassword))
{echo "Access GRANTED\n";}
else {echo "Access DENIED\n";}
?>
```

Challenge #4 - Insecure protocol

The following protocol is used for transmitting secret information between a client and a server.



a. Identify weaknesses and possible vulnerabilities in the protocol.

b. Suggest improvements to be compliant with FIPS 140-2 standards.

c. Implement the protocol in any chosen programming language.

AES ECB mode isn't recommended most of the cases (deterministic enryption for each block, use CTR mode instead).

SHA256 is much shorter than the original secret X or Y, someone would brute force the possible hash outputs. We should use the shared secret as a seed for a PRG to create a common key for AES.

I choose Diffie-Hellman for public key encryption and decryption purposes.

Challenge #5 - Linear cryptanalysis

Linear cryptanalysis described by Mitsuru Matsui who first applied the technique to the FEAL cipher in EUROCRYPT '92. We have an 8 bits plaintext, ciphertext and key (P,C,K). We know the following linear expressions:

$$P \oplus C = 0x01010101$$
$$P_1 \oplus P_4 \oplus P_3 \oplus C_1 \oplus C_5 = K_4$$
$$P_3 \oplus P_6 \oplus P_1 \oplus C_1 \oplus C_3 = K_8$$
$$P_3 \oplus P_6 \oplus P_8 \oplus C_2 \oplus C_8 = K_6$$
$$P_3 \oplus P_2 \oplus P_7 \oplus C_5 \oplus C_8 = K_1$$
$$P_5 \oplus P_4 \oplus P_7 \oplus C_6 \oplus C_2 = K_7$$
$$P_7 \oplus P_3 \oplus P_1 \oplus C_3 \oplus C_8 = K_4$$
$$P_1 \oplus P_3 \oplus P_5 \oplus C_7 \oplus C_7 = K_2$$
$$P_5 \oplus P_8 \oplus P_7 \oplus C_2 \oplus C_3 = K_1$$
$$P_7 \oplus P_3 \oplus P_7 \oplus C_1 \oplus C_7 = K_3$$
$$P_6 \oplus P_7 \oplus P_2 \oplus C_5 \oplus C_1 = K_7$$
$$P_1 \oplus P_8 \oplus P_6 \oplus C_3 \oplus C_4 = K_8$$
$$P_1 \oplus P_3 \oplus P_7 \oplus C_2 \oplus C_1 = K_5$$
$$P_3 \oplus P_5 \oplus P_1 \oplus C_8 \oplus C_3 = K_3$$
$$P_2 \oplus P_6 \oplus P_7 \oplus C_2 \oplus C_6 = K_2$$
$$P_8 \oplus P_1 \oplus P_7 \oplus C_4 \oplus C_7 = K_5$$
$$P_1 \oplus P_2 \oplus P_3 \oplus C_4 \oplus C_5 = K_6$$

a. Find a valid Plaintext, Ciphertext and Key.

b. Implement the solution in any chosen programming language.

P ^ C = 0b01010101 -> Every odd bit is equal and every even bit is inequal in P and C.

Because of this I only have to iterate through P and K because I can calculate C from P (same bits for 1,3,5,7 and switch bits for 2,4,6,8).

```
def generatePossiblePKandC_lists():
```

```
    pkList = []
    cList = []

    for i in range(256): #iterate through the possible P values

        p = 8 * [0] #values in a list (example: [0,0,0,1,0,1,0,0])
        for j, digit in enumerate(bin(i)[-1:1:-1]):
            p[-1-j] = int(digit)

        #calculate C for a given P (P ^ C == 0b01010101)
        c  = 8 * [0]
        for j, val in enumerate(zip(c,p)):
            if j % 2 == 0:
                c[j] = val[1]
            else:
                if val[0] == val[1]:
                    c[j] = 1

        pkList.append(p)
        cList.append(c)

    return (pkList, cList)

def linearTest(P,C,K):
    #indeces from zero
    l1  = P[0] ^ P[3] ^ P[2] ^ C[0] ^ C[4] == K[3]
    l2  = P[2] ^ P[5] ^ P[0] ^ C[0] ^ C[2] == K[7]
    l3  = P[2] ^ P[5] ^ P[7] ^ C[1] ^ C[7] == K[5]
    l4  = P[2] ^ P[1] ^ P[6] ^ C[4] ^ C[7] == K[0]
    l5  = P[4] ^ P[3] ^ P[6] ^ C[5] ^ C[1] == K[6]
    l6  = P[6] ^ P[2] ^ P[0] ^ C[2] ^ C[7] == K[3]
    l7  = P[0] ^ P[2] ^ P[4] ^ C[6] ^ C[6] == K[1]
    l8  = P[4] ^ P[7] ^ P[6] ^ C[1] ^ C[2] == K[0]
    l9  = P[6] ^ P[2] ^ P[6] ^ C[0] ^ C[6] == K[2]
    l10 = P[5] ^ P[6] ^ P[1] ^ C[4] ^ C[0] == K[6]
    l11 = P[0] ^ P[7] ^ P[5] ^ C[2] ^ C[3] == K[7]
    l12 = P[0] ^ P[2] ^ P[6] ^ C[1] ^ C[0] == K[4]
    l13 = P[2] ^ P[4] ^ P[0] ^ C[7] ^ C[2] == K[2]
    l14 = P[1] ^ P[5] ^ P[6] ^ C[1] ^ C[5] == K[1]
    l15 = P[7] ^ P[0] ^ P[6] ^ C[3] ^ C[6] == K[4]
    l16 = P[0] ^ P[1] ^ P[2] ^ C[3] ^ C[4] == K[5]

    return (l1 and l2 and l3 and l4 and l5 and l6 and l7 and l8
            and l9 and l10 and l11 and l12 and l13 and l14 and l15 and l16)

def findAndPrintAllValidPCK():
```

```
    pkList, cList = generatePossiblePKandC_lists()

    for i, pc in enumerate(zip(pkList, cList)):
        p = pc[0]
        c = pc[1]

        for k in pkList:
            if linearTest(p, c, k):
                print(p, c, k)

def findFirstValidPCK():
    pkList, cList = generatePossiblePKandC_lists()

    for i, pc in enumerate(zip(pkList, cList)):
        p = pc[0]
        c = pc[1]

        for k in pkList:
            if linearTest(p, c, k):
                return (p, c, k)

print("All the valid Plaintext, Ciphertext and Key values for the given linear expressions:\
print("       Plaintexts               Ciphertexts                Keys")
findAndPrintAllValidPCK()
```

```
All the valid Plaintext, Ciphertext and Key values for the given linear expressions:

       Plaintexts               Ciphertexts                Keys
[0, 0, 0, 0, 1, 0, 1, 1] [0, 1, 0, 1, 1, 1, 1, 0] [0, 1, 1, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 1, 0, 1] [0, 0, 0, 1, 0, 0, 0, 0] [1, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 1, 1, 1, 0, 1, 0] [1, 1, 1, 0, 1, 1, 1, 1] [0, 1, 1, 1, 1, 1, 1, 0]
[1, 1, 1, 1, 0, 1, 0, 0] [1, 0, 1, 0, 0, 0, 0, 1] [1, 0, 0, 0, 1, 1, 1, 1]
```

Challenge #6 - MySQL3.23 hash cracking

We have the following MySQL 3.23 hash: 789abffc71d4fbbe

a. Recover the cleartext password.

b. Recommend a more secure hash function.

I used the MySQL323 Collider 1.1 program from Tobtu's website (source: https://tobtu.com/mysql323.php)

The code snippet that I used:

```
/kripto/mysql323-collider/MySQL323 Collider$ ./mysql323collider64 -m 2048 -t 4 -
h 789abffc71d4fbbe
Initializing...
Took 10.84 sec
12.45 Pp/s [25.1% 25.0% 25.0% 24.9%]
789abffc71d4fbbe:2344584f5e723c642a3438544a:#DXO^r<d*48TJ

Crack time:    13.184 seconds
Average speed: 12.35 Pp/s
```

a) We can see that #DXO^r<d*48TJ will be a good cleartext password for 789abffc71d4fbbe hash.

b) For password hashing, Argon2, PBKDF2, Scrypt or Bcrypt would be a better choice.

Challenge #7 - Cascade Ciphers

"Cascade Ciphers: The Importance of Being First" written by Maurer and Massey in 1993. The article can be downloaded from the internet.

What is the main conclusion of the article? (25-30 sentences)

Product ciphers: encrypting the plaintext multiple times with the same key.

Cascade ciphers: encrypting the plaintext multiple times with statistically independent keys.

Cascade ciphers are at least as strong as their first component.

If a cascades's ciphers commute: the cascade ciphers are at least as strong as their most-difficult-to-break component.

Additive stream ciphers do commute, and this fact is used to suggest a strategy for designing secure practical ciphers.

Note that although it can be proved "only" that a cryptographic stream-cipher chain is always at least as strong as the strongest link, it can be reasonably conjectured that the cascade is usually much stronger.

Using multiple encryption with different methods (with different necessary assumptions for security like RSA with factoring large numbers and Diffie-Hellman with discrete logarithm problem) can make our system more secure (if methods fail in the future our system will be safe with at least one strong method which stands the test of time).