How to encrypt text with a password in python?

Asked 6 years, 1 month ago   Modified 5 months ago   Viewed 77k times

32

Surprisingly difficult to find a straight answer to this on Google.

I'm wanting to collect a piece of text and a message from a user such as `1PWP7a6xgoYx81VZocrDr5okEEcnqKkyDc` `hello world`.

Then I want to be able to encrypt/decrypt the message with the text somehow so that I can save it in my database and not worry about the data being exposed if my website gets hacked,
`encrypt('1PWP7a6xgoYx81VZocrDr5okEEcnqKkyDc', 'hello world')`
`decrypt('1PWP7a6xgoYx81VZocrDr5okEEcnqKkyDc', <encrypted_text>)`

Is there a simple way to achieve this with python and please can someone provide/direct me to an example.

Perhaps an example of how to create public/private key pairs using a seed such as `'1PWP7a6xgoYx81VZocrDr5okEEcnqKkyDc'` ?

Many thanks in advance :)

EDIT: Just to be clear I'm looking for a way to encrypt my users data in a determanistic way not obfuscate the message.

If that means I have to generate a PGP/GPG pub/pri key pair on the fly by using the text `1PWP7a6xgoYx81VZocrDr5okEEcnqKkyDc` as a seed then that's fine but what's the method to do this?

python   encryption   cryptography

Share   Improve this question   Follow   edited Mar 3, 2017 at 0:25   asked Mar 3, 2017 at 0:02
derrend
4,036 4 27 40

2 "encrypt my users data in a determanistic way" - not secure. If every encryption of the same plaintext produces the same ciphertext, it'll be really easy to spot identical plaintexts in your database.
– user2357112 Mar 3, 2017 at 0:33

1 @user2357112 I'll be using a unique text string to encrypt each message so there will be no identical plaintexts. – derrend Mar 3, 2017 at 0:40

Even if it isn't an exact duplicate, the other question seems to contain useful answers. Several of the answer seems to be to use `pycrypto` for the encryption itself and perhaps `base64` to get a string representation of the result. – John Coleman Mar 3, 2017 at 0:54

3 @JohnColeman: Even if there are useful pieces, unless you already know what you're doing, it's impossible to recognize which ones are useful and how to use them properly. DES is broken, the top AES answer uses ECB mode, and the AES answer after it doesn't demonstrate proper IV handling. The accepted answer uses the freaking Vigenere cipher, broken for centuries and breakable by hand. – user2357112 Mar 3, 2017 at 12:20

2 @derrend: Identical plaintext blocks map to identical ciphertext blocks, so you get things like the ECB penguin. – user2357112 Mar 3, 2017 at 12:48

Report this ad

6 Answers

Sorted by: Highest score (default)

55

+100

Here's how to do it properly in CBC mode, including PKCS#7 padding:

```
import base64
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
from Crypto import Random

def encrypt(key, source, encode=True):
    key = SHA256.new(key).digest()  # use SHA-256 over our key to get a proper-
sized AES key
    IV = Random.new().read(AES.block_size)  # generate IV
    encryptor = AES.new(key, AES.MODE_CBC, IV)
    padding = AES.block_size - len(source) % AES.block_size  # calculate needed
padding
    source += bytes([padding]) * padding  # Python 2.x: source += chr(padding) *
padding
    data = IV + encryptor.encrypt(source)  # store the IV at the beginning and
encrypt
    return base64.b64encode(data).decode("latin-1") if encode else data
```

Join Stack Overflow to find the best answer to your technical question, help others answer theirs.

Sign up

```
sized AES key
    IV = source[:AES.block_size]  # extract the IV from the beginning
    decryptor = AES.new(key, AES.MODE_CBC, IV)
    data = decryptor.decrypt(source[AES.block_size:])  # decrypt
    padding = data[-1]  # pick the padding value from the end; Python 2.x:
ord(data[-1])
    if data[-padding:] != bytes([padding]) * padding:  # Python 2.x: chr(padding)
* padding
        raise ValueError("Invalid padding...")
    return data[:-padding]  # remove the padding
```

It's set to work with `bytes` data, so if you want to encrypt strings or use string passwords make sure you `encode()` them with a proper codec before passing them to the methods. If you leave the `encode` parameter to `True` the `encrypt()` output will be base64 encoded string, and `decrypt()` source should be also base64 string.

Now if you test it as:

```
my_password = b"secret_AES_key_string_to_encrypt/decrypt_with"
my_data = b"input_string_to_encrypt/decrypt"

print("key:  {}".format(my_password))
print("data: {}".format(my_data))
encrypted = encrypt(my_password, my_data)
print("\nenc:  {}".format(encrypted))
decrypted = decrypt(my_password, encrypted)
print("dec:  {}".format(decrypted))
print("\ndata match: {}".format(my_data == decrypted))
print("\nSecond round....")
encrypted = encrypt(my_password, my_data)
print("\nenc:  {}".format(encrypted))
decrypted = decrypt(my_password, encrypted)
print("dec:  {}".format(decrypted))
print("\ndata match: {}".format(my_data == decrypted))
```

your output would be similar to:

```
key:  b'secret_AES_key_string_to_encrypt/decrypt_with'
data: b'input_string_to_encrypt/decrypt'

enc:  7roSO+P/4eYdyhCbZmraVfc305g5P8VhDBOUDGrXmHw8h5ISsS3aPTGfsTSqn9f5
dec:  b'input_string_to_encrypt/decrypt'

data match: True

Second round....

enc:  BQm8FeoPx1H+bztlZJYZH9foI+IKAorCXRsMjbiYQkqLWbGU3NU50OsR+L9Nuqm6
dec:  b'input_string_to_encrypt/decrypt'

data match: True
```

Now, this is much better than ECB but... if you're going to use this for communication - don't! This is more to explain how it should be constructed, not really to be used in a production environment and especially not for communication as its missing a crucial ingredient - message authentication. Feel free to play with it, but you should not roll your own crypto, there are well vetted protocols that will help you avoid the common pitfalls and you should use those.

Share Improve this answer Follow   edited Oct 13, 2017 at 0:13   answered May 27, 2017 at 3:09
zwer
24.6k 3 48 65

2 'there are well vetted protocols that will help you avoid the common pitfalls and you should use those' you mean there is a simpler/more robust way to achieve this? if so could you please link me to it? – derrend May 27, 2017 at 4:31

1 @TessellatingHeckler - yes, CBC actually needs it, and if it's not random and unique (to the key used) CBC becomes susceptible to multitude of attacks. As for Damon's comments, I'm afraid he doesn't know what he's talking about - in CBC mode, IV is used to XOR the first 'plaintext' block, and since it's a chaining mode if you can't retrieve the first block you cannot retrieve any other - meaning, if you encrypt your IV with the data there is no way to decrypt it. There is a way to obfuscate your IV but that exceeds the comment section size and, besides, there is absolutely no reason to do it. – zwer May 27, 2017 at 5:26

3 **Warning**: using SHA as derivation method for the key (a KBKDF) is fine for input strings with *very high amount of entropy* but not for normal passwords or phrases. CBC is fine for storage in place (e.g. in a database) but **not** for transport mode security: rely on TLS instead. – Maarten Bodewes Jul 7, 2018 at 23:30

4 This might be a great solution, but it looks unnecessarily complicated. I see no reason not to use the `cryptography` library (cryptography.io/en/latest) which encrypts/decrypts in one line of code. The package is statically linked so all dependencies are included. – expz Sep 4, 2018 at 23:08

1 @Ferris - because rolling your own crypto is, usually, a bad idea. In particular, using just this to secure data in transit would make your protocol susceptible to replay attacks, keystream substitution, padding oracle attacks and possibly other attacks. Furthermore, as previously noted in the comments, it doesn't include a safe key derivation for low-entropy data so it's susceptible to brute force attacks. There are whole groups of smarter people than me who spent decades to iron out the details so, for production use, please refer to their vetted protocols—like the TLS. – zwer Aug 18, 2020 at 5:36

11

Based on zwer's answers but shows an example attempt to deal with the case where the source text is exactly a multiple of 16 (`AES.block_size`). However @zwer explains in a comment how this code will **BREAK THE ENCRYPTION** of your text by not padding your source text appropriately, making your pipeline insecure.

Code:

```
from builtins import bytes
import base64
```

Join Stack Overflow to find the best answer to your technical question, help others answer theirs. Sign up

```
    def encrypt(string, password):
        """
        It returns an encrypted string which can be decrypted just by the
        password.
        """
        key = password_to_key(password)
        IV = make_initialization_vector()
        encryptor = AES.new(key, AES.MODE_CBC, IV)

        # store the IV at the beginning and encrypt
        return IV + encryptor.encrypt(pad_string(string))

    def decrypt(string, password):
        key = password_to_key(password)

        # extract the IV from the beginning
        IV = string[:AES.block_size]
        decryptor = AES.new(key, AES.MODE_CBC, IV)

        string = decryptor.decrypt(string[AES.block_size:])
        return unpad_string(string)

    def password_to_key(password):
        """
        Use SHA-256 over our password to get a proper-sized AES key.
        This hashes our password into a 256 bit string.
        """
        return SHA256.new(password).digest()

    def make_initialization_vector():
        """
        An initialization vector (IV) is a fixed-size input to a cryptographic
        primitive that is typically required to be random or pseudorandom.
        Randomization is crucial for encryption schemes to achieve semantic
        security, a property whereby repeated usage of the scheme under the
        same key does not allow an attacker to infer relationships
        between segments of the encrypted message.
        """
        return Random.new().read(AES.block_size)

    def pad_string(string, chunk_size=AES.block_size):
        """
        Pad string the peculirarity that uses the first byte
        is used to store how much padding is applied
        """
        assert chunk_size  <= 256, 'We are using one byte to represent padding'
        to_pad = (chunk_size - (len(string) + 1)) % chunk_size
        return bytes([to_pad]) + string + bytes([0] * to_pad)
    def unpad_string(string):
        to_pad = string[0]
        return string[1:-to_pad]

    def encode(string):
        """
        Base64 encoding schemes are commonly used when there is a need to encode
        binary data that needs be stored and transferred over media that are
        designed to deal with textual data.
        This is to ensure that the data remains intact without
        modification during transport.
```

```
def decode(string):
    return base64.b64decode(string.encode("latin-1"))
```

Tests:

```
def random_text(length):
    def rand_lower():
        return chr(randint(ord('a'), ord('z')))
    string = ''.join([rand_lower() for _ in range(length)])
    return bytes(string, encoding='utf-8')

def test_encoding():
    string = random_text(100)
    assert encode(string) != string
    assert decode(encode(string)) == string

def test_padding():
    assert len(pad_string(random_text(14))) == 16
    assert len(pad_string(random_text(15))) == 16
    assert len(pad_string(random_text(16))) == 32

def test_encryption():
    string = random_text(100)
    password = random_text(20)
    assert encrypt(string, password) != string
    assert decrypt(encrypt(string, password), password) == string
```

Share Improve this answer Follow

edited Sep 4, 2021 at 21:18
hobs
18k 10 81 103

answered Feb 4, 2018 at 1:20
Ignacio Tartavull
341 2 12

1 Thank you for your answer sir :) @zwer would you please mind taking a moment to review if this new answer does indeed improve on your previous answer and cover use cases in which the source is a multiple of 16? Thanks you :) – derrend Feb 5, 2018 at 3:24

3 @derrend - sorry for the delay, SO failed to notify me of your comment and I just noticed it. There's no bug in my example pertaining to the source size - when it's exactly a multiply of `AES.block_size` it returns `AES.block_size` as padding ensuring that PKCS#7 padding is present as per specification. Ignacio's `(chunk_size - (len(string) + 1)) % chunk_size` will, however, return zero padding for all multiples of `AES.block_size - 1` (15 bytes in this case) which **will break** the encryption/decryption process for sources of that size. – zwer Feb 26, 2018 at 13:12

1 @zwer much appreciated sir, thank you very much for the response and clarification :) Ignacio, your effort is still very much appreciated also. – derrend Feb 27, 2018 at 13:42

Thanks, fixed the issue. let me know if you want me to update the answer. – Ignacio Tartavull Feb 27, 2018 at 20:05

1

1. If you are going to use mentioned database to authorise users, you should use hashes or message digests of user's passwords, instead of 2 way encryption algorithms, that would make your data hard to use even in case of db leakage.
2. You cannot use above method to protect data that needs to be decrypted at some point, but even then you can use more secure way than just encrypting user passwords using some fixed key (which is the worst method). Take a look at OWASP's Password Storage Cheat Sheet.

As you wrote "I want to be able to encrypt/decrypt the message", I'm attaching a simple python source (tested under 2.7) for encr/decr using Blowfish.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
import os
from Crypto.Cipher import Blowfish     # pip install pycrypto

BS = 8
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s : s[0:-ord(s[-1])]

def doEncrypt(phrase, key):
    c1  = Blowfish.new(key, Blowfish.MODE_ECB)
    return c1.encrypt(pad(phrase))

def doDecrypt(phrase, key):
    c1  = Blowfish.new(key, Blowfish.MODE_ECB)
    return unpad(c1.decrypt(phrase))

def testing123(phrase, key):
    encrypted = doEncrypt(phrase, key)
    decrypted = doDecrypt(encrypted, key)
    assert phrase == decrypted, "Blowfish ECB enc/dec verification failed"
    print ("Blowfish ECB enc/dec verified ok")
    print ('phrase/key(hex)/enc+dec: {}/{}/{}'.format(phrase, key.encode('hex'),
decrypted))

if __name__== "__main__":
    phrase= 'Ala ma kota, a kot ma AIDS.'
    key= os.urandom(32)
    testing123(phrase, key)
```

Share Improve this answer Follow

edited May 27, 2017 at 16:19
Nathan Tuggy
2,234 27 30 38

answered May 27, 2017 at 15:59
internety
344 3 9

1 ECB? Blowfish? Insecure! – Maarten Bodewes Jul 7, 2018 at 23:26

You can do it by using two of the built-in functions on the standard Python library. The first one is it to its corresponding unicode code (an integer). Two simple examples of the usage of this function are provided:

```
>>> ord('a')
    97

>>> ord('b')
    98
```

Then, you also have the inverse function of ord(): **chr( )**. And as you can imagine it works all the way around: it has a unicode code as an input (integer) and gets the corresponding unicode character (string):

```
>>> chr(97)
    'a'

>>> chr(98)
    'b'
```

Then you can do a simple encription by adding or substracting by some arbitrary integer... in this case, the number 2:

NOTE: Watch out in not utting very big values or you'll get an error id you reach a negative nber, for example.

```
def encrypt(message):
    newS=''
    for car in message:
        newS=newS+chr(ord(car)+2)
    return newS


print(encrypt('hello world'))
```

And getting as a result:

```
jgnnq"yqtnf
```

Now you can copy and past the same function and generate the decrypt function. In this case, it requires, obviously, to substract by 2:

```
def decrypt(message):
    newS=''
    for car in message:
        newS=newS+chr(ord(car)-2)
```

```
print(decrypt('jgnnq"yqtnf'))
```

And the result will be the original message again:

```
'hello world'
```

This would be a great way to encrypt messages to non programmers. However, anyone with a little of programming knowledge could write a program that varied the integer we used until they found we have just added (2) to the unicode characters to encrypt the code...

In order to avoid that, I would propose two more complex alternatives.

1. The first one is the simplest: it consists in applying a different sum value to the chr function depending on the position of the character (for example, adding 2 to each unicode code when it occupies an even position in the string and substracting 3 when sits on an odd position).

2. The second one will generate the maximum security. It will consist on adding or substracting every unicode code for a number that will be randomly generated for each character. It will require to store an array of values to decript back the message. Make sure, then, this array of values is not available to third parties.

There it goes a possible solution for **1**.:

```
def encryptHard(message):
newS=''
for i in range(len(message)):
  if i%2==0:
    newS=newS+chr(ord(message[i])+2)
  else:
    newS=newS+chr(ord(message[i])-3)
return newS


print(encryptHard('hello world'))
```

And the result would be:

```
jbniqyltif
```

With the information hereby privided the decrypting script is obvious, so I won't bother you with coping, pasing and changing two values.

Finally, let's go into an in-depth-analysis of the second more complex alternative. With this one we can say that the encription will be almost indefitable. The idea is to vary the value we add or range of numbers the chr( ) function admits, so do not try to play with other numbers o you will definitely get an error).

In this case, my proposal also randomizes the operation (sum or subtract), and avoids that the final number be a 0 (i.e. we would get an original character). Finally, its also returns a list with the numers it has been subtracted to, something you will need in order to decrypt the message back.

The chances that you get the same encrypted message if you call two times this function using the same message of length **n** are somewhat near to **255^n**... So don't worry (I say somewhat, as the algorithm created would actually generate more repeated values on the low-end or high-end range of values, for example, in case the most frequent characters were not centered in this distrubution unicode caracrer set (from 0 to 255), which is the case. However, the program, though not perfect, works flawlessly and protects the information.

```
import random as r
def encryptSuperHard(message):
  newS=''
  l_trans=[]
  for car in message:
    code=ord(car)
    add_subtract=r.choice([True,False])
    if add_subtract:
      transpose=r.randint(0,code-1)
      newS=newS+chr(code-transpose)
      l_trans=l_trans+[-transpose]
    else:
      transpose=r.randint(code+1,255)
      newS=newS+chr(code+transpose)
      l_trans=l_trans+[transpose]
  return newS, l_trans

print(encryptSuperHard('hello world'))
```

In this case, this random encrypting script I've made has returned this two value tuple, where the first value is the encrypted message and the second one is the value that has "transposed" every character in order of apearance.

```
('A0ŤłY\x10řG;,à', [-39, -53, 248, 214, -22, -16,     226, -40, -55, -64, 124])
```

Decrypting, in this case would need to take the encrypred message and the list and proceed as follows:

```
def decryptSuperHard(encriptedS,l):
  newS=''
  for i in range(len(l)):
    newS=newS+chr(ord(encriptedS[i])-l[i])
  return newS
```

And the results goes back to:

> hello world

```
print(deccryptSuperHard('A0ŤłY\x10řG;,à', [-39, -53, 248, 214, -22, -16,     226,
-40, -55, -64, 124])
```

Share Improve this answer Follow

answered May 27, 2017 at 5:08
blackcub3s
169 4

2 I appreciate the time and trouble you went to in order to create this answer but I'm quite sure this would be a terrible and severely insecure solution to my original question, having to store the array of values needed to decipher the text for instance is not realistic and introduces further complications. Thank you for trying though :) – derrend May 27, 2017 at 5:17

6 Congratulations, you've just discovered a rolling Caesar cipher, 2 millennia later. – zwer May 27, 2017 at 5:37

-1, it's pretty clear this is not secure. It would take most people an entire 15 minutes to reverse engineer this and decrypt anything used by this "encryption" algorithm. – Matt Messersmith May 27, 2017 at 6:03

@derrend, you're welcome. I am sorry the more complex solution I came up with is not feasible. I am not a computer scientist, nor a cybersecurity expert, but I thought It might generate some ideas. Thank you for your time reading it, I enjoyed thinking it! – blackcub3s May 27, 2017 at 11:41

0

Have gou considered using the cryptography package? Here's a simple example using Fernet encryption from their README:

```
from cryptography.fernet import Fernet

key = Fernet.generate_key()
f = Fernet(key)
token = f.encrypt(b"A secret message")
f.decrypt(token)
```

Based on this answer, the following AES256-GCM solution is even safer, although it requires a nonce:

```
import secrets
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

# Generate a random secret key (AES256 needs 32 bytes)
```

```
ciphertext = nonce + AESGCM(key).encrypt(nonce, b"Message", b"")

# Decrypt (raises InvalidTag if using wrong key or corrupted ciphertext)
msg = AESGCM(key).decrypt(ciphertext[:12], ciphertext[12:], b"")
```

To install the cryptography package:

```
pip install cryptography
```

Cheers, Cocco

Share Improve this answer Follow edited Nov 5, 2022 at 23:07 answered Nov 5, 2022 at 22:47

coccoinomane
827 8 22

-2

Here is my solution for anyone who may be interested:

```
from Crypto.Cipher import AES  # pip install pycrypto
import base64

def cypher_aes(secret_key, msg_text, encrypt=True):
    # an AES key must be either 16, 24, or 32 bytes long
    # in this case we make sure the key is 32 bytes long by adding padding and/or
slicing if necessary
    remainder = len(secret_key) % 16
    modified_key = secret_key.ljust(len(secret_key) + (16 - remainder))[:32]
    print(modified_key)

    # input strings must be a multiple of 16 in length
    # we achieve this by adding padding if necessary
    remainder = len(msg_text) % 16
    modified_text = msg_text.ljust(len(msg_text) + (16 - remainder))
    print(modified_text)

    cipher = AES.new(modified_key, AES.MODE_ECB)  # use of ECB mode in enterprise
environments is very much frowned upon

    if encrypt:
        return base64.b64encode(cipher.encrypt(modified_text)).strip()

    return cipher.decrypt(base64.b64decode(modified_text)).strip()


encrypted = cypher_aes(b'secret_AES_key_string_to_encrypt/decrypt_with',
b'input_string_to_encrypt/decrypt', encrypt=True)
print(encrypted)
print()
print(cypher_aes(b'secret_AES_key_string_to_encrypt/decrypt_with', encrypted,
encrypt=False))
```

```
b'secret_AES_key_string_to_encrypt'
b'input_string_to_encrypt/decrypt '
b'+IFU4e4rFWEkUlOU6sd+y8JKyyRdRbPoT/FvDBCFeuY='

b'secret_AES_key_string_to_encrypt'
b'+IFU4e4rFWEkUlOU6sd+y8JKyyRdRbPoT/FvDBCFeuY=    '
b'input_string_to_encrypt/decrypt'
```

Share Improve this answer Follow

answered May 27, 2017 at 0:36
derrend
4,036 4 27 40

2 Do not use ECB mode. Ever! It's not frowned upon in enterprise environments, it's frowned upon everywhere you want anything secure. ECB leaks so much data and is susceptible to so much attacks that some libraries refuse to even include it as an option. – zwer May 27, 2017 at 1:07

1 Btw. here's how to use CBC (on files, but remove the file stream and you can use the same code for in-memory encryption as well): stackoverflow.com/a/44126075/7553525 – zwer May 27, 2017 at 1:17

@zwer I'm no python expert, would you possibly mind at all providing a modified example to achieve the same thing in a secure way using CBC? I'd appreciate it greatly :) I'll pay you 100 reputation points. – derrend May 27, 2017 at 2:02

It would be quite easy if you used the same key on different pieces of data and/or different blocks of data, especially if the attacker has access to some of the plain text data to compare. While, technically, one can use ECB in a secure fashion, it's more trouble than it's worth so - ECB is like meth, not even once! – zwer May 27, 2017 at 2:05

@zwer it is a different key each time but in some instances it is the same piece of text being encrypted, potentially hundreds of times :/ – derrend May 27, 2017 at 2:12