CPSC 526 Winter 2025

Assignment 3

Due date: posted on D2L
Weight: 18% of your final grade.
Group work is allowed but not required. Maximum group size is 2 students.

This assignment is about symmetric cryptography. You will be given a Python program enkrypt.py that encrypts data using AES-128-CTR with a provided password. Your task will be to write three different programs that will decrypt data produced by enkrypt.py. The first program will decrypt data using a password; the second program will brute-force guess a password from encrypted data; and the third program will try to decrypt incorrectly encrypted data without learning the password.

Starter code (enkrypt.py)

Clone the starter code repository:

```
$ git clone https://gitlab.com/cpsc526/w25/a3-dekrypt.git
$ cd a3-dekrypt
```

The repo contains a fully functioning program, enkrypt.py, which encrypts data using a password. It accepts the password on the command line, then reads data from standard input. It encrypts the data using AES-128-CTR and writes the encrypted data to standard output. For example, the following command will encrypt the file inp/sample1.txt using the password “cpsc526”, and write the result to file out/sample1.txt.enc:

```
$ cat inp/sample1.txt
What is the answer to life, universe and everything?
$ ./enkrypt.py cpsc526 < inp/sample1.txt > out/sample1.txt.enc
```

The output will be a binary file, which you can inspect using xxd(1):

```
$ xxd out/sample1.txt.enc
00000000: 73d4 b368 ae33 2cce 6c69 82b3 5e7b 99dd  s..h.3,.li..^{..
00000010: 2416 fb6b b2e6 6380 0eba 15bd 7d8f beb9  $..k..c.....}...
00000020: f22b 2687 4756 8673 0542 5965 4daa cab2  .+&.GV.s.BYeM...
00000030: daea cba3 d616 1cd4 5ec1 2d27 78bf ba67  ........^.-'x..g
00000040: d535 d188 42fd fdfc ccc3 cde9 1bf3 464c  .5..B.........FL
00000050: 00be 6a03 f1                             ..j..
```

Please note that enkrypt.py can encrypt any data, including binary files.

Random salt and IV

The enkrypt.py program generates two random 16-byte nonces: salt and IV. The salt is used to stretch the user supplied password into a 16-byte key suitable for AES-128 encryption, using PBKDF2 key-derivation algorithm with 10 iterations. The IV value is used for the initialization vector needed by the CTR mode of operation. Using random IV for each encryption ensures that the same input encrypted multiple times results in different cyphertexts:

```
$ echo "hello" | ./enkrypt.py cpsc526 | xxd -s 32 -p
d3c492de0d64
$ echo "hello" | ./enkrypt.py cpsc526 | xxd -s 32 -p
d27f400e06cb
```

nce the decryption of a ciphertext would be impossible without knowing both the salt and IV used during encryption, enkrypt.py writes salt and IV as the first 32 bytes of the output. The IV vector occupies the first 16 bytes, and the salt is written to the next 16 bytes. For example, in the sample output above, the randomly generated IV is “73d4..99dd” and the salt is “2416..beb9”. This also means that enkrypt.py generates output that is 32 bytes longer than the input it receives, and the ciphertext starts at 33rd byte.

```
$ wc -c inp/sample1.txt out/sample1.txt.enc
 53 inp/sample1.txt
 85 out/sample1.txt.enc
```

Task 1 – Decryption with password (decrypt1.py)

Write a program dekrypt1.py which will decrypt data encrypted using enkrypt.py. The dekrypt1.py will accept a password on command line, read data on standard input, decrypt it, and write it to standard output. Your dekrypt1.py must be able to correctly decrypt any data encrypted using enkrypt.py, when the correct password is provided.

For example, to decrypt out/sample1.txt.enc we created above, you could execute the following command:

```
$ ./dekrypt1.py cpsc526 < out/sample1.txt.enc
What is the answer to life, universe and everything?
```

To check the output is correct for larger and/or binary files, you could compare SHA256 hashes:

```
$ sha256sum < inp/tower.png
e471da531b4e9b13e6f59b99df1a005b33bf58a800b3bc87ff0cdefa1612207a  -
$ ./enkrypt.py YYC < inp/tower.png > out/tower.png.enc
$ ./dekrypt1.py YYC < out/tower.png.enc | sha256sum
e471da531b4e9b13e6f59b99df1a005b33bf58a800b3bc87ff0cdefa1612207a  -
```

Here is another quick way to check the correctness of your program for short messages:

```
$ echo "hello world" | ./enkrypt.py YYC | ../dekrypt1.py YYC
hello world
```

Hints

This should be a straightforward task. Most of your code should be similar, and some parts even identical, to the code in enkrypt.py. Feel free to re-use any parts of enkrypt.py in your solution.

Task 2 – Brute-force password guessing (decrypt2.py)

Write a program dekrypt2.py, which will try to guess the password for a given encrypted data. You can assume the encrypted data was created using enkrypt.py, from a text input (ASCII characters only).

The input to your program will be a password pattern, provided on the command line, and encrypted data, provided via standard input. Your program will then read the first 32 bytes, corresponding to the nonces (salt and IV). Then it will read in 4096 bytes of the ciphertext (or as much as possible if the ciphertext is shorter than 4K). The rest of the input will be ignored.

Your program will then generate every password that matches the password pattern and use each of the generated passwords to decrypt the 4K buffer. If the decryption is correct, the program will output the corresponding password. To decide if an decryption is correct, your program will simply check if every one of the decrypted bytes is a valid ASCII character, i.e. in the range [0-127]. You can use the built-in bytes.isacii() function for this.

This is summarized using following pseudocode:

Pseudocode for dekrypt2.py

```
read salt (16 bytes), and iv (16 bytes)
read 4096 bytes into buffer (if input shorter than 4096, read entire input)
for each password that matches password-pattern:
    decrypt buffer into plainblock, using password, salt and iv
    if plainblock contains only ascii characters:
        print password
```

The password pattern may contain one or more underscore characters “_”, serving as a kind of a single character wild card. Your program will generate password guesses by replacing each underscore with either a single digit character [0-9], or an empty string. For example, if your program is given the partial password “cs_”, it will try the following 11 password guesses: cs0, cs1, cs2, cs3, cs4, cs5, cs6, cs7, cs8, cs9 and cs. Your program should not output the same password twice.

For example, you could guess the password for the out/sample1.txt.enc file like this:

```
$ ./dekrypt2.py cpsc___ < out/sample1.txt.enc
found password that seems to work: 'cpsc526'
$ ./dekrypt2.py _cpsc___ < out/sample1.txt.enc
found password that seems to work: 'cpsc526'
$ ./dekrypt2.py cp_sc__2_ < out/sample1.txt.enc
found password that seems to work: 'cpsc526'
```

If the program does not find a working password, it should print a relevant message:

```
$ ./dekrypt2.py cpsc < out/sample1.txt.enc
no password candidates found
$ ./dekrypt2.py _cpsc_ < out/sample1.txt.enc
no password candidates found
```

Detecting non-ASCII characters in the decrypted data is not a bulletproof way of deciding whether the password is correct. However, the probability of incorrectly reporting a password is 2^{-len}, where len is the length of the buffer that your program decrypts. If the plaintext contains at least 4KB, the probability of dekrypt2.py reporting false positives is negligible (2^{-4096}). Naturally, on short inputs dekrypt2.py may report many false positives:

```
$ echo tinytext | ./enkrypt.py cpsc526 | ./dekrypt2.py ___
found password that seems to work: '156'
found password that seems to work: '363'
found password that seems to work: '50'
found password that seems to work: '1'
```

Note: enkrypt.py uses only 10 iterations for PBKDF2, which is usually low. This was done on purpose – to make the assignment a bit easier. Low value of iterations will make the key-stretching faster, allowing more efficient brute-force password guessing. In a real application, PBKDF2 would need to be used with a much higher number of iterations (500,000+) to make brute-force attacks more difficult. Feel free to experiment with higher values for the iterations and compare much slower your dekrypt2.py would run.

Task 3 – Decryption without password (decrypt3.py)

In order for CTR mode of operation to be secure, it is important that we do not re-use the same nonces. Reuse of nonces would lead to the same weakness that One-Time-Pad cipher has with a reused secret key. For example, if Eve was able to obtain a valid plaintext-ciphertext pair, Eve would be able to recover the secret key. The recovered key could then be used to decipher any other ciphertext (provided it was encrypted using the same key). That is why enkrypt.py generates random salt and IV nonces for each encryption.

If enkrypt.py reused the same nonces for multiple encryptions, an attacker might be able to decrypt some files without even learning the password. For example, consider two plaintext files P1 and P2. Suppose someone encrypted both files with the same password and nonces (IV and salt), and stored the results in files C1 and C2, respectively. If the attacker got hold of files P1, C1 and C2, but not P2, the attacker could recover the contents of P2 using P1, C1 and C2, without learning the password.

To help you create such encryptions (with reused nonces), enkrypt.py can be invoked with ‘-nonce N’ command line option. This option disables random nonce generation, and forces the program to generate nonces deterministically from N. For example, you can encrypt 2 files with the same password and nonces like this:

```
$ ./enkrypt.py -nonce 123 unknown < inp/1984.txt > out/1984.txt.enc
$ ./enkrypt.py -nonce 123 unknown < inp/secret.txt > out/secret.txt.enc
```

You can verify that both files were created using the same nonces, by inspecting their first 32 bytes:

```
$ xxd -l 32 out/1984.txt.enc
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 007b  ...............{
$ xxd -l 32 out/secret.txt.enc
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 007b  ...............{
```

Write a program dekrypt3.py that will take 3 filenames on command line as input: P1, C1 and C2. Assume that file C1 is the encrypted version of file P1, and C2 is the encrypted version of some unknown file P2. Your program will examine the contents of P1, C1 and C2 and then decide whether any contents of P2 can be computed. If P2 can be at least partially recovered, your program will output the recovered part of P2. If none of the contents of P2 can be recovered, your program will output an error message indicating so. For example, dekrypt3.py can be used to recover contents of scret.txt from the example above:

```
$ ./dekrypt3.py inp/1984.txt out/1984.txt.enc out/secret.txt.enc
The answer to life,
universe and everything
is 42.
```

Here is an example where the program would fail because of mismatched nonces:

```
$ ./dekrypt3.py inp/1984.txt out/secret.txt.enc out/sample1.txt.enc
Error: cannot decrypt
```

Hints

Review the materials on OTP cipher, and CTR mode of operation. This task does not require use of any modern cryptography libraries or techniques.

Allowed 3rd party code and libraries

- You may use any library/package installed on the Linux lab computers.
- You may use any code given to you by your instructor or your TA, including the starter code.

Group work & demo

You will be asked to demo your assignments individually. The time for your demo will be arranged by your TAs. During the demo you will be asked to demonstrate the functionality of your code. You may also be asked to adjust some functionality of your code to show you are familiar with the code you submitted.

You are allowed to work on this assignment with another student (max. group size is 2). Just beware that during the demo you will be asked to demonstrate familiarity with all code you submit. If you decide to group up with another student, make sure that both of you understand your code. If you are unable to explain how your code works during the demo, you may receive reduced, or even zero, marks for the assignment.

Submissions

Make sure all your code works on the Linux lab machines.

All group members must submit 4 files on D2L:

dekrypt1.py	Implementation of task 1
dekrypt2.py	Implementation of task 2
dekrypt3.py	Implementation of task 3
readme.[txt|pdf|docx]	Write names of all group members at the top. If needed, include any special instructions needed to run your code. Do not use any other file formats!

Marking

Category	**Marks**
dekrypt1.py – works correctly on at least 1 non-trivial test case	10
– correctly decrypts all text files	10
– correctly decrypts all files (text and binary)	10
dekrypt2.py – correctly guesses at least one password (passes 1 non-trivial test case)	20
– correctly guesses passwords with one underscore at the end	10
– correctly guesses passwords with 2-6 underscores at the end	10
– correctly guesses passwords with any 6 wildcards, and **under 85s** on cslinuxlab computers	10
– reports duplicate passwords	-10 (penalty)
– unable to handle patterns without underscores	-5 (penalty)
dekrypt3.py – correctly decrypts at least one test case	10
– works correctly on 2 test cases	5
– works correctly on all test cases	5

V2

Other Penalties

Non-functioning code (e.g. code does not run on Linux machines in the lab)	-100
Occasional crashes	-5/per, up to -20
Ugly or unreadable code	Up to -15
Missing/inadequate documentation	Up to -15
Missing or incomplete readme.txt	Up to -20
Code not uploaded to D2L	-100

General information about all assignments

- All assignments are due on the date listed on D2L. Late penalties will be applied as described in the outline and during the first week of classes.
- Extensions may be granted only by the course instructor.
- Most assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
- All programs you submit must run on departmental Linux machines. If we are unable to run your code on the departmental Linux machines, you will receive 0 marks.
- Assignments must reflect your own, or your group's individual work. Here are some examples of what you are not allowed to do for assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions (including code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly, you are not allowed to AI tools to generate your code nor answers. This list is not exclusive. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k.html .
- We may use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Appendix – examples of password patterns

Here are some examples of partial passwords and corresponding password guesses that your program needs to generate:

Password pattern	Guesses
pass	pass
cpsc__	cpsc00 cpsc01 cpsc02 cpsc03 cpsc04 cpsc05 cpsc06 cpsc07 cpsc08 cpsc09 cpsc0 cpsc10 cpsc11 cpsc12 cpsc13 cpsc14 cpsc15 cpsc16 cpsc17 cpsc18 cpsc19 cpsc1 cpsc20 cpsc21 cpsc22 cpsc23 cpsc24 cpsc25 cpsc26 cpsc27 cpsc28 cpsc29 cpsc2 cpsc30 cpsc31 cpsc32 cpsc33 cpsc34 cpsc35 cpsc36 cpsc37 cpsc38 cpsc39 cpsc3 cpsc40 cpsc41 cpsc42 cpsc43 cpsc44 cpsc45 cpsc46 cpsc47 cpsc48 cpsc49 cpsc4 cpsc50 cpsc51 cpsc52 cpsc53 cpsc54 cpsc55 cpsc56 cpsc57 cpsc58 cpsc59 cpsc5 cpsc60 cpsc61 cpsc62 cpsc63 cpsc64 cpsc65 cpsc66 cpsc67 cpsc68 cpsc69 cpsc6 cpsc70 cpsc71 cpsc72 cpsc73 cpsc74 cpsc75 cpsc76 cpsc77 cpsc78 cpsc79 cpsc7 cpsc80 cpsc81 cpsc82 cpsc83 cpsc84 cpsc85 cpsc86 cpsc87 cpsc88 cpsc89 cpsc8 cpsc90 cpsc91 cpsc92 cpsc93 cpsc94 cpsc95 cpsc96 cpsc97 cpsc98 cpsc99 cpsc9 cpsc
cpsc	0cpsc0 0cpsc1 0cpsc2 0cpsc3 0cpsc4 0cpsc5 0cpsc6 0cpsc7 0cpsc8 0cpsc9 0cpsc 1cpsc0 1cpsc1 1cpsc2 1cpsc3 1cpsc4 1cpsc5 1cpsc6 1cpsc7 1cpsc8 1cpsc9 1cpsc 2cpsc0 2cpsc1 2cpsc2 2cpsc3 2cpsc4 2cpsc5 2cpsc6 2cpsc7 2cpsc8 2cpsc9 2cpsc 3cpsc0 3cpsc1 3cpsc2 3cpsc3 3cpsc4 3cpsc5 3cpsc6 3cpsc7 3cpsc8 3cpsc9 3cpsc 4cpsc0 4cpsc1 4cpsc2 4cpsc3 4cpsc4 4cpsc5 4cpsc6 4cpsc7 4cpsc8 4cpsc9 4cpsc 5cpsc0 5cpsc1 5cpsc2 5cpsc3 5cpsc4 5cpsc5 5cpsc6 5cpsc7 5cpsc8 5cpsc9 5cpsc 6cpsc0 6cpsc1 6cpsc2 6cpsc3 6cpsc4 6cpsc5 6cpsc6 6cpsc7 6cpsc8 6cpsc9 6cpsc 7cpsc0 7cpsc1 7cpsc2 7cpsc3 7cpsc4 7cpsc5 7cpsc6 7cpsc7 7cpsc8 7cpsc9 7cpsc 8cpsc0 8cpsc1 8cpsc2 8cpsc3 8cpsc4 8cpsc5 8cpsc6 8cpsc7 8cpsc8 8cpsc9 8cpsc 9cpsc0 9cpsc1 9cpsc2 9cpsc3 9cpsc4 9cpsc5 9cpsc6 9cpsc7 9cpsc8 9cpsc9 9cpsc cpsc0 cpsc1 cpsc2 cpsc3 cpsc4 cpsc5 cpsc6 cpsc7 cpsc8 cpsc9 cpsc

12_34_	120340 120341 120342 120343 120344 120345 120346 120347 120348 120349 12034
	121340 121341 121342 121343 121344 121345 121346 121347 121348 121349 12134
	122340 122341 122342 122343 122344 122345 122346 122347 122348 122349 12234
	123340 123341 123342 123343 123344 123345 123346 123347 123348 123349 12334
	124340 124341 124342 124343 124344 124345 124346 124347 124348 124349 12434
	125340 125341 125342 125343 125344 125345 125346 125347 125348 125349 12534
	126340 126341 126342 126343 126344 126345 126346 126347 126348 126349 12634
	127340 127341 127342 127343 127344 127345 127346 127347 127348 127349 12734
	128340 128341 128342 128343 128344 128345 128346 128347 128348 128349 12834
	129340 129341 129342 129343 129344 129345 129346 129347 129348 129349 12934
	12340 12341 12342 12343 12344 12345 12346 12347 12348 12349 1234