Breaking Crypto

Inleveren

Voor deze weektaak lever je het volgende in:

- Individuele .py-bestanden met de code die je voor de opdrachten hebt gemaakt, voorzien van eigen commentaar
- .pdf-bestand met de code + commentaar van alle opdrachten
- .pdf-bestand met antwoorden op de **dikgedrukte** vragen
- Een .pdf van je lab journal

Instructies voor de weektaak:

- Lever voor elke opdracht de volledige code in! Alleen de uiteindelijke code is hiervoor nodig; bij bv. opdracht 5 hoef je niet voor elke tussenstap een nieuw bestand op te slaan.
- Maak voor elke opdracht gebruik van de bijgeleverde bestanden, en zorg dat je de assert-statements laat staan in de versie de je inlevert.
- De eerste drie opgaven zijn relatief makkelijk en kunnen worden gemaakt door bestaande packages te gebruiken. Doe hier ook vooral je voordeel mee (en lees de bijbehorende documentatie!), anders raak je in de knoop met de tijd. Dit is de warming up voor opgaves 4 en 5, waar je meer zelf zult moeten ontwerpen.
- Elke opgave heeft ook een **dikgedrukte** open vraag. Zorg dat je deze beantwoordt in een apart bestand.
- TIP: Maak alle vragen op volgorde!

1. Base64 codering

Hoewel de eerste cryptografische technieken zoals de Caesar Cipher werkten met letters, opereren moderne cryptografische functies altijd direct op bits. Omdat binair voor mensen niet prettig werkt, kunnen we sleutels en plain- en cipherteksten die we willen printen encoderen in bv. base64. Hierbij worden de bits opgedeeld in blokken van 6 bits, die per blok omgezet worden naar een bijbehorend b64-karakter. Deze week gaan we dit soort encoderingen meermaals gebruiken, dus het is goed om hier even mee te beginnen.

Extra Reading: Geeks for Geeks: encoding & decoding hash strings, Assert Statements

a. **Wat is het verschil tussen b'hello world` en "hello world"?**

b. In bestand *1 - base64.py* staat de functie `string_to_b64()` voor het converteren van strings naar base64-codering. Maak deze functie af. De assert-statements geven aan of je de functie goed hebt geïmplementeerd.
   Hint: bestaat hier al een relevant package voor?

c. Maak ook de functie `b64_to_string()` af.

d. **Kun je encryptie vervangen met encodering? Waarom zou dit wel of niet een probleem zijn?**

e. **BONUS:** Base64 encodering is eigenlijk best simpel! Schrijf zelf een functie die de stappen van base64-encodering implementeert.

2. Binary XOR

Een belangrijke operatie in menig cryptografisch algoritme is de XOR, oftewel de *Exlusive OR*. In opgave 4 zullen we dit ook gaan toepassen, dus maken we nu alvast een functie om dit te kunnen doen.

a. Gegeven de plaintekst byte P=10111010 en de key byte K=10010011, voer de volgende operaties uit op papier:
   $C_A = P$ AND K
   $C_O = P$ OR K
   $C_X = P$ XOR K
   Herhaal nu de operaties door (C_A AND *K), (C_O* OR *K), en (C_X* XOR *K) uit te voeren. Vergelijk het resultaat van de drie operaties;* **wat is er uniek aan de uitkomst van C_X XOR K, en waarom zou dit nuttig zijn voor cryptografie?**

b. Schrijf een functie die twee inputs XOR't door in het bestand *2 – binary XOR.py* de functie `fixed_length_xor()` af te maken.
   Hint: Python heeft hier al een operator *voor. Kijk maar eens naar de documentatie.*

c. De sleutel is zelden precies even groot als de tekst zelf (zoals bij One-Time Pads wel het geval is). Breid nu de functie `repeating_key_xor()` uit zodat vóór het uitvoeren van de XOR de parameter `key` wordt verlengd met (delen van) zichzelf, zodat deze even lang is als de parameter `text`.
   Bijvoorbeeld: de key HAY *is drie tekens lang, en de tekst* HELLOWORLD *is tien tekens lang. In dit geval zou je de sleutel dus moeten uitbreiden naar* HAYHAYHAYH.

3. ECB-mode AES

De *Advanced Encryption Standard*, oftewel AES, is zoals de naam al impliceert dé moderne standaard voor symmetrische (private-key) encryptie. Er zijn echter veel keuzes in de implementatie van AES, zoals de sleutellengte of keuze van de *mode of operation* die je kiest; ECB (Electronic Cipher Block), CBC (Cipher Block Chain), GCM (Galois Counter Mode) – om maar een paar te noemen. Modes of operation zijn niet zomaar een kwestie van persoonlijke voorkeur: een groot deel wordt ondertussen niet meer veilig geacht, zoals bijvoorbeeld ECB.

Extra reading: Stackoverflow, Medium.com

In het bestand *file3.txt* zit een base64-geencodeerde ciphertekst die is versleuteld met AES in ECB mode, met de sleutel `SECRETSAREHIDDEN`.

a. **Op basis van de sleutel kun je bepalen of de ciphertekst is versleuteld met AES-128, AES-192, of AES-256. Welke is het, en hoe weet je dat?**
   Hint: Wat zijn de lengte-eisen voor een AES sleutel? En uit hoeveel bits bestaat een ASCII-karakter?
   NB. Het antwoord op deze vraag zul je nodig hebben voor 3b.

b. Maak in het bestand "3 – ECB Mode AES.py" de functie `ECB_decrypt()` af, en gebruik deze om de inhoud van 3.txt te ontsleutelen. Je kunt de assert statement gebruiken om te controleren of je de opdracht goed hebt gemaakt.

c. **BONUS:** Schrijf ook een functie `ECB_encrypt()`
   Hint: let goed op de volgorde van de/encoderen en ontsleutelen!
   Hint #2: pycryptodome *is handig voor deze opdracht*.

4. CBC-mode

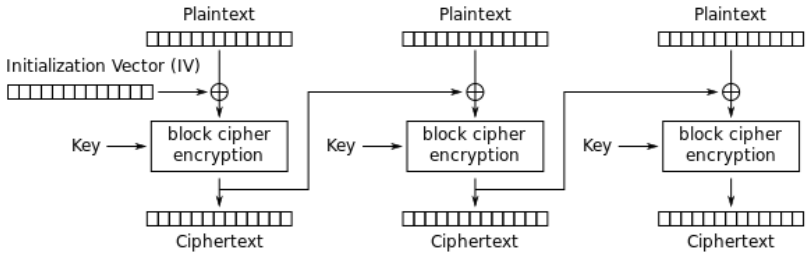
Een grote zwakte van ECB-mode is dat bij een gegeven sleutel hetzelfde plaintekst-blok altijd wordt versleuteld tot hetzelfde ciphertekst-blok. Dat betekent dat het mogelijk is om herhalende blokken te detecteren, en dus uit de ciphertekst informatie over de plaintekst te infereren zonder daarvoor de sleutel nodig te hebben (De befaamde ECB pinguïn is hier een goed voorbeeld van!). CBC mode probeert dit probleem op te lossen door elk blok vóór het versleutelen te XOR'en met de ciphertekst van het vorige blok (het eerste blok wordt ge-XOR'd met een unieke waarde genaamd *Initialization Vector*). Hieronder zie je een schematische weergave van CBC-encryptie en -decryptie.
NB. CBC-mode is weer kwetsbaar voor andere aanvallen, maar is wel aanzienlijk beter dan ECB-mode!

a. **Leg in je eigen woorden duidelijk uit deze extra stap in CBC de bovengenoemde kwetsbaarheid in ECB-mode oplost**

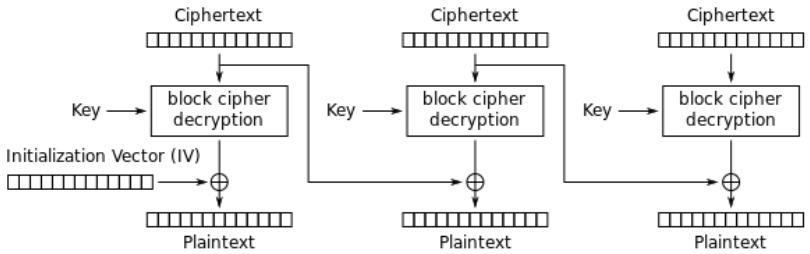
b. Implementeer zelf een versie van CBC-mode door de functie `CBC_decrypt()` in bestand *4 – CBC mode AES.py* af te maken. Maak hiervoor gebruik van `repeating_key_xor()` uit opdracht 2 en `ECB_decrypt()` uit opdracht 3. Je kunt de assert statement gebruiken om te controleren of je functie klopt. De afbeelding hieronder kan handig zijn voor het bepalen van de volgorde van je stappen (let op dat je een decryptie-functie schrijft!)
   Hint: vergeet niet om altijd te opereren op bytes, niet strings of base64-encoded bytes-objects!

c. BONUS: Schrijf ook een functie `CBC_encrypt()`

d. **BONUS:** De IV (Initialization Vector) moet bij elk versleuteld bericht opnieuw gegenereerd worden. Waarom is het een probleem als je maar één keer een IV genereert en die steeds opnieuw gebruikt?



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

5. Adaptive Chosen Plaintext Attack on AES-ECB

Na het implementeren van twee encryptie-technieken sluiten we af met het kraken van één van deze technieken met een zogeheten *Adaptive Chosen-Plaintext Attack* (ACPA). In een ACPA heeft een aanvaller toegang tot een oracle dat plaintekst inputs accepteert en de ciphertekst daarvan teruggeeft. Door nauwkeurig geschreven plainteksten op te stellen kan een aanvaller daarmee informatie uit het oracle vissen. In het bestand *5 – Breaking ECB.py* staat zo'n oracle, namelijk de functie `ECB_oracle()`. Deze functie voegt een geheime tekst toe aan de input-plaintekst, versleutelt dit alles met een 'geheime' sleutel, en geeft de resulterende ciphertekst terug.

Jouw doel is om een functie te schrijven die met een Adaptive Chosen Plaintext Attack de geheime tekst van `ECB_oracle()` ontdekt. Doe dit met de volgende stappen (**Pas het orakel & de sleutel niet aan!**):

NB. Hoeveel bits is een ASCII-karakter ook alweer?

a. Breid de functie `find_block_length()` uit zodat deze de orakel-functie een steeds grotere reeks van dezelfde bytes voert (e.g. "X", dan "XX", "XXX", etc.), zo'n reeks bytes heet **padding**. Doe dit tot je ziet dat het begin van de ciphertekst niet meer verandert; dit levert je de blocksize die de orakel-functie gebruikt.

b. **Leg uit hoe je door de vorige opdracht kon zien wat de blocksize is. Met welke eigenschap van ECB mode heeft dit te maken?**

c. Voer de orakel-functie een padding van een formaat dat één byte kleiner is dan de blocksize (e.g. bij een blocksize van 4 stuur je de bytestring "XXX"). Dit geeft je je "doelciphertekst".
   Hint: Vergeet niet om je blok te vullen met bytes en niet een string!

d. **Wat zal de oracle-functie in de laatste byte van dit blok plaatsen?**
   NB. Het antwoord op deze vraag moet je snappen om de rest te kunnen maken. Je practicumdocent kan al je vragen hierover beantwoorden.

e. Ga nu de ciphertekst van alle mogelijke combinaties van [padding] + [laatste byte] plainteksten bij langs (Als je bloklengte 4 bytes zou zijn, zou je dus de plainteksten "XXXA", "XXXB", "XXXC", etc. proberen) tot je een combinatie vindt die overeen komt met het eerste blok van je doelciphertekst. Je hebt nu de eerste byte van de geheime tekst ontdekt!
   Hint: Hoeveel mogelijke waardes bestaan er voor deze laatste byte? Hoe 'loop' je door al die waardes heen?

f. Herhaal stap c & e voor de resterende bytes, totdat je de geheime tekst hebt gevonden. Doe dit door je padding steeds één byte kleiner te maken en aan te vullen met de ontdekte letter uit de geheime tekst.
   Hint: Je zal je functie per blok te werken moeten laten gaan.

g. **BONUS:** Probeer te bedenken in wat voor omstandigheid dit soort aanval nuttig zou zijn.

Zie volgende pagina voor meer uitleg over hoe deze aanval werkt!

Waarom werkt deze aanval?
Onthoud dat bij ECB de blokken volledig onafhankelijk zijn van elkaar; hierdoor is het mogelijk om de ciphertekst blok voor blok aan te vallen. Stel dat de geheime tekst 'HALLODAAR' is, en de bloklengte 4. Als je de plaintekst XXX aanlevert, zal deze eerst worden aangevuld met de geheime tekst, wat resulteert in de tekst 'XXXHALLODAAR'. Wanneer deze wordt opgedeeld in blokken van vier krijg je de blokken 'XXXH', 'ALLO', en 'DAAR'. Het eerste blok wordt dus *precies* aangevuld met de eerste letter van de geheime tekst. Laten we zeggen dat dit eerste blok de ciphertekst %j3K oplevert. Nu kun je simpelweg gokken wat deze eerste letter is tot je een plaintekst vindt waarbij het eerste blok van de ciphertekst ook %j3K is:

XXXA ---> j7AE | XXXB ---> #(3K | XXXC ---> akf* | XXXD ---> qs0d | XXXE ---> pOK$
XXXF ---> #i8D | XXXG ---> 18ia | XXXH ---> %j3K Succes!

Als je daarna de plaintekst XX invoert wordt het eerste blok aangevuld met de eerste 2 letters van de geheime tekst, HA, en wordt het eerste blok XXHA. Omdat je weet dat de eerste letter H is kun je de plainteksten XXHA, XXHB, XXHC, XXHD, etc. proberen totdat je weer de juiste ciphertekst tegenkomt! Dit proces kun je herhalen tot je de ciphertekst hebt. Wat nou als de geheime tekst langer is dan één blok? Eigenlijk verandert er dan maar weinig; je zult alleen even goed moeten nadenken over hoe je controleert of je de juiste letter hebt gevonden.