Piotr Szewczyk - CryptoHack_Challenges walkthrough part. 3 (wersja PL)

AES

SYMMETRIC CIPHERS

1. Keyed Permutations

   W tym zadaniu przedstawiono podstawową ideę szyfru blokowego AES jako "keyed permutation" - permutacji zależnej od klucza.Oznacza to, że dla każdej możliwej wartości klucza AES wyznacza unikalne odwzorowanie (permutację) między przestrzenią wszystkich możliwych bloków wejściowych (plaintextów) a bloków wyjściowych (ciphertextów). Ważną właściwością tej operacji jest jednoznaczność odwzorowania: każdemu blokowi wejściowemu odpowiada dokładnie jeden blok wyjściowy i odwrotnie. Dzięki temu możliwe jest odwracanie szyfrowania (deszyfrowanie). Matematyczny termin opisujący taką jednoznaczną zależność między elementami dwóch zbiorów to **bijection (bijekcja).**

2. Resisting Bruteforce

   Najlepszym atakiem na AES przy użyciu jednego klucza jest atak **biclique** - zmniejsza on nieco skuteczność ochrony AES-128 (do około 126,1 bitów), ale nie stanowi praktycznego zagrożenia.

3. Structure of AES

   AES-128 operuje na blokach 128-bitowych (16 bajtów), które są reprezentowane jako macierz 4×4 bajtów.

   Kolejne bajty tekstu są wypełniane wierszami macierzy (tzw. row-major order). W kodzie znajdują się dwie funkcje:

- bytes2matrix() - konwertuje 16 bajtów na macierz 4×4,
- matrix2bytes() - wykonuje operację odwrotną, czyli łączy macierz z powrotem w 16 bajtów.

  Wynik:

  crypto{inmatrix}

4. Round Keys

   Krok AddRoundKey jest jedynym etapem AES, w którym klucz jest faktycznie mieszany z danymi.
   Polega on na bitowym XOR między aktualnym stanem (state) a kluczem rundy (round key).

   Fragment kodu:

   ```
   def add_round_key(s, k):
     return [[s[i][j] ^ k[i][j] for j in range(4)] for i in range(4)]
   ```

   Operacja XOR jest odwracalna, co oznacza, że ten sam kod można wykorzystać zarówno do szyfrowania, jak i deszyfrowania.

   Po zastosowaniu add_round_key oraz konwersji do bajtów (matrix2bytes) uzyskano:

   crypto{r0undk3y}

5. Confusion through Substitution

   Krok SubBytes wprowadza do AES element nieliniowości, który zapewnia confusion - utrudnienie matematycznego powiązania klucza z szyfrogramem. Każdy bajt stanu jest zastępowany wartością z tablicy S-Box (Substitution Box), będącej wynikiem funkcji nieliniowej opartej na działaniu w polu Galois $GF(2^8)$. Odwrotna operacja (używająca *inv_s_box*) jest wykorzystywana podczas deszyfrowania.

   Fragment kodu:

   ```
   def sub_bytes(s, sbox):
     return [[sbox[byte] for byte in row] for row in s]
   ```

   Po zastosowaniu odwrotnego S-Boxa (inv_s_box) do podanej macierzy otrzymano:

   crypto{l1n34rly}

6. Diffusion through Permutation

   Dwa kolejne etapy AES - ShiftRows i MixColumns - są odpowiedzialne za zapewnienie właściwości diffusion (rozproszenia informacji).
   Ich celem jest to, aby każda zmiana jednego bajtu w plaintextcie wpływała na wiele bajtów w ciphertextcie po kilku rundach.

- ShiftRows - przesuwa kolejne wiersze macierzy o odpowiednią liczbę bajtów:
  - 1. wiersz – bez zmian
  - 2. wiersz – w lewo o 1
  - 3. wiersz – w lewo o 2
  - 4. wiersz – w lewo o 3

- *inv_shift_rows* = wykonuje przesunięcia w odwrotnym kierunku (w prawo).
  Fragment kodu dla odwrotnej operacji:

```
def inv_shift_rows(s):
  s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
  s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
  s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]
```

   MixColumns miesza kolumny poprzez mnożenie ich przez stałą macierz w ciele $GF(2^8)$, dzięki czemu wszystkie bajty kolumny wpływają na siebie nawzajem.
   Odwrotna wersja *inv_mix_columns* stosuje transformację odwracalną.
   Po wykonaniu *inv_mix_columns* i *inv_shift_rows* uzyskano wynik:

   crypto{d1ffUs3R}

7. Bringing It All Together

   W ramach zadania polegającego na implementacji pełnego procesu deszyfrowania AES-128 stworzono kod zawierający funkcje generowania kluczy rundowych oraz operacje odwrotne przekształceń AES. Proces deszyfrowania rozpoczął się od rozwinięcia klucza głównego do jedenastu kluczy rundowych, po czym wykonano początkową operację AddRoundKey z ostatnim kluczem rundowym. Następnie przeprowadzono dziewięć głównych rund, w każdej wykonując kolejno operacje InvShiftRows, InvSubBytes, AddRoundKey

i InvMixColumns. Ostatnia runda została zrealizowana bez przekształcenia InvMixColumns. Końcowy stan macierzy przekonwertowano na postać bajtową, uzyskując po deszyfrowaniu zaszyfrowanego tekstu flagę:

crypto{MYAES128}

SYMMETRIC STARTER

1. Modes of Operation Starter

   Cel zadania
   Celem jest zrozumienie, jak działają tryby pracy AES oraz jakie zagrożenia wynikają z niepoprawnego udostępniania API. Aplikacja szyfruje flagę AES-ECB i jednocześnie pozwala odszyfrować dowolny ciphertext tym samym kluczem.

   Wykorzystana podatność
   Publiczny endpoint decrypt() tworzy oracle deszyfrujący. Wystarczy pobrać ciphertext flagi i przekazać go do decrypt(), aby poznać plaintext.

   Wynik:
   crypto{bl0ck_c1ph3r5_4r3_f457_!}

   Wnioski:
   Nigdy nie wolno wystawiać publicznego deszyfrowania dla tego samego klucza co szyfrowanie. ECB nie zapewnia bezpieczeństwa strukturalnego.

2. Passwords as Keys

   Cel zadania

   Pokazanie, dlaczego hasła nie mogą być kluczami kryptograficznymi bez użycia mocnych KDF. Klucz AES tworzony był jako MD5 hasła z wordlisty.

   Atak OFFLINE

   1. Pobranie ciphertextu.

2. Wczytanie słownika.

3. Dla każdego słowa: MD5 → AES-ECB → sprawdzenie, czy plaintext zaczyna się od crypto{.

Wynik: FLAG = **crypto{bl0ck_c1ph3r5_4r3_f457_!}**

Wnioski

MD5 jest szybkie → łatwe do brute-force. Klucze oparte na hasłach muszą korzystać z KDF (PBKDF2, scrypt, Argon2).

BLOCK CIPHERS 1

1. ECB CBC WTF

Cel zadania
Pokazanie różnicy między trybami ECB i CBC oraz skutków błędów implementacyjnych. Serwer szyfrował flagę w trybie CBC, ale udostępniał dodatkowy endpoint decrypt() działający w trybie ECB tym samym kluczem.

Wykorzystana podatność
Podatność polega na tym, że można:

1. Pobierać ciphertext flagi szyfrowanej w CBC.
2. Przekazać dokładnie ten ciphertext do endpointu decrypt().
3. Endpoint odszyfruje każdy blok oddzielnie w trybie ECB.

Ponieważ decryption oracle ujawnia „surowe” D(E(C_i)), możliwe jest późniejsze wykonanie pełnego odwrócenia CBC lokalnie:

P_i = D(C_i) XOR C_{i-1}

Wynik: plaintext flagi został odzyskany. **crypto{3cb_5uck5_4v01d_17_!!!!!}**

Wniosek: mieszanie trybów szyfrowania jest niebezpieczne — różne API nie mogą używać tego samego klucza w różnych trybach.

2. ECB ORACLE
   Cel zadania
   Zobrazowanie klasycznego ataku „byte-at-a-time” na AES-ECB.
   Serwer szyfruje: AES_ECB(PLAINTEXT || SECRET_FLAG).
   Można dostarczyć własny prefix, dzięki czemu możliwe jest wypozycjonowanie bajtów flagi w określonym miejscu bloku.

   Wykorzystana podatność
   ECB szyfruje każdy blok niezależnie, więc identyczne bloki plaintextu prowadzą do identycznych bloków ciphertextu. Umożliwia to:

   1. Kontrolowanie pierwszego bloku przez dopasowanie długości prefixu.
   2. Zgadywanie bajtu flagi: porównując ciphertext naszego zgadywanego bloku z ciphertextem rzeczywistym.
   3. Rekonstrukcję flagi bajt po bajcie. **crypto{p3n6u1n5_h473_3cb}**

   Wniosek: ECB absolutnie nie nadaje się do szyfrowania sekretów, jeśli atakujący może kontrolować część plaintextu.

3. FLIPPING COOKIE
   Cel zadania
   Pokazanie ataku „bit-flipping” w CBC, który umożliwia zmianę zaszyfrowanych danych bez znajomości klucza.

   Opis podatności
   W CBC obowiązuje:

   P_1 = D(C_1) XOR IV

   Oznacza to, że modyfikacja IV wpływa *dokładnie* na pierwszy blok plaintextu.
   Serwer zwraca cookie zakodowane jako:

   admin=False;expiry=...

   A celem jest uzyskanie:

   admin=True;

   Rozwiązanie:

1. Pobranie cookie oraz IV.
2. XOR:
   IV XOR (oryginalny tekst) XOR (tekst, jaki chcemy wymusić)
3. Przesłanie sfałszowanego IV → serwer odczytuje poprawny plaintext z ustawioną wartością admin=True.

crypto{4u7h3n71c4710n_15_3553n714l}

Wniosek: poprawne użycie CBC wymaga stosowania MAC (HMAC / AES-GCM) - sam CBC nie gwarantuje integralności.

4. Lazy CBC

   Cel zadania

   Pokazanie, jak błędne działanie CBC po stronie serwera prowadzi do całkowitego złamania klucza.

   Opis podatności

   Serwer:

   1. Implementuje CBC niepoprawnie.
   2. W przypadku nieprawidłowego plaintextu zwraca go w treści błędu (!).
   3. Można celowo skonstruować ciphertext, który po stronie serwera zdeszyfruje się w sposób kontrolowany:

   C0 || 0...0 || C0

   Po odszyfrowaniu:

   D(C0) XOR IV
   D(0...0) XOR C0
   D(C0) XOR C1

   Po porównaniach otrzymuję różnicę P0 XOR P2, która ujawnia klucz (znany błąd implementacyjny). **crypto{50m3_p30pl3_d0n7_7h1nk_IV_15_1mp0r74n7_?}**

   Wniosek: błędne API, które ujawnia plaintext w błędach, prowadzi do natychmiastowego złamania systemu.

5. Triple DES

Cel zadania
Pokazać problem „kluczy słabych” oraz konstrukcji 2-key 3DES.

Opis podatności
Szyfrowanie 3DES (EDE) z kluczem:

K = (00...00 || FF...FF)

prowadzi do błędów wynikających z:

1. Efektów redukcji klucza (weak keys)
2. Struktury 2DES/3DES podatnej na meet-in-the-middle

Atak:
1. Uzyskano ciphertext flagi zaszyfrowany 3DES.
2. Następnie zaszyfrowano flagę ponownie tym samym kluczem.
3. Wynik zdradza właściwości słabego klucza oraz strukturę 3DES.

Wniosek: Triple DES ma znane słabości konstrukcyjne i nie powinien być używany. **crypto{n0t_4ll_k3ys_4r3_g00d_k3ys}**

STREAM CIPHERS

1. Symmetry

   Cel zadania: Demonstracja ataku na tryb AES-OFB (działający jak szyfr strumieniowy), w którym atakujący może zaszyfrować dowolny tekst tym samym wektorem inicjującym (IV), który został użyty do zaszyfrowania sekretu.
   Podatność: Tryb OFB (Output Feedback) generuje strumień klucza (keystream) zależny od klucza i IV. Jeśli ten sam IV zostanie użyty do zaszyfrowania dwóch różnych wiadomości (P1 i P2), to:
   C1 = P1 XOR Keystream
   C2 = P2 XOR Keystream
   Przekształcenie:
   C1 XOR C2 = (P1 XOR Keystream) XOR (P2 XOR Keystream)
   C1 XOR C2 = P1 XOR P2
   ponieważ strumień klucza redukuje się (Keystream XOR Keystream = 0).
   Atak:
   - Pobrano zaszyfrowaną flagę (C_flag) oraz jej IV z publicznego endpointu encrypt_flag.
   - Wykorzystano drugi publiczny endpoint (encrypt), aby zaszyfrować wiadomość złożoną z samych bajtów zerowych (P_chosen) przy użyciu tego samego IV, otrzymując C_chosen.
   - Obliczono C_flag XOR C_chosen, co jest równe P_flag XOR P_chosen.
   - Ponieważ P_chosen składa się z samych zer, wynik operacji XOR daje bezpośrednio P_flag.

   Wynik:

   crypto{0fb_15_5ymm37r1c4l_!!!11!}

2. Bean Counter

   Cel zadania: Demonstracja ataku "known-plaintext" (ze znanym tekstem jawnym) przeciwko niestandardowemu szyfrowi strumieniowemu (implementującemu logikę CTR za pomocą ECB), który ponownie wykorzystuje krótki strumień klucza.

   Podatność: Szyfr wykonuje operację XOR na tekście jawnym z keystreamem, który jest krótszy niż wiadomość i powtarza się cyklicznie. Jest to wariant ataku na tzw. "running key cipher".

Atak:

- Zidentyfikowano znany fragment tekstu jawnego – standardowy 16-bajtowy nagłówek pliku PNG (PNG_PREFIX).
- Poprzez operację XOR na tym prefiksie oraz pierwszych 16 bajtach szyfrogramu (C[0:16] XOR P[0:16]) odzyskano 16-bajtowy strumień klucza (Keystream).
- Wykorzystano odzyskany Keystream do odszyfrowania całego szyfrogramu, zakładając, że keystream powtarza się cyklicznie na całej długości wiadomości.
- Zdeszyfrowane bajty zapisano jako plik PNG, który zawierał flagę.

Wynik:
crypto{hex_bytes_beans}

3. CTRIME

Cel zadania: Przeprowadzenie ataku typu CRIME (Compression Ratio Info-leak Made Easy) na endpoint szyfrujący, który kompresuje dane przed szyfrowaniem.

Podatność: Serwer stosuje kompresję (zlib) na tekście jawnym zawierającym dane kontrolowane przez atakującego oraz sekretną flagę, a następnie szyfruje wynik w trybie AES-CTR. Długość szyfrogramu ujawnia stopień kompresji. Lepsza kompresja (czyli krótszy szyfrogram) pojawia się wtedy, gdy atakujący odgadnie poprawny fragment sekretu, umożliwiając algorytmowi kompresji znalezienie powtórzeń.

Atak:

- Zaimplementowano pętlę typu "byte-at-a-time" zaczynając od znanego prefiksu crypto{.
- W każdej iteracji wysyłano do serwera dwie wersje danych:
  – jedną ze znanym prefiksem i znakiem-wypełniaczem (np. flag + "*")
  – drugą ze znanym prefiksem i testowaną literą alfabetu (np. flag + c)
- Porównywano długości zwróconych szyfrogramów. Jeżeli szyfrogram dla testowanej litery był krótszy niż dla wersji z wypełniaczem, oznaczało to, że litera jest poprawna (spowodowała lepszą kompresję).
- Poprawną literę dołączano do budowanego prefiksu i proces powtarzano dla kolejnych pozycji, aż do odzyskania całej flagi.

Wynik: **crypto{CRIME_571ll_p4y5}**

4. Logon Zero

   Cel zadania: Wykorzystanie błędu logiki w systemie resetowania hasła i uwierzytelniania w protokole opartym na CFB-8.
   Podatność: Aplikacja sieciowa umożliwiała zresetowanie hasła przy użyciu tokenu. Odkryto, że przesłanie tokenu składającego się z 28 bajtów zerowych (\x00 * 28) powodowało ustawienie hasła użytkownika na pusty ciąg znaków.
   Atak:
   - Nawiązano połączenie z serwerem.
   - W pętli wysyłano polecenie reset_password z tokenem wypełnionym zerami.
   - Bezpośrednio po resecie wysyłano polecenie authenticate z pustym hasłem ("").
   - Serwer, po odebraniu reset_password z zerowym tokenem, akceptował uwierzytelnienie przy użyciu pustego hasła i zwracał flagę.

   Wynik: **crypto{Zerologon_Windows_CVE-2020-1472}**

5. Stream of Consciousness

   Cel zadania: Demonstracja ataku na szyfr strumieniowy, który wielokrotnie używa tego samego strumienia klucza (atak typu "many-time pad").
   Podatność: Serwer szyfrował wiele różnych wiadomości (w tym flagę) przy użyciu tego samego klucza i statycznego licznika/nonce (AES-CTR). Inicjalizacja Counter.new(128) odbywała się poza funkcją encrypt, co powodowało ponowne wykorzystanie tego samego strumienia klucza (keystream) przy każdym wywołaniu.
   Atak:
   - Pobrano 100 szyfrogramów z serwera i odfiltrowano je do 22 unikalnych.
   - Zastosowano technikę "crib dragging": założono, że flaga zaczyna się od znanego ciągu b'crypto{'.
   - Przetestowano to założenie dla każdego szyfrogramu Ci, obliczając:
     Pj = (Ci XOR Cj) XOR P_i_crib
     dla wszystkich pozostałych j.
   - Jeśli wszystkie wynikowe Pj zawierały drukowalne znaki, założenie było poprawne i identyfikowano szyfrogram flagi.

- Kontynuowano zgadywanie kolejnych "cribów" w innych częściowo odszyfrowanych wiadomościach, stopniowo odzyskując cały keystream i wszystkie teksty jawne, w tym pełną flagę.

Wynik: **crypto{k3y57r34m_r3u53_15_f474l}**

6. Dancing Queen

Cel zadania: Implementacja ataku typu "known-plaintext" na niestandardową implementację szyfru strumieniowego ChaCha20 w celu odzyskania klucza.
Podatność: Posiadanie jednej pary (tekst jawny, szyfrogram) zaszyfrowanej ChaCha20 umożliwia odtworzenie klucza, jeśli dostępne są odwrotne przekształcenia szyfru. ChaCha20 generuje blok strumienia klucza (Keystream_Block) poprzez 10 rund przetwarzania stanu początkowego.
Atak:

- Zaimplementowano pełną strukturę szyfru ChaCha20, w tym funkcje _inner_block, _quarter_round oraz operacje odwrotne: _inner_block_inv oraz _quarter_round_inv.
- Obliczono pierwszy 64-bajtowy blok keystreamu, korzystając z relacji:
  Keystream = P_known XOR C_known
  gdzie P_known i C_known to dostarczona wiadomość i jej szyfrogram.
- Otrzymany Keystream potraktowano jako końcowy stan po 10 rundach szyfrowania.
- Zastosowano 10 odwrotnych rund (_inner_block_inv) na tym stanie, aby odzyskać stan początkowy.
- Ze stanu początkowego, który ma ustalony format (stałe, klucz, licznik, nonce), wyodrębniono 32-bajtowy klucz główny.
- Użyto odzyskanego klucza oraz drugiego podanego IV (iv2) do odszyfrowania właściwego szyfrogramu flagi (flag_enc).

Wynik: **crypto{M1x1n6_r0und5_4r3_1nv3r71bl3!}**

7. Oh SNAP

Cel zadania: Przeprowadzenie ataku statystycznego na implementację RC4, wykorzystując słabości algorytmu KSA (Key Scheduling Algorithm).
Podatność: Serwer używał RC4 z kluczem w postaci konkatenacji długiego nonce kontrolowanego przez atakującego oraz nieznanej flagi (key = nonce || flag). Słabości KSA RC4 powodują, że pierwsze bajty strumienia klucza są statystycznie skorelowane z bajtami klucza.

Atak:

- Zaimplementowano funkcję PC4 symulującą częściowe wykonanie KSA RC4 na podstawie znanego prefiksu klucza (nonce + stopniowo odgadywana flaga). Funkcja generowała przybliżony strumień klucza.
- Zebrano 120 próbek (NDATA) z serwera = każda zawierała losowy nonce i odpowiadające mu pierwsze 32 bajty rzeczywistego keystreamu (NKS).
- Rozpoczęto atak „byte-at-a-time", przyjmując jako prefiks crypto{.
- Dla każdego kandydata na kolejny bajt flagi (guess) wykonywano:
  - symulację PC4 dla wszystkich 120 próbek,
  - używając klucza nonces[i] + known_flag + guess.
- Zliczano przypadki, w których przybliżony keystream z symulacji pokrywał się z rzeczywistym strumieniem klucza z serwera („trafienia").
- Bajt guess z najwyższą liczbą trafień uznawano za poprawny i dopisywano go do odzyskiwanej flagi.

Wynik: **crypto{w1R3d_equ1v4l3nt_pr1v4cy?!}**