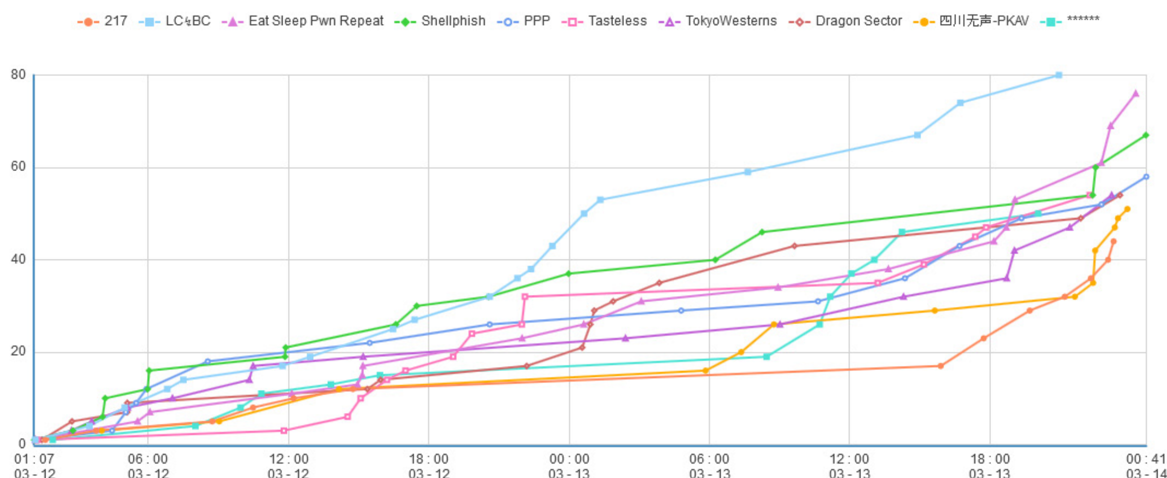
0CTF 2016 Quals – People's Square

0CTF to zawody organizowane przez 0ops – najlepszą chińską drużynę według rankingu CTFtime.org. Tegoroczny finał w Szanghaju był poprzedzony internetowymi kwalifikacjami, w których wzięły udział 874 drużyny z całego świata. Kwalifikacje wygrała drużyna LC↯BC będąca fuzją 2 mocnych rosyjskich zespołów, polscy liderzy z Dragon Sector uplasowali się na miejscu 7., a nasz zespół zajął miejsce 13. W trakcie zawodów trafiliśmy na ciekawy problem kryptograficzny, którego rozwiązaniem postanowiliśmy podzielić się z czytelnikami.



CTF	0CTF 2016 Quals https://ctf.0ops.net
Waga CTFtime.org	41,07 (https://ctftime.org/event/262)
Liczba drużyn (z niezerową liczbą punktów)	874
System punktacji zadań	Od 1 punktu (bardzo proste) do 10 punktów (trudne)
Liczba zadań	26
Podium	1. LC↯BC (Rosja) – 80 pkt. 2. Eat Sleep Pwn Repeat (Niemcy) – 76 pkt. 3. Shellphish (Stany Zjednoczone) – 67 pkt.
Zadanie	People's Square (Crypto 6p)

O ZADANIU

Mimo że zadanie było zaliczone do kategorii Kryptografia, to w materiałach, które zostały do niego dostarczone, znajdowała się tylko aplikacja szyfrująca w postaci skompilowanego pliku binarnego (co sugerowałoby zadanie z kategorii Reverse Engineering). W związku z tym pierwsza część zadania opierała się o inżynierię wsteczną i analizę dostarczonego programu. Warto pochwalić autorów za przygotowanie 3 wersji aplikacji – pod Windows, pod Linuxa oraz pod OS X, aby ułatwić debugowanie. Niestety w naszym przypadku okazało się to utrudnione, ponieważ aplikacja korzystała z nowych instrukcji Asemblera, niewspieranych przez nasze domowe komputery, a tym samym pozostawała nam jedynie statyczna analiza zdekompilowanego kodu.

Część kryptograficzna zadania była związana z algorytmem AES (Advanced Encryption Standard), który jest jednym z najpopularniejszych szyfrów symetrycznych i generalnie jest uważany za bezpieczny. W przypadku prezentowanego zadania algorytm był lekko zmodyfikowany, co pozwalało na odzyskanie klucza szyfrującego i zdekodowanie flagi.

W ramach materiałów do zadania otrzymaliśmy logi z działania aplikacji, które zawierały zbiór 2048 zaszyfrowanych tekstów oraz zaszyfrowaną flagę do zadania:

```
ciphertext for 0 is:
6b f9 8e 66 17 db d1 1e 60 19 08 4d b1 14 4e 2f
ciphertext for 1 is:
9d 25 dc dc ce 3e a9 72 f9 12 89 fe c8 52 71 0b
Correct!
( ... i tak 1024 razy ... )
Now I will give you the flag:
af 93 ce ae 1f 1e 7a 13 26 d6 05 51 97 3c 46 1b c9 b1 56 9c 2c
df d5 5a c6 ca 33 46 31 fb 19 73
```

ANALIZA WSTECZNA APLIKACJI SZYFRUJĄCEJ

Po dekompilacji dostarczonego programu analizę rozpoczęliśmy od samej procedury szyfrującej, spodziewając się podatności właśnie w tym miejscu:

```
const __m128i *__fastcall sub_4013C5(const __m128i *a1, __int64 a2) {
  __m128i v2;
  __m128i v3;
  __m128i v6;
  const __m128i *result;
  signed __int64 i;
  __int128 v11;
  __int128 v12;
  __int128 v13;
```

```
  __int128 v14;
  __int128 v15;
  __int128 v16;
  __int128 v17;
  __int128 v18;

  v2 = _mm_load_si128((const __m128i *)a2);
  _mm_store_si128((__m128i *)&v12, _mm_loadu_si128(a1));
  _mm_store_si128((__m128i *)&v13, v2);
  _mm_store_si128(
    (__m128i *)&v11,
    _mm_xor_si128(_mm_load_si128((const __m128i *)&v13),
      _mm_load_si128((const __m128i *)&v12)));
  for ( i = 1LL; (unsigned __int64)i <= 3; ++i ) {
    v3 = _mm_load_si128((const __m128i *)(16 * i + a2));
    _mm_store_si128((__m128i *)&v14, _mm_load_si128((const __
    m128i *)&v11));
    _mm_store_si128((__m128i *)&v15, v3);
    _XMM0 = _mm_load_si128((const __m128i *)&v14);
    __asm { aesenc  xmm0, [rbp+var_40] }
    _mm_store_si128((__m128i *)&v11, _XMM0);
  }
  v6 = _mm_load_si128((const __m128i *)(a2 + 64));
  _mm_store_si128((__m128i *)&v16, _mm_load_si128((const __m128i
  *)&v11));
  _mm_store_si128((__m128i *)&v17, v6);
  _XMM0 = _mm_load_si128((const __m128i *)&v16);
  __asm { aesenclast xmm0, [rbp+var_20] }
  _mm_store_si128((__m128i *)&v11, _XMM0);
  _mm_store_si128((__m128i *)&v18, _mm_load_si128((const __m128i
  *)&v11));
  result = a1;
  _mm_storeu_si128((__m128i *)a1, _mm_load_si128((const __m128i
  *)&v18));
  return result;
}
```

Kod składa się jedynie z wczytywania zmiennych, jednej operacji XOR oraz instrukcji Asemblera `aesenc` oraz `aesenclast` (sprzętowej implementacji AESa dostępnej dla niektórych procesorów).

Po długim upraszczaniu doprowadziliśmy powyższy kod do postaci:

```
void realEncrypt(uint128_t *data, uint128_t *key) {
  uint128_t state = data ^ key[0];
  for (int i = 1; i < 4; ++i) {
    state = aesenc(state, key[i];
  }
  *data = aesenclast(state, key[4]);
}
```

Ten kod to faktycznie poprawnie zaimplementowany AES. Różnica między przedstawionym w zdekompilowanym kodzie algorytmem a jego standardową (bezpieczną) wersją to liczba rund szyfrowania. Dla 128-bitowego klucza standardowo używa się 10 rund, podczas gdy w naszym kodzie widzimy, że rundy są tylko 4 (warunek `i < 4`).

Po dekompilacji i uproszczeniu funkcji `main` aplikacji uzyskujemy (bardzo skrócona wersja):

```
int main() {
  uint64_t key;
  char ciphertextFor0[16];
  char ciphertextFor1[16];
  char encryptedFlag[32];
  putEncryptedFlagIntoBuffer(&encryptedFlag);
  uint64_t v10 = 0;
  sub_400A74(&v10);
  generateKey(&v10, &key);
  uint64_t v6 = 0;
  uint64_t initTime = time(0);
  for (int i = 0; i < 1024; i++) {
    memsetAndEncrypt(&key, &ciphertextFor0, &ciphertextFor1, i,
    initTime);
    unsigned int randomBit = rand() & 1;
    char *v1;
    if (randomBit) {
      v1 = &ciphertextFor0;
    } else {
      v1 = &ciphertextFor1;
    }
    hexdump(v1, 0x10);
    puts("0 or 1?");
    user_bit = getchar() - 48;
    puts("ciphertext for 0 is: ");
    hexdump(&ciphertextFor0, 0x10);
    puts("ciphertext for 1 is: ");
    hexdump(&ciphertextFor1, 0x10);
    if ( user_bit == randomBit ) {
      puts("Correct!");
      ++v6;
    }
    else {
      puts("Incorrect!");
    }
  }
  if ( v6 == 1024 ) {
    puts("Now I will give you the flag:");
    realEncrypt(&encryptedFlag[0], &key);
    realEncrypt(&encryptedFlag[16], &key);
    hexdump(&encryptedFlag, 32);
  }
  return 0;
}
```

W kodzie widzimy, że główna pętla wykonywana jest 1024 razy, a w każdym jej obrocie szyfrowane są 2 zestawy danych. Następnie na ekranie wyświetlany jest zaszyfrowany jeden z zestawów, a użytkownik udziela odpowiedzi, czy jest to zestaw szyfrowany z parametrem 0, czy 1. Jeśli użytkownik udzieli samych poprawnych odpowiedzi, aplikacja wyświetla zaszyfrowaną flagę. Log z takiego właśnie wykonania został dostarczony razem z zadaniem.

Przeanalizujmy teraz, jak wyglądają szyfrowane zestawy danych:

```
void memsetAndEncrypt(uint64_t key, char ciphertext0[16],
    char ciphertext1[16], uint64_t iter,
    uint64_t initTime) {
  memset(ciphertext0, 0, 0x10uLL);
  memset(ciphertext1, 1, 0x10uLL);
  memcpy((char *)ciphertext0 + 8,  &iter,     4uLL);
  memcpy((char *)ciphertext1 + 8,  &iter,     4uLL);
  memcpy((char *)ciphertext0 + 12, &initTime, 4uLL);
  memcpy((char *)ciphertext1 + 12, &initTime, 4uLL);
  realEncrypt(ciphertext0, key);
  realEncrypt(ciphertext1, key);
}
```

Oba zestawy danych mają po 16 bajtów. Z kodu wynika, że pierwszy zestaw danych ma format:

0x0101010101010101	i	initTime

A drugi zestaw:

0x0000000000000000	i	initTime

Parametr `initTime` jest stały dla wszystkich zestawów, więc zestawy z 1 różnią się między sobą jedynie licznikiem pętli i tak samo zestawy z 0 różnią się jedynie licznikiem pętli. Będzie to bardzo istotne podczas selekcji danych do ataku kryptograficznego.

SZYFROWANIE ALGORYTMEM AES

Żeby zrozumieć przebieg ataku, konieczne jest wyjaśnienie, w jaki sposób działa szyfrowanie AES.

Algorytm jest szyfrem blokowym i działa na blokach rozmiaru 16 bajtów. Oznacza to, że dane wejściowe są dzielone na 16 bajtowe fragmenty, które następnie są kodowane jeden po drugim. AES ma kilka różnych trybów działania, które definiują, w jaki sposób postępować z danymi złożonymi z wielu bloków, niemniej w naszym przypadku wszystkie zestawy danych mają dokładnie 16 bajtów, więc ten problem nas nie dotyczy.

KEYLIGHT

KURSY MULTIMEDIALNE

Modelowanie gada z BLENDER'em

[LOW POLY]

WWW.KEYLIGHT.COM.PL

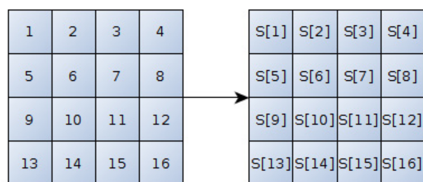
Modelowanie humanoida z

BLENDER'em

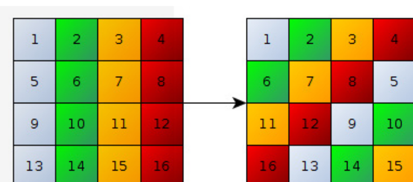
[LOW POLY]

AES operuje na wspomnianych 16 bajtowych blokach jako tablicy 4x4 i wykonuje na nich 4 operacje:

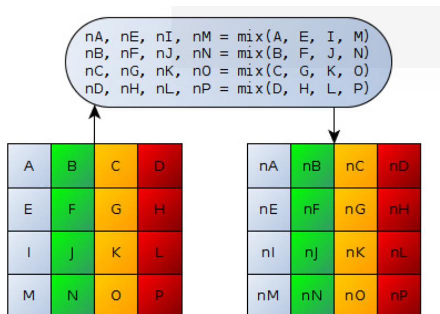
› `SubBytes` – każdy bajt w macierzy jest zamieniany na inny wybrany bajt z macierzy, korzystając z ustalonej tablicy asocjacyjnej (tzw. substitution box):



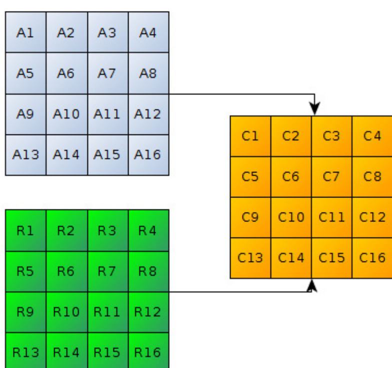
› `ShiftRows` – macierz jest przesuwana w wierszach o numer wiersza pozycji w lewo:



› `MixColumns` – wartości macierzy w kolumnach są mieszane (konkretnie mnożone przez pewien wielomian w skończonym ciele). Ważne jest to, że ostateczne wartości w wynikowej komórce tablicy zależą tylko od innych wartości w tej samej kolumnie:



› `AddRoundKey` – każda wartość w macierzy jest xorowana z wartością z klucza rundy (round key). Czyli zostaje wykonana operacja `C = A xor R`:



Nie będziemy opisywać szczegółowo algorytmu generacji klucza rundy, ponieważ nie jest on istotny dla naszego ataku, niemniej należy wspomnieć, że jest to algorytm deterministyczny i odwracalny. Znając klucz rundy, jesteśmy w stanie wyliczyć cały klucz szyfrowania.

WYJAŚNIENIE PODATNOŚCI I WEKTORA ATAKU

Opisywany tutaj atak można odszukać w literaturze pod hasłem „square attack" lub „integral cryptanalysis".

Załóżmy, że możemy uzyskać szyfrogramy dla 256 wybranych tekstów. Każdy z wybranych tekstów jest przygotowany tak, że wszystkie bajty są stałe, poza jednym, który przyjmuje po kolei wszystkie możliwe wartości, na przykład:

```
00 00 00 00 00 ... 00
01 00 00 00 00 ... 00
02 00 00 00 00 ... 00
03 00 00 00 00 ... 00
04 00 00 00 00 ... 00
..
FE 00 00 00 00 ... 00
FF 00 00 00 00 ... 00
```

Rozważmy, jak będą wyglądały te bloki podczas szyfrowania AESem. Przed rozpoczęciem szyfrowania wszystkie szyfrogramy wyglądają tak:

A	C	C	C
C	C	C	C
C	C	C	C
C	C	C	C

Gdzie C oznacza bajty, które są stałe na danej pozycji dla każdego tekstu, a A oznacza, że bajt przyjmuje wszystkie możliwe wartości (od 0 do 255), dla kolejnych tekstów. Dla podanego wyżej przykładu wszystkie bajty C będą miały wartość 0, a bajt A będzie przyjmował wartości 00, 01, 02, 03, ..., FE, FF.

Rozpoczynamy pierwszą rundę szyfrowania. Po wykonaniu operacji `SubBytes` tablica będzie miała postać:

A	C	C	C
C	C	C	C
C	C	C	C
C	C	C	C

Jak widać, nic się nie zmieniło – stałe bajty pozostały sobie równe, a pierwszy bajt dalej przyjmuje wszystkie możliwe wartości. Nie znaczy to, że `SubBytes` nie wykonało żadnej operacji, ale że wszystkie bajty zostały zmienione w taki sam sposób.

Każdy stały bajt `C` został zamieniony na tą samą wartość `S[C]`, a kolejne wartości dla A przeiterują po wszystkich wartościach tablicy asocjacyjnej S, więc przyjmą wszystkie możliwe wartości.

Następna jest operacja `ShiftRows`:

A	C	C	C
C	C	C	C
C	C	C	C
C	C	C	C

Podobnie jak wcześniej – komórki przechodzą z miejsca w miejsce, ale w naszej schematycznej tabeli wszystko pozostaje bez zmian – dalej

wszystkie bajty poza jednym są identyczne dla każdego szyfrogramu. To zmienia się po następnym kroku, `MixColumns`:

A	C	C	C
A	C	C	C
A	C	C	C
A	C	C	C

Tutaj już wartości C nie będą takie same, ale będą nadal takie same dla każdego z 256 testowanych tekstów na odpowiadających pozycjach. Analogicznie bajty oznaczone jako A będą znowu przyjmowały wszystkie możliwe wartości (czyli np. pierwszy bajt w drugim wierszu przyjmie wartość 1 dokładnie dla 1 szyfrowanego tekstu, wartość 2 dokładnie dla 1 szyfrowanego tekstu itd.).

Po ostatniej operacji w tej rundzie, AddKey, sytuacja pozostaje taka sama:

A	C	C	C
A	C	C	C
A	C	C	C
A	C	C	C

Xor ze stałą wartością zawsze da taki sam wynik, więc wszystkie C będą nadal sobie równie, a xor z kolejnymi wartościami bajtów nadal da nam wszystkie wartości.

Następnie rozpoczyna się druga runda szyfrowania. Po `SubBytes`:

A	C	C	C
A	C	C	C
A	C	C	C
A	C	C	C

Po `ShiftRows`:

A	C	C	C
C	C	C	A
C	C	A	C
C	A	C	C

Tutaj nareszcie jakaś zmiana – operacja `ShiftRows` pozmieniała pozycje bajtów A i C. Teraz w każdej kolumnie występuje jedna wartość A.

Po `MixColumns`:

A	A	A	A
A	A	A	A
A	A	A	A
A	A	A	A

w tym momencie wszystko jest już wymieszane dokładnie i wszystkie bajty przyjmują wszystkie możliwe wartości.

Ostatnia operacja w tej rundzie, czyli `AddKey`, nie zmienia sytuacji w naszej tablicy z powodów wyjaśnionych wcześniej dla kroku `AddKey` w rundzie pierwszej.

Podobnie trzecia runda, `SubBytes` i `ShiftRows` zostawia nas w tym samym stanie:

A	A	A	A
A	A	A	A
A	A	A	A
A	A	A	A

Teraz następuje ważny moment – trzecia operacja `MixColumns`. Miesza ona wszystkie kolumny i psuje naszą własność A:

S	S	S	S
S	S	S	S
S	S	S	S
S	S	S	S

Pojawia się jednak inna ważna własność (oznaczona tutaj jako S) – jeśli ustalimy jakieś pole (np. lewy górny róg) i dla wszystkich szyfrogramów weźmiemy wartość z tego pola, a następnie zxorujemy je ze sobą, to otrzymamy zero. Brzmi skomplikowanie, ale pseudokod powinien wszystko wyjaśnić:

```
result = 0
j = 0  # który bajt szyfrogramu rozważamy (0..15)
for i in range(256):
  result ^= szyfrogramy[i][j]

assert result == 0  # po xorowaniu wszystkich wartości wynikiem
jest zero
```

Następna operacja (AddRoundKey) zachowuje tą własność. Niestety, w czwartej rundzie mamy znowu operacje `SubBytes`, `ShiftRows` i `AddRoundKey` (`MixColumns` w tej rundzie nie występuje), które nawet tą ostatnią własność niwelują.

Po co więc był ten cały wstęp? Zauważmy, że zarówno operacje `SubBytes`, jak i `ShiftRows` są odwracalne – problemem jest jedynie `AddRoundKey` – tej operacji nie da się odwrócić, nie znając klucza rundy. Możemy próbować go zgadnąć, ale równie dobrze moglibyśmy próbować zgadnąć cały klucz AESa, co jest niepraktyczne ze względu na duże uniwersum 128-bitowych kluczy.

Nasuwa się pytanie, czy możliwe jest zgadywanie klucza rundy po jednym bajcie. Okazuje się, że tak. Możemy zgadnąć pierwszy bajt klucza, zdeszyfrować ostatnią rundę z pierwszych bajtów szyfrogramów i sprawdzić, czy zachodzi własność S.

W pseudokodzie:

```
j = 0  # który bajt klucza rozważamy
candidates = []
for key_guess in range(256):
  result = 0
  for i in range(256):
    encrypted_byte = ciphertexts[i][j]
    decrypted_byte = decrypt_single_round(key_guess,
    encrypted_byte)
    result ^= decrypted_byte
  if result == 0:
    # własność "S" spełniona - prawdopodobnie zgadliśmy bajt poprawnie
    candidates.append(key_guess)
```

Jeśli zgadniemy bajt klucza poprawnie, mamy pewność, że wynik naszego testu (xor ustalonego bajtu po dekodowaniu ostatniej rundy we wszystkich szyfrogramach) wyniesie zero.

Ostatni problem stanowi fakt, że wynik może przypadkowo wynieść zero także w przypadku błędnego klucza – w przypadku złego strzału bajt przyjmie pseudolosową wartość z zakresu 0..255, więc statystycznie raz na 256 prób trafi się zero. Ponieważ zgadujemy 256 razy, statystycznie dla każdego bajtu klucza będzie dwóch kandydatów – jeden poprawny, a drugi, który w teście daje zero przypadkowo.

Następnie możemy powtórzyć przytoczoną operację 16 razy – dla kolejnych bajtów klucza. Skoro średnio uzyskamy dwie pasujące wartości, w tym jedną poprawną, dla każdego bajtu klucza, złożoność ataku wyniesie 2^{16}, czyli znacznie mniej niż 2^{128} przy ataku brute-force na całe uniwersum kluczy. Po uzyskaniu poprawnego klucza rundy możemy na jego podstawie wyliczyć klucz szyfrowania.

Czy ten atak możemy zastosować w przedstawionym zadaniu? Jak najbardziej! Jeśli weźmiemy pierwsze 256 zestawów danych dla 1 lub dla `0`, zauważymy, że wszystkie różnią się tylko na jednym bajcie – tym zależnym od licznika pętli `i`, a sam licznik przechowuje kolejne liczby 0-255.

IMPLEMENTACJA

W celu implementacji ataku najpierw musieliśmy zaimplementować lekko zmodyfikowanego AESa (z 4 rundami zamiast 10). Jest to dość nudny i powtarzalny kod, więc nie będziemy go tutaj reprodukować (do znalezienia w literaturze albo na naszym githubie – https://github.com/p4-team/ctf/tree/master/2016-03-12-0ctf/peoples_square#pl-version). Zdefiniowane są tam m.in. funkcje `encrypt4rounds` i `decrypt4rounds` (szyfrowanie/deszyfrowanie zmodyfikowanym, cztero-rundowym AESem) oraz `round2master` (odzyskiwanie klucza szyfrowania z klucza rundy).

Natomiast samo serce ataku wygląda tak:

```
def decryptSingleRound(ct, byteGuess, byteIndex):
  """ decrypt round with guessed round key """
  t = ct[byteIndex] ^ byteGuess
  return invsbox[t]

def integrate(index, ciphertexts):
  potential = []
  for candidateByte in range(256):
    sum = 0
    for ciph in ciphertexts:
      oneRoundDecr = decryptSingleRound(ciph, candidateByte,
      index)
      sum ^= oneRoundDecr
    if sum == 0:
      potential.append(candidateByte)
  return potential

def integralAttack(ciphertexts):
  candidates = []
  for i in range(16):
    candidates.append(integrate(i, ciphertexts))
  print 'candidates', candidates
  for roundKey in product(*candidates):
    masterKey = round2master(roundKey)
    plain = ''.join(chr(c) for c in
    decrypt4rounds(ciphertexts[1], masterKey))
    # wiemy że \0\0\0\0 występuje w plaintexcie, a szansa na to, że
    # przypadkowo się pojawi po deszyfrowaniu złym kluczem, jest
    bardzo mała
    if '\0\0\0\0' in plain:
      print 'solved', masterKey
      return masterKey
```

Funkcja `integralAttack(ciphertexts)` to główna funkcja przeprowadzająca atak. Chcemy odzyskać za jej pomocą klucz użyty do szyfrowania. Pierwszy krok to szukanie kandydatów na poszczególne bajty klucza głównego – używana jest do tego funkcja `integrate`. Drugi krok to wypróbowanie wszystkich możliwych kluczy rundy (wszystkie kombinacje kandydatów) i sprawdzenie, który z nich poprawnie deszyfruje kryptogram.

Druga ważna funkcja to `integrate(i, ciphertexts)`. Jej zadanie to znalezienie kandydatów na i-ty bajt klucza rundy. Zasada jej działania została opisana dokładnie w poprzednim rozdziale, więc tylko podsumowanie: wiemy, że jeśli poprawnie zgadniemy i-ty bajt klucza ostatniej rundy i zdeszyfrujemy za jego pomocą po jednym bajcie wszystkich kryptogramów, to zbiór tych zdeszyfrowanych bajtów będzie miał szczególną właściwość – po xorowaniu ich wszystkich razem wynikiem będzie zero. Możemy więc sprawdzić wszystkie możliwe wartości bajtu (w końcu jest ich tylko 256) i sprawdzić, kiedy będzie spełniony ten warunek. Jedyna pułapka jest taka, że wynik może też wynieść 0 przypadkowo, przy deszyfrowaniu złym kluczem. Nie psuje nam to jednak algorytmu, wystarczy zwracać listę kandydatów na bajt klucza (statystycznie będzie średnio dwóch kandydatów na każdy bajt), i to właśnie robimy.

Po uruchomieniu kodu dla dostarczonych danych dostajemy taki oto wynik:

```
candidates [[95, 246], [246], [1, 99], [78, 187], [123], [106],
[98, 223], [96], [211], [44, 63, 102], [192, 234], [167], [9,
135, 234], [36], [146, 166], [107]]
solved [23, 74, 34, 20, 64, 53, 100, 117, 220, 227, 160, 55,
163, 23, 237, 75]
```

Świetnie, teraz wystarczy zdeszyfrować flagę uzyskanym kluczem – „solved", to odzyskany klucz, a zaszyfrowana flaga była podawana w materiałach do zadania:

```
0CTF{~R0MAN_l0VES_B10CK_C1PHER~}
```

PODSUMOWANIE

Opisane przez nas zadanie było jednym z ciekawszych problemów, z którymi się spotkaliśmy. Wymagało dobrej znajomości dwóch kategorii: kryptografii oraz inżynierii wstecznej, a dodatkowo dotyczyło powszechnie stosowanych, nowoczesnych metod szyfrowania.

Zadanie wymagało od nas dość szczegółowego poznania kroków algorytmu, a także ich implementacji.

Kluczowa w rozwiązaniu zadania była obserwacja, że mamy do czynienia z AESem o zmniejszonej liczbie rund, oraz fakt, że dysponujemy szyfrogramami dla bardzo szczególnie dobranych tekstów. Pozwoliło to na odszukanie odpowiednich pozycji w literaturze omawiających podobne modyfikacje algorytmu i ich podatności.

Jarosław Jedynak, Stanisław Podgórski

Rozwiązanie zadania People's Square zostało nadesłane przez p4, polski zespół CTF-owy, który w chwili redagowania tego artykułu zajmował 2-gie miejsce w generalnej klasyfikacji CTFTime.org.

https://ctftime.org/team/5152