..

README.md	typo	a year ago
integral.py	Add trace writeup, in both polish and english	a year ago
people_square.tar.gz	fixed dir name	a year ago

README.md

People's Square (Crypto 6p)

```
People's Square (A.K.A. shenmhin guangshan in Shanghai Dialect)
is a large public square in the Huangpu District of Shanghai,
China.
We know Talent Yang is the king of People's Square. Now he
provides you a strange guessing game, and he also demonstrates
his talent by giving you the result of how he tackles this task.
Can you show your talent to decrypt the secret?
```

###ENG PL

We start by unpacking and analysing binary that we were given in challenge. In fact, that binary wasn't working on our computers (not everyone of us has newest laptop model), but in spite of it we managed to decompile it.

Decompilation turned out to be only first step of analysis, because (as it's often the case with automatic decompilers) code was not very readable. It's difficult to analyse something like that:

```
const __m128i *__fastcall sub_4013C5(const __m128i *a1, __int64 a2)
{
  __m128i v2; // xmm0@1
  __m128i v3; // xmm0@2
  __m128i v6; // xmm0@4
  const __m128i *result; // rax@4
  signed __int64 i; // [sp+18h] [bp-98h]@1
  __int128 v11; // [sp+30h] [bp-80h]@1
  __int128 v12; // [sp+40h] [bp-70h]@1
  __int128 v13; // [sp+50h] [bp-60h]@1
  __int128 v14; // [sp+60h] [bp-50h]@2
  __int128 v15; // [sp+70h] [bp-40h]@2
  __int128 v16; // [sp+80h] [bp-30h]@4
  __int128 v17; // [sp+90h] [bp-20h]@4
  __int128 v18; // [sp+A0h] [bp-10h]@4

  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;
  }
```

We started by manually refactoring provided code to something more manageable. We arrived at something like this:

```
  __int64 realMain()
  {
    char keyProbably;
    char ciphertextFor0[16];
    char ciphertextFor1[16];
    __int64 v14 = 0LL;
    __int64 v16 = 0LL;
    sub_400ABE(&v14); // put encrypted flag into buffer
    __int64 v10 = 0LL;
    sub_400A74(&v10);
    generateKey((const __m128i *)&v10, (__int64)&keyProbably);
    __int64 v6 = 0LL;
    __int64 initTime = time(0LL);
    for (__int64 i = 0LL; i <= 0x3FF; i++)
    {
      memsetAndEncrypt((__int64)&keyProbably, &ciphertextFor0, &ciphertextFor1, i, initTime);
      unsigned int v5 = rand() & 1;
      char *v1;
      if (v5) {
        v1 = &ciphertextFor0;
      } else {
        v1 = &ciphertextFor1;
      }
      hexdump((__int64)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 == v5 )
      {
        puts("Correct!");
        ++v6;
      }
      else
      {
        puts("Incorrect!");
      }
    }
    if ( v6 == 1024 )
    {
      puts("Now I will give you the flag:");
      realEncrypt((const __m128i *)&v14, (__int64)&keyProbably); // decrypt flag
      realEncrypt((const __m128i *)&v16, (__int64)&keyProbably);
      hexdump((__int64)&v14, 0x20uLL);
    }
    return 0;
  }

  void memsetAndEncrypt(__int64 keyProbably, char ciphertext0[16], char ciphertext1[16], __int64 iter, __int64 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((const __m128i *)ciphertext0, keyProbably);
    realEncrypt((const __m128i *)ciphertext1, keyProbably);
```

```
}

void realEncrypt(__m128i *a1, __m128i *key)
{
  __int128 state;

  *state = a1 ^ key[0];
  for (int i = 1LL; i <= 3; ++i) {
    *state = aesenc(state, key[i];
  }
  *a1 = aesenclast(state, key[4]);
}
```

Beautiful, if compared to abomination we had earlier. (For example, function that I pasted earlier is now called "realEncrypt" - that's right, that mess of code is only 4 lines long after refactoring).

As you can see - all this code is only AES encrypting some data. RealEncrypt is really traditional AES, but it's using only 4 rounds - instead of 10 rounds, as proper AES128 should be (and only because of it we were able to crack it)

Reiterating, in pseudocode, our program can be summarized with something like this:

```
uint32_t initTime = time()

for (uint32_t i = 0; i < 1024; i++) {
    // 16 byte array: [ 8 ones ] [ i (4 bytes) ] [ initTime (4 bytes) ]
    uint8_t ones[16];
    memset(ones, 0, 16);
    memcpy(ones + 8, &i, 4);
    memcpy(ones + 12, &initTime, 4);

    // 16 byte array: [ 8 zeroes ] [ i (4 bytes) ] [ initTime (4 bytes) ]
    uint8_t zeroes[16];
    memset(zeroes, 0, 16);
    memcpy(zeroes + 8, &i, 4);
    memcpy(zeroes + 12, &initTime, 4);

    bool bit = rand() % 2;
    if (bit == 0) {
        print(encrypt(zeroes));
    } else {
        print(encrypt(zeroes));
    }

    print("guess?");
    bool guess = read();
    if (guess == bit) {
        goodGuesses++;
    }

    print(encrypt(zeroes));
    print(encrypt(ones));
}

if (goodGuesses == 1024) {
    print("you won");
    print encrypt(encryptedFlag); // will actually decrypt flag for us
}
```

Where encrypt is our modified, 4-round AES.

How would we begin cracking something like that? It turns out we can use well-known cryptographic attack technique, called `square attack` or `integral cryptanalysis`.

To understand how this attack is supposed to work, we need to first get a high level (simplified) overview of AES.

So AES is working on 128bit blocks. These 128 bits are treated as 4x4 byte array, and we are performing some "operations" on that array.

What kind of operations?

1. SubBytes - every byte in matrix is substituted with another byte, using fixed associatve table (so-called substitution box, aka sbox). So given matrix:

in_matrix: 1 2 3 4 6 7 8 5 11 12 9 10 16 13 14 15

we end up with:

```
out_matrix:
 S[1]  S[2]  S[3]  S[4]
 S[6]  S[7]  S[8]  S[5]
S[11] S[12]  S[9] S[10]
S[16] S[13] S[14] S[15]
```

ShiftRows - rows of matrix are shifted by `row_number` positions to the left. For example given matrix:

```
in_matrix:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

we arrive to:

```
out_matrix:
 1  2  3  4
 6  7  8  5
11 12  9 10
16 13 14 15
```

MixColumns - values in matrix columns are mixed with each other (obvious, I know) (to be precise, columns are treated as polynominals and multiplied by fixed polynominal, but that's not important right now) What's important is that final values in column depends only on another values in that column:

```
in_matrix:
A  E  I  M
B  F  J  N
C  G  K  O
D  H  L  P

nA, nB, nC, nD = mix(A, B, C, D)
nE, nF, nG, nH = mix(E, F, G, H)
nI, nJ, nK, nL = mix(I, J, K, L)
nM, nN, nO, nP = mix(M, N, O, P)

out_matrix:
nA nE nI nM
nB nF nJ nN
nC nG nK nO
nD nH nL nP
```

AddRoundKey - we are adding round key to each byte in matrix (don't look at me like that, it's not my fault that every operation is already descriptively named). The catch is, instead of traditional addition we are using 'xor' operation (in technical words, that's because we are operating on galios field, and 'xor' is addition analogy there).

```
in_matrix:
A  E  I  M
B  F  J  N
C  G  K  O
D  H  L  P

round_key:
R0 R1 R2 R3
R4 R5 R6 R7
R8 R9 RA RB
RC RD RE RF

out_matrix:
A^R0  E^R1  I^R2  M^R3
B^R4  F^R5  J^R6  N^R7
C^R8  G^R9  K^RA  O^RB
D^RC  H^RD  L^RE  P^RF
```

Why bother with all that theory? Because now we are going to describe the actual attack.

Assume that we can get ciphertexts for 256 chosen (by us) plaintexts - in such a way, that every byte is constant among all plaintexts, except one byte that gets every possible byte value.

Example of such set could be for example:

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

Then we can represent AES state at the beginning with the following matrix:

```
X   C   C   C
C   C   C   C
C   C   C   C
C   C   C   C
```

Important - C means that byte is the same on corresponding positions in matrix for every considered plaintext from our set. And X means that byte at that position gets *every possible byte value* for some considered plaintexts.

So, let's check what AES will do with our set of chosen plaintexts: Initial value.

```
X   C   C   C
C   C   C   C
C   C   C   C
C   C   C   C
```

As mentioned, first byte is taking every possible value, and rest of bytes is constant.

After SubBytes:

```
X   C   C   C
C   C   C   C
C   C   C   C
C   C   C   C
```

Of course, every 'C' is changed to the same value (so they are still constant). What's next?

After ShiftRows:

```
X   C   C   C
C   C   C   C
C   C   C   C
C   C   C   C
```

Nothing's changed.

After MixColumns:

```
X   C   C   C
X   C   C   C
X   C   C   C
X   C   C   C
```

Now we're getting somewhere. Now 'C' values are not exactly the same when looking at single state, but they get the same value for every plaintext in our choosen set - ant that's what matters. Similarly, every byte marked as 'X' will take every possible value for one of our plaintexts. I hope that's clear, let's continue.

After AddKey:

```
X   C   C   C
X   C   C   C
X   C   C   C
X   C   C   C
```

Now for second round:

After SubBytes:

```
X   C   C   C
X   C   C   C
X   C   C   C
X   C   C   C
```

After ShiftRows

```
X   C   C   C
C   C   C   X
C   C   X   C
C   X   C   C
```

After MixColumns:

```
X   X   X   X
X   X   X   X
X   X   X   X
X   X   X   X
```

After AddKey:

```
X   X   X   X
X   X   X   X
X   X   X   X
X   X   X   X
```

What happend now? Now *every byte* is taking every possible value. Horrible! Or is it?

That means that if we fix any state position (for example, 2nd row and 2nd column) and xor them: (State_xx means state for xx-th plaintext from our set)

```
state_01[2][2] ^ state_02[2][2] ^ state03_[2][2] ^ ... ^ stateFF[2][2] = 0
```

Then we will always get 0! That's because we know that that byte is taking every possible byte, value, so we are xoring (in random order) numbers

```
0 ^ 1 ^ 2 ^ 3 ... ^ 0xFE ^ 0xFF
```

So what? After all, our AES has 4 rounds, not 3.

But we can *guess* one byte of key to *decrypt* one byte of encrypted data after 4th round. From previos relationship we can conclude that if we guess the byte correctly, decrypt some byte in ciphertext with it, and xor it all together, we must get 0.

Then we can repeat that process 16 times - once fo each byte of key. On average we will find 2 good values (one correct, and 256 * 1/256 chance of false positive). We can just try every possibility - that will give us 2^16 complexity of attack - a LOT better than naive 2^128.

We can also quite easily recover the master cipher key from any round key.

Can we use that attack in this challenge? Of course - if we take first 256 plaintexts, we can see that every one of them is differing only on ony byte - lowest byte of 'i'.

So we can implement that attack in python (AES implementation skipped here, but it will be provided with full code in writeup)

```
def integrate(index):
    potential = []

    for candidateByte in range(256):
        sum = 0
        for ciph in ciphertexts:
            oneRoundDecr = backup(ciph, candidateByte, index)  # decrypt one round of one byte
            sum ^= oneRoundDecr     # xor result with sum
        if sum == 0:   # if sum is equal to 0 - candidateByte stays a candidate
            potential.append(candidateByte)
    return potential


from itertools import product
def integral():
    candidates = []
    for i in range(16):
        candidates.append(integrate(i))    # compute all possible candidate bytes for all positions
    print 'candidates', candidates
    for roundKey in product(*candidates):  # check all possibilities
        masterKey = round2master(roundKey)
        plain = ''.join(chr(c) for c in decrypt4rounds(ciphertexts[1], masterKey))
        if '\0\0\0\0' in plain:  # we know that plaintext contains 4 '0' bytes, and it's unlike to be accident
            print 'solved', masterKey
```

After running it for provided ciphertexts we get following result:

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

Awesome! Now we can just decrypt encrypted flag (we have key, after all).

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

Full code used in our attack included in integral.py file.

Code template and used attack was mostly borrowed from lecture http://opensecuritytraining.info/Cryptanalysis.html (CC-by-sa 3.0 license). For that reason, our code is published on CC 3.0 BY SA license, but writeup text and our code (fragments in this writeup) use our usual license.

###PL version

Najpierw analizujemy binarkę którą dostajemy w zadaniu. Co prawda nie działała ona na większości naszych komputerów za bardzo (nie wszyscy mamy najnowsze laptopy), udało nam się ją zdekompilować.

Dekompilacja to dopiero pierwszy krok analizy, ponieważ jak to z automatyczną dekompilacją zawsze - kod był bardzo nieczytelny. Ciężko o analizę z funkcjami takimi jak taka:

```
const __m128i *__fastcall sub_4013C5(const __m128i *a1, __int64 a2)
{
  __m128i v2; // xmm0@1
  __m128i v3; // xmm0@2
  __m128i v6; // xmm0@4
  const __m128i *result; // rax@4
  signed __int64 i; // [sp+18h] [bp-98h]@1
  __int128 v11; // [sp+30h] [bp-80h]@1
  __int128 v12; // [sp+40h] [bp-70h]@1
  __int128 v13; // [sp+50h] [bp-60h]@1
  __int128 v14; // [sp+60h] [bp-50h]@2
  __int128 v15; // [sp+70h] [bp-40h]@2
  __int128 v16; // [sp+80h] [bp-30h]@4
  __int128 v17; // [sp+90h] [bp-20h]@4
  __int128 v18; // [sp+A0h] [bp-10h]@4
```