Branch: master ctf / 2015-10-18-hitcon / forensic_250_puzzleng / Create new file Find file History

msm-code Add english version for puzzleng Latest commit 1fbb3d4 on Oct 19 2015

..

README.md	Add english version for puzzleng	2 years ago
encrypt	Add forensics 250 puzzleng:	2 years ago
flag.puzzle	Add forensics 250 puzzleng:	2 years ago
puzzleng.tgz	Add forensics 250 puzzleng:	2 years ago
result.png	Add forensics 250 puzzleng:	2 years ago

README.md

Puzzleng (forensic, 150p, 24 solves)

> Next Generation of Puzzle!

> puzzleng-edb16f6134bafb9e8b856b441480c117.tgz

PL

ENG

Z dołączonego do zadania pliku tgz wypakowujemy dwa kolejne - encrypt (to binarka, elf) oraz flag.puzzle (nieznany typ pliku). Łatwo domyślić się że jeden to kod którym zostały zaszyfrowane dane, a drugi to same zaszyfrowane dane.

Przepisujemy kod szyfrujący do C żeby móc go dokładniej przeanalizować:

```
int main(int argc, char *argv[]) {
    char hash[20];
    assert(argc == 3);
    int password_len = strlen(argv[1]);
    SHA1(argv[1], password_len, hash);
    stream = fopen(argv[2], "r");
    assert(stream);
    fseek(stream, 0LL, 2);
    int data_len = ftell(stream);
    rewind(stream);
    for (int i = 0; i <= 19; ++i) {
        for (int j = 0; j < (data_len + 19) / 20; ++j) {
            int chr = fgetc(stream);
            if (chr == -1) break;
            putchar(chr ^ hash[i]);
        }
    }
}
```

Jak widać działanie jest bardzo proste - dzieli plik wejściowy na 20 bloków, i każdy z nich xoruje z innym bajtem. Bajty z którymi szyfruje są losowe (wynik SHA1) więc ich nie zgadniemy.

Ale wydaje się to być banalne, albo wręcz trywialne zadanie. W końcu xorowanie po bajcie to jedna z najsłabszych metod szyfrowania jaką można wymyślić. Zakładamy więc że dane które zostały zaszyfrowane to plaintext, i piszemy na szybko dekryptor (źródło już nie istnieje, ale pomysł był prosty - dla każdego bloku sprawdzenie, z jakim bajtem go trzeba xorować żeby po xorowaniu wszystkie bajty były plaintekstem). Bardzo się zawiedliśmy - nie ma takich bajtów, więc dane które zostały zaszyfrowane nie są plaintextem.

W przypływie natchnienie sprawdzamy co innego - czy da się znaleźć taki bajt, że po xor-owaniu go z pierwszym blokiem w wyniku będzie gdzieś "IHDR". I nie myliliśmy się - dane które otrzymaliśmy to zaszyfrowany plik .png:

```
s = open('flag.puzzle', 'rb').read()

chunk_len = (1135+19)/20
chunks = [s[chunk_len*i:chunk_len*(i+1)] for i in range(20)]

def xor(a, b):
    return ''.join(chr(ord(ac) ^ ord(bc)) for ac, bc in zip(a, b))

for i in range(len(chunks)):
    c = chunks[i]
    for b in range(256):
        xx = xor(c, chr(b)*10000)
        if 'IHDR' in xx:
            print i, b, xx, xx.encode('hex')
```

Co teraz? Wiemy już z jakim bajtem był xorowany pierwszy blok, ale mamy 19 do zgadnięcia. Zrobiliśmy to samo dla 'IDAT' i 'IEND', zgadując kolejne dwa bajty. Niestety to ślepa uliczka - pikseli obrazka w ten sposób nie zgadniemy (a przynajmniej nie mieliśmy pomysłu żadnego).

Dlatego poszliśmy inną drogą - wiemy jak zaczynają się dane chunka IDAT (to stream zlibowy), bo mamy ich fragment:

```
start = '789CEDCF418AE4300C05D0DCFFD2358BC6485F76'.decode('hex')
```

Stream zlibowy prawdopodobnie nie zadziała dla wszystkich danych - możemy próbować deszyfrować drugi blok, i patrzeć kiedy będzie leciał wyjątek podczas dekompresji:

```
known = {
    0: 101,
    1: 48
}

curr = start
for ndx in range(2, 20):
    c = chunks[ndx]
    if not known.has_key(ndx):
        for i in range(256):
            xx = xor(c, chr(i)*10000)
            try:
                zl = zlib.decompressobj().decompress(curr+xx)
            except:
                continue
            known[ndx] = i
    print known[ndx]
    xxok = xor(c, chr(known[ndx])*10000)
    curr = curr + xxok
```

Był to bardzo obiecujący sposób, ale niestety skończył się niepowodzeniem - o ile zawartość trzeciego bloku odzyskaliśmy (był tylko jeden bajt który nie powodował wyjątku), przy czwartym i dalej bloku za wiele danych dekompresowało się poprawnie.

Więc wykorzystaliśmy to co wiedzieliśmy o obrazku (wyciągniętą z rozszyfrowanej sekcji IDAT) - miał szerokość 912 pikseli oraz jednobitową paletę (czyli prawdopodobnie czarno-biały). I teraz cechą plików png, jest to że na początku każdego wiersza danych znajduje się filtr którym są one traktowane. Mniejsza o technikalia, wynika z tego że co 115 bajt w zdekompresowanych danych powinien być równy '\x00' (tzn. nie musiał jeśli byłyby użyte inne filtry, ale po analizie fragmentów danych które mieliśmy zauważyliśmy że tutaj używany wszędize jest filtr 0, jak zazwyczaj w png).

```
def testraw(raw):
    for i in range(len(raw) / 115):
        if raw[i*115] != '\0':
            return False
    return True
```

Nowa wersja:

```
curr = start
for ndx in range(2, 20):
    c = chunks[ndx]
    if not known.has_key(ndx):
        for i in range(256):
            xx = xor(c, chr(i)*10000)
            try:
                zl = zlib.decompressobj().decompress(curr+xx)
            except:
                continue
            if testraw(zl):
                known[ndx] = i
    print known[ndx]
    xxok = xor(c, chr(known[ndx])*10000)
    curr = curr + xxok
```

Niestety, było to dalej niewystarczające - ciągle więcej niż jeden bajt spełniał nasze warunki, więc musieliśmy je jakoś oceniać. Jako że zauważyliśmy że w błędnie zdekompresowanych plikach na końcu znajdywały się głównie zera (czarne piksele), wartościowaliśmy po ilości nieczarnych pikseli w wyniku po dekompresji. Ostateczna wersja:

```
curr = start
for ndx in range(2, 20):
    clean()
    c = chunks[ndx]
    if not known.has_key(ndx):
        mingap = 0
        for i in range(256):
            xx = xor(c, chr(i)*10000)
            try:
                zl = zlib.decompressobj().decompress(curr+xx)
            except:
                continue
            if testraw(zl):
                gap = len([True for x in zl[-500:] if x != '\x00'])
                if gap > mingap:
                    known[ndx] = i
                    mingap = gap
    print known[ndx]
    xxok = xor(c, chr(known[ndx])*10000)
    curr = curr + xxok
```

Udało się, dostaliśmy jakieś dane. Po zapisaniu ich do pliku otrzymaliśmy piękny QR code:

Po dekodowaniu:

```
hitcon{qrencode -s 16 -o flag.png -l H --foreground 8F77B5 --background 8F77B4}
```

ENG version

We unpacked two files from tgz attached to task: encrypt (elf binary) and flag.puzzle (unknown file). It was obvious to us that first file is binary used to encrypt some data, and second file is result of that encryption.

We disassembled and rewritten binary to C to simplify analysis:

```
int main(int argc, char *argv[]) {
    char hash[20];
    assert(argc == 3);
    int password_len = strlen(argv[1]);
    SHA1(argv[1], password_len, hash);
    stream = fopen(argv[2], "r");
    assert(stream);
    fseek(stream, 0LL, 2);
    int data_len = ftell(stream);
    rewind(stream);
    for (int i = 0; i <= 19; ++i) {
        for (int j = 0; j < (data_len + 19) / 20; ++j) {
            int chr = fgetc(stream);
            if (chr == -1) break;
            putchar(chr ^ hash[i]);
        }
```

```
    }
}
```

Encryption method is really simple - program splits input file to 20 blocks of equal length, than xors each of them with another byte.

At this moment challenge seems to be very easy, almost trivial - after all xoring data with single byte is one of weakest existing methods of encryption. We assumed that encrypted data was plain text, and tried to bruteforce bytes in sha1 that was used to xor them. If our assumption was true, we could find byte such that ciphertext_byte ^ byte is printable ascii for each byte in ciphetext - but it was not possible. That means, data that was encrypted is not textual.

So we checked another possiblity - we tried bruteforcing bytes for first block, but this time we hoped for 'IHDR' in decrypted string. And we were right - our encrypted data used to be .png file.

```
s = open('flag.puzzle', 'rb').read()

chunk_len = (1135+19)/20
chunks = [s[chunk_len*i:chunk_len*(i+1)] for i in range(20)]

def xor(a, b):
    return ''.join(chr(ord(ac) ^ ord(bc)) for ac, bc in zip(a, b))

for i in range(len(chunks)):
    c = chunks[i]
    for b in range(256):
        xx = xor(c, chr(b)*10000)
        if 'IHDR' in xx:
            print i, b, xx, xx.encode('hex')
```

Now what? We know byte that was used to encrypt first block, but we have 19 more to go. We used the same method to find 'IDAT' and 'IEND', leaving us with 17 unknown bytes.

That turned out to be dead end, so we tried something completely different. We know first few bytes of IDAT section (because we decrypted block with IDAT)

```
start = '789CEDCF418AE4300C05D0DCFFD2358BC6485F76'.decode('hex')
```

We know that this is zlib stream, and that next block, after decrypting and appending to this fragment, should form correct zlib stream as well (otherwise it will probably throw some exception)

```
known = {
    0: 101,
    1: 48
}

curr = start
for ndx in range(2, 20):
    c = chunks[ndx]
    if not known.has_key(ndx):
        for i in range(256):
            xx = xor(c, chr(i)*10000)
            try:
                zl = zlib.decompressobj().decompress(curr+xx)
            except:
                continue
            known[ndx] = i
    print known[ndx]
    xxok = xor(c, chr(known[ndx])*10000)
    curr = curr + xxok
```

This method was really promising, but we didn't get very far with it. We decrypted third block with ease (only one byte didn't throw exception), but fourth and later blocks decompression threw exceptions much sparser and left us with too much posibilities to bruteforce.

So we used our knowledge about image (that we acquired after decrypting IDAT section) - it's 912 pixels wide, 912 pixels height, and have 1bit palette (black and white probably). We remembered that each row in decompressed raw png data starts with 'filter byte' (this byte allows encoders/decoders to preprocess raw pixel data, potentially reducing final file size). After

analysing blocks that we managed to decompress, we concluded that all filter bytes are equal to 0, so every 115-th byte in decompressed data should be equal to '\x00'. Using this knowledge we updated our decryptor:

```
def testraw(raw):
    for i in range(len(raw) / 115):
        if raw[i*115] != '\0':
            return False
    return True
```

New version:

```
curr = start
for ndx in range(2, 20):
    c = chunks[ndx]
    if not known.has_key(ndx):
        for i in range(256):
            xx = xor(c, chr(i)*10000)
            try:
                zl = zlib.decompressobj().decompress(curr+xx)
            except:
                continue
            if testraw(zl):
                known[ndx] = i
    print known[ndx]
    xxok = xor(c, chr(known[ndx])*10000)
    curr = curr + xxok
```

Alas, that still wasn't it - too many bytes passed this test. We noticed that incorrectly decompressed data contained a lot of zeroes at the end. So we decided to grade possible solutions according to number of '\x00' bytes at the end (the less the better). Final version:

```
curr = start
for ndx in range(2, 20):
    clean()
    c = chunks[ndx]
    if not known.has_key(ndx):
        mingap = 0
        for i in range(256):
            xx = xor(c, chr(i)*10000)
            try:
                zl = zlib.decompressobj().decompress(curr+xx)
            except:
                continue
            if testraw(zl):
                gap = len([True for x in zl[-500:] if x != '\x00'])
                if gap > mingap:
                    known[ndx] = i
                    mingap = gap
    print known[ndx]
    xxok = xor(c, chr(known[ndx])*10000)
    curr = curr + xxok
```

We did it, that method gave us correct byte that each block was xored with (stored in `known` dictionary). After saving decrypted data to file, we get beautiful QR code:

After decoding:

```
hitcon{qrencode -s 16 -o flag.png -l H --foreground 8F77B5 --background 8F77B4}
```