Challenge #11 Solution

by Claudiu Teodorescu

The purpose of the challenge is to successfully decrypt the secret stored in the binary resource 124 and extract the embedded email address.

The application expects one argument, interpreted as a 32-bit number, to be passed in. The least-significant byte of the input parameter is combined with the `Salt` in the `EncryptedKeyMaterial` to generate the correct salt to be used in the key derivation scheme.

The decryption process starts by computing the `MasterKey`, i.e. the key from which all the other Key Encryption Keys (KEKs) are derived from.

The `MasterKeyRecord` consists of the three 16-byte keys that are combined to derive a 16-byte `MasterKey`. The `MasterKeyRecord` data is stored in the binary resource 120 and it has the following structure:

```
struct MasterKeyRecord {
uint8_t         Key1[0x10];
uint8_t         Key2[0x10];
uint8_t         Key3[0x10];
};
```

Figure 1: MasterKeyRecord Structure

```
Key1        : 16 bytes

Key2        : 16 bytes

Key3        : 16 bytes


0000 45 0E 29 A6 B4 C4 F1 77 4E 31 C1 EB E1 C2 3C 87 E.)....wN1....<.

0010 9A BD 53 6A 7D D0 16 14 D2 7B 2E CD 54 21 B7 5F ..Sj}....{..T!._

0020 AC 91 8D 29 20 53 86 15 7A 98 04 A0 4E 08 54 A6 ...) S..z...N.T.
```

Figure 2: Master Key Record

The algorithm to derive the `MasterKey` based on the `MasterKeyRecord` is described in the Figure 3:

```
for (uint32_t i = 0; i < 16; ++i)

  MasterKey[i] = MasterKeyRecord.Key1[i] ^ MasterKeyRecord.Key2[i] ^ MasterKeyRecord.Key3[i]
```

Figure 3: Master Key Generation Algorithm

The `EncryptedKeyMaterial` data, containing all the encrypted KEK records, is stored in the binary resource 121, and it has the following format:

```
struct EncryptedKeyMaterial {
      uint32_t              Sig1;
      uint32_t              Sig2;
      uint32_t              Iterations;
      uint32_t              Rounds;
      uint8_t               Salt[0x10];
      uint8_t               KMHash[0x10];
      struct KeyMaterial    Keks[0x20];

};
```

Figure 4: EncryptedKeyMaterial Structure

The `Sig1` and `Sig2` fields make up the "FLARE-ON" signature and the `KMHash` represents the MD5 hash of the `Keks` array data.

The block cipher used in this challenge is `RC5` in `CBC` mode. The `Rounds` field in the `EncryptedKeyMaterial` structure specifies the number of rounds the `RC5` algorithm is using to construct the decryption schedule.

The challenge uses a Password Based Key Derivation scheme to generate symmetric RC5 keys. The C implementation of the derivation key algorithm for 16-byte keys is shown in the Figure 5:

```
#define HASHBYTES 16

PBKDF(uint8_t derivedkey[HASHBYTES], uint8_t hmackey[HASHBYTES], uint8_t
salt[HASHBYTES], uint32_t iterations, uint32_t index) {

  uint8_t tempHash[HASHBYTES + sizeof(uint32_t)];

  uint8_t temp[HASHBYTES];

  memcpy(tempHash, sizeof(tempHash), salt, HASHBYTES);

  *(uint32_t*)(&tempHash[HASHBYTES]) = _byteswap_ulong(index);

  MD5_HMAC(derivedkey, HASHBYTES, hmackey, HASHBYTES, tempHash, sizeof(tempHash));

  for (uint32_t i = 1; i < iterations; ++i) {

    MD5_HMAC(temp, HASHBYTES, hmackey, HASHBYTES, derivedkey, HASHBYTES);

    for (uint32_t j = 0; j < HASHBYTES; ++j)
```

```
      derivedkey[j] ^= temp[j];
    }
}
```

Figure 5: 16-byte Key Derivation Algorithm

The C implementation of the derivation key algorithm for a variable length key is shown in the Figure 6:

```
void DeriveKey(const void *pwd, uint pwdsize, uint8_t salt[HASHBYTES], uint32_t
iterations, void *derivedkey, uint32_t derivedkeysize) {
  uint8_t hash[HASHBYTES];
  uint8_t hmackey[HASHBYTES];
  uint32_t blocks = derivedkeysize / HASHBYTES;
  if (derivedkeysize % HASHBYTES)
    blocks++;
  MD5(pwd, pwdsize, hmackey, HASHBYTES);
  uint32_t genSize = 0;
  uint32_t curSize = 0;
  for (uint32_t i = 0; i < blocks; ++i) {
    PBKDF(hash, hmackey, salt, iterations, i + 1);
    if (genSize + HASHBYTES <= derivedkeysize)
      curSize = HASHBYTES;
    else
      curSize = derivedkeysize - genSize;
    memcpy((uint8_t*)(derivedkey)+genSize, curSize, hash, curSize);
    genSize += curSize;
  }
}
```

Figure 6: Variable Length Key Derivation Algorithm

The MD5 algorithm implemented in the challenge is changed to generate a modified hash by toggling the endianness of the four 32-bit components of the real hash.

The `Salt` field in `EncryptedKeyMaterial` structure is combined with the 32-bit command line parameter by performing a bit-wise OR of the least-significant bytes. The correct 32-bit input number should have the least-significant byte equal to 205 (0xCD).

The 16-byte `MasterKey`, the generated `Salt` and the `Iterations` field in `EncryptedKeyMaterial` structure are passed to the `DeriveKey` function to generate a 16-byte key that decrypts the key encryption key (kek) at index 0 in the `Keks` array.

A decrypted `KeyMaterial` entry in the `Keks` array has the following structure:

```
struct KeyMaterial {
      uint32_t       Index;
      uint32_t       BaseIter;
      uint8_t        Kek[8];
      uint8_t        Salt[0x10];
      uint8_t        Hash[0x10];
};
```

Figure 7: KeyMaterial Structure

The `BaseIter` field is used to compute the real iteration count using the following algorithm:

```
uint32_t CurrentIter = EncryptedKeyMaterial.Iter;

for (uint32_t i = 1; i < 0x20; ++i) {

    CurrentIter = (i / 0x10) * CurrentIter + EncryptedKeyMaterial.Keys[i - 1].BaseIter;
```

Figure 8: Iteration Generation Algorithm

The `Index` field represents the index of the record in the `Keks` array. The iteration count computed based on `BaseIter` field, the 8-byte `Kek` field and `Salt` field are used to derive the key for the next record in the `Keks` array using the `DeriveKey` function.

The `Hash` field represents the modified MD5 hash of the data in the corresponding record, excluding the stored `Hash` field data.

The Figure 9 shows the `EncryptedKeyMaterial` data structure found in the binary resource 121:

```
Sig1        :  4 bytes

Sig2        :  4 bytes

Iterations  :  4 bytes

Rounds      :  4 bytes

Salt        :  16 bytes

KMHash      :  16 bytes

Keys        :  32 * sizeof(KeyMaterial)
```

```
0000 46 4C 41 52 45 2D 4F 4E 00 80 00 00 0F 00 00 00 FLARE-ON........
0010 00 77 A9 B5 28 3A C9 52 0D 07 BA 20 3A DD 58 48 .w..(:.R... :.XH
0020 65 E8 15 23 EF 9E CF C1 9F 59 89 DB 52 09 0F 1C e..#.....Y..R...
0030 73 DF AD F2 37 58 F4 EA F5 FB 80 C8 A4 14 19 11 s...7X..........
0040 9D BC E4 54 C6 7D AD CE CF DF 6B 18 20 5B CF A3 ...T.}....k. [..
0050 94 27 AA 02 E8 76 23 BF D7 8C 42 D9 90 E7 34 4B .'...v#...B...4K
0060 58 01 E2 F4 FA 7D 0C D2 DC 04 7C 63 6F 87 24 48 X....}....|co.$H
0070 84 BC A4 50 88 01 78 89 34 F3 B8 43 6F C1 86 7D ...P..x.4..Co..}
0080 2D AE 3B 78 2F 26 22 C6 79 C1 B9 76 EE 54 6D C8 -.;x/&".y..v.Tm.
0090 58 12 86 11 90 74 95 A8 E1 D2 F6 A8 99 74 1F 25 X....t.......t.%
00A0 B3 EA 30 A7 98 D0 2D 13 85 E7 F3 76 DA AC CE BA ..0...-....v....
00B0 A8 21 53 77 2D 33 40 E8 17 1C D5 B6 06 D5 8E 3C .!Sw-3@........<
00C0 47 F0 6E 43 36 4A A8 BF DE AF 3D BC 20 84 30 66 G.nC6J....=. .0f
00D0 90 28 01 6C 20 FF 2C 11 EE FB AF C7 B9 35 2D 0A .(.l .,......5-.
00E0 E8 80 88 EF 06 57 70 3A EB 08 D9 0B 6D 8B DE 72 .....Wp:....m..r
00F0 47 94 8E 78 43 49 03 44 50 BE 42 54 D7 9D 37 8E G..xCI.DP.BT..7.
0100 79 0C 2E 08 8E FA BE 4B 92 31 CF 8D 9D 8B F5 D7 y......K.1......
0110 49 94 DF 1D 35 CF 55 4C 0F 0B D3 31 57 2C 45 6E I...5.UL...1W,En
0120 1D 01 DF 99 A2 86 7C 7E 1E C5 D6 AA BC 9C DF CD ......|~........
```

Figure 9: Key Material

The binary resource 122 contains the 16 encrypted data keys (DEKs), the size of each key being 16 bytes. Since there are 32 KEK records in the `Keks` array in `EncryptedKeyMaterial` structure but only 16 DEKs it means that you need to decrypt only the first 16 KEK records to decrypt all the DEKs. And thus, you would be waiting a very long time if you let the program run and decrypt all 32 KEK records.

The structure of the `DEKs` data in binary resource 122 is should below:

```
struct DecryptionKeyRecrord {
        uint8_t DecryptionKey[0x10];
};

struct DecryptionKeyRecords {
```

```
	struct DecryptionKeyRecrord DecryptionKeys[0x10];
};
```

Figure 10: DecryptionKeyRecord Structure

```
DecryptionKeyRecord[0] : 16 bytes

DecryptionKeyRecord[1] : 16 bytes

…

DecryptionKeyRecord[15] : 16 bytes


0000 8C F5 C6 82 15 DD BE 3C 8E FC 83 35 0E 55 20 D9 .......<...5.U .
0010 D8 6D 70 20 83 FF 2D F9 66 C0 47 56 40 84 5D 23 .mp ..-.f.GV@.]#
0020 59 25 FC B1 85 CF 05 DB 93 FA 34 2F 29 3B 2F 38 Y%........4/);/8
0030 91 89 5F 59 80 39 5A 61 D7 02 4D 05 5D 4E F0 AD .._Y.9Za..M.]N..
0040 65 81 61 A3 39 D5 42 D0 EC F4 EC E7 B9 67 B2 30 e.a.9.B......g.0
0050 24 68 B3 A3 04 40 8B 0B B2 6D 50 96 C8 90 E8 28 $h...@...mP....(
0060 0A FB 89 8A FE F1 AA 91 81 09 2D 4D 9C 00 99 47 ..........-M...G
0070 7C 0D 83 64 60 6E 28 30 95 60 0E AE D8 50 07 7A |..d`n(0.`...P.z
0080 FD 38 1B 0D EA 24 2B 9B A3 00 EA 80 1F 43 7E 73 .8...$+......C~s
0090 8C 9B D4 5E 81 F4 A8 84 D0 35 F8 46 B5 8E EB C5 ...^.....5.F....
00A0 6A C4 D1 2A 8C 48 AA 6D 08 D9 2F E4 6F DF 72 25 j..*.H.m../.o.r%
00B0 CA 53 EE EA F5 D7 9E 06 C6 0C 91 18 6C D2 52 3E .S..........l.R>
00C0 BF 19 DC D6 0E F5 3D 1B ED 11 9E 91 3A 63 BC B7 ......=.....:c..
00D0 41 97 B2 6C 30 66 01 B1 65 12 8D BC C0 BC 2E CC A..l0f..e.......
00E0 2F 7A CD DB 60 AC 4B 87 F9 BE AA 06 DE 01 3C 5E /z..`.K.......<^
00F0 31 6D 01 61 C5 F6 BD D0 1E D2 23 B4 DE FA 45 2A 1m.a......#...E*
```

Figure 11: Decryption Key Records

The binary resource 124 contains the encrypted content of the secret. Only one `DEK` from the `DecryptionKeyRecords` can successfully decrypt the secret, using `RC5` in `CBC` mode. The first 256 bytes of the secret are shown in the Figure 12:

```
00000000 00 F9 A7 AD 70 B5 78 C6 34 B0 FC 12 3B 54 5B 78 ....p.x.4...;T[x
```

```
00000010 94 02 65 7E BC C1 19 C9 A2 65 79 23 FF E0 C2 D9 ..e~.....ey#....
00000020 9C 13 5C 39 D6 65 0D 9B 8C A5 5C F7 AA 40 F3 3C ..\9.e....\..@.<
00000030 A3 FC 73 31 85 F9 72 BB 7F 07 51 36 3F 6F 76 9D ..s1..r...Q6?ov.
00000040 99 47 89 E9 79 55 5B 6E BB 8F F1 D8 60 A6 24 41 .G..yU[n....`.$A
00000050 D3 FD 5F 33 5F 4A F1 C7 38 02 7D 86 98 D6 C0 96 .._3_J..8.}.....
00000060 3A E7 77 2A 2D 41 F8 8D 41 CE 75 F8 02 55 23 E9 :.w*-A..A.u..U#.
00000070 6A 29 4D 7F 25 71 84 49 EB 13 E2 F8 31 4D 94 D2 j)M.%q.I....1M..
00000080 56 27 21 F6 8F 0B A0 D9 08 31 22 6A 5F 97 F8 6B V'!......1"j_..k
00000090 BB 91 02 30 60 0D F6 08 83 99 00 54 A1 59 05 28 ...0`......T.Y.(
000000A0 70 33 69 C7 DD 9A AF 32 70 C9 8B 82 C0 D1 D5 DC p3i....2p.......
000000B0 E0 01 38 E7 0E 47 E6 52 97 C9 BA 9A 52 1B 52 10 ..8..G.R....R.R.
000000C0 DD F6 42 4C 28 9C 47 28 C8 8D D7 1A 83 18 C2 8A ..BL(.G(........
000000D0 83 21 DA C1 2C 22 0A 18 1F 3F 3E 30 06 F4 7D F2 .!..,"...?>0..}.
000000E0 CA 6E 8C CB 99 42 D8 91 E3 D3 EF B6 C7 00 CC DE .n...B..........
000000F0 7C 10 24 C5 25 C2 05 6F FA 22 D7 F2 7E 70 74 A3 |.$.%..o."..~pt.
```

Figure 12: Secret data

The last step of the challenge is to find the index of the DEK that will unlock the secret containing the email. The algorithm to determine the index uses the `MasterKey` data, `CurrentIter` count, a constant `0x31000C01`, a global variable `Fails` and `Div` and `BitCount` functions. All of the aforementioned data primitives are all constants except `CurrentIter` and `Fails`. `CurrentIter` depends on the index of the last record in the `Keks` array that has been decrypted. `Fails` is a global variable that is initialized to 0 and incremented every time a record in the `Keks` array fails to decrypt correctly (the computed hash of the record doesn't match the hash stored in the record).

The `Div` function is implemented to throw a `DivideByZero` exception when the divisor is 0.

The Figure 13 shows the algorithm used to compute the index of the correct DEK:

```
uint32_t Fails = 0;


class DivideByZero {
public:
  DivideByZero(uint val) : Value(val) {}
```

```
  uint GetValue() { return Value; }
private:
  uint Value;
};

uint32_t Div(uint x, uint y) {
  if (y == 0)
  {
    throw DivideByZero(x);
  }
  return x / y;
}

uint32_t GetIndex(uint8_t masterkey[HASHBYTES]) {
  const uint32_t EASY_VALUE = 0x31000C01;
  uint32_t dwVal = (uint32_t*)(masterkey)[1] | HASHBYTES;
  uint32_t index = 0;
  try {
    if (CurrentIter >> BitCount((uint32_t*)(masterkey)[2] | EASY_VALUE)) {
      uint dwVal1 = Div(dwVal,
                        CurrentIter>>BitCount((uint32_t*)(masterkey)[3]));
      index += (dwVal1 >> 16) + (Fails > 0);
    }
    else
      index += (dwVal >> 8) + (Fails > 0);
    index %= 16;
  }
  catch (DivideByZero divZero) {
    index += BitCount(divZero.GetValue()) >> 1;
```

```
    }
    return index;
}
```

Figure 13: GetIndex Algorithm

The expression `BitCount((uint32_t*)(masterkey)[3])` always evaluates to `0x1A` `(BitCount(0x7edfebfb))`. If the `CurrentIter` is either `0x01000000` or `0x02000000`, the expression `CurrentIter>>0x1A`, evaluates to 0. Due to the division by 0, an exception is thrown and the index of the DEK key is computed in the `catch` block. `divZero.GetValue()` returns the constant stored in `dwVal`, i.e. `0x766147f9`, and the index of the DEK is evaluated to 9 `(BitCount(0x766147f9) = 0x12, 0x12 >> 1 = 9)`.

So, the record at index 9 in the `Keks` array contains the 8-byte `Kek` that represents the key that decrypts the DEK at index 9 in the `DecryptionKeys` array. The RC5-39 rounds algorithm, in CBC mode, decrypts the secret shown in Figure 14:

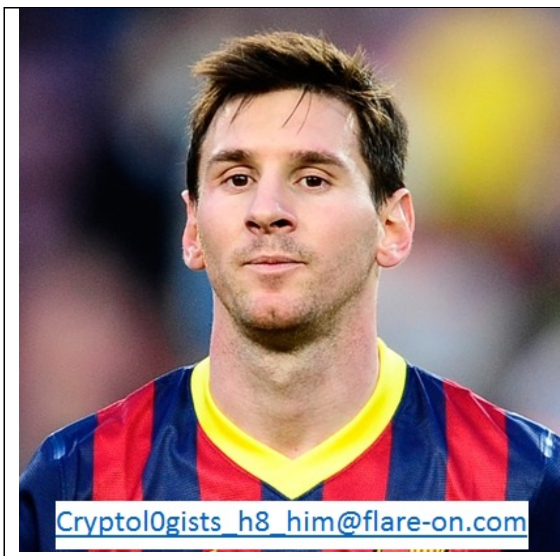

Figure 14: Successful Decryption