Learn to quickly detect RC4 encryption in (malicious) binaries

0xc0decafe.com/2020/12/23/detect-rc4-in-malicious-binaries

December 23, 2020

RC4 (also known as *ARC4*) is a simple stream cipher. It was designed in the late 1980s and its internals became known to the public in the mid-1990s. While it is a very simple and fast crypto algorithm, security researchers have discovered multiple flaws in it throughout the years. Today, it is just another broken stream cipher.

However, it is still used by software systems in the wild. Many malware families use it for encryption or better said: just for obfuscation purposes. Due to its simplicity and speed, malware authors embed it directly in their source code or statically link it into their binaries. For instance, *ZLoader* utilizes it to decrypt its configuration and *Smokeloader* encrypts its network traffic with this stream cipher.

Even though they could utilize one of the crypto APIs offered by Windows like the WinCrypt* functions, malware authors likely prefer this way as another way to ensure malware analysts' job security.

In contrast to other ciphers, *RC4* does not rely on any constants that make it easy for tools like findcrypt-yara to detect it in the binary. Nevertheless, tools like capa that take the assembly code structure of the binary into account are capable of detecting *RC4*. More on how *capa* does this later on.

Detection possibility: Key-Scheduling Algorithm (*KSA*)

While explaining *RC4* is out of scope of this blog post (Wikipedia does a great job!), one of the most interesting parts of the algorithm is *Key-Scheduling Algorithm* (KSA). In a nutshell, it initializes an internal array based on the provided key that is later utilized by another algorithm to encrypt / decrypt. In pseudo code *KSA* looks like this (taken from Wikipedia):

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

The internal array S contains all possible byte values from `0x00` to `0xFF`. It is permuted in the *KSA*. This usually compiles down to something like the following:

```
0000ED5276                          loc_ED5276:                           ; CODE XREF: rc4+3F↓j
0000ED5276 41 88 03                             mov     [r11], al
0000ED5279 FF C0                                inc     eax
0000ED527B 49 FF C3                             inc     r11
0000ED527E 3D 00 01 00 00                       cmp     eax, 100h
0000ED5283 72 F1                                jb      short loc_ED5276
0000ED5285 45 8B F2                             mov     r14d, r10d
0000ED5288 45 8B CA                             mov     r9d, r10d
0000ED528B 40 0F B6 F6                          movzx   esi, sil
0000ED528F 4C 8D 1C 24                          lea     r11, [rsp+108h+var_108]
0000ED5293
0000ED5293                          loc_ED5293:                           ; CODE XREF: rc4+80↓j
0000ED5293 45 0F B6 03                          movzx   r8d, byte ptr [r11]
0000ED5297 33 D2                                xor     edx, edx
0000ED5299 41 8B C1                             mov     eax, r9d
0000ED529C F7 F6                                div     esi
0000ED529E 41 FF C1                             inc     r9d
0000ED52A1 0F B6 0C 2A                          movzx   ecx, byte ptr [rdx+rbp]
0000ED52A5 41 03 CE                             add     ecx, r14d
0000ED52A8 41 03 C8                             add     ecx, r8d
0000ED52AB 44 0F B6 F1                          movzx   r14d, cl
0000ED52AF 42 8A 04 34                          mov     al, [rsp+r14+108h+var_108]
0000ED52B3 41 88 03                             mov     [r11], al
0000ED52B6 49 FF C3                             inc     r11
0000ED52B9 46 88 04 34                          mov     [rsp+r14+108h+var_108], r8b
0000ED52BD 41 81 F9 00 01 00 00                 cmp     r9d, 100h
0000ED52C4 72 CD                                jb      short loc_ED5293
0000ED52C6 45 8B CA                             mov     r9d, r10d
0000ED52C9 85 DB                                test    ebx, ebx
0000ED52CB 74 3B                                jz      short loc_ED5308
0000ED52CD 4C 8B DB                             mov     r11, rbx
0000ED52D0
0000ED52D0                          loc_ED52D0:                           ; CODE XREF: rc4+C2↓j
0000ED52D0 41 8D 41 01                          lea     eax, [r9+1]
0000ED52D4 44 0F B6 C8                          movzx   r9d, al
0000ED52D8 42 0F B6 14 0C                       movzx   edx, [rsp+r9+108h+var_108]
0000ED52DD 41 8D 04 12                          lea     eax, [r10+rdx]
0000ED52E1 44 0F B6 D0                          movzx   r10d, al
0000ED52E5 42 8A 04 14                          mov     al, [rsp+r10+108h+var_108]
0000ED52E9 42 88 04 0C                          mov     [rsp+r9+108h+var_108], al
0000ED52ED 42 88 14 14                          mov     [rsp+r10+108h+var_108], dl
0000ED52F1 42 0F B6 0C 0C                       movzx   ecx, [rsp+r9+108h+var_108]
0000ED52F6 03 CA                                add     ecx, edx
0000ED52F8 0F B6 C1                             movzx   eax, cl
0000ED52FB 8A 0C 04                             mov     cl, [rsp+rax+108h+var_108]
0000ED52FE 30 0F                                xor     [rdi], cl
0000ED5300 48 FF C7                             inc     rdi
0000ED5303 49 FF CB                             dec     r11
0000ED5306 75 C8                                jnz     short loc_ED52D0
```

The first and second blocks are the actual *KSA*. Note the two `cmp` instructions (`cmp eax, 100h` and `cmp r9d, 100h`). These are part of the two `for` loops are seen in the pseudo-code (lines 1 and 5). The third block is the *Pseudo-random generation algorithm* (*PRGA*) used to encrypt/decrypt the plain/ciphertext. I won't go much more into the details of the *PRGA*, please refer to the great Wikipedia article on RC4.

Detect RC4 encryption with yara

These two `for` loops in the *KSA* are something where we could detect the presence of *RC4* in the binary. But mind possible false positives! For years, I utilized a simple *yara* rule to detect this stream cipher.

```
rule rc4_ksa
 {
     meta:
         author = "Thomas Barabosch"
         description = "Searches potential setup loops of RC4's KSA"
     strings:
         $s0 = { 3d 00 01 00 00 } // cmp eax, 256
         $s1 = { 81 f? 00 01 00 00 } // cmp {ebx, ecx, edx}, 256
         $s2 = { 48 3d 00 01 00 00 } // cmp rax, 256
         $s3 = { 48 81 f? 00 01 00 00 } // cmp {rbx, rcx, …}, 256
     condition:
         any of them
 }
```

As you can see, this rule targets exactly the `cmp` instructions found in the *KSA*. While there may be better ways to do this, this is still a very fast approximation.

Detect RC4 encryption with capa

Nowadays, we have tools like *capa* that do a better job. But how does *capa* does it? I've promised to tell you: on one side, *capa* detects if a binary is linked against OpenSSL or imports WinCrypt functions. This is trivial as you can see in the rule linked-against-openssl.yml, which performs simple string matching:

```
rule:
  meta:
    name: linked against OpenSSL
    namespace: linking/static/openssl
    author: william.ballenthin@fireeye.com
    scope: file
    examples:
      - 6cc148363200798a12091b97a17181a1
  features:
    - or:
      - string: RC4 for x86_64, CRYPTOGAMS by <appro@openssl.org>
      - string: AES for x86_64, CRYPTOGAMS by <appro@openssl.org>
      - string: DSA-SHA1-old
```

On the other side, *capa* detects the KSA and PRGA algorithms of RC4 based on the assembly. This is more interesting since *capa* takes the structure of the binary into account. The rule encrypt-data-using-rc4-ksa.yml detects the *KSA* as follows:

```
rule:
  meta:
    name: encrypt data using RC4 KSA
    namespace: data-manipulation/encryption/rc4
    author: moritz.raabe@fireeye.com
    scope: function
    att&ck:
      - Defense Evasion::Obfuscated Files or Information [T1027]
    mbc:
      - Cryptography::Encrypt Data::RC4 [C0027.009]
      - Cryptography::Encryption Key::RC4 KSA [C0028.002]
    examples:
      - 34404A3FB9804977C6AB86CB991FB130:0x403D40
      - C805528F6844D7CAF5793C025B56F67D:0x4067AE
      - 9324D1A8AE37A36AE560C37448C9705A:0x404950
      - 782A48821D88060ADF0F7EF3E8759FEE3DDAD49E942DAAD18C5AF8AE0E9EB51E:0x405C42
      - 73CE04892E5F39EC82B00C02FC04C70F:0x40646E
  features:
    - or:
      - and:
        - basic block:
          - and:
            - description: initialize S
            # misses if regular loop is used,
            # however we cannot model that a loop contains a certain number
            - characteristic: tight loop
            - or:
              - number: 0xFF
              - number: 0x100
        - or:
          - match: calculate modulo 256 via x86 assembly
          # compiler may do this via zero-extended mov from 8-bit register
          - count(mnemonic(movzx)): 2 or more
        - or:
          - description: modulo key length
          - mnemonic: div
          - mnemonic: idiv
      - and:
        - description: optimized, writes DWORDs instead of bytes
        - or:
          - number: 0xFFFEFDFC
          - mnemonic: sub
        - or:
          - number: 0x03020100
          - mnemonic: add
        - number: 0x4040404
```

capa detects *RC4* in two ways. The first way consists of three parts (lines 20-29).

- a basic block with a tight loop counting to `0xFF` or `0x100`
- a match against another rule *calculate modulo 256 via x86 assembly* or two or more `movzx` mnemonics

- either a `div` or a `idiv` mnemonics that are utilized by the *KSA* for the module of `keylength` (see pseudo algorithm of *KSA*)

The second way detects optimizations where instead of bytes DWORDs are written by the *KSA* (lines 38-46). For instance, the password cracker John optimizes the K*S*A like this (see opencl_rc4.h). It comprises an initialized array of 64 DWORDs:

```
#ifdef RC4_IV32
__constant uint rc4_iv[64] = { 0x03020100, 0x07060504, 0x0b0a0908, 0x0f0e0d0c,
                               0x13121110, 0x17161514, 0x1b1a1918, 0x1f1e1d1c,
                               0x23222120, 0x27262524, 0x2b2a2928, 0x2f2e2d2c,
                               0x33323130, 0x37363534, 0x3b3a3938, 0x3f3e3d3c,
                               0x43424140, 0x47464544, 0x4b4a4948, 0x4f4e4d4c,
                               0x53525150, 0x57565554, 0x5b5a5958, 0x5f5e5d5c,
                               0x63626160, 0x67666564, 0x6b6a6968, 0x6f6e6d6c,
                               0x73727170, 0x77767574, 0x7b7a7978, 0x7f7e7d7c,
                               0x83828180, 0x87868584, 0x8b8a8988, 0x8f8e8d8c,
                               0x93929190, 0x97969594, 0x9b9a9998, 0x9f9e9d9c,
                               0xa3a2a1a0, 0xa7a6a5a4, 0xabaaa9a8, 0xafaeadac,
                               0xb3b2b1b0, 0xb7b6b5b4, 0xbbbab9b8, 0xbfbebdbc,
                               0xc3c2c1c0, 0xc7c6c5c4, 0xcbcac9c8, 0xcfcecdcc,
                               0xd3d2d1d0, 0xd7d6d5d4, 0xdbdad9d8, 0xdfdedddc,
                               0xe3e2e1e0, 0xe7e6e5e4, 0xebeae9e8, 0xefeeedec,
                               0xf3f2f1f0, 0xf7f6f5f4, 0xfbfaf9f8, 0xfffefdfc };
#endif
```

Now we can understand where the constants `0xFFFEFDFC` and `0x03020100` come from. Such an optimized version of *RC4* was actually utilized in the original XBOX bootloader (there we can also see the utilization of the DWORD `0x4040404`).

Let me tell you a tiny anecdote. A couple of years ago, a junior coworker reversed the whole *RC4* algorithm in a malicious binary and told some colleagues and me that they analyzed a custom crypto algorithm. We told them that this was just plain old *RC4*. The coworker was a little bit upset but they likely learned a lot from their tiny adventure in *RC4*. I hope that now you are capable of spotting *RC4* in (malicious) binaries and I've just saved you a couple of hours of reversing.