"Guardians of the Secrets: Analyzing Cryptographic Algorithms in Malicious Code"

Muhammad Ahmad

[Kill_The_Malware]

RC4 Unveiled : The Three Stages of Secrets - KSA, PRGA, and XOR Operations"

- Rc4 have 3 stages
  - KSA [key Scheduling Algorithm]
  - PRGA [Pseudo Random Generation Algo]
  - XOR operations
- KSA Stage:
  - During initialization, a list of values from 0 to 255 is created. These values are rearranged by swapping them based on calculations involving two indices, S[j] and S[i].
- PRGA [Pseudo Random Generation Algo]
  - Generates key stream by rearranging values in the list.
  - Produces bytes, limited up to 256.
  - Using modulo % with 256 for simplicity.
  - 3 iterations using 256, 1 swapping step.

```
RC4(const std::vector<unsigned char>& key) : S(256), i(0), j(0) {
    for (int i = 0; i < 256; ++i) {
        S[i] = i;
    }

    int j = 0;
    for (int i = 0; i < 256; ++i) {
        j = (j + S[i] + key[i % key.size()]) % 256;
        std::swap(S[i], S[j]);
    }
}
```

KSA

```
unsigned char generateByte() {
    i = (i + 1) % 256;
    j = (j + S[i]) % 256;
    std::swap(S[i], S[j]);
    return S[(S[i] + S[j]) % 256];
}
```

PRGA

```
for i in range(N):
    encrypted_byte = plaintext[i] XOR keystream[i]
    ciphertext.append(encrypted_byte)
```

XOR

Rc4 in Dharma malware

- Rc4 code in C

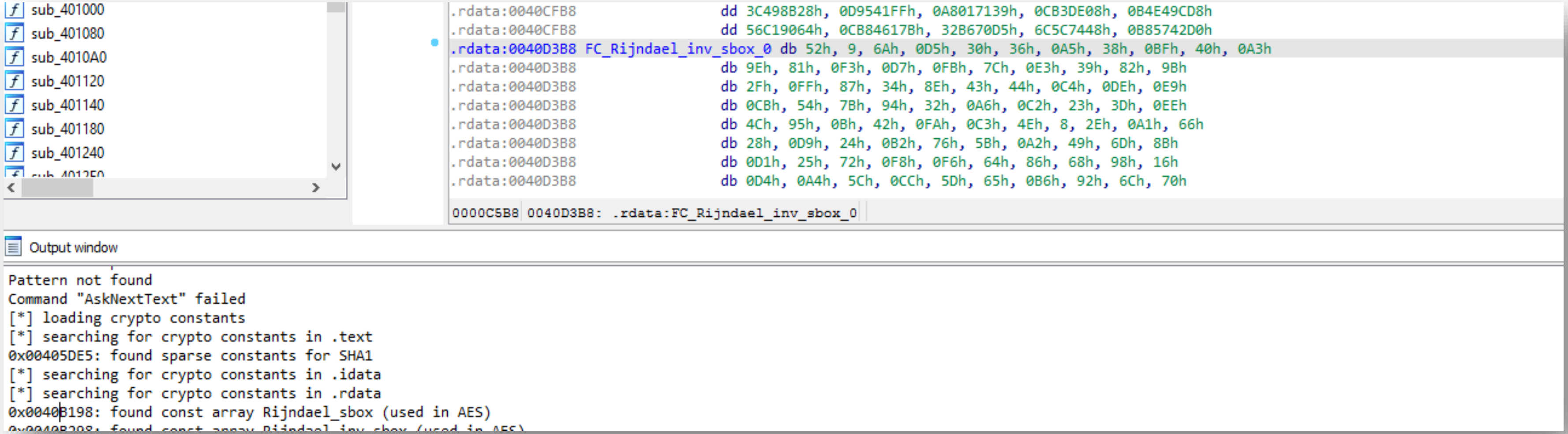
```
def rc4_encrypt(data, key):
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    i = 0
    j = 0
    encrypted = bytearray()
    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        keystream_byte = S[(S[i] + S[j]) % 256]
        encrypted.append(byte ^ keystream_byte)

    return bytes(encrypted)

key = b'SecretKey'
plaintext = b'This is a secret message.'

encrypted = rc4_encrypt(plaintext, key)
```

Dharma Code analysis

```
sub_401000(v17, 256);
v3 = 0;
v4 = a1;
v5 = v19;
v7 = v4 - v6;
do
{
  v8 = &v17[v3];
  v9 = *(_BYTE *)((v3 & 0x1F) + v5);
  v8[v7] = v3++;
  *v8 = v9;
}
while ( v3 < 256 );
v10 = v18;
v11 = (unsigned __int8 *)v18;
v12 = &v17[-v18];
do
{
  v13 = *v11;
  v14 = (v21 + (char)v11[(_DWORD)v12] + *v11) % 256;
  result = *(_BYTE *)(v14 + v10);
  *v11++ = result;
  v16 = v20-- == 1;
  v21 = v14;
```

AES Revealed:Unmasking AES with Find-Crypt

- Discovering the AES algorithm in malware samples is made easier by examining lookup tables like S-Boxes or T-Tables, depending on the implementation of AES.
- To expedite the search process, tools such as Find-Crypt can be used. Findcrypt is an IDA Python plugin designed for recognizing algorithm constants.
- In the specific case mentioned (REvil), Find-Crypt not only identified constants related to the Salsa20 encryption algorithm but also recognized AES tables, which are indicative of AES encryption being used within the malware.
- string Rijndael_inv_sbox which are associated with the AES encryption algorithm checkout for AES Source Code https://android.googlesource.com/platform/external/openssh/+/idea133/rijndael.c



The Hidden Secrets of Blowfish Implementation

- Discovering the Smilarities include the presence of two loops, each iterating 18 times, which are employed for initializing the P-Array.
- Additionally, there is a loop that iterates four times, with an internal loop that runs 256 times, serving the purpose of initializing the S-Boxes.

Assembly Code

```
memmove(v16 + 4, &unk_5605C0, 0x48u);
memmove(v16 + 22, &unk_560608, 0x1000u);
v11 = v17;
v13 = 0;
v10 = 0;
for ( i = 0; i < 0x12; ++i )
{
  v13 = 0;
  v7 = 4;
  while ( v7-- )
  {
    v13 = (v13 << 8) | (unsigned __int8)*v11++;
    if ( ++v10 == Size )
    {
      v10 = 0;
      v11 = v17;
    }
  }
  v16[i + 4] ^= v13;
}
v8 = 0;
v9 = 0;
for ( i = 0; i < 0x12; ++i )
{
  sub_422610(&v8);
  v16[i++ + 4] = v8;
  v16[i + 4] = v9;
}
for ( j = 0; j < 4; ++j )
{
  for ( k = 0; k < 256; ++k )
  {
    sub_422610(&v8);
0021282 sub_421D20:47 (421E82)
```

BlowFish.c

```
memcpy(keystruct->p,p_perm,sizeof(WORD) * 18);
memcpy(keystruct->s,s_perm,sizeof(WORD) * 1024);

  // Combine the key with the P box. Assume key is standard 448 bits
(56 bytes) or less.
  for (idx = 0, idx2 = 0; idx < 18; ++idx, idx2 += 4)
     keystruct->p[idx] ^= (user_key[idx2 % len] << 24) |
(user_key[(idx2+1) % len] << 16)
                          | (user_key[(idx2+2) % len] << 8) |
(user_key[(idx2+3) % len]);
  // Re-calculate the P box.
  memset(block, 0, 8);
  for (idx = 0; idx < 18; idx += 2) {
     blowfish_encrypt(block,block,keystruct);
     keystruct->p[idx] = (block[0] << 24) | (block[1] << 16) |
(block[2] << 8) | block[3];
     keystruct->p[idx+1]=(block[4] << 24) | (block[5] << 16) |
(block[6] << 8) | block[7];
  }
  // Recalculate the S-boxes.
  for (idx = 0; idx < 4; ++idx) {
     for (idx2 = 0; idx2 < 256; idx2 += 2) {
        blowfish_encrypt(block,block,keystruct);
        keystruct->s[idx][idx2] = (block[0] << 24) | (block[1] << 16)
|
                                  (block[2] << 8) | block[3];
        keystruct->s[idx][idx2+1] = (block[4] << 24) | (block[5] <<
16) |
                                   (block[6] << 8) | block[7];
```