Automating Pikabot’s String Deobfuscation | ThreatLabz

By Nikolaos Pantazopoulos

Published: 2024-04-08 · Archived: 2026-04-02 10:47:52 UTC

Technical Analysis

Strings obfuscation

The steps for decrypting a Pikabot string are relatively simple. Each string is decrypted only when required (in other words, Pikabot does not decrypt all strings at once). Pikabot follows the steps below to decrypt a string:

1. Pushes on the stack the encrypted string array.
2. Initializes the RC4 encryption algorithm. The RC4 key is different for each string (with very few exceptions).
3. Pikabot takes the decrypted RC4 output, decodes it using Base64 after replacing all instances of the character ‘`_`’ (underscore) with ‘`=`’ (equal) and decrypts it using the AES-CBC algorithm. The AES key and initialization vector (IV) are the same for all strings.

ANALYST NOTE: *There are encrypted strings, which are encrypted only with the RC4 algorithm.*

Figure 1 shows the code used to decrypt the string, `Kernel32.dll`.

```
1232    Encrypted_Array[0] = 0x94BA738E;
1233    Encrypted_Array[1] = 0x54E67CE0;
1234    Encrypted_Array[2] = 0xFB3986A0;
1235    v107 = 0;
1236    Encrypted_Array[3] = 0x6E7E32CE;
1237    Encrypted_Array[4] = 0xB7EE6DED;
1238    Encrypted_Array[5] = 0xE0B6D82A;
1239    strcpy(&rc4_key[4], "bad file descriptor");
1240    do
1241    {
1242      v232[v107 + 8] = v107;
1243      ++v107;
1244    }
1245    while ( v107 < 0x100 );
1246    v108 = v245;
1247    for ( i20 = 0; i20 < 0x100; ++i20 )
1248    {
1249      v110 = v232[i20 + 8];
1250      v108 *= 1353392136;
1251      v111 = (LOBYTE(v243[0]) + rc4_key[i20 % 0x13 + 4] + v110);
1252      v243[0] = v111;
1253      v232[i20 + 8] = v232[v111 + 8];
1254      v232[v111 + 8] = v110;
1255    }
1256    v243[0] = 0;
1257    v112 = byte_44D64C[0];
1258    v113 = v244;
1259    dword_44D604 = v108;
1260    LOBYTE(v108) = 0;
1261    for ( loop_counter = 0; loop_counter < 0x18; ++loop_counter )
1262    {
1263      v108 = (v108 + 1);
1264      v115 = 0;
1265      if ( v112 )
1266      {
1267        while ( ++v115 <= 1662 )
1268        {
1269          if ( !byte_44D64C[v115] )
1270            goto LABEL_114;
1271        }
1272        dword_44D604 = v113 - 275141729;
1273      }
1274  LABEL_114:
1275      v116 = v232[v108 + 8];
1276      v113 = (v113 ^ 0xD1A529C6) - 2129106622;
1277      v243[0] = (v116 + LOBYTE(v243[0]));
1278      v232[v108 + 8] = v232[v243[0] + 8];
1279      v232[v243[0] + 8] = v116;
1280      RC4_Decrypted_String[loop_counter] = *(Encrypted_Array + loop_counter) ^ v232[(v116 + v232[v108 + 8]) + 8];// Kernel32.dll
1281    }
1282    RC4_Decrypted_String[24] = 0;
1283    dword_44D5F0 = v113 | 0xDDAF5034;
1284    Referenceof_RC4_Decrypted_String = RC4_Decrypted_String;
1285  }
1286  decrypted_string = base64_decode_aes_decrypt_string(Referenceof_RC4_Decrypted_String);
```


Figure 1: Example Pikabot string decryption for `Kernel32.dll`.

Figure 2 shows the function that first decrypts the AES key and IV. The RC4 decrypted string passed to the function is then Base64 decoded, and is finally decrypted using AES.

```
LOBYTE(v10) = 0;
for ( j = 0; j < 0x3D; ++j )
{
  v10 = (v10 + 1);
  v12 = 116;
  v43 = aes_key[v10 + 64];
  v34 = v43;
  v13 = (v9 + v43);
  v14 = L"text file busy";
  v35 = v13;
  while ( (v12 - 48) <= 9u )
  {
    v12 = *++v14;
    if ( !*v14 )
      goto LABEL_13;
  }
  v8 &= 0xA3051FFB;
LABEL_13:
  v8 += 521471124;
  junk_data_16(v14);
  v9 = v35;
  aes_key[v10 + 64] = aes_key[v35 + 64];
  aes_key[v9 + 64] = v43;
  aes_key[j] = *(v39 + j) ^ aes_key[(v34 + aes_key[v10 + 64]) + 64];// 8udKGMtQHxWEL=nfSJ0oLjQP=JrxOgvkaVSUs7sd+aizzbigv7Fovx
}
aes_key[61] = 0;
dword_44D59C = v8 & 0x217B6741;
v37[0] = -792047170;
v37[1] = 229441174;
v37[2] = 1585924466;
v37[3] = 1523544232;
strcpy(&v31[1], "GlobalCollection");
v37[4] = -631102418;
v37[5] = -1494269296;
v37[6] = 2039842685;
v37[7] = -2015108016;
v37[8] = -1680064469;
v37[9] = 2086525404;
v37[10] = -540614038;
v37[11] = -1145579218;
v38 = -3215;
LOBYTE(v15) = 0;
for ( k = 0; k < 0x100; ++k )
  aes_key[k + 64] = k;
for ( m = 0; m < 0x100; ++m )
{
  v18 = aes_key[m + 64];
  v15 = (v15 + *(&v31[1] + (m & 0xF)) + v18);
  aes_key[m + 64] = aes_key[v15 + 64];
  aes_key[v15 + 64] = v18;
}
LOBYTE(v19) = 0;
LOBYTE(v20) = 0;
for ( index = 0; index < 0x32; ++index )
{
  v20 = (v20 + 1);
  v22 = aes_key[v20 + 64];
  v19 = (v19 + v22);
  aes_key[v20 + 64] = aes_key[v19 + 64];
  aes_key[v19 + 64] = v22;
  *(aes_IV + index) = *(v37 + index) ^ aes_key[(v22 + aes_key[v20 + 64]) + 64];// 3=XfbWjYdFdqAsyD76UliliZfsQzYqOdPPGbuhTIWGc
}
loop_counter = v33;
input_string = v36;
BYTE2(v41) = 0;
while ( loop_counter < sizeof_input_string )
{
  if ( input_string[loop_counter] == '_' )
    input_string[loop_counter] = '=';
  ++loop_counter;
}
size = get_size(input_string);
sizeof_base64_decoded_data = base64_decode(input_string_1, size);
sizeof_decrypted_string = aes_decryption(input_string, sizeof_base64_decoded_data, aes_key, aes_IV);
input_string[sizeof_decrypted_string] = 0;
return sizeof_decrypted_string;
```


Figure 2: Pikabot Base64 decoding and AES decryption function.

Decrypting Pikabot strings

The following information is required to decrypt a Pikabot string:

- The AES key and IV of a binary sample.

- The RC4 encrypted array of each string.
- The RC4 key of each encrypted string.
- The string's size.

Our approach relies on IDA's microcode. This decision helped us with several problems such as:

- IDA's microcode converts the assignment/copy of the RC4 key into a `strcpy` function. In the assembly level, this could either be multiple `mov` or `rep` instructions. As a result, it would make the detection and extraction harder and more challenging.
- Extracting the RC4 encrypted array. Since IDA reconstructs the stack, it makes it much easier to search and extract the encrypted array.

IDA's microcode brings other limitations (for example, decompilation failure for a function) but no such issues were encountered for the parts of the code we wanted to analyze.

In the sections below, we describe how each component was extracted.

Extracting the AES key/IV

For the extraction of the AES key and IV, we iterate all analyzed functions and discard any function, whose size is not in the range of 600 and 1,600 bytes.

Next, we scan the functions for the following patterns:

- Existence of RC4 encryption. This is the same heuristic we use for detecting encrypted RC4 strings.
- Existence of values 0x3D and 0x5F (used before Base64 decoding the string) that are used with microcode opcodes `m_stx` and `m_jnz` respectively.

Lastly, if all of the patterns above match, then the handler for decrypting a Pikabot string is invoked. For the classification of the key and the IV, we apply the following checks:

- The number of decrypted strings from the identified function must be two. Otherwise, the identified function is incorrect.
- The longest string is marked as the AES key (by taking the first 32-bytes) and the remaining decrypted string as the IV (by taking the first 16-bytes).

Extracting the RC4 encrypted array

Pikabot constructs the RC4 encrypted array by pushing it onto the stack and then decrypting it. Our approach involves the following steps for detecting each part of the array:

- Use the detected RC4 encryption block address as a starting point.
- Search for the microcode opcode `m_add` in the decryption instruction. The detected microcode holds the starting stack offset of the encrypted array.
- Start iterating backwards and search for the microcode opcodes `m_mov/m_call`, the second opcode is used in case the data is copied via a `strcpy` or `memcpy` instruction. If the stack offset matches, then we save

the data and update the stack offset. This process is repeated until the reconstructed encrypted array has the expected size.

Extracting the RC4 encrypted array size

The length of the encrypted array is extracted in a similar way as the encrypted array. The detection pattern is:

- Use the detected RC4 encryption block address as a starting point.
- Search for the microcode opcodes `m_jb`, `m_jae`, and `m_setb`, and use the immediate constant number in the instruction as a size.

Extracting the RC4 key

Extracting the RC4 key of each string proved to be the most challenging part while creating the plugin. In our first attempt, we were extracting the RC4 key after detecting the initialization of the RC4 algorithm. However, this approach had the following issues:

- **Incorrect extraction of the RC4 key:** In many cases, an invalid/junk string was placed in-between the correct RC4 key and the RC4 algorithm initialization.
- **Incorrect detection of RC4 initialization code block:** For example, if the size of the encrypted array was 256 bytes then an incorrect RC4 key would be detected.

Instead of trying to detect the RC4 key by detecting the initialization of the RC4 algorithm, we decided to extract all strings from each targeted function. Then, we decrypted the RC4 encrypted array with each extracted RC4 key and validated the decrypted output by applying the following checks:

- If it matches the expected string size.
- If all characters of the string are readable.

ANALYST NOTE: *After successful decryption, the RC4 key is marked and not reused in order to limit any false-positives. For example, if the decrypted string does not have any junk characters.*

Explore more Zscaler blogs
