Summary:

This report is on the Ransomware as a Service REvil, detailing their techniques, any changes to previous samples they used and their obfuscation techniques. While writing this analysis, I used a reiteration of my personal analysis as well as additional documentation of changes I saw in the code base. While there are slight differences between the two, this is worth looking into as the code flow changes during this analysis.

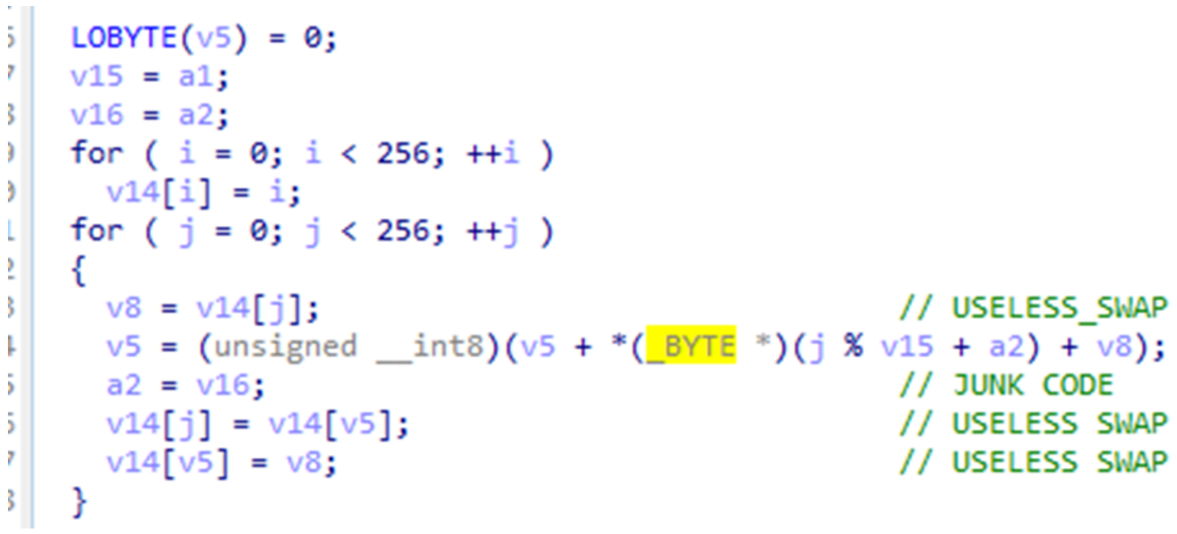
Sample Hash

Sha256:0c10cf1b1640c9c845080f460ee69392bfaac981a4407b607e8e30d2ddf903e8

Tactics and Techniques used for obfuscation

Junk Code

While this is a minor detail, the changes to add junk code to established algorithms did slow down my analysis since adding junk code increases the difficulty of pattern matching to more dated samples.

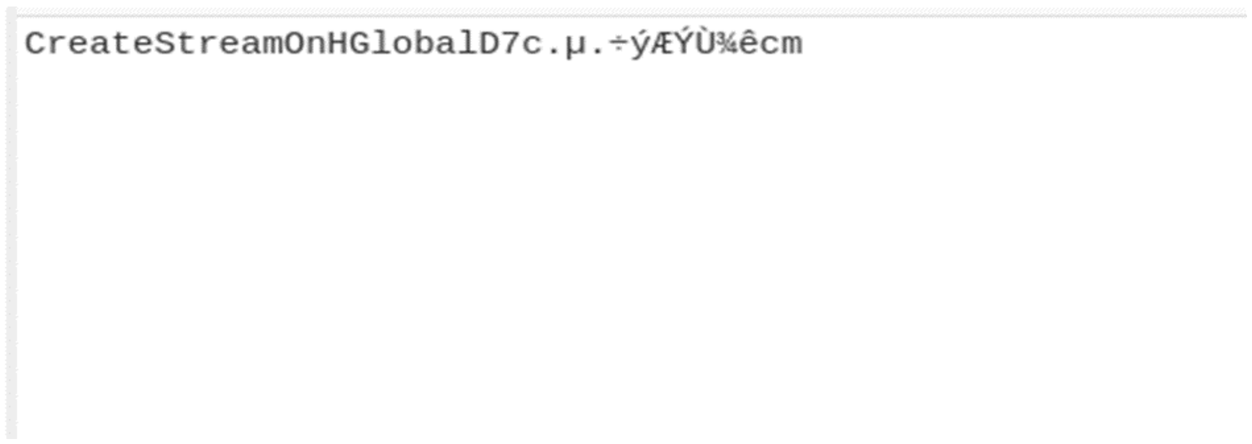


```
LOBYTE(v5) = 0;
v15 = a1;
v16 = a2;
for ( i = 0; i < 256; ++i )
  v14[i] = i;
for ( j = 0; j < 256; ++j )
{
  v8 = v14[j];                                      // USELESS_SWAP
  v5 = (unsigned __int8)(v5 + *(_BYTE *)(j % v15 + a2) + v8);
  a2 = v16;                                         // JUNK CODE
  v14[j] = v14[v5];                                 // USELESS SWAP
  v14[v5] = v8;                                     // USELESS SWAP
}
```

Looking at this sample we see swaps which try to change the way that the RC4 decryption algorithm appears. Again, these changes are minor, but the technique causes the code to seem more complex. These kinds of operations could be put in place to make the code analysis more difficult.

RC4 Decryption

All string decryption is done via the RC4 algorithm. Connecting back to the previous technique of adding excess code, a majority of the data decrypted is junk data. This makes it harder to read and an analyst can overlook this fact. The usage of junk code and data is apparent to slow down the analysis and prevent automation.

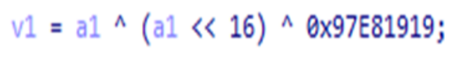


```
CreateStreamOnHGlobalD7c.µ.÷ýÆÝÙ¾êcm
```

An example of this is the screencap above, part of this is garbage text. I believe that creating a modified RC4 script,which the malware has already done, is the best way to bypass this tactic.

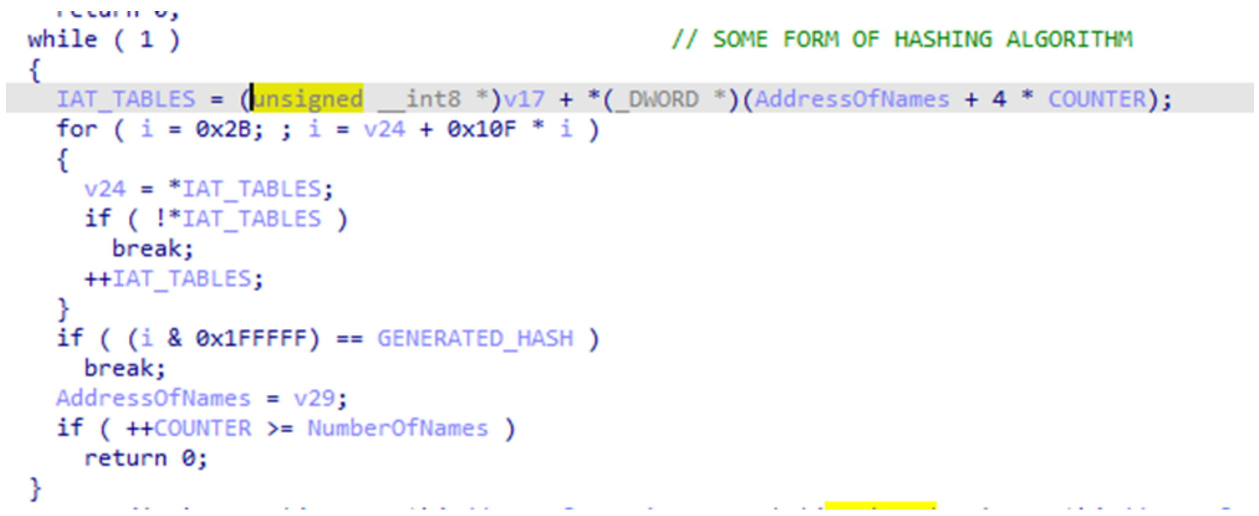
API Hashing

API Hashing utilizes 195 API which are obfuscated hashes. It dynamically allocates these hashes by using the _LIST_Entry and the InMemoryOrderModuleList structure. First the DWORD which is assigned a number is transformed or deobfuscated.



```
v1 = a1 ^ (a1 << 16) ^ 0x97E81919;
```

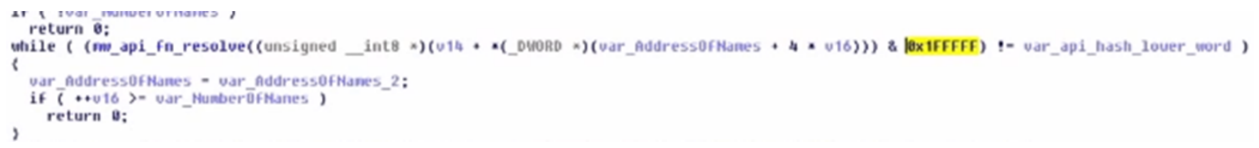
This image shows the lower DWORD part of the result. The XOR value I have seen in this is changed between the samples. This is done to stop quick automation.



```
while ( 1 )                                    // SOME FORM OF HASHING ALGORITHM
{
  IAT_TABLES = (unsigned __int8 *)v17 + *(_DWORD *)(AddressOfNames + 4 * COUNTER);
  for ( i = 0x2B; ; i = v24 + 0x10F * i )
  {
    v24 = *IAT_TABLES;
    if ( !*IAT_TABLES )
      break;
    ++IAT_TABLES;
  }
  if ( (i & 0x1FFFFF) == GENERATED_HASH )
    break;
  AddressOfNames = v29;
  if ( ++COUNTER >= NumberOfNames )
    return 0;
}
```

This result is then tested against the hash generated later in the code or against a specific function name.

The hashing algorithm is simple but it should be noted that the structure of the code I saw in my analysis is different from the analysis I have seen previously. To see more of these differences I would direct you to OALabs who did an analysis video here, (https://www.youtube.com/watch?v=hM2Zvsak3GM)



```
  return 0;
while ( (mw_api_fn_resolve((unsigned __int8 *)(v14 + *(_DWORD *)(var_AddressOfNames + 4 * v16))) & 0x1FFFFF) != var_api_hash_lower_word )
{
  var_AddressOfNames = var_AddressOfNames_2;
  if ( ++v16 >= var_NumberOfNames )
    return 0;
}
```

In this image, you can see there is a separate subroutine in the code. The subroutine algorithm is the same as previous iterations, but the difference should be noted for pattern matching to other samples. The API hashing deobfuscation script can be found on my GitHub.

CRC32 Hashing of Configuration File

The program takes a CRC32 hash of the configuration file before using, this has been documented before so I will not be detailing it in this post.. If you want to read more about this, I would suggest this post.

Conclusion:

The main purpose of this analysis was to look at the changes I saw in REvil from previous samples but also to practice looking at API hashing as well as having my own documentation of the sample to look back on. The main purpose of REvil

is to create ransomware so I will not go into detail about that part of the sample, but the nature of the configuration files and how it went about obfuscating itself was quite interesting.