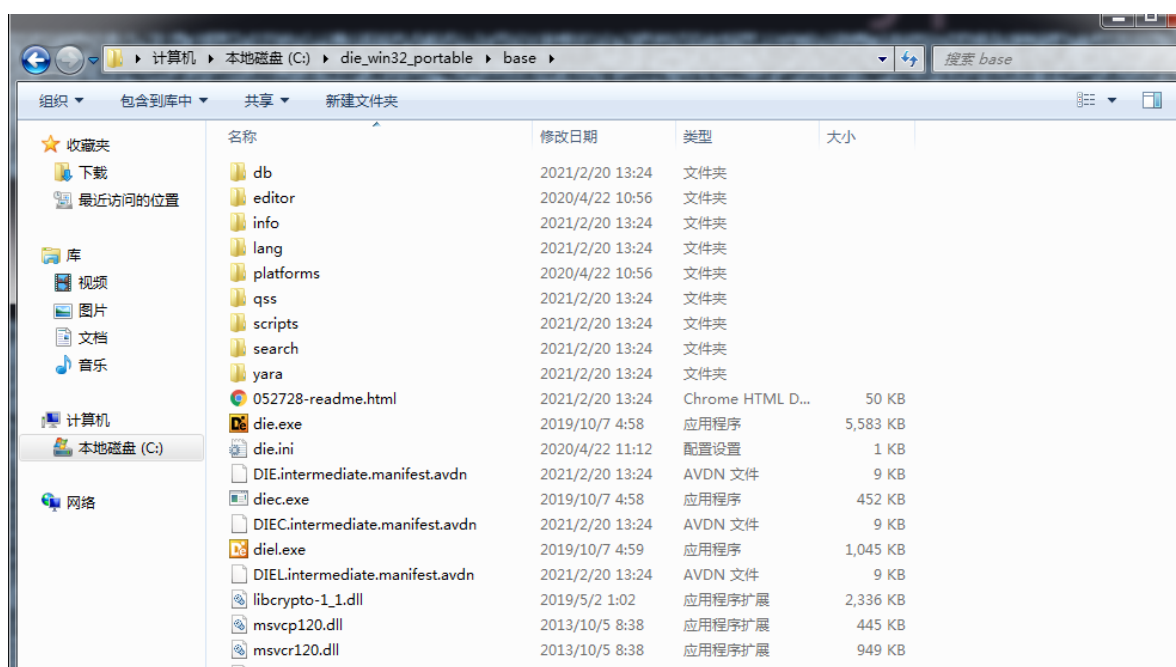


Avaddon勒索
解密工具
解密工具原理
解密工具优化
关于文件大小的疑惑
Avaddon勒索加密流程补充
解密工具实现
相关资料

Avaddon勒索

该勒索病毒使用C++语言进行编写，采用RSA-2048和AES-256加密算法对文件进行加密，加密库使用的是Windows自带的CryptAPI



被该勒索加密后的文件后缀为avdn

解密工具

国外安全研究人员发布了一款Avaddon勒索病毒解密工具，解密工具源代码地址：

<https://github.com/JavierYuste/AvaddonDecryptor>

经过测试，这个工具确实是可以解密被Avaddon勒索加密的文件，下面是我输出的解密时的日志

```

8 Removing signature
9 Size of the encrypted file with signature: 156184
0 Size of the encrypted file without signature: 155648
1 Decrypting file
2 Decrypting file C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\1049\License_SysClrTypes.rtf.avdn with k
3 Decrypting C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\1049\License_SysClrTypes.rtf.avdn
4 Truncating to 152187
5 > Found file C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\2052\License_SysClrTypes.rtf.avdn
6 Signature: b'\n#\x02\x00\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x07\x03\x03\x01\x01\x01\x89\x05'
7 Original file size unpacked: 140042 (0x2230a)
8 Hardcoded signature: 0x1030307
9 Removing signature
0 Size of the encrypted file with signature: 147992
1 Size of the encrypted file without signature: 147456
2 Decrypting file
3 Decrypting file C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\2052\License_SysClrTypes.rtf.avdn with k
4 Decrypting C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\2052\License_SysClrTypes.rtf.avdn
5 Truncating to 140042
6 > Found file C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\3082\License_SysClrTypes.rtf.avdn
7 Signature: b'\xac\xaa\x01\x00\x00\x00\x00\x00\x02\x00\x00\x01\x00\x00\x00\x07\x03\x01\x01\x01\x89\x05'
8 Original file size unpacked: 109228 (0x1aaac)
9 Hardcoded signature: 0x1030307
0 Removing signature
1 Size of the encrypted file with signature: 115224
2 Size of the encrypted file without signature: 114688
3 Decrypting file
4 Decrypting file C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\3082\License_SysClrTypes.rtf.avdn with k
5 Decrypting C:\Program Files (x86)\Microsoft SQL Server\90\Shared\Resources\3082\License_SysClrTypes.rtf.avdn
6 Truncating to 109228
7
8 --- SUMMARY ---
9 Total files: 141136
0 Decrypted files: 4212
1 Time: 182.6574169
2

```

想要解密被加密的文件 需要具备下面几个条件

1. 当前存活的勒索进程
2. 勒索进程的dump文件
3. 一份被加密的文件
4. 一份被加密文件的原始文件

然后调用下面这个命令

```
python3 main.py -f <encrypted_file> -o <original_file> -d <memory_dump> --folder
<folder_to_decrypt>
```

就可以解密机器上所有的被加密文件了（需要修改源码中写死的三个路径 才能把解密脚本跑起来）

解密工具原理

目前我的需求是把这个解密能力集成的到公司的工具里，再来分析一下代码

```

def main():
    # 解析命令行参数
    args = p.parse_args()
    # 获取被加密的原始文件名
    original_file = os.path.abspath(args.original)
    # 获取被加密的文件名
    encrypted_file = os.path.abspath(args.file)
    # 分割被加密文件的文件名和后缀
    filename, file_extension = os.path.splitext(encrypted_file)
    # 获取dump文件路径
    dump = os.path.abspath(args.dump)
    # 这个是密钥特征码
    pattern = "106600000100000020000000"

    # 在dump文件中搜索特征码 返回偏移列表
    # Get a list of offsets of the matches with searchbin
    offsets = search_pattern(dump=dump, pattern=pattern)
    print(f"Offsets: {offsets}")

    # 从搜索到的列表中 获取每一个可能的key
    # Get each possible key from the list of offsets
    possible_keys, pointers_to_possible_keys = get_keys_from_offsets(dump=dump, offsets=offsets)
    print(f"Pointers to possible keys: {pointers_to_possible_keys}")
    print(f"Possible keys: {possible_keys}")

```

首先在dump文件中搜索所有可能的密钥，然后返回一个偏移列表，再根据这个偏移列表，去拿到所有的key

```
Offsets: [7465731, 32681323, 32687019, 32688443, 32691291, 32700547, 32705531, 35507403, 35508115, 35511675, 35512387, 35521643, 35524491, 35530899, 35537307, 35545851, 35556531, 35563
Structure found: 10660000010000002000000005011b700
Structure found: 1066000001000000200000000d0319704
Structure found: 106600000100000020000000010489704
Structure found: 1066000001000000200000000a04a9704
Structure found: 1066000001000000200000000c0589704
Structure found: 1066000001000000200000000e07c9704
Structure found: 106600000100000020000000060909704
Structure found: 106600000100000020000000030c1d604
Structure found: 1066000001000000200000000f8c3d604
Structure found: 1066000001000000200000000e0bd1d604
Structure found: 1066000001000000200000000a084d604
Structure found: 1066000001000000200000000d0f8d604
Structure found: 1066000001000000200000000f03d704
```

输出的日志显示offset有90个，也就是说有90个AES的密钥

```
# 尝试每一个密钥 一直到解密成功
# Try each key till success
success = False
i = 0
possible_keys = list(dict.fromkeys(possible_keys))
while not success and i < len(possible_keys):
    # Decrypt file 解密文件 调用c++程序
    decrypted_file = decrypt_file(data['encrypted_truncated_file'], possible_keys[i])

    # Truncate to original size 截断为原始大小
    with open(decrypted_file, "r+b") as f:
        f.truncate(data["original_size"])
    # Compare with the original file 与原始文件比较
    success = filecmp.cmp(decrypted_file, original_file, shallow=True)

#不成功则继续解密文件
if not success:
    i = i + 1
```

然后利用搜索到的key去解密文件，每解密一次，都去和源文件进行比对，比对成功则说明密钥正确

```
# 如果成功 打印出正确的密钥 并开始解密整个文件
if success:
    print(f"[SUCCESS] Found the correct symmetric key: {possible_keys[i]}")
    os.remove(data['encrypted_truncated_file'])
    decrypt_whole_system(args.folder, possible_keys[i], file_extension)
else:
    shutil.copy(f"{encrypted_file}.backup_copy", encrypted_file)
    os.remove(f"{encrypted_file}.backup_copy")
    print("[FAIL] Did not find the correct symmetric key")
```

比对成功之后，开始解密整个系统的文件。

```
def decrypt_file(file, key):
    print(f"\tDecrypting file {file} with key {key}")
    with open("key_bytes", "wb") as f:
        f.write(key)
    key_path = os.path.abspath("key_bytes")
    # Invoke C++ program, which decrypts a specified file with a given key 调用c++程序，用给定的密钥解密指定的文件
    abs_path = os.path.abspath(file)
    filename, file_extension = os.path.splitext(abs_path)
    print(f"\tDecrypting {abs_path}")
    subprocess.run(["DecryptFile.exe", abs_path, filename, key_path], stdout=subprocess.PIPE)
    if os.stat(abs_path).st_size > 0x100000:
        with open(abs_path, "r+b") as f:
            mm = mmap.mmap(f.fileno(), 0)
            with open(filename, "r+b") as f2:
                # Memory-map the file, size 0 means whole file
                mm.seek(0x100000)
                f2.seek(0, 2)
                while mm.tell() < mm.size() and (mm.size() - mm.tell()) > 0x2000:
                    f2.write(mm.read(0x2000))
                if mm.tell() < mm.size():
                    f2.write(mm.read(mm.size() - mm.tell()))
            mm.close()
    return filename
```

解密文件时，传入被加密的文件路径和解密后的文件路径以及密钥文件路径，然后调用DecryptFile.exe对文件进行解密

需要特殊处理的是，如果文件大小大于0x100000个字节，那么解密完成之后需要将0x100000字节后的数据全部复制到解密后的文件。

也就是说这个勒索实际上只会加密前0x100000个字节，这个细节在目前已有的分析报告中并没有提及。

```

59 def decrypt_whole_system(rootdir, key, extension):
60     print("Decrypting whole system")
61     total_files = 0
62     total_encrypted_files = 0
63     start_time = time.perf_counter()
64     for subdir, dirs, files in os.walk(os.path.abspath(rootdir)):
65         for file in files:
66             total_files += 1
67             file_path = os.path.abspath(os.path.join(subdir, file))
68             if (file_path.endswith(extension)) and "$Recycle.Bin" not in file_path:
69                 try:
70                     print(f"> Found file {file_path}")
71                     total_encrypted_files += 1
72                     data = get_signature_data(file_path)
73                     print(f"\tRemoving signature")
74                     data['encrypted_truncated_file'] = remove_signature(file_path)
75                     print("\tDecrypting file")
76                     decrypted_file = decrypt_file(data['encrypted_truncated_file'], key)
77                     print(f"\tTruncating to {data['original_size']}")
78                     with open(decrypted_file, "r+b") as f:
79                         f.truncate(data["original_size"])
80                     os.remove(file_path)
81                 except OSError:
82                     print("Permissions denied?")
83                 pass
84
85     print(f"\n--- SUMMARY ---")

```

解密完成之后，将文件截断为原始文件大小

那么这里其实有一个问题，为什么可以根据内置的特征码搜索到AES密钥？那个密钥的特征码是哪来的？

作者在代码中给出了这样一句注释

```

# Dump the process with procdump.exe -ma <PID>
# Pattern to search for (part of the key_data_s structure, in particular alg_id,
# flags and key_size):
# 106600000100000020000000

```

根据这个提示，找到了这个结构体，原文出处：<https://forums.codeguru.com/showthread.php?79163-Structure-of-HCRYPTKEY-Data>

```

struct key_data_s
{
    void *unknown; // XOR-ed
    uint32_t alg;
    uint32_t flags;
    uint32_t key_size;
    void* key_bytes;
};

```

勒索采用的是AES256，那 alg = 0x00006610, keysize=0x00000020, flags= 0x1

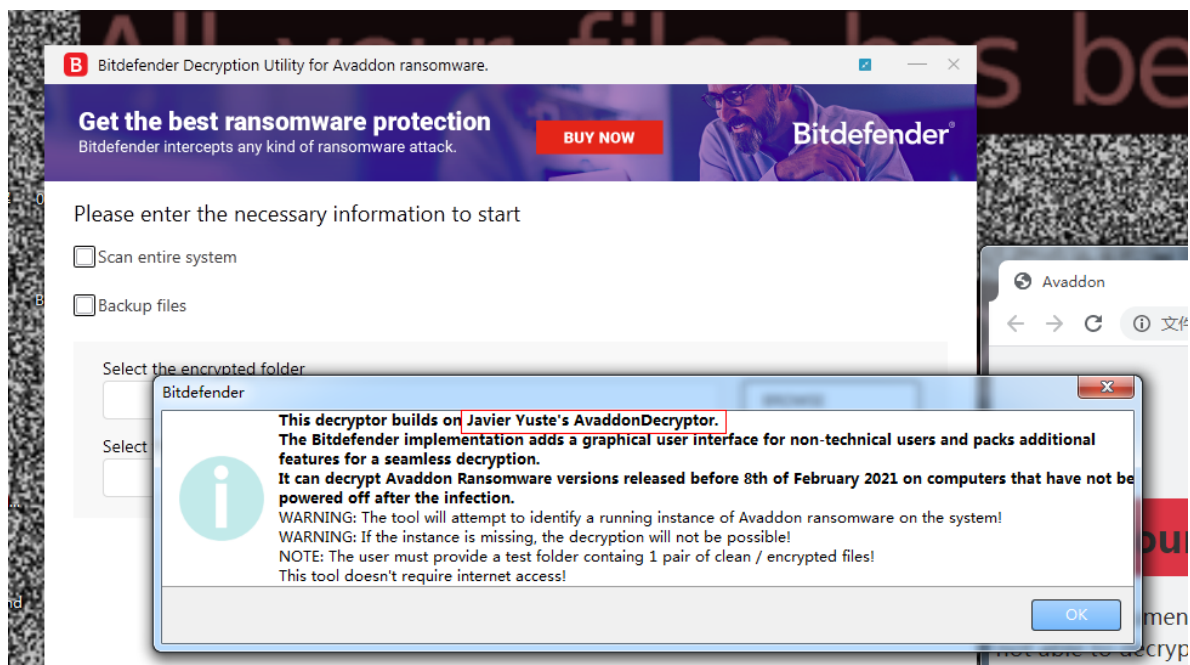
则特征值对应：106600000100000020000000

这个结构体来自于CryptApi，作者是逆向了cryptsp.dll和rsaenh.dll这两个dll得到的这个数据结构。

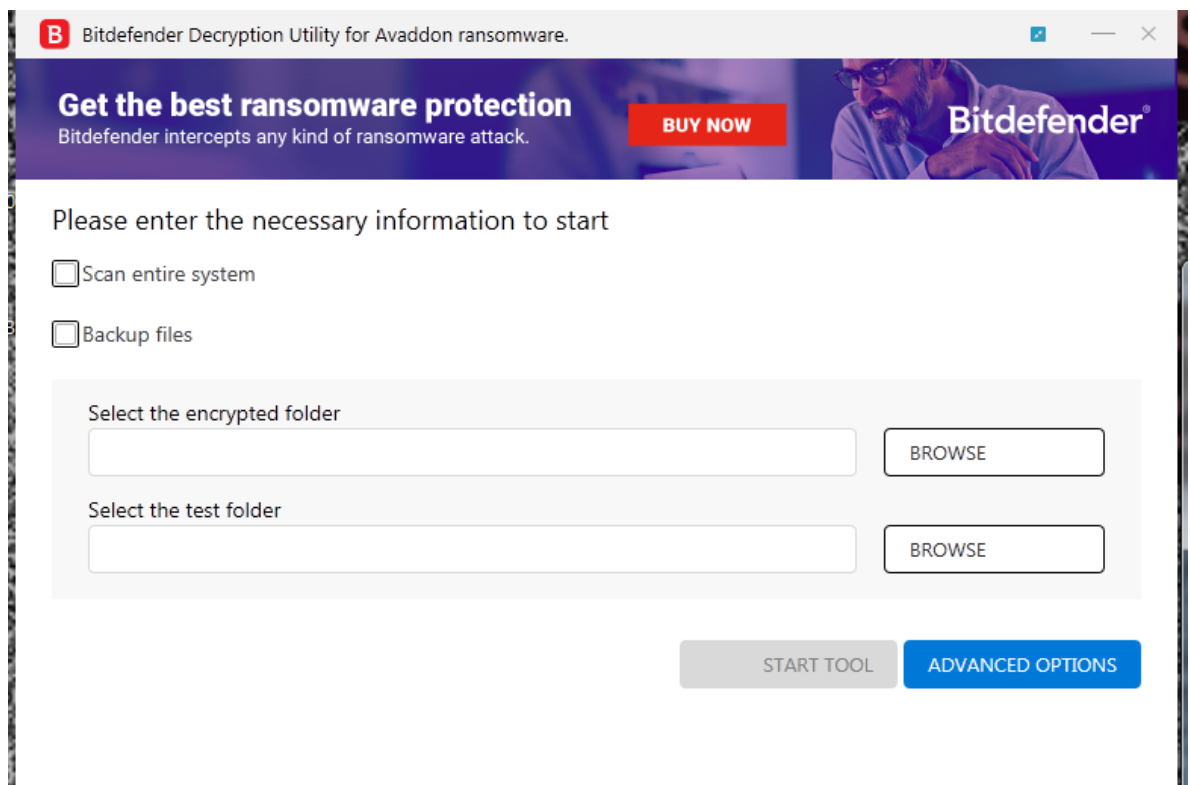
也就是说，这种在内存中暴力搜索密钥去解密被加密文件的方式，只适用于调用了CryptApi，并且随机生成密钥的情况。

那么有没有可能将作者的源码进行优化呢？答案是有

解密工具优化



在这之后，bd也针对该勒索发布了一款解密工具，根据提示，也是基于上面的代码做了一个图形化的工具而已。



工具只需要一个被加密文件和被加密前的源文件，不需要选择进程，不需要dump文件，就能对整个文件夹进行解密，但是我这里测试是解密失败的。分析一下这个工具有没有什么可借鉴的地方

```

if ( Process32FirstW(v4, &pe) )
{
    while ( !getAesKeys_sub_1000A460(v1, pe.th32ProcessID) )
    {
        if ( !Process32NextW(v4, &pe) )
        {
            v6 = 0;
            goto LABEL_26;
        }
    }
}

```

bd的做法是遍历整个系统的进程

```

{
    |
    v8 = (const void **)v2[20];
    while ( 1 )
    {
        v26 = Buffer.RegionSize + v2[20];
        if ( (signed int)(v26 - (_DWORD)v8) < 12 )
            break;
        v9 = 8;
        v29 = (const void **)(v26 - 12);
        v10 = &v37;
        v11 = v8;
        while ( 1 )
        {
            v12 = *v11;
            if ( *v11 != (const void *)*v10 )
                break;

```

直接在内存里匹配密钥，

```

do
{
    v21 = *v20;
    v22 = (_BYTE *)sub_1000D392(32);
    lpAddress = v22;
    if ( ReadProcessMemory(hObject, v21, v22, 0x20u, &NumberOfBytesRead) && NumberOfBytesRead )
    {
        if ( *v22 || v22[1] || v22[2] )
        {
            v23 = v30 + 0x18;
            v24 = (_DWORD *)v30[25];
            if ( v24 == (_DWORD *)v30[26] )
            {
                sub_1000C3E0(v23, (unsigned int)v24, &lpAddress);
            }
            else
            {
                *v24 = v22;
                v23[1] += 4;
            }
        }
        else
        {
            sub_1000D1A8(v22);
        }
    }
    ++v20;
}

```

如果匹配完成，就直接读取，然后调用解密程序。

关于文件大小的疑惑

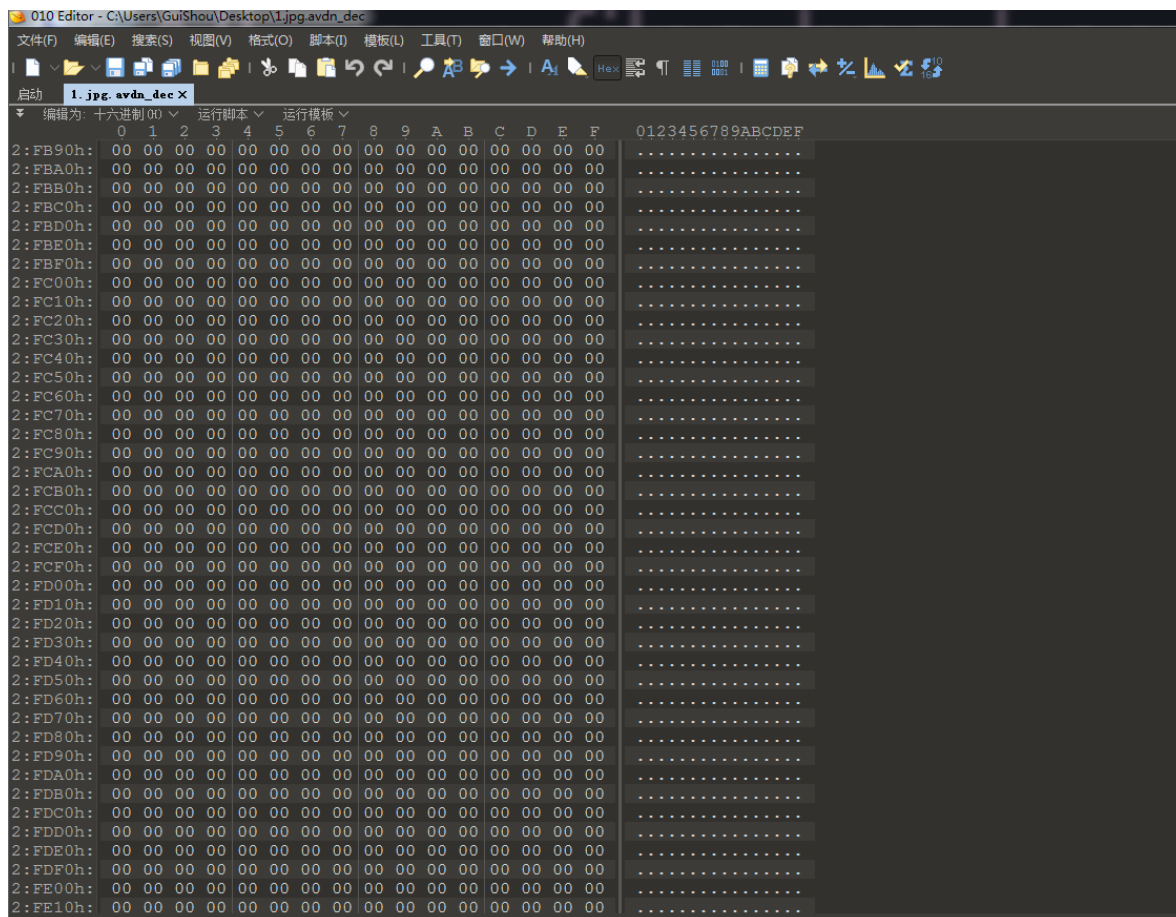
根据网上的分析报告提示


```
explorer.exe_108.dll Microsoft.SqlServer.ConnectionInfo.xml avdn X Pe_x86_to_file_topo
Edit As: Hex Run Script Run Template
0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
1:FF60h: 7D DE 1A 6A 12 DE 99 57 5E F8 F5 17 ED 32 FA 52 )P.j.P^W^øö.i2úR
1:FF70h: 07 E6 13 13 B9 DC 9A 63 FD 52 91 5E 5A OE A5 CO .æ..üšcýR^Z.¥À
1:FF80h: 89 2F 67 24 62 87 1F F9 62 D3 3B 3F AE FO 64 AA %/g$b+.ùbó;?@ød^
1:FF90h: B2 25 25 1C 3C 0F 02 36 2C 03 10 BA A8 CE 29 68 ^%ö.iU,.d~æZ.×
1:FFA0h: AF 9F 54 1F 3C 0F 02 36 2C 03 10 BA A8 CE 29 68 ^YT.\m.6 h!°"i)h
1:FFB0h: 14 FO OA 27 14 C2 7E 61 A5 B2 07 DE 46 00 9E CE .8.'Â~a~^P.F.žİ
1:FFC0h: 78 D3 5A 86 OE EE 7B 74 64 5D 59 EC ED E5 46 A4 xÓZ+.i(tdjYiiaF#
1:FFD0h: 02 27 43 69 63 AA 7D 12 63 A3 7D 57 98 C4 3D AB .'Cic^}.c£)W~Å=«
1:FFE0h: B0 C3 A1 53 C5 A6 02 FC F6 8B A7 A0 B9 7E C4 48 °Å;SÅ!.úö<S ^~ÅH
1:FFF0h: 9D 17 01 83 OA 84 11 29 A6 F5 40 B4 BC OF AC EE ...f...);öø^~i
2:0000h: EF 29 E4 10 80 D3 73 AF 12 3D D5 DF CO 1B FE 40 /)ä.eóö~.=öBÀ.pö
2:0010h: E3 CA 92 EC F3 9B 31 16 27 B1 A4 14 A6 D5 B8 20 äE'io>1.'±x.¡ö
2:0020h: FA BF D4 OD AE 51 45 E8 45 70 5C A6 4E FB CE 2C úö.öQEëEp\;Nüî,
2:0030h: 39 C5 D6 8D 81 85 85 OC AA 90 FF 4D 3F EE FB AC 9Åö.....^yM?iû~
2:0040h: 84 4E 12 FA OA OC DA C8 07 3E DE 6A 96 DC FE 85 „N.ú..Üë.>Pj~Üp.
2:0050h: 77 6F 3A 25 78 73 F1 5C DD C2 DD 12 4A 88 1A EE wo:~xsH\YÄY.J^i
2:0060h: 0B 1A B8 EF 20 A5 68 15 62 51 F8 F4 1E 52 E7 73 ...i Wh.bQëö.Rçs
2:0070h: EA 27 B6 91 07 7E 0B OD OE 4D D0 B5 CC 5F 24 F7 é'¶~...Möüî_ç÷
2:0080h: 9B FF FB BA B5 FA E0 A4 1D A9 62 20 19 3D 48 31 >yú°púä~.öb =H1
2:0090h: 31 >-/'ás[ö'8Söy'J>
2:00A0h: A1 öö'a~ÅjÖRiWzä.6i
2:00B0h: E4 FB 28 E5 6F 65 65 65 65 65 65 65 65 65 65 65 äü(ÄWl.*;e.úf.=Å
2:00C0h: CF 00 C4 77 6F 65 65 65 65 65 65 65 65 65 65 65 I.Äw~c^).ë-i+Ü.
2:00D0h: B3 78 48 1F 6F 65 65 65 65 65 65 65 65 65 65 65 ^xH..ûz;-gÅWCBÁ^
2:00E0h: 09 B9 05 2E OE 7B D3 73 1C E7 4F D5 A6 18 B9 DF .^...{ós.çöÖ!..B
2:00F0h: 9D 40 E2 AO OD 64 EB 99 1B 7B 7E 7D EB 34 48 67 .öÄ..dë^..(~)ë4Hg
2:0100h: 78 42 65 FO OE 05 A9 C3 C8 D1 33 2E 35 2F 56 A3 xBeö..öÄëN3.5/Vë
2:0110h: 55 77 CC 7B 12 39 B6 13 83 32 E8 2C CE 1C 7D 21 Uwî(.9¶.f2è,î.)!
2:0120h: C6 66 63 23 FO 4F 53 E4 ED 44 A5 OF 42 D9 CD 04 Efc#öOSaidW.BÜÍ.
2:0130h: E9 OA 9F 76 5C OD F1 62 4B 22 EC BB 77 A7 F2 1C é.Yv\..HbK"i»wSö.
2:0140h: 7D 76 C2 EA 1F 9C 90 56 EC DE 4B 11 8C 1E BA 4B }vÅë.æ.VipK.Ö.°K
2:0150h: B3 AE D7 2C F7 69 EC 68 OC 93 AF E5 B7 21 1C 93 ^öx,÷iîh.^~Ä.!.^
2:0160h: 8C AB B8 D8 E4 D9 B4 12 E6 FB FE DB A9 D2 26 6A Ö'öaÜ'.æüpÜöÖg
2:0170h: 40 CD D8 C3 AD A3 CB B3 E8 AA 87 AF A1 14 7A 17 öíoÄ-ëE^ë^+~;z.
2:0180h: A5 61 63 89 1C 73 36 1B F1 9F 11 42 B2 76 FA A1 Vact..s6.ÄY.B^vú;
2:0190h: 71 D8 OF 3E ED EC B7 11 84 48 1A 59 31 3E 52 D6 qö.>iî..H.Y1>RÖ
2:01A0h: 7A BE 5C 7F 28 E5 EO C3 90 4E E1 A7 A5 56 BD F8 zK\..(ääÄ.NáS¥V~ö
2:01B0h: 5B FC 41 8B 63 9C 4A C9 A5 CE B8 57 7B 02 7C 81 [üA<coeJëWî,W(.|.
2:01C0h: BA 9D B7 DF 4A 1C E2 C1 64 A2 58 85 41 06 9C D6 °..öJ.äÄdcX..A.æö
2:01D0h: AB FE FF 71 98 48 FF 4F 3E 19 56 F3 A7 D3 16 2C «pÿq~HyO>.VóSÓ.,
2:01E0h: 4F AB 5F 23 40 94 9A 74 D3 FO 18 38 OE 83 5B 02 O«_#ö'ätöó.8.f[.
2:01F0h: B6 EA 20 F6 87 60 49 85 F6 68 DD 11 FB E8 53 71 qé ö+`I.öhY.üëSq
2:0200h: 0B EB 01 00 00 00 00 00 00 02 00 00 01 00 00 00 .ë.....
2:0210h: 07 03 03 01 01 01 BC 02 .....K.
```

最后附加的24个字节中前4个字节是原始文件大小，但这个大小似乎不太对。如果原始文件大小是0x01EB0B，那么

```
0x01EB0B-512-24=0x1E8F3
```

但上图被加密文件数据的大小是0x20000，这个文件大小这么来看的话是对不上的。直到我即将完成我的解密工具的时候，解密出来的文件在末尾总是会出现一堆0。

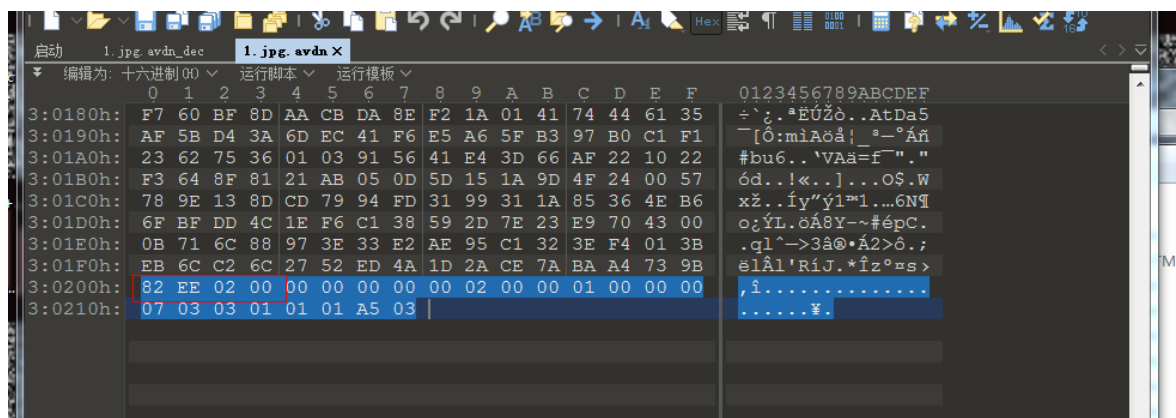


末尾的这一堆0，直接导致了解密后的文件和解密前的文件md5对比失败，一开始我以为是程序逻辑上的bug，但是后来发现并不是。实际上

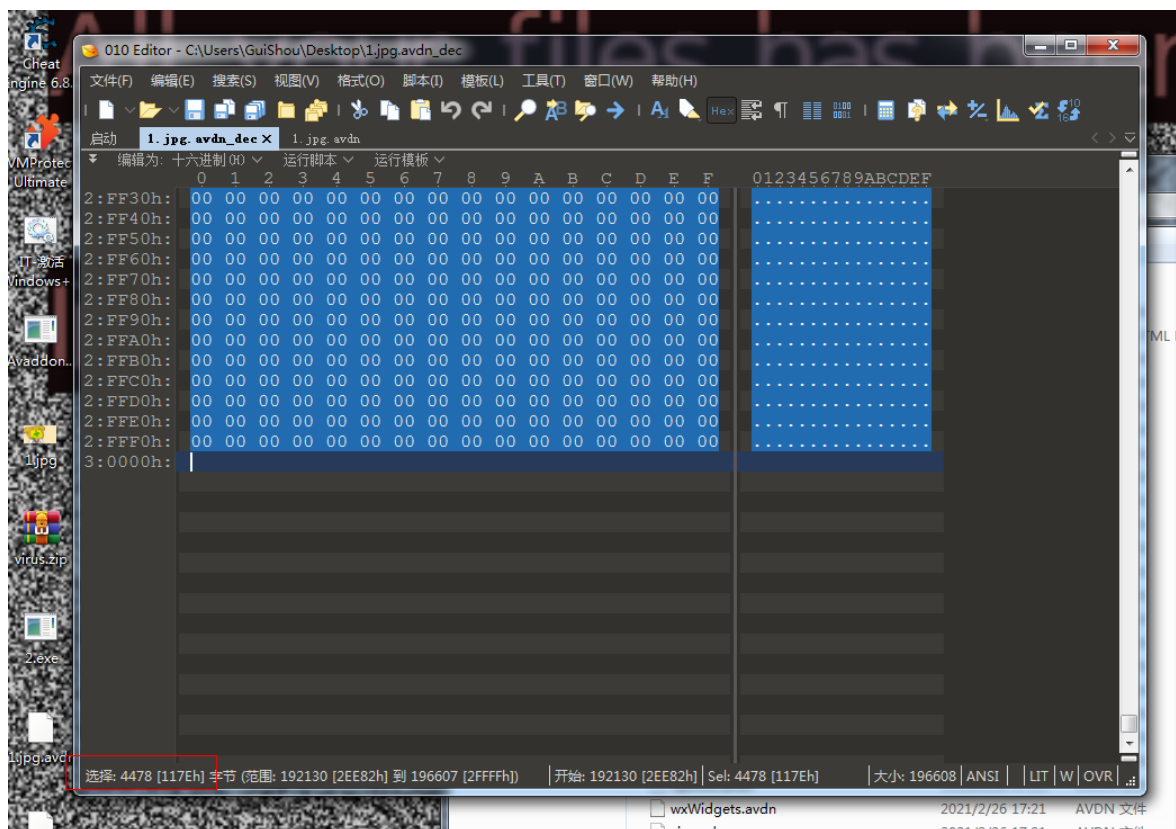
文件的计算方式如下：

加密后的文件大小(小于0x100000)=原始文件大小+填充0的字节数+512+24

以下面某个文件为例：



- 原始文件大小为：0x2EE82
- 加密后的文件大小为:0x30218



- 填充0的字节数为4478

$0x30218 = 0x2EE82 + 4478 + 512 + 24$

最后填充的0实际上是为了对齐到0x2000个字节，因为该样本每次会加密0x2000个字节，如果不足2000那么在加密的过程可能会导致异常退出。

Avaddon勒索加密流程补充

解决了文件大小的问题，这里对Avaddon勒索的AES加密流程做一个补充。

对于文件大小小于0x10000的文件，首先会在文件末尾填充0，将大小补齐到0x2000，然后将样本进行加密处理，每次加密0x2000个字节

对于文件大小小于0x10000的文件，不需要进行填充，直接加密前0x10000个字节，0x10000以后的部分不做加密处理

解密工具实现

那么到这里，已经填完了所有的坑，可以做一个相对来说最优化的解决方案。整个解密流程如下：

```
BOOL Check(LPCTSTR lpszFile);
```

首先判断是否是该家族的加密文件，判断条件有三个，两个末尾24字节写死的特征码和文件大小的计算是否满足条件

```
void GetValidPid();
```

首先获取有效进程的PID，遍历整个进程，并且获取进程映像文件的md5，将md5和注册表启动项中的映像文件做对比，如果对比成功，说明可能是潜在的勒索进程

```
ScanAddress(hProcess, (char*)"106600000100000020000000", 0);
```

接着遍历所有有效进程，搜索特征码，获取到所有可能的Key

```
BOOL GetUniqueKey(LPCTSTR szSourceFile, LPCTSTR szEncryptedFile);
```

用所有可能的Key文件去解密被加密文件，如果解密出来的文件和源文件md5一致，那么视为密钥获取成功

```
void DecryptAllFiles(LPCTSTR FileDirectory);
```

开始解密整个需要解密的目录

```
//勒索解密函数
| BOOL Decrypt(LPCWSTR lpszOldFile, LPCWSTR lpszNewFile);
| //Aes解密
| BOOL AesDecrypt(LPCWSTR lpszOldFile, LPCWSTR lpszNewFile);
| //获取原始文件大小
| DWORD GetOriginalFileSize(LPCWSTR lpszFile);
| //特征码搜索
| uintptr_t ScanAddress(HANDLE process, char *markCode, int nOffset, unsigned long dwReadLen = 4, uintptr_t StartAddr = 0, uintptr_t EndAddr = 0);
| //从偏移中获取有效密钥
| void GetValidKeyFromOffset(HANDLE hProcess);
| //计算密钥
| BOOL CalcKey(LPCTSTR szSourceFile, LPCTSTR szEncryptedFile);
| string byteToHexStr(unsigned char byte_arr[], int arr_len);
| void HexString2Bytes(const std::string& hex, BYTE* bytes);
| //计算文件的md5
| string CalcMd5(LPCTSTR lpFilePath);
| //获取唯一密钥
| BOOL GetUniqueKey(LPCTSTR szSourceFile, LPCTSTR szEncryptedFile);
| //获取启动项md5
| void GetBootInfoMd5(vector<string>& md5arr);
| //获取有效的进程ID
| void GetValidPid();
| //判断是否为被加密文件
| BOOL Check(LPCTSTR lpszFile);
| //提升为Debug权限
| BOOL EnableDebugPrivilege();
| //挂起进程
| void SuspendProcess(DWORD dwPid, BOOL isSuspend);
| //重新设置文件大小
| void ResetFileSize(LPCWSTR lpszFile, LONG lDistanceToMove, DWORD dwMoveMethod);
| //解密所有文件
| void DecryptAllFiles(LPCTSTR FileDirectory);
```

整个解密工具写了快1100行代码，花了6天左右。工程这里就不发了，记录一下整个过程和一些踩过的坑。

相关资料

勒索分析: <https://www.freebuf.com/articles/others-articles/249109.html>

解密工具: <https://github.com/JavierYuste/AvaddonDecryptor>

勒索解密工具分析: https://mp.weixin.qq.com/s?_biz=MzA4ODEyODA3MQ==&mid=2247486514&idx=1&sn=6464b9066980a6c045a33ef58dd5b6b0&chksm=902fa31aa7582a0c88242eb816fb466cbb7f94fe33b3e8b14f1f769d6d493961eaf9721cc87d#rd

