Aura Stealer #1 36bytesmademelosemymind

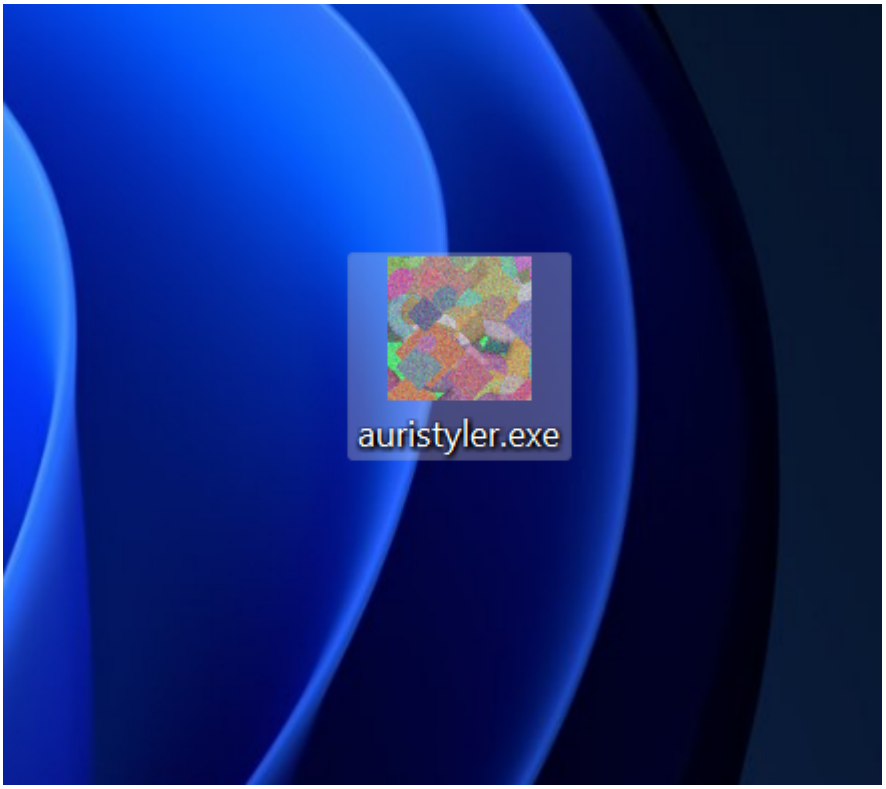
Published: 2025-10-22 · Archived: 2026-04-06 01:06:56 UTC

Hello it’s me again

today i gonna share my thought process and experience with aura stealer

i was looking for some new malware with either a virtual machine or some neat obfuscation. lookin through some forums and news i saw a stealer named `Aura Stealer` mentioned couple of times. it does not seem to be a big stealer yet but lets see how lumma doing in the next couple of months with all the law enforcement on their asses.

sooo i decided to focus on that one i got the hash from this blog https://foresiet.com/blog/aura-stealer-malware-analysis/



the random generated icon looks kinda cool tho :3

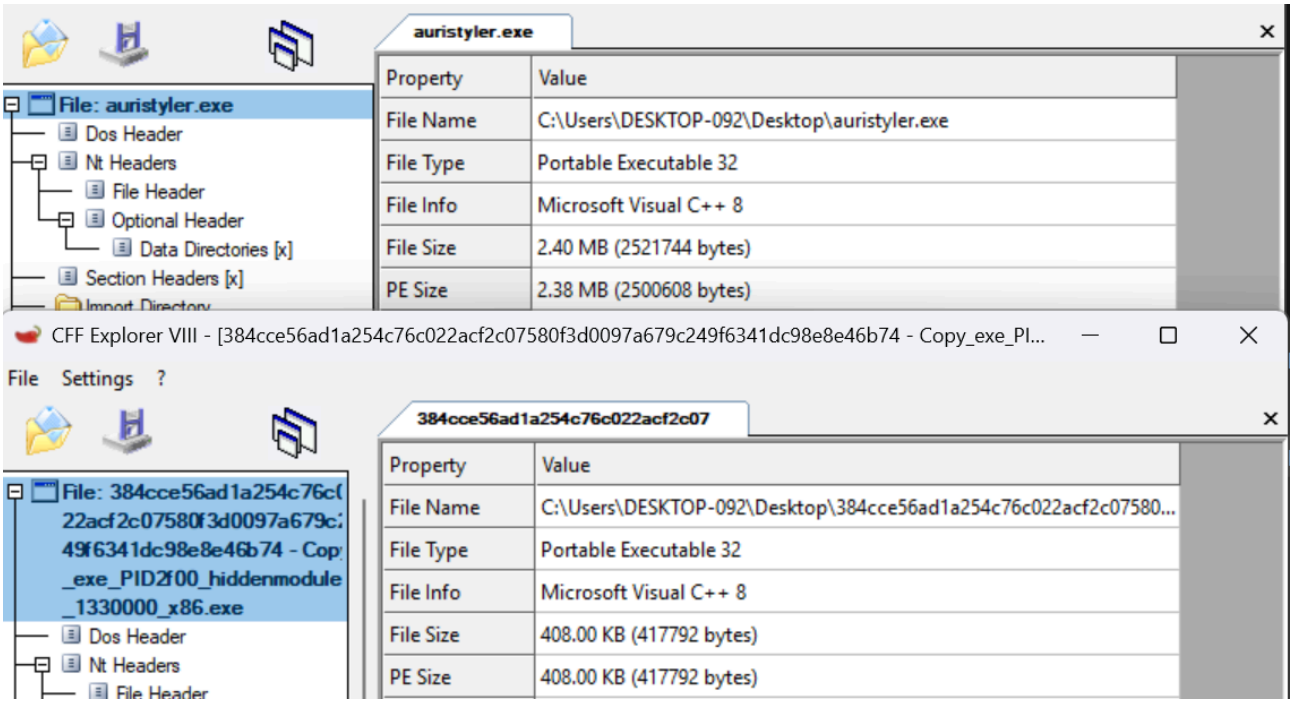
Functions

Function name	Segment	Start
sub_54C871	.text	0054C871
sub_54C684	.text	0054C684
sub_54C5F7	.text	0054C5F7
sub_54C5E0	.text	0054C5E0
sub_54C5A0	.text	0054C5A0
sub_54C160	.text	0054C160
sub_54C14A	.text	0054C14A
sub_54C09C	.text	0054C09C
sub_54C03D	.text	0054C03D
sub_54BFB9	.text	0054BFB9
sub_54BF73	.text	0054BF73
sub_54BF47	.text	0054BF47
sub_54BEF3	.text	0054BEF3
sub_54BEB7	.text	0054BEB7
sub_54BD3C	.text	0054BD3C
sub_54BCD3	.text	0054BCD3
sub_54BC10	.text	0054BC10
sub_54BA1A	.text	0054BA1A
sub_54B907	.text	0054B907
sub_54B8C4	.text	0054B8C4
sub_54B898	.text	0054B898
sub_54B69E	.text	0054B69E
sub_54B53D	.text	0054B53D
sub_54B4DF	.text	0054B4DF
sub_54B4B2	.text	0054B4B2
sub_54B3C0	.text	0054B3C0
sub_54AF44	.text	0054AF44
sub_54AF30	.text	0054AF30
sub_54AEE4	.text	0054AEE4
sub_54AE1C	.text	0054AE1C
sub_54AE13	.text	0054AE13
sub_54ADAE	.text	0054ADAE
sub_54AD45	.text	0054AD45
sub_54AD22	.text	0054AD22
sub_54ACF1	.text	0054ACF1
sub_54ACA5	.text	0054ACA5
sub_54ABF9	.text	0054ABF9
sub_54AB6D	.text	0054AB6D
sub_54AB05	.text	0054AB05
sub_54A97B	.text	0054A97B
sub_54A95D	.text	0054A95D
sub_54A936	.text	0054A936
sub_54A90E	.text	0054A90E
sub_54A8CC	.text	0054A8CC
sub_54A8C0	.text	0054A8C0
sub_54A8BD	.text	0054A8BD

sub_54A833	.text	0054A833
sub_54A7CB	.text	0054A7CB
sub_54A7C0	.text	0054A7C0
sub_54A740	.text	0054A740
sub_54A6ED	.text	0054A6ED
sub_54A6CC	.text	0054A6CC
sub_54A66C	.text	0054A66C
sub_54A5EB	.text	0054A5EB
sub_54A591	.text	0054A591
sub_54A568	.text	0054A568
sub_54A3A3	.text	0054A3A3
sub_54A383	.text	0054A383
sub_54A285	.text	0054A285

Line 3664 of 3908, /sub_413320

after seeing this i was a bit overwhelmed. to make sure this was the real aurastealer and not some packer/crypter i ran it and dumped it with pd32 https://github.com/glmcdona/Process-Dump



after dumping we can see that the file went from 2.40MB to 408B which gave me hope

the binary is still obfuscated but much less. so after traveling some functions and examining the startin route i found this.

```
int start()
{
  cookiething();
  return sub_1333EA0();
}
```

```
int __usercall sub_1333EA0@<eax>(int a1@<esi>, int a2@<ebx>, int a3@<edi>)
{
  _DWORD *v4; // eax
  _DWORD *v5; // esi
  int *v6; // eax
  int *v7; // esi
  int v8; // edi
  int v9; // esi
  _DWORD *v10; // eax
  char v11; // [esp+10h] [ebp-24h]
  UINT v12; // [esp+14h] [ebp-20h]

  if ( !(unsigned __int8)sub_13342F5(1) || (LOBYTE(a2) = 0, v11 = sub_13342C3(), dword_138F9CC == 1) )
  {
    sub_13345E7(a2, a3, a1, 7u);
    goto LABEL_20;
  }
  if ( dword_138F9CC )
  {
    LOBYTE(a2) = 1;
  }
  else
  {
    dword_138F9CC = 1;
    if ( sub_133856C(&dword_1342158, &dword_1342174) )
      return 255;
    sub_1338541(&dword_134214C, &dword_1342154);
    dword_138F9CC = 2;
  }
  sub_133444C(v11);
  v4 = sub_13345DB();
  v5 = v4;
  if ( *v4 && sub_13343B5((int)v4) )
    ((void (__thiscall *)(_DWORD, _DWORD, int, _DWORD))*v5)(*v5, 0, 2, 0);
  v6 = (int *)sub_13345E1();
  v7 = v6;
  if ( *v6 && sub_13343B5((int)v6) )
    sub_133746B(*v7);
  v8 = sub_1338528();
  v9 = *(_DWORD *)sub_13385F6();
  v10 = (_DWORD *)sub_13385F0();
  a1 = sub_133348B(*v10, v9, v8);
  if ( !sub_1334701() )
  {
LABEL_20:
    sub_1337491(a1);
    sub_1337455(v12);
    __debugbreak();
  }
  if ( !(_BYTE)a2 )
    sub_1337446();
  sub_1334469(1, 0);
  return a1;
}
```

here we can see the `return a1` this means that `a1` is most likely important ;3 so lets look into that.

```
int sub_1332D17()
{
  int v1; // [esp+0h] [ebp-8h] BYREF
  int v2; // [esp+4h] [ebp-4h] BYREF

  OutputDebugStringA("RunImage");
  DO_SMTH_WITH_HUGE_BYTE((char *)&v1, &v2);
  sub_1332D5A(byte_1349030, 285851, byte_1349010);
  return sub_133103C(1, v1, v2);
}
```

`RunImage` sounds good lets dig deeper.

```
char *__cdecl DO_SMTH_WITH_HUGE_BYTE(char *a1, int *a2)
{
  char *result; // eax
  unsigned int v3; // esi
  char *v4; // edi
  char *v5; // ecx
  char v6; // al
  char *v7; // esi
  void *v8; // ebx
  int v9; // esi
  int v10; // [esp+0h] [ebp-8h] BYREF
  int v11; // [esp+4h] [ebp-4h] BYREF

  result = a1;
  if ( a1 )
  {
    if ( a2 )
    {
      v3 = 0;
      *(_DWORD *)a1 = 0;
      *a2 = 0;
      result = (char *)sub_1337140(285851);
      v4 = result;
      if ( result )
      {
        do
        {
          v5 = &v4[v3];
          v6 = byte_1349030[v3] ^ byte_1349010[v3 & 0x1F];
          ++v3;
          *v5 = v6;
        }
        while ( v3 < 0x45C9B );
        if ( (unsigned int)dword_1349000 <= 0x45C9B
          && (v7 = &v4[dword_1349000],
              v10 = 285851 - dword_1349000,
              v11 = dword_1349004,
              (v8 = (void *)sub_1337140(dword_1349004)) != 0) )
        {
          v9 = sub_133345C(v8, &v11, v7, &v10);
          sub_1337130(v4);
          if ( v9 )
          {
            return (char *)sub_1337130(v8);
          }
          else
          {
            *(_DWORD *)a1 = v8;
            result = (char *)v11;
            *a2 = v11;
          }
        }
        else
        {
          return (char *)sub_1337130(v4);
        }
      }
    }
  }
```

here we can see some xoring from some byte array going on nice. That is a good indicator ^-^

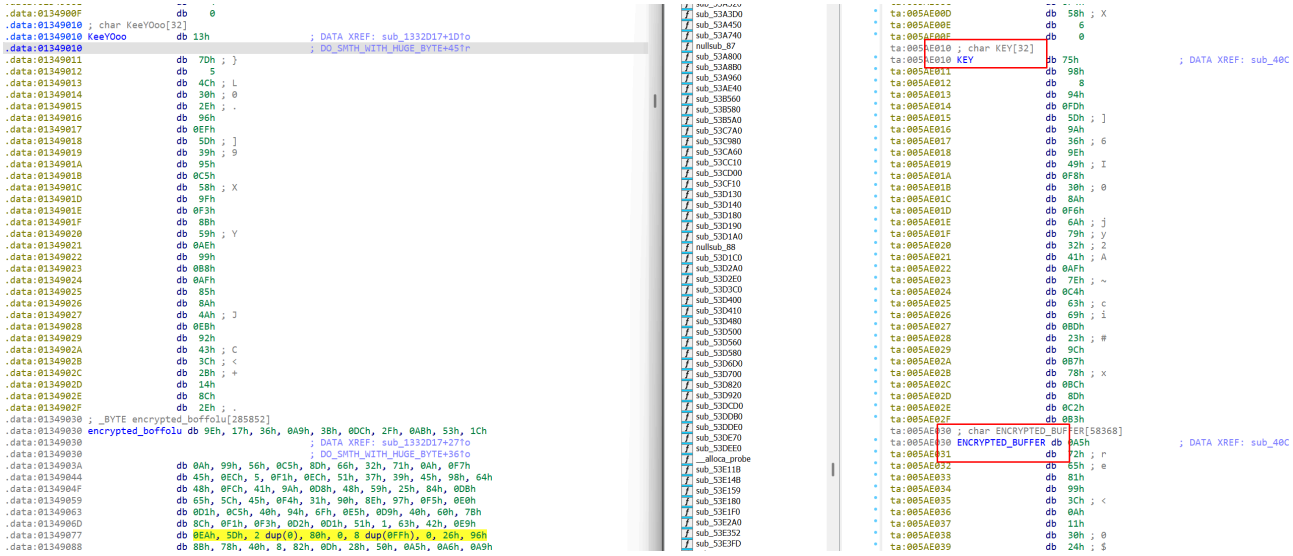
```
  while ( i < 285851 );
  if ( (unsigned int)dword_1349000 <= 285851
    && (v7 = &decryptBAFFA[dword_1349000],
        size_compressed = 285851 - dword_1349000,
        v11 = dword_1349004,
        (v8 = (void *)allocate_mem(dword_1349004)) != 0) )
  {
    v9 = decompress(v8, &v11, v7, &size_compressed);
    sub_1337130(decryptBAFFA);
    if ( v9 )
    {
      return (char *)sub_1337130(v8);
    }
    else
    {
      *(_DWORD *)a1 = v8;
      result = (char *)v11;
      *a2 = v11;
    }
```

i assume this is decompression because of the size calculation and would fit into typical malware thingis.

`size_compressed = 285851 - dword_1349000`

first i tried `decrypt_xor -> decompress` but that did not work o_O sooo lets find out why:

if we look into our first file (right picture) we can find the same unpacking mechanism like in our dumped file (left picture) just with another key and another byte array.



lets write some python script to extract that.

```
.text:0040D01F                                          ; .text:0040D23B↓j ...
.text:0040D01F                  mov     edx, [esp+8]
.text:0040D023                  mov     eax, edx
.text:0040D025                  and     eax, 1Fh
.text:0040D028                  movzx   eax, byte_5AE010[eax]
.text:0040D02F                  xor     al, byte_5AE030[edx]
.text:0040D035                  mov     ecx, [esp+4]
.text:0040D039                  mov     [ecx+edx], al
.text:0040D03C                  inc     edx
.text:0040D03D                  cmp     edx, 415988
.text:0040D043                  jz      loc_40DEE1
.text:0040D049
```

^ for the pattern

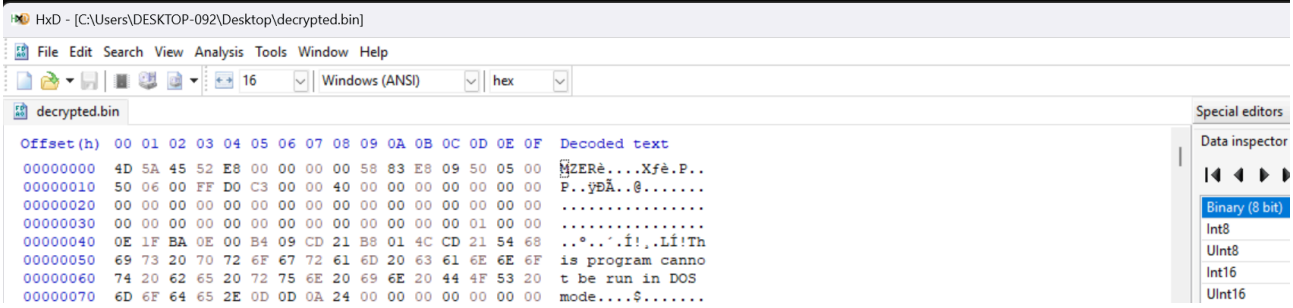
```
 import sys
import pefile
import struct
import os

PATTERNS = [
    {
        'name': 'xor_decrypt_loop',
        'asm': [
            'movzx eax, byte ptr [eax + KEY_ADDR]',
            'xor al, byte ptr [edx + BLOB_ADDR]'
        ],
        'sig': '0F B6 80 ?? ?? ?? ?? 32 82 ?? ?? ?? ??',
        'key_offset': 3,
        'blob_offset': 9,
        'size_check': {
            'asm': 'cmp edx, SIZE',
            'sig': '81 FA ?? ?? ?? ??',
            'value_offset': 2,
            'search_range': 0x100
        }
    }
]

def parse_sig(sig):
    parts = sig.split()
    pat = []
    for p in parts:
        if p == '??':
            pat.append(None)
        else:
            pat.append(int(p, 16))
    return pat
```

```
def find_pat(data, sig):
    pat = parse_sig(sig)
    for i in range(len(data) - len(pat)):
        match = True
        for j, b in enumerate(pat):
            if b is not None and data[i + j] != b:
                match = False
                break
        if match:
            return i
    return -1

def find_addrs(exe_path):
    pe = pefile.PE(exe_path)

    for sec in pe.sections:
        if b'.text' not in sec.Name:
            continue

        data = sec.get_data()

        for pat in PATTERNS:
            off = find_pat(data, pat['sig'])
            if off == -1:
                continue

            key_start = off + pat['key_offset']
            key_va = struct.unpack('<I', data[key_start:key_start + 4])[0]

            blob_start = off + pat['blob_offset']
            blob_va = struct.unpack('<I', data[blob_start:blob_start + 4])[0]

            sz_check = pat['size_check']
            search_end = off + sz_check['search_range']
            sz_off = find_pat(data[off:search_end], sz_check['sig'])
            if sz_off == -1:
                continue

            sz_start = off + sz_off + sz_check['value_offset']
            sz = struct.unpack('<I', data[sz_start:sz_start + 4])[0]

            return key_va, blob_va, sz

    return None, None, None
def decrypt_exe(exe_path):
    pe = pefile.PE(exe_path)
```

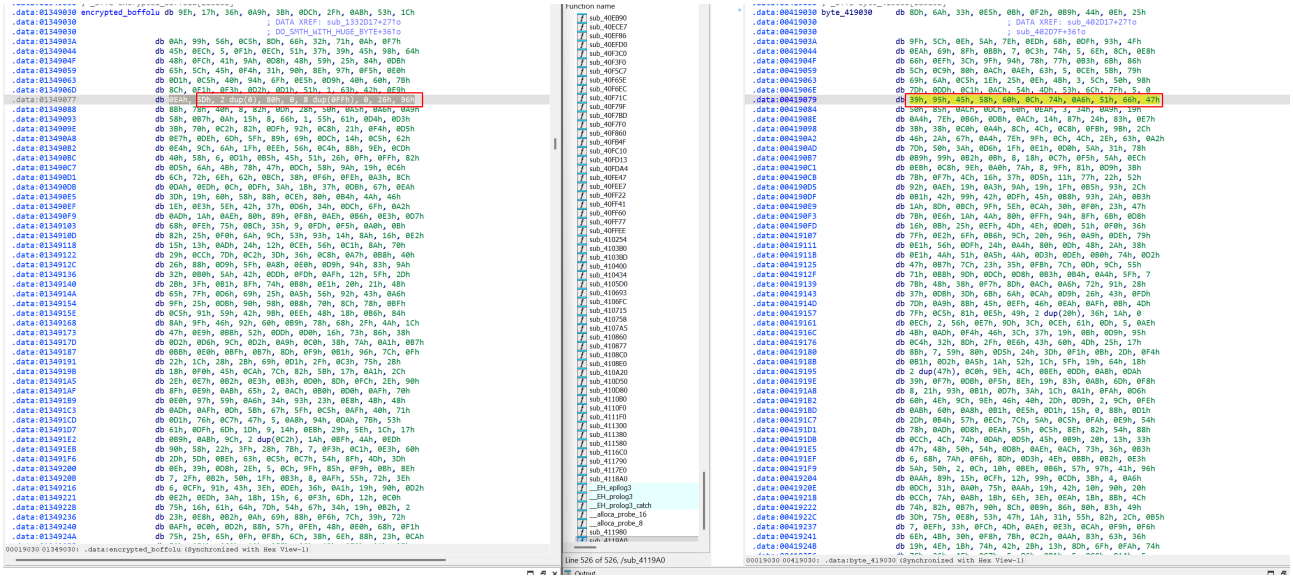
```
    key_va, blob_va, sz = find_addrs(exe_path)

    if not all([key_va, blob_va, sz]):
        print("Pattern not found")
        return False

    print(f"Key address: 0x{key_va:08X}")
    print(f"Blob address: 0x{blob_va:08X}")
    print(f"Size: 0x{sz:X} ({sz} bytes)")

    key_rva = key_va - pe.OPTIONAL_HEADER.ImageBase
    blob_rva = blob_va - pe.OPTIONAL_HEADER.ImageBase

    key = pe.get_data(key_rva, 32)
    enc = pe.get_data(blob_rva, sz)

    if not key or not enc:
        print("Failed to extract data")
        return False

    print(f"Key: {' '.join(f'{b:02X}' for b in key[:32])}")
    print(f"First 32 bytes of encrypted data: {' '.join(f'{b:02X}' for b in enc[:32])}")

    dec = bytearray(sz)
    for i in range(sz):
        dec[i] = enc[i] ^ key[i & 0x1F]

    off = 0x4F4
    if off < len(dec) and dec[off:off + 2] == b'MZ':
        dec = dec[off:]

    out = os.path.join(os.path.expanduser("~"), "Desktop", "decrypted.bin")
    with open(out, "wb") as f:
        f.write(dec)

    print(f"Wrote {len(dec)} bytes to {out}")
    return True

def main():
    if len(sys.argv) != 2:
        sys.exit(1)

    decrypt_exe(sys.argv[1])
```

```
if __name__ == "__main__":
    main()
```

```
PS C:\Users\DESKTOP-092\Desktop\DUMPAURA> python .\dump.py .\384cce56ad1a254c76c022acf2c07580f3d0097a679c249f6341dc98e8e46b74.exe
Key address: 0x005AE010
Blob address: 0x005AE030
Size: 0x658F4 (415988 bytes)
Key: 75 98 08 94 FD 5D 9A 36 9E 49 F8 30 8A F6 6A 79 32 41 AF 7E C4 63 69 BD 23 9C B7 78 BC 8D C2 B3
First 32 bytes of encrypted data: A5 72 65 81 99 3C 0A 11 30 24 40 88 FA 12 28 D9 98 62 10 FE B8 A5 49 B0 05 A4 86 D1 7F 18 6C 85
Wrote 414720 bytes to C:\Users\DESKTOP-092\Desktop\decrypted.bin
PS C:\Users\DESKTOP-092\Desktop\DUMPAURA>
```



now we can compare the clean decrypted.bin with our initial dumped file

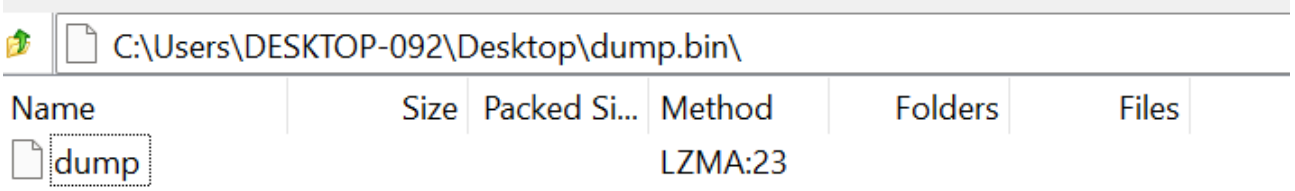


yeah this confirms the problem we had before. the dumped file already is decrypted so we just need to use lzma on that. the problem we had before was that we basically encrypted it again kek.

*i wanted to write a full unpacker but for some reason the lzma was corrupted and 36 bytes were different - after 12h i could not figure out why. if you do pls tell me, this shit keeps me up at night ty ;3 that means we can just dump the left buffer and extract at the starting lzma header offset and then decompress it.

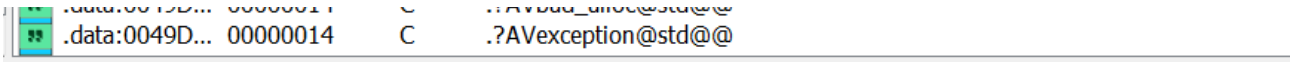
```
 import ida_bytes
import os

addr = 0x1349078
size = 0x45C9B - 0x48
```

```
data = ida_bytes.get_bytes(addr, size)

if data:
    output_path = os.path.join(os.path.expanduser("~"), "Desktop", "dump.bin")
    with open(output_path, "wb") as f:
        f.write(data)
    print("Wrote %d bytes to %s" % (len(data), output_path))
else:
    print("Failed to read data at 0x%X" % addr)
```



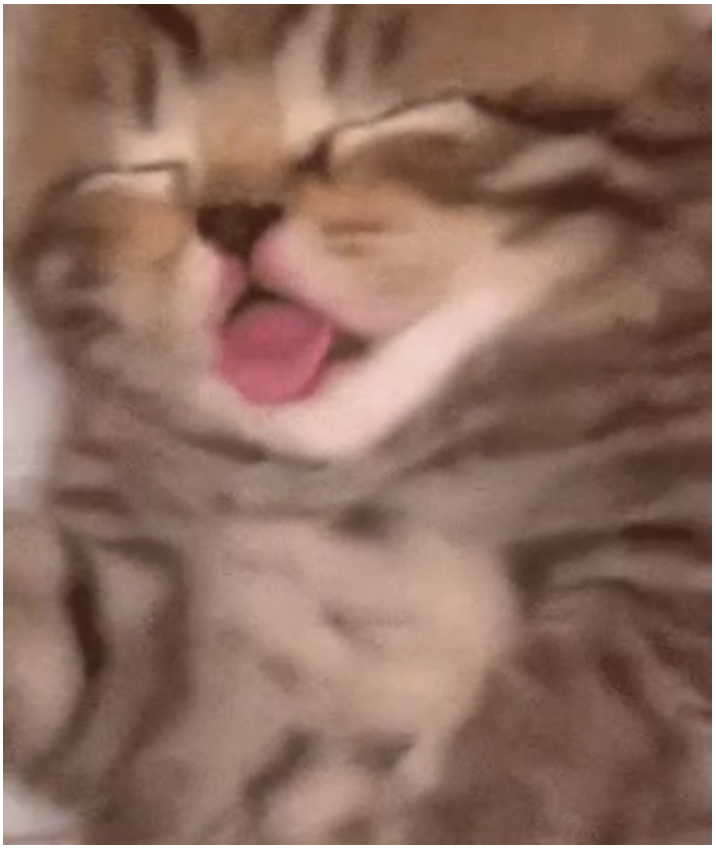
soooOoo lets open dump in ida and look at our new stage. first i wanted to make sure that this is the real stealer and looked for some indicators.

Address	Length	Type	String
.rdata:0048...	0000000A	C	RtlUnwind
.rdata:0048...	0000000D	C	SetEndOfFile
.rdata:0048...	00000018	C	SetEnvironmentVariableW
.rdata:0048...	0000001B	C	SetFileInformationByHandle
.rdata:0048...	00000011	C	SetFilePointerEx
.rdata:0048...	0000000D	C	SetLastError
.rdata:0048...	0000000D	C	SetStdHandle
.rdata:0048...	0000001C	C	SetUnhandledExceptionFilter
.rdata:0048...	00000006	C	Sleep
.rdata:0048...	00000011	C	TerminateProcess
.rdata:0048...	00000009	C	TlsAlloc
.rdata:0048...	00000008	C	TlsFree
.rdata:0048...	0000000C	C	TlsGetValue
.rdata:0048...	0000000C	C	TlsSetValue
.rdata:0048...	00000019	C	UnhandledExceptionFilter
.rdata:0048...	0000000D	C	VirtualAlloc
.rdata:0048...	00000014	C	WideCharToMultiByte
.rdata:0048...	0000000E	C	WriteConsoleW
.rdata:0048...	0000000A	C	WriteFile
.rdata:0048...	0000001C	C	GdipCreateBitmapFromHBITMAP
.rdata:0048...	00000011	C	GdipDisposeImage
.rdata:0048...	00000016	C	GdipSaveImageToStream
.rdata:0048...	0000000C	C	RegCloseKey
.rdata:0048...	0000000E	C	RegOpenKeyExW
.rdata:0048...	00000011	C	RegQueryValueExW
.rdata:0048...	0000000D	C	KERNEL32.dll
.rdata:0048...	0000000C	C	gdiplus.dll
.rdata:0048...	0000000D	C	ADVAPI32.dll
.data:0048E...	0000001B	C	
.data:0048E...	0000001A	C	bcdefghijklmnopqrstuvwxyz
.data:0048E...	00000005	C	ABCDE
.data:0048E...	00000016	C	FGHIJKLMNOPQRSTUVWXYZ
.data:0048E...	0000000F	C	
.data:0048E...	0000000C	C	
.data:0048E...	0000001B	C	abcdefghijklmnopqrstuvwxyz
.data:0048E...	0000000C	C	ABCDEFGHIJKL
.data:0048E...	0000000F	C	MNOPQRSTUVWXYZ
.data:0048E...	0000000C	C	WAVAUATVWUSH
.data:0048F...	00000006	C	@qL4O
.data:0048F...	0000000C	C	[]_^A\\A]A^A_
.data:0048F...	0000000E	C	AWAVATVWUSHcA<
.data:0048F...	0000000A	C	[]_^A\\A^A_
.data:0048F...	00000005	C	1w4OH
.data:0048F...	00000005	C	wyR+H
.data:0048F...	00000008	C	1kWickKH
.data:0048F...	0000000C	C	APPBKEYREAD
.data:0048F...	0000000D	C	APPBKEYWRITE
.data:0048F...	0000000C	C	BROWSERTYPE
.data:0048F...	00000043	C	C:\\Program Files\\BraveSoftware\\Brave-Browser\\Application\\brave.exe
.data:00490...	00000008	C	C:\\Progr
.data:00490...	0000002E	C	am Files\\Google\\Chrome\\Application\\chrome.exe
.data:00490...	0000003D	C	C:\\Program Files (x86)\\Microsoft\\Edge\\Application\\msedge.exe
.data:00490...	00000008	C	0\vP\np\t`\b
.data:00490...	00000008	C	\n0\tP\bp\a`
.data:00495...	00000005	C	XTOw9
.data:00495...	00000005	C	A[~6\x1B
.data:00495...	00000005	C	OU#U\\
.data:0049B...	00000016	C	.?AVlogic_error@std@@
.data:0049D...	0000001F	C	.?AVbad_array_new_length@std@@
.data:0049D...	00000014	C	.?AVbad_alloc@std@@



looking at the strings we can see some chromium browser paths which should be enough. I wonder if APPBYKEY has anything to do with Chrome's Appbound encryption & decryption O_o

the next blogs about aura will cover the obfuscation and config decrypter.



SHA-256: `bac52ffc8072893ff26cdbf1df1ecbcbb1762ded80249d3c9d420f62ed0dc202`

thanks for reading!

greetz to ma homies

```
 eversince
deluks
and c0ner0ne
```

you are cool and badass >:3

Source: https://blog.xyris.mov/posts/aura-stealer-%231-36bytesmademelosemymind/