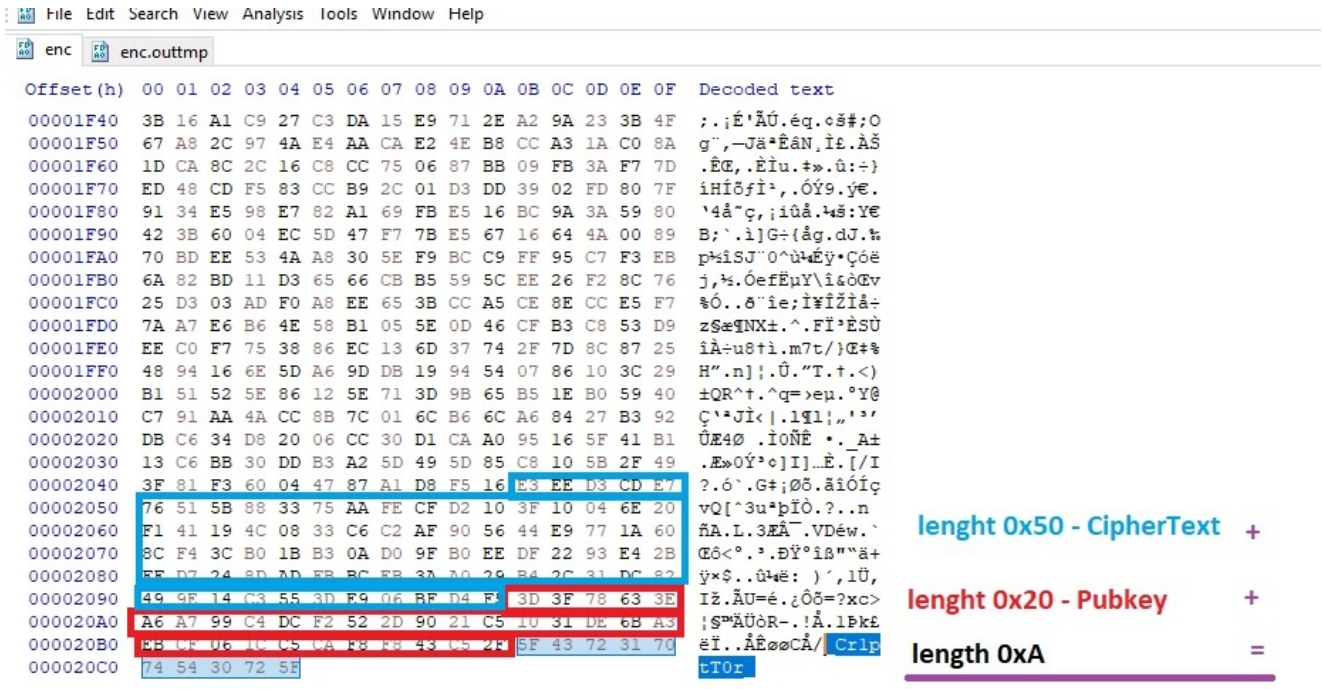
DE-Cr1pt0r tool - The Cr1pt0r ransomware decompiled decryption routine

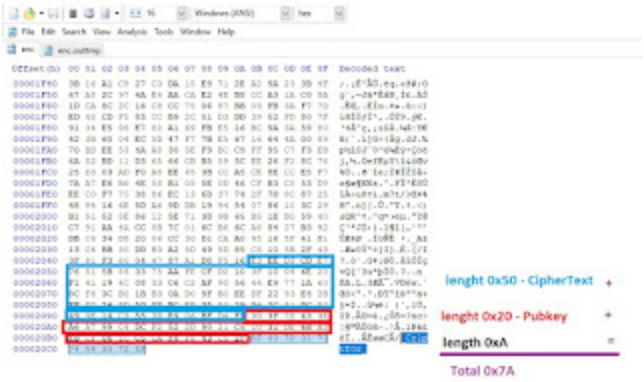
resolverblog.blogspot.com/2019/03/de-cr1pt0r-tool-cr1pt0r-ransomware.html



Hello Everybody,
after so many articles(1 - 2 - 3) about my research on this Cr1ptor ransomware finally there is a tiny way to decrypt your files.

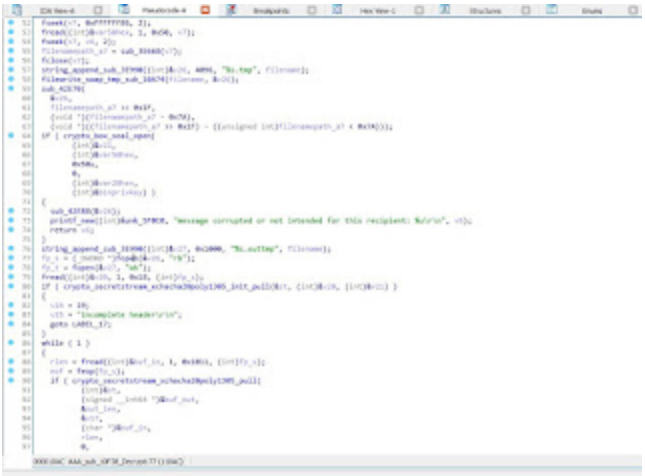
SPOILER ALERT:
This is a very early alpha release, is destined to programmers not directly to the victims.
Calm down, this will not be quick and/or easy at all but there is only a theorical chance.
Probably you'll need few months, years, your son's life of computational work to brute the key. **This is not a solution**.

Let's start from the beginning:
as I wrote in the last article I got chance to have a pair of valid keys to run some tests on my Raspberry PI VM.
Before to talk about the source code, I need you to focus on the encrypted files's structure:



Basically this ransomware append after encryption 0x7A bytes. This is important because of this:
This is how the Decryption routine looks initially (where I made some gusses..):

Studying the code from here is completely INSANE.....well, I did it anyway (tens hours of hard work) helped by the libsodium documentation I figured out the exact pseudo code of the decompilation routine built in the ransomware and what do it exactly do.
After a few IDA corrections here is where I landed:



Most of functions now looks familiar, especially those concerning libsodium and files manipulation.
Starting from here, I've ported the code into a C application to reproduce the decryption.Once figured out which kind of encryption the ransomware adopted, I've started to write a C program and from the libsodium documentation there was something interesting:

```
static int
decrypt(const char *target_file, const char *source_file,
        const unsigned char key[crypto_secretstream_xchacha20poly1305_KEYBYTES])
{
    unsigned char  buf_in[CHUNK_SIZE + crypto_secretstream_xchacha20poly1305_ABYTES];
    unsigned char  buf_out[CHUNK_SIZE];
    unsigned char  header[crypto_secretstream_xchacha20poly1305_HEADERBYTES];
    crypto_secretstream_xchacha20poly1305_state st;
    FILE          *fp_t, *fp_s;
    unsigned long long out_len;
    size_t         rlen;
    int            eof;
    int            ret = -1;
    unsigned char  tag;

    fp_s = fopen(source_file, "rb");
    fp_t = fopen(target_file, "wb");
    fread(header, 1, sizeof header, fp_s);
    if (crypto_secretstream_xchacha20poly1305_init_pull(&st, header, key) != 0) {
        goto ret; /* incomplete header */
    }
    do {
        rlen = fread(buf_in, 1, sizeof buf_in, fp_s);
        eof = feof(fp_s);
        if (crypto_secretstream_xchacha20poly1305_pull(&st, buf_out, &out_len, &tag,
                                                       buf_in, rlen, NULL, 0) != 0) {
            goto ret; /* corrupted chunk */
        }
        if (tag == crypto_secretstream_xchacha20poly1305_TAG_FINAL && ! eof) {
            goto ret; /* premature end (end of file reached before the end of the stream) */
        }
        fwrite(buf_out, 1, (size_t) out_len, fp_t);
    } while (! eof);

    ret = 0;
ret:
    fclose(fp_t);
    fclose(fp_s);
    return ret;
}
```

Well, this looks similar to our ransomware implementation, except for the fact that he's doing some manipulation on the top of the pseudo code, in fact this code example is not sufficient.
A sealed box implementation seems to anticipate the code we seen:

Sealed boxes

Example

```
#define MESSAGE (const unsigned char *) "Message"
#define MESSAGE_LEN 7
#define CIPHERTEXT_LEN (crypto_box_SEALBYTES + MESSAGE_LEN)

/* Recipient creates a long-term key pair */
unsigned char recipient_pk[crypto_box_PUBLICKEYBYTES];
unsigned char recipient_sk[crypto_box_SECRETKEYBYTES];
crypto_box_keypair(recipient_pk, recipient_sk);

/* Anonymous sender encrypts a message using an ephemeral key pair
 * and the recipient's public key */
unsigned char ciphertext[CIPHERTEXT_LEN];
crypto_box_seal(ciphertext, MESSAGE, MESSAGE_LEN, recipient_pk);

/* Recipient decrypts the ciphertext */
unsigned char decrypted[MESSAGE_LEN];
if (crypto_box_seal_open(decrypted, ciphertext, CIPHERTEXT_LEN,
                         recipient_pk, recipient_sk) != 0) {
    /* message corrupted or not intended for this recipient */
}
```

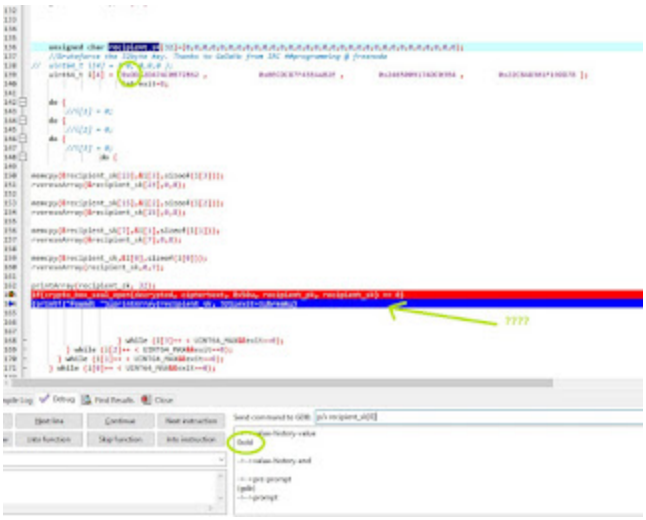
Good. We now have so many pieces of the puzzle. Its time to put them togheter.
What do we need to decrypt the files?

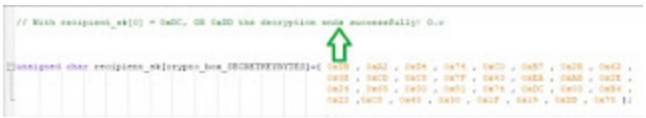
Take a closer look at the "crypto_box_seal_open" function.
Do you remember the encrypted structure?
CIPHERTEXT_LEN, from the bottom of file is 0x50. We have it.
recipient_pk, from the bottom of the file and is 0x20. We have it.
recipient_sk, from the end of.....No. Unfortunaly we haven't the secret key.
The result decrypted array is then used to decrypt the rest of the file more or less as decribed on the libsodium documentation secret-key_cryptography -> "Stream encryption/file encryption" on github.
To procede with

```
crypto_secretstream_xchacha20poly1305_init_pull(&st, header, key) != 0)
```

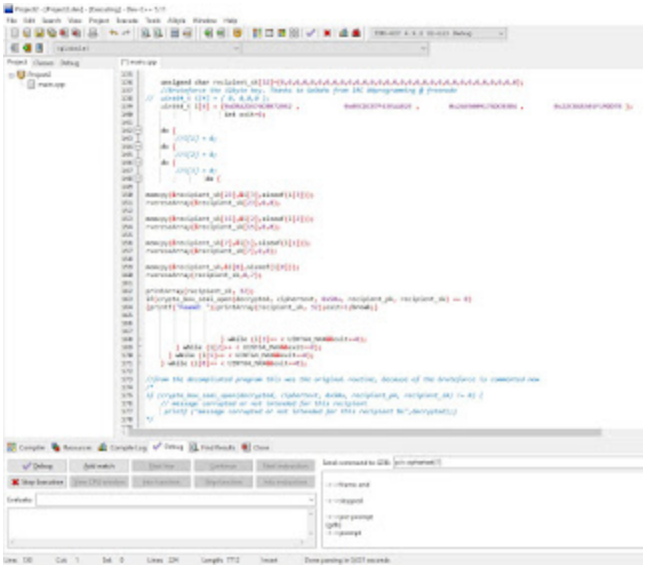
we need the header and the key. The header is actually stored into the encrypted file as the same as the example shows. So we have it.
the key....the key is the "decrypted" crypto_box_seal_open resoult! We have it.

Since the fact I had a working keypair, I had everything I need to run some tests with the good old DEV-C++ IDE.
Once set up the code, I found a very strange behaviour of libsodium which brings me to a correct decryption with 3 different private keys!(!?!?!?!?!?) O.o

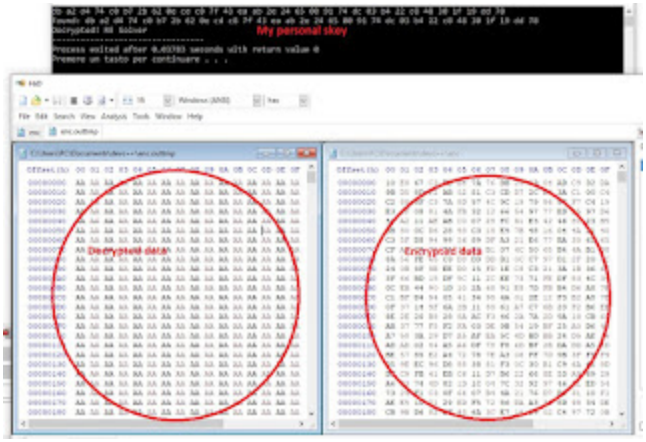




Is due to a libsodium bug?! IDK!
I hope some of you knows (and tells me) the reason of such behaviour, by the way victims does not have the private key and this strange behaviour of libsodium motivated me to implement a brute force routine into the code (to MAYBE find a working decryption sk with humanly acceptable timing).



And here is the result (skipping the brute force routine)!



(and yes, the original file was filled by a junky 0xAA 😊)Here is the main source (I do not share the encryption routine to avoid a Cr1pt0r x86 porting)

```
#include
#include
#include
#include
#include
#include
#include
#include
#define UINT64_MAX (18446744073709551615ULL)
#define CHUNK_SIZE 4096

#define crypto_stream_chacha20_ietf_KEYBYTES 32U

void rvereseArray(unsigned char *arr, int start, int end)
{
    while (start < end)
    {
        unsigned char temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
void printArray(unsigned char arr[], int size)
{
  int i;
  for (i=0; i < size; i++)
    printf("%02x ", arr[i]);

  printf("\n");
}

static int
decrypt(const char *target_file, const char *source_file, const unsigned char
key[crypto_secretstream_xchacha20poly1305_KEYBYTES])
{
    unsigned char  buf_in[CHUNK_SIZE + crypto_secretstream_xchacha20poly1305_ABYTES];
    unsigned char  buf_out[CHUNK_SIZE];
    unsigned char  header[crypto_secretstream_xchacha20poly1305_HEADERBYTES];
    crypto_secretstream_xchacha20poly1305_state st;

    FILE          *fp_t, *fp_s, *fp_s1;
    unsigned long long out_len;
    size_t         rlen;
    int            eof;
    int            ret = -1;
    unsigned char  tag = 0x0;


#define MESSAGE (const unsigned char *) "Message"

#define MESSAGE_LEN 15
#define CIPHERTEXT_LEN (crypto_box_SEALBYTES + MESSAGE_LEN)
```

```
unsigned char recipient_pk[crypto_box_PUBLICKEYBYTES];
//My pk
//unsigned char recipient_pk[crypto_box_PUBLICKEYBYTES];={0x3D , 0x3F , 0x78 , 0x63 ,
0x3E , 0xA6 , 0xA7 , 0x99 , 0xC4 , 0xDC , 0xF2 , 0x52 , 0x2D , 0x90 , 0x21 , 0xC5 ,
0x10 , 0x31 , 0xDE , 0x6B , 0xA3 , 0xEB ,
              //0xCF , 0x06 , 0x1C , 0xC5 , 0xCA , 0xF8 , 0xF8 , 0x43 , 0xC5 ,
0x2F};//; /* Bob's public key */
//
//recipient_sk decrypt the files also with recipient_sk[0]=0xDB (the original byte)
than 0xDD and also 0xDC
// unsigned char recipient_sk[crypto_box_SECRETKEYBYTES]={ 0xDB , 0xA2 , 0xD4 , 0x74
, 0xC0 , 0xB7 , 0x2B , 0x62 , 0x0E , 0xCD , 0xC8 , 0x7F , 0x43 , 0xEA , 0xAB , 0x2E ,
0x24 ,
//0x65 , 0x00 , 0x91 , 0x74 , 0xDC , 0x03 , 0xB4 , 0x22 , 0xC8 , 0x48 , 0x30 , 0x1F ,
0x19 , 0xDD , 0x78 }; //; /* Bob's secret key */
unsigned char recipient_sk[32]=
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

unsigned char ciphertext[80];
//var50h

long filelen;
fp_s1 = fopen("enc", "rb");
   if(fp_s1==0){ printf("Encrypted files not found"); return 1;}
    fseek(fp_s1, 0xFFFFFFD6, SEEK_END);
    //filelen = ftell(fp_s1);
  fread(recipient_pk, 1, 0x20, fp_s1);
  fseek(fp_s1, 0xFFFFFF86, SEEK_END);

  fread(ciphertext, 1, 0x50, fp_s1);

//because the new key is 32dec bytes long
unsigned char decrypted[32];



//brute start
//Bruteforce the 32byte key. Thanks to GeDaMo from IRC ##programming @ freenode
//
 uint64_t i[4] = { 0, 0,0,0 };
// uint64_t i[4] = {0xDBA2D474C0B72B62 ,    0x0ECDC87F43EAAB2E ,
0x2465009174DC03B4 ,    0x22C848301F19DD78 };
      int exit=0;
 printf("DE-Cr1pt0r Tool By RE-Solver @solver_re:\r\n Bruteforcing \r\n ");
  do {
   i[1] = 0;
  do {
   i[2] = 0;
 do {
  i[3] = 0;
      do {
```

```
memcpy(&recipient_sk[23],&i[3],sizeof(i[3]));
rvereseArray(&recipient_sk[23],0,8);

memcpy(&recipient_sk[15],&i[2],sizeof(i[2]));
rvereseArray(&recipient_sk[15],0,8);

memcpy(&recipient_sk[7],&i[1],sizeof(i[1]));
rvereseArray(&recipient_sk[7],0,8);

memcpy(&recipient_sk,&i[0],sizeof(i[0]));
rvereseArray(recipient_sk,0,7);

//printArray(recipient_sk, 32);
if(crypto_box_seal_open(decrypted, ciphertext, 0x50u, recipient_pk, recipient_sk) ==
0)
{printf("Found: ");printArray(recipient_sk, 32);exit=1;break;}



           } while (i[3]++ < UINT64_MAX&&exit==0);
        } while (i[2]++ < UINT64_MAX&&exit==0);
   } while (i[1]++ < UINT64_MAX&&exit==0);
 } while (i[0]++ < UINT64_MAX&&exit==0);
 //END brute
//from the decompilated program this was the original routine, because of the
bruteforce is added as a comment now
/*
if (crypto_box_seal_open(decrypted, ciphertext, 0x50u, recipient_pk, recipient_sk) !=
0) {
    // message corrupted or not intended for this recipient
     printf ("message corrupted or not intended for this recipient %s",decrypted);}
*/

    fp_s = fopen(source_file, "rb");
    fp_t = fopen(target_file, "wb");
    if(fp_s==0 || fp_t==0){printf("Encrypted files not found");
 goto ret;}
    fread(header, 1, 0x18, fp_s);
    if (crypto_secretstream_xchacha20poly1305_init_pull(&st, header, decrypted) != 0)
{
        goto ret; /* incomplete header */
    }
    do {
        rlen = fread(buf_in, 1, 0x1011, fp_s);
        eof = feof(fp_s);
        //tag = 0x0;
        int value=crypto_secretstream_xchacha20poly1305_pull(&st, buf_out, &out_len,
&tag, buf_in, rlen, NULL,0);
        if (value != 0) {
            goto ret; /* corrupted chunk */
        }
        if (tag == 3 && ! eof) { //crypto_secretstream_xchacha20poly1305_TAG_FINAL ->
3
            goto ret; /* premature end (end of file reached before the end of the
stream) */
```

```
        }
        fwrite(buf_out, 1, (size_t) out_len, fp_t);
    } while (! eof);

    ret = 0;
ret:
    fclose(fp_t);
    fclose(fp_s);
    fclose(fp_s1);
    return ret;
}

int
main(void)
{
    unsigned char key[crypto_secretstream_xchacha20poly1305_KEYBYTES];

    if (sodium_init() != 0) {
        return 1;
    }
    crypto_secretstream_xchacha20poly1305_keygen(key);

    if (decrypt("enc.outtmp", "enc.tmp", 0x0) != 0) {
     printf("Something goes wrong.");
        return 1;
    }
    printf("Decrypted! RE Solver");
    return 0;
}
```

IDE: DEV-C++
libsodium library: libsodium-1.0.17-mingw.tar.gz
Remember to link the library into the Project/Project Options

Compiled tool: https://www.sendspace.com/file/275c70
sha256: 4066fa0d402a8458f7784e89ba979929ee1d7efd761b3cabe9705784aa8af865

usage: Copy an encrypted file into the same folder of the tool and rename it as enc (with no extensions). Copy the same encrypted file and rename it as enc.tmp and strip the last 0x7A from the end of the file. If you're lucky within some weeks you'll have the key printed on the console and the encrypted.outtmp decrypted file created on the same folder dir.

Next step: create a file named privkey and write the hex key (with no spaces) into a text file and put it in the Cr1pt0r folder. From the same folder, rename the file pubkey as pubkey_backup and turn on your D-Link nas again.

Note: My GF is waiting me since days, she has been so patient. A special Thanks to her. 😊
I'm sorry but I do not support the tool usage or others kind of requests.Since the fact that code is released under GPL, everyone can compile, improve, modify the code. (And I hope it

happens).

Follow me on Twitter @solver_re
Hire me! Job offers are welcome.
Cheers,
RE Solver