

DE-Cr1pt0r tool - The Cr1pt0r ransomware decompiled decryption routine

Archived: 2026-04-05 14:26:11 UTC

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.](#)

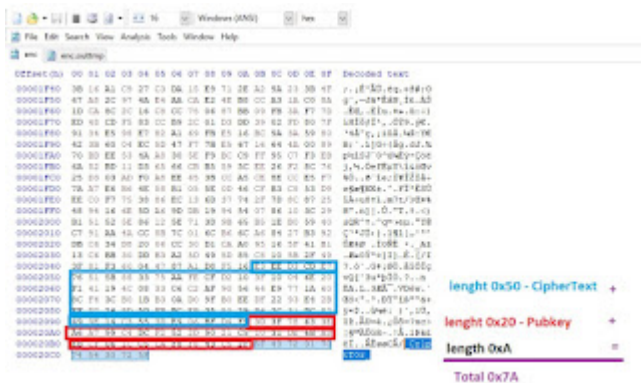
Calm down, this will not be quick and/or easy at all but there is only a theoretical chance.

[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.):

```

41 v1 = fopen(v1, "w");
42 if (v1 < 0)
43 {
44     printf_func(100|k00k_SPOCK, "never opening file: %s\n", v1);
45     return v1;
46 }
47
48 sub_41440([signed int]v1, v1, 0x00);
49 sub_41440(v1, -42, 2);
50 sub_41440([int]v1, v1, 0);
51 sub_41440(v1, -125, 2);
52 sub_41440(v1, -125, 2);
53 sub_41440([int]v1, v1, 0);
54 sub_41440(v1, -42, 2);
55 v1 = sub_41440(v1);
56 sub_41440(v1);
57 sub_41440([int]v1, 0x00, "lib.smp", v1);
58 FILE* file_smp; sub_41440(v1, &v1);
59 sub_41440(v1, v1, 0);
60 if (sub_41440(v1, &v1, 0x00, 0x00, 0x00, 0x00))
61 {
62     sub_41440(v1);
63     printf_func(100|k00k_SPOCK, "message corrupted or not intended for this recipient: %s\n", v1);
64     return v1;
65 }
66
67 sub_41440([int]v1, 0x00, "lib.smp", v1);
68 v1 = (int)v1;
69 v1 = fopen(v1, "w");
70 sub_41440([int]v1, v1, 24, (int)v1);
71 if (sub_41440(v1, &v1, 0x00, 0x00))
72 {
73     v1 = 20;
74     v1 = "Incomplete header!\n";
75     goto LABEL_17;
76 }
77 while (v1 < 1)
78 {
79     v1 = sub_41440([int]v1, v1, 0x00, (int)v1);
80     v1 = sub_41440(v1);
81     if (sub_41440(v1, (signed int)v1, 0x00, 0x00, (char *)v1, v1, 0, 0x00))
82     {
83         v1 = "corrupted chunk!\n";
84         v1 = 20;
85         LABEL_17:
86         printf_func(100|v1, v1, (int)k00k_SPOCK);
87         sub_41440(v1);
88         sub_41440(v1);
89         sub_41440(v1 == 0, "...decrypt...", 100, "lib_decrypt");
90     }
91     if (v1 == 1)
92     {
93         break;
94     }
95     printf_func(100|k00k_SPOCK, v1, v1, v1);
96     if (v1 < 1)
97     {
98         goto LABEL_18;
99     }
100

```

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:

```

512 Fopen(v1, "w");
513 Fwrite(v1, 1, 0x00, v1);
514 Fclose(v1);
515 FILE* file_smp;
516 string_append_sub_41440([int]v1, 0x00, "lib.smp", file_smp);
517 FILE* file_smp; sub_41440(file_smp, &v1);
518 sub_41440(v1);
519
520 FILE* file_smp;
521 FILE* file_smp;
522 FILE* file_smp;
523 FILE* file_smp;
524 FILE* file_smp;
525 FILE* file_smp;
526 FILE* file_smp;
527 FILE* file_smp;
528 FILE* file_smp;
529 FILE* file_smp;
530 FILE* file_smp;
531 FILE* file_smp;
532 FILE* file_smp;
533 FILE* file_smp;
534 FILE* file_smp;
535 FILE* file_smp;
536 FILE* file_smp;
537 FILE* file_smp;
538 FILE* file_smp;
539 FILE* file_smp;
540 FILE* file_smp;
541 FILE* file_smp;
542 FILE* file_smp;
543 FILE* file_smp;
544 FILE* file_smp;
545 FILE* file_smp;
546 FILE* file_smp;
547 FILE* file_smp;
548 FILE* file_smp;
549 FILE* file_smp;
550 FILE* file_smp;
551 FILE* file_smp;
552 FILE* file_smp;
553 FILE* file_smp;
554 FILE* file_smp;
555 FILE* file_smp;
556 FILE* file_smp;
557 FILE* file_smp;
558 FILE* file_smp;
559 FILE* file_smp;
560 FILE* file_smp;
561 FILE* file_smp;
562 FILE* file_smp;
563 FILE* file_smp;
564 FILE* file_smp;
565 FILE* file_smp;
566 FILE* file_smp;
567 FILE* file_smp;
568 FILE* file_smp;
569 FILE* file_smp;
570 FILE* file_smp;
571 FILE* file_smp;
572 FILE* file_smp;
573 FILE* file_smp;
574 FILE* file_smp;
575 FILE* file_smp;
576 FILE* file_smp;
577 FILE* file_smp;
578 FILE* file_smp;
579 FILE* file_smp;
580 FILE* file_smp;
581 FILE* file_smp;
582 FILE* file_smp;
583 FILE* file_smp;
584 FILE* file_smp;
585 FILE* file_smp;
586 FILE* file_smp;
587 FILE* file_smp;
588 FILE* file_smp;
589 FILE* file_smp;
590 FILE* file_smp;
591 FILE* file_smp;
592 FILE* file_smp;
593 FILE* file_smp;
594 FILE* file_smp;
595 FILE* file_smp;
596 FILE* file_smp;
597 FILE* file_smp;
598 FILE* file_smp;
599 FILE* file_smp;
600 FILE* file_smp;
601 FILE* file_smp;
602 FILE* file_smp;
603 FILE* file_smp;
604 FILE* file_smp;
605 FILE* file_smp;
606 FILE* file_smp;
607 FILE* file_smp;
608 FILE* file_smp;
609 FILE* file_smp;
610 FILE* file_smp;
611 FILE* file_smp;
612 FILE* file_smp;
613 FILE* file_smp;
614 FILE* file_smp;
615 FILE* file_smp;
616 FILE* file_smp;
617 FILE* file_smp;
618 FILE* file_smp;
619 FILE* file_smp;
620 FILE* file_smp;
621 FILE* file_smp;
622 FILE* file_smp;
623 FILE* file_smp;
624 FILE* file_smp;
625 FILE* file_smp;
626 FILE* file_smp;
627 FILE* file_smp;
628 FILE* file_smp;
629 FILE* file_smp;
630 FILE* file_smp;
631 FILE* file_smp;
632 FILE* file_smp;
633 FILE* file_smp;
634 FILE* file_smp;
635 FILE* file_smp;
636 FILE* file_smp;
637 FILE* file_smp;
638 FILE* file_smp;
639 FILE* file_smp;
640 FILE* file_smp;
641 FILE* file_smp;
642 FILE* file_smp;
643 FILE* file_smp;
644 FILE* file_smp;
645 FILE* file_smp;
646 FILE* file_smp;
647 FILE* file_smp;
648 FILE* file_smp;
649 FILE* file_smp;
650 FILE* file_smp;
651 FILE* file_smp;
652 FILE* file_smp;
653 FILE* file_smp;
654 FILE* file_smp;
655 FILE* file_smp;
656 FILE* file_smp;
657 FILE* file_smp;
658 FILE* file_smp;
659 FILE* file_smp;
660 FILE* file_smp;
661 FILE* file_smp;
662 FILE* file_smp;
663 FILE* file_smp;
664 FILE* file_smp;
665 FILE* file_smp;
666 FILE* file_smp;
667 FILE* file_smp;
668 FILE* file_smp;
669 FILE* file_smp;
670 FILE* file_smp;
671 FILE* file_smp;
672 FILE* file_smp;
673 FILE* file_smp;
674 FILE* file_smp;
675 FILE* file_smp;
676 FILE* file_smp;
677 FILE* file_smp;
678 FILE* file_smp;
679 FILE* file_smp;
680 FILE* file_smp;
681 FILE* file_smp;
682 FILE* file_smp;
683 FILE* file_smp;
684 FILE* file_smp;
685 FILE* file_smp;
686 FILE* file_smp;
687 FILE* file_smp;
688 FILE* file_smp;
689 FILE* file_smp;
690 FILE* file_smp;
691 FILE* file_smp;
692 FILE* file_smp;
693 FILE* file_smp;
694 FILE* file_smp;
695 FILE* file_smp;
696 FILE* file_smp;
697 FILE* file_smp;
698 FILE* file_smp;
699 FILE* file_smp;
700 FILE* file_smp;
701 FILE* file_smp;
702 FILE* file_smp;
703 FILE* file_smp;
704 FILE* file_smp;
705 FILE* file_smp;
706 FILE* file_smp;
707 FILE* file_smp;
708 FILE* file_smp;
709 FILE* file_smp;
710 FILE* file_smp;
711 FILE* file_smp;
712 FILE* file_smp;
713 FILE* file_smp;
714 FILE* file_smp;
715 FILE* file_smp;
716 FILE* file_smp;
717 FILE* file_smp;
718 FILE* file_smp;
719 FILE* file_smp;
720 FILE* file_smp;
721 FILE* file_smp;
722 FILE* file_smp;
723 FILE* file_smp;
724 FILE* file_smp;
725 FILE* file_smp;
726 FILE* file_smp;
727 FILE* file_smp;
728 FILE* file_smp;
729 FILE* file_smp;
730 FILE* file_smp;
731 FILE* file_smp;
732 FILE* file_smp;
733 FILE* file_smp;
734 FILE* file_smp;
735 FILE* file_smp;
736 FILE* file_smp;
737 FILE* file_smp;
738 FILE* file_smp;
739 FILE* file_smp;
740 FILE* file_smp;
741 FILE* file_smp;
742 FILE* file_smp;
743 FILE* file_smp;
744 FILE* file_smp;
745 FILE* file_smp;
746 FILE* file_smp;
747 FILE* file_smp;
748 FILE* file_smp;
749 FILE* file_smp;
750 FILE* file_smp;
751 FILE* file_smp;
752 FILE* file_smp;
753 FILE* file_smp;
754 FILE* file_smp;
755 FILE* file_smp;
756 FILE* file_smp;
757 FILE* file_smp;
758 FILE* file_smp;
759 FILE* file_smp;
760 FILE* file_smp;
761 FILE* file_smp;
762 FILE* file_smp;
763 FILE* file_smp;
764 FILE* file_smp;
765 FILE* file_smp;
766 FILE* file_smp;
767 FILE* file_smp;
768 FILE* file_smp;
769 FILE* file_smp;
770 FILE* file_smp;
771 FILE* file_smp;
772 FILE* file_smp;
773 FILE* file_smp;
774 FILE* file_smp;
775 FILE* file_smp;
776 FILE* file_smp;
777 FILE* file_smp;
778 FILE* file_smp;
779 FILE* file_smp;
780 FILE* file_smp;
781 FILE* file_smp;
782 FILE* file_smp;
783 FILE* file_smp;
784 FILE* file_smp;
785 FILE* file_smp;
786 FILE* file_smp;
787 FILE* file_smp;
788 FILE* file_smp;
789 FILE* file_smp;
790 FILE* file_smp;
791 FILE* file_smp;
792 FILE* file_smp;
793 FILE* file_smp;
794 FILE* file_smp;
795 FILE* file_smp;
796 FILE* file_smp;
797 FILE* file_smp;
798 FILE* file_smp;
799 FILE* file_smp;
800 FILE* file_smp;
801 FILE* file_smp;
802 FILE* file_smp;
803 FILE* file_smp;
804 FILE* file_smp;
805 FILE* file_smp;
806 FILE* file_smp;
807 FILE* file_smp;
808 FILE* file_smp;
809 FILE* file_smp;
810 FILE* file_smp;
811 FILE* file_smp;
812 FILE* file_smp;
813 FILE* file_smp;
814 FILE* file_smp;
815 FILE* file_smp;
816 FILE* file_smp;
817 FILE* file_smp;
818 FILE* file_smp;
819 FILE* file_smp;
820 FILE* file_smp;
821 FILE* file_smp;
822 FILE* file_smp;
823 FILE* file_smp;
824 FILE* file_smp;
825 FILE* file_smp;
826 FILE* file_smp;
827 FILE* file_smp;
828 FILE* file_smp;
829 FILE* file_smp;
830 FILE* file_smp;
831 FILE* file_smp;
832 FILE* file_smp;
833 FILE* file_smp;
834 FILE* file_smp;
835 FILE* file_smp;
836 FILE* file_smp;
837 FILE* file_smp;
838 FILE* file_smp;
839 FILE* file_smp;
840 FILE* file_smp;
841 FILE* file_smp;
842 FILE* file_smp;
843 FILE* file_smp;
844 FILE* file_smp;
845 FILE* file_smp;
846 FILE* file_smp;
847 FILE* file_smp;
848 FILE* file_smp;
849 FILE* file_smp;
850 FILE* file_smp;
851 FILE* file_smp;
852 FILE* file_smp;
853 FILE* file_smp;
854 FILE* file_smp;
855 FILE* file_smp;
856 FILE* file_smp;
857 FILE* file_smp;
858 FILE* file_smp;
859 FILE* file_smp;
860 FILE* file_smp;
861 FILE* file_smp;
862 FILE* file_smp;
863 FILE* file_smp;
864 FILE* file_smp;
865 FILE* file_smp;
866 FILE* file_smp;
867 FILE* file_smp;
868 FILE* file_smp;
869 FILE* file_smp;
870 FILE* file_smp;
871 FILE* file_smp;
872 FILE* file_smp;
873 FILE* file_smp;
874 FILE* file_smp;
875 FILE* file_smp;
876 FILE* file_smp;
877 FILE* file_smp;
878 FILE* file_smp;
879 FILE* file_smp;
880 FILE* file_smp;
881 FILE* file_smp;
882 FILE* file_smp;
883 FILE* file_smp;
884 FILE* file_smp;
885 FILE* file_smp;
886 FILE* file_smp;
887 FILE* file_smp;
888 FILE* file_smp;
889 FILE* file_smp;
890 FILE* file_smp;
891 FILE* file_smp;
892 FILE* file_smp;
893 FILE* file_smp;
894 FILE* file_smp;
895 FILE* file_smp;
896 FILE* file_smp;
897 FILE* file_smp;
898 FILE* file_smp;
899 FILE* file_smp;
900 FILE* file_smp;
901 FILE* file_smp;
902 FILE* file_smp;
903 FILE* file_smp;
904 FILE* file_smp;
905 FILE* file_smp;
906 FILE* file_smp;
907 FILE* file_smp;
908 FILE* file_smp;
909 FILE* file_smp;
910 FILE* file_smp;
911 FILE* file_smp;
912 FILE* file_smp;
913 FILE* file_smp;
914 FILE* file_smp;
915 FILE* file_smp;
916 FILE* file_smp;
917 FILE* file_smp;
918 FILE* file_smp;
919 FILE* file_smp;
920 FILE* file_smp;
921 FILE* file_smp;
922 FILE* file_smp;
923 FILE* file_smp;
924 FILE* file_smp;
925 FILE* file_smp;
926 FILE* file_smp;
927 FILE* file_smp;
928 FILE* file_smp;
929 FILE* file_smp;
930 FILE* file_smp;
931 FILE* file_smp;
932 FILE* file_smp;
933 FILE* file_smp;
934 FILE* file_smp;
935 FILE* file_smp;
936 FILE* file_smp;
937 FILE* file_smp;
938 FILE* file_smp;
939 FILE* file_smp;
940 FILE* file_smp;
941 FILE* file_smp;
942 FILE* file_smp;
943 FILE* file_smp;
944 FILE* file_smp;
945 FILE* file_smp;
946 FILE* file_smp;
947 FILE* file_smp;
948 FILE* file_smp;
949 FILE* file_smp;
950 FILE* file_smp;
951 FILE* file_smp;
952 FILE* file_smp;
953 FILE* file_smp;
954 FILE* file_smp;
955 FILE* file_smp;
956 FILE* file_smp;
957 FILE* file_smp;
958 FILE* file_smp;
959 FILE* file_smp;
960 FILE* file_smp;
961 FILE* file_smp;
962 FILE* file_smp;
963 FILE* file_smp;
964 FILE* file_smp;
965 FILE* file_smp;
966 FILE* file_smp;
967 FILE* file_smp;
968 FILE* file_smp;
969 FILE* file_smp;
970 FILE* file_smp;
971 FILE* file_smp;
972 FILE* file_smp;
973 FILE* file_smp;
974 FILE* file_smp;
975 FILE* file_smp;
976 FILE* file_smp;
977 FILE* file_smp;
978 FILE* file_smp;
979 FILE* file_smp;
980 FILE* file_smp;
981 FILE* file_smp;
982 FILE* file_smp;
983 FILE* file_smp;
984 FILE* file_smp;
985 FILE* file_smp;
986 FILE* file_smp;
987 FILE* file_smp;
988 FILE* file_smp;
989 FILE* file_smp;
990 FILE* file_smp;
991 FILE* file_smp;
992 FILE* file_smp;
993 FILE* file_smp;
994 FILE* file_smp;
995 FILE* file_smp;
996 FILE* file_smp;
997 FILE* file_smp;
998 FILE* file_smp;
999 FILE* file_smp;
1000 FILE* file_smp;

```

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[OHMM_SIZE + crypto_secretstream_xchacha20poly1305_ABYTES];
    unsigned char buf_out[OHMM_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 together. 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, f

rom the end of

.....

No. Unfortunately 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)
```

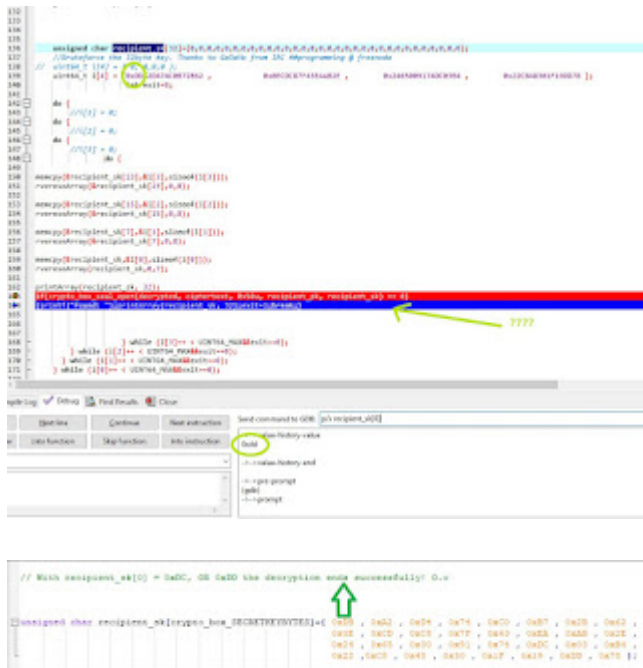
we need the header and the key. The header is actually stored into the encrypted file as the same as the example shows. So we have it.

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

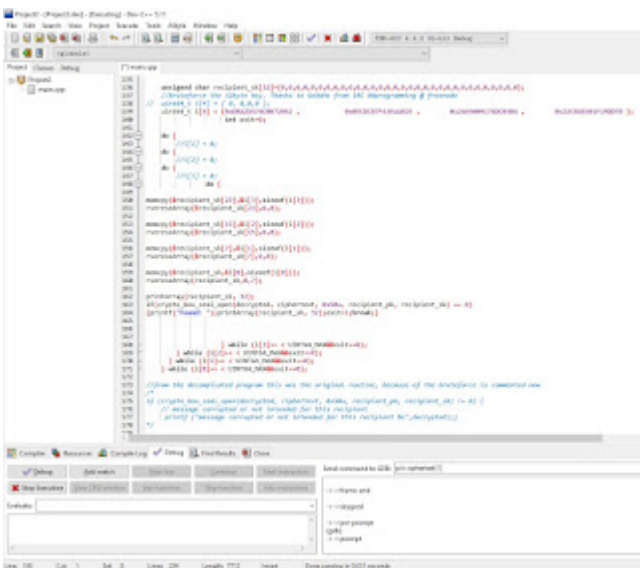


Is due to a libsodium bug?! IDK!

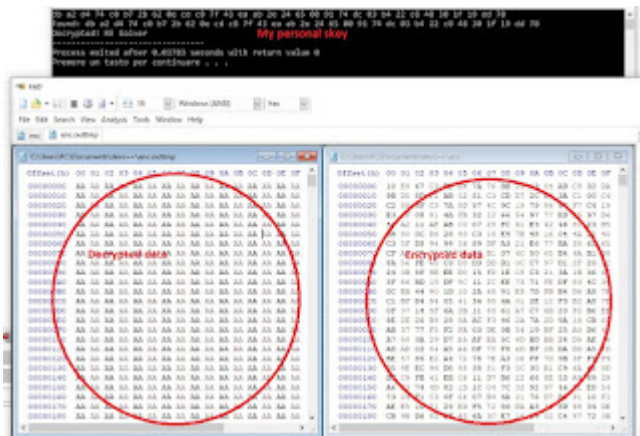
I hope some of you knows (and tells me) the reason of such behaviour, by the way victims does not have the private key and

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 rreverseArray(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
{
    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 ,
```

```
        //0xCF , 0x06 , 0x1C , 0xC5 , 0xCA , 0xF8 , 0xF8 , 0x43 , 0xC5 , 0x2F};//; /* Bob's pul
//
//recipient_sk decrypt the files also with recipient_sk[0]=0xDB (the original byte) than 0xDD and al
// unsigned char recipient_sk[CRYPTO_BOX_SECRETKEYBYTES]={ 0xDB , 0xA2 , 0xD4 , 0x74 , 0xC0 , 0xB7 ,
//0x65 , 0x00 , 0x91 , 0x74 , 0xDC , 0x03 , 0xB4 , 0x22 , 0xC8 , 0x48 , 0x30 , 0x1F , 0x19 , 0xDD ,
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, 0xFFFFFFFFD6, SEEK_END);
    //filelen = ftell(fp_s1);
    fread(recipient_pk, 1, 0x20, fp_s1);
    fseek(fp_s1, 0xFFFFFFFF86, 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 ,    0x22C848
    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]));
```

```
rreverseArray(&recipient_sk[7],0,8);

memcpy(&recipient_sk,&i[0],sizeof(i[0]));
rreverseArray(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 decompiled program this was the original routine, because of the bruteforce is added as
/*
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, r
        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

Source: <https://resolverblog.blogspot.com/2019/03/de-cr1pt0r-tool-cr1pt0r-ransomware.html>