Recovering some files encrypted by LockBit 2.0

skyblue.team/posts/hive-recovery-from-lockbit-2.0/

Oct 15, 2021 · 1066 words · 6 minute read

The LockBit 2.0 ransomware has been incredibly "productive" these last few months: their technique is well automated, and the list of compromised companies keeps growing every day.

In order to reduce the destructiveness of their payload, most ransomware operators do not encrypt every single file on a system; instead, they set out a set of rules, for example:

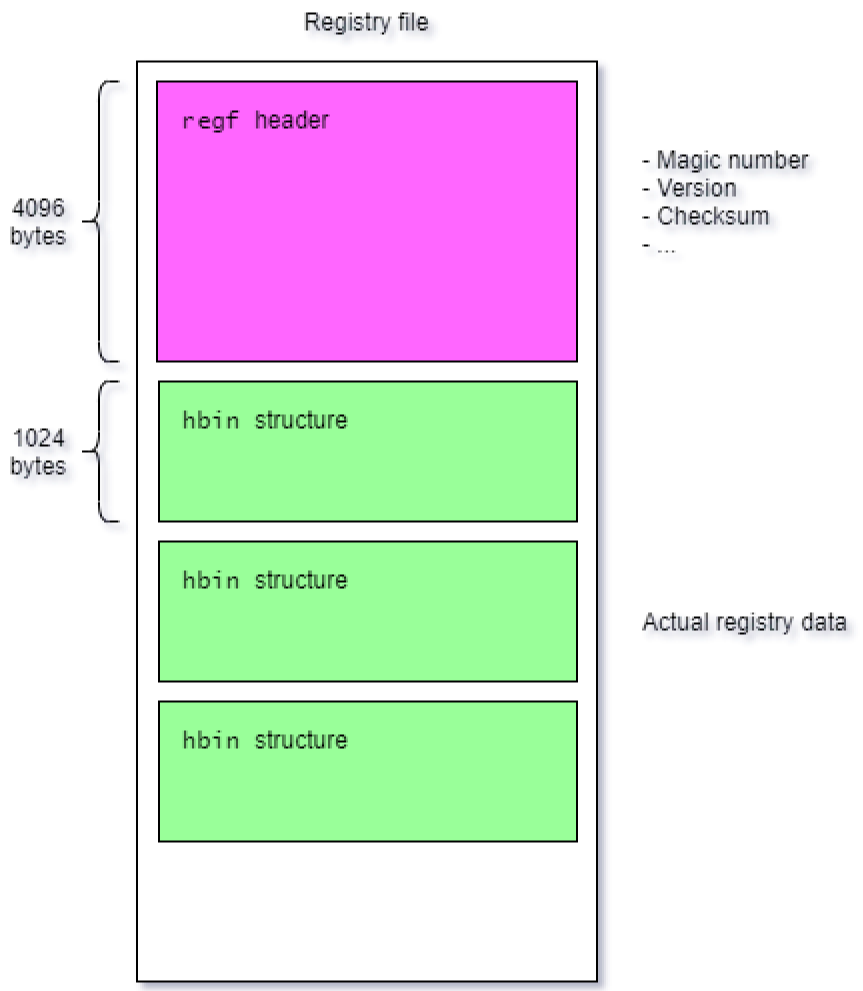
- Only encrypt files with specific extensions: `.docx` , `.cpp` , `.db` , `.log`
- Don't encrypt files in `C:\Windows\System32` , in order to keep a semi-working machine (otherwise how would the users read the ransom note?)

LockBit 2.0 has a pretty interesting quirk though: as an optimization, only the first 4KiB (4096 bytes) of each file are encrypted. This is usually enough to lock away important data and make file recovery a pain. It also speeds up the encryption process.

We have also observed that the LockBit 2.0 ransomware is pretty generous in the extension list it encrypts: even user hive files (`NTUSER.DAT`) are encrypted, which is a pain if we want to extract useful data from it. But registry hives can be pretty big, could we maybe recover some data anyway?

Registry hive structure 🔗

In order to understand how we can recover the hives, we must first have a look at how registry hives are stored on-disk. Willi Ballenthin's `python-registry` has some good explanations, including a text file from a certain B.D. which goes over the structure of hives for both Windows 95 and NT. This document tells us that NT optimized hive loading by making the header the typical size of a page, *4KiB*.



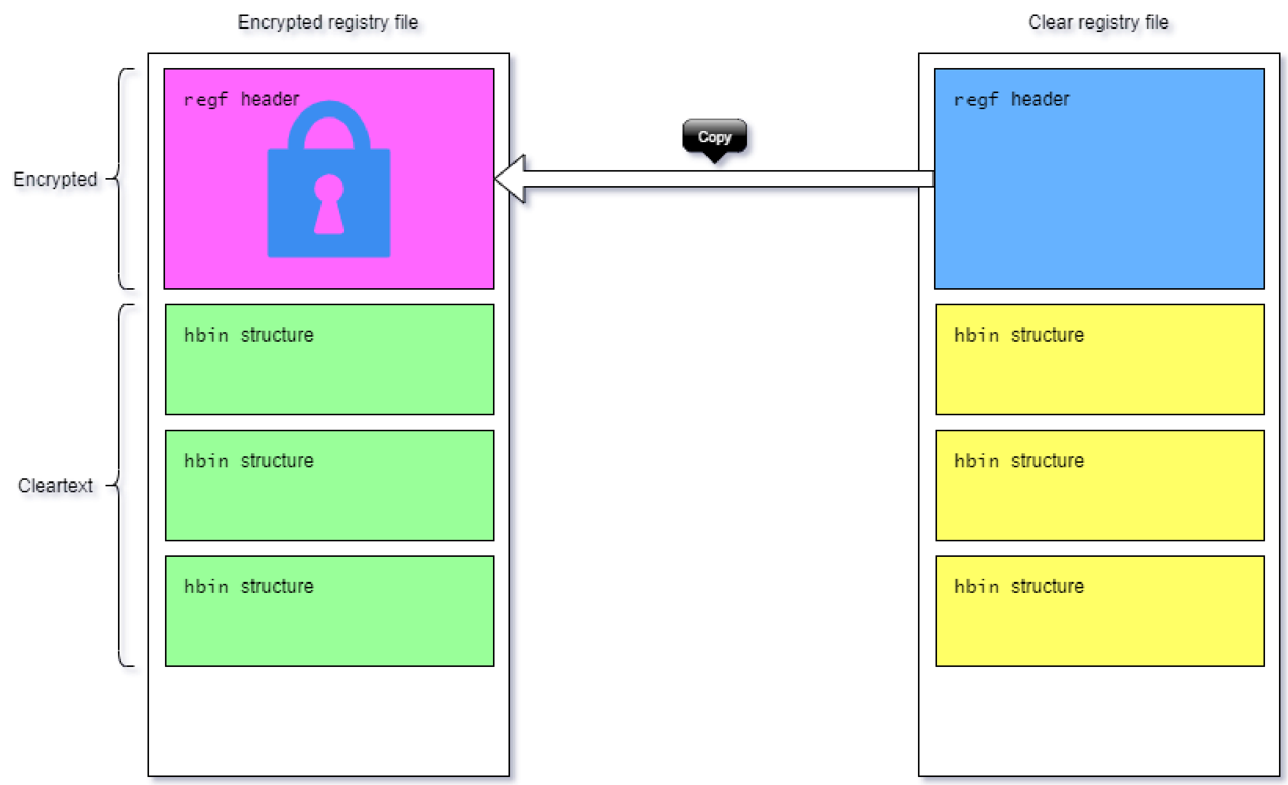
This means that LockBit only encrypts the header, and doesn't touch the actual data of the hive. Is it possible to restore the header? To answer this question, we must list everything the header contains:

- A magic number (`regf`)
- Sequence numbers (used for inconsistency detection)
- Modification timestamp
- Version numbers
- Hive name
- Hive flags
- Header checksum

Basically, we can see that the header is mostly self-contained: there's no reference to `hbin` offset, or a global hive checksum. There should be no problem restoring the header by copying it from another hive of the same type 😊

Restoring the hive 🔗

Simply by copying over the first 4096 bytes from another, clear `NTUSER.DAT`, we were able to entirely recover all our user hives!



```
$ regrip.py --ntuser ./ntuser_recovered.dat userassist | wc -l
84
```

It works! RegRippy will be confused when trying to give you the user names when extracting data from `NTUSER.DAT`, because it guesses them based on the hive name, which has been copied over from a clean hive. Other than that, everything works as expected, and all data is accessible.

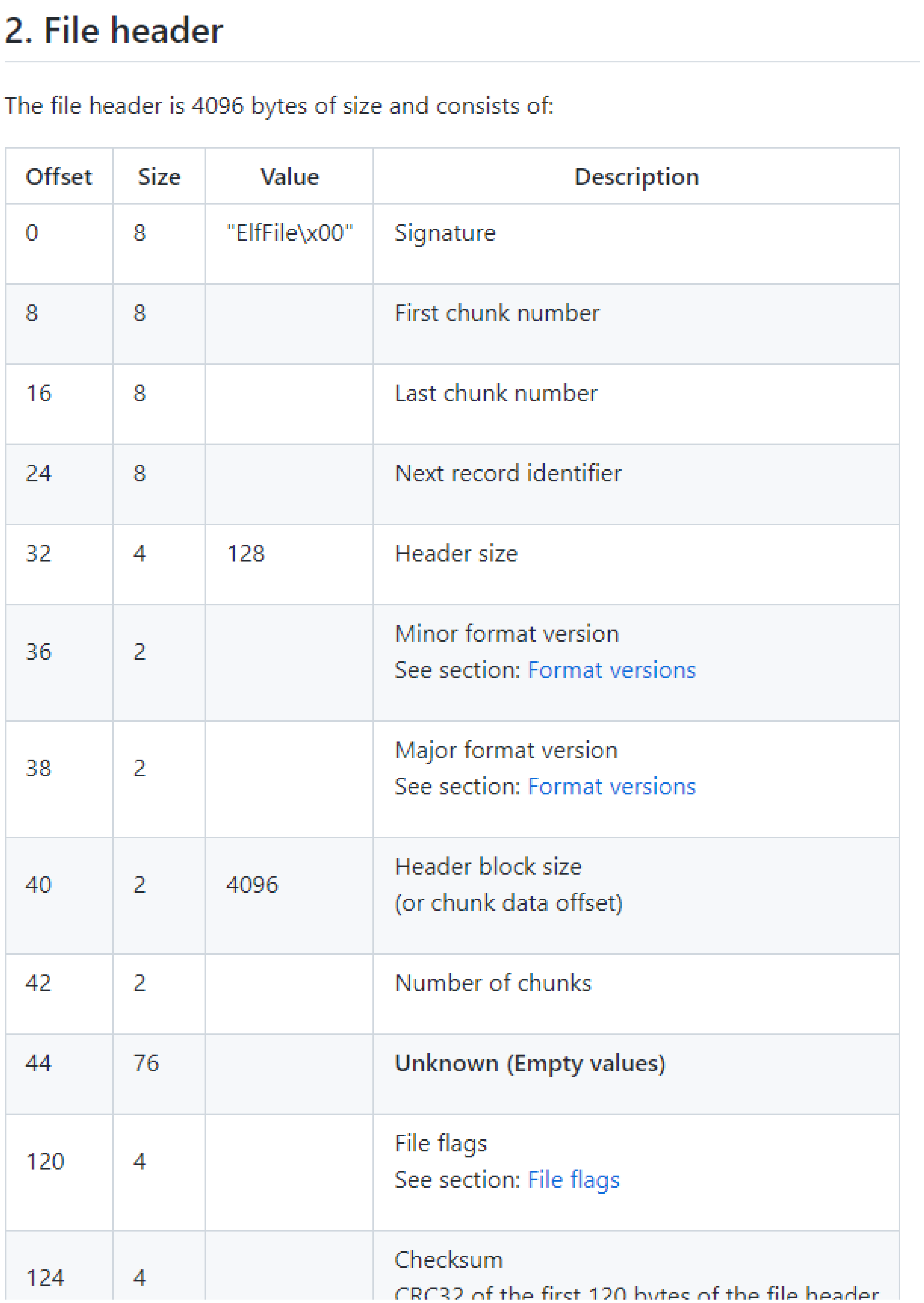
If you ever encounter this issue, here's a script which can restore an encrypted `NTUSER.DAT` hive: it's basically rebuilding the header and replacing it to create a clean hive.

```
#!/usr/bin/env python3

import argparse

def main():
    parser = argparse.ArgumentParser(description="Fix encrypted hives by repairing
the header (only for NTUSER.DAT)")

    parser.add_argument("--user", type=str, help="The user name to store in the
header (default: JohnDoe)", default="JohnDoe")
    parser.add_argument("hive_path", type=str, help="Encrypted hive path")
    parser.add_argument("output", type=str, help="Where to output the fixed hive")

    args = parser.parse_args()

    hive_name = "??\\C:\\Users\\" + args.user + "\\ntuser.dat"
    encoded_hive_name = hive_name.encode("utf-16-le")
    if len(encoded_hive_name) > 64:
        encoded_hive_name = encoded_hive_name[:64]
    else:
        encoded_hive_name += b"\x00" * (64 - len(encoded_hive_name))

    header =
b"regfH\x1E\x00\x00\x48\x1E\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x0

    header += encoded_hive_name
    header +=
b"\x43\xBE\x11\x44\xFF\x07\xE8\x11\x92\x75\xEA\x28\xA0\xD0\x3E\x60\x43\xBE\x11\x44\xFF

    header += b"\x00"*316
    header += b"\xD8\xC9\x70\x75"
    header += b"\x00"*3584

    with open(args.hive_path, "rb") as h1:
        with open(args.output, "wb") as h2:
            data = h1.read()
            without_header = data[4096:]
            h2.write(header)
            h2.write(without_header)

    print("Done! Hive written to", args.output)


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

(Update) Extending the technique to other file types

In passing, @citronneur mentioned that EVTX files also had a 4 KiB header. Maybe they could be reconstructed as well?

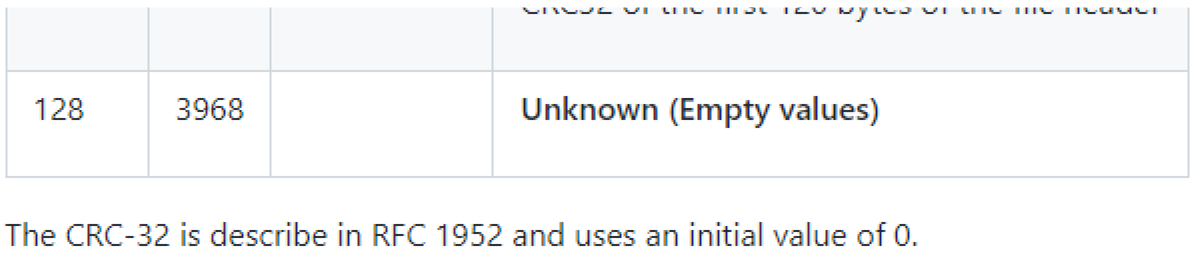
When investigating Windows file formats, it's always a good idea to check Joachim Metz's `libyal` repositories. In that case, bingo! `libevtx` exists, with some very detailed documentation.



2. File header

The file header is 4096 bytes of size and consists of:

Offset	Size	Value	Description
0	8	"ElfFile\x00"	Signature
8	8		First chunk number
16	8		Last chunk number
24	8		Next record identifier
32	4	128	Header size
36	2		Minor format version See section: Format versions
38	2		Major format version See section: Format versions
40	2	4096	Header block size (or chunk data offset)
42	2		Number of chunks
44	76		**Unknown (Empty values)**
120	4		File flags See section: File flags
124	4		Checksum CRC32 of the first 120 bytes of the file header



			CRC32 of the first 120 bytes of the file header
128	3968		**Unknown (Empty values)**

The CRC-32 is describe in RFC 1952 and uses an initial value of 0.

Basically, an EVTX file is composed of several chunks, and each chunk contains a number of records. Each record has an ID, which is unique across all chunks.

We assume the first chunk number is always 0. To get the last chunk number, we will search for the signature `"ElfChnk\x00"` and count its occurrences. We assume chunks are numbered in increasing order, starting from zero.

To get the last record ID, we first get the last chunk (easy, because each chunk has a fixed size), and parse the offset to the last record from its header. We then parse the record at this offset to extract its ID.

The checksum is a simple CRC32 of the first 120 bytes of the file header. With this, we are able to recreate all the data from the encrypted file header and read the events!

And here is a Python script which does just that:

```
#!/usr/bin/env python3

import argparse
import binascii
import sys


def get_number_of_chunks(data):
    count = 0
    needle = b"ElfChnk\x00"
    for offset in range(len(data) - len(needle)):
        if data[offset : offset + len(needle)] == needle:
            count += 1

    return count


def get_chunk(data, n):
    data = data[4096:]  # get rid of header
    print("[+] Getting chunk", n)
    chunk = data[n * 65536 : (n + 1) * 65536]
    assert chunk[:8] == b"ElfChnk\x00"
    return chunk


def get_last_record(chunk):
    offset = int.from_bytes(chunk[44:48], byteorder="little")
    print(f"[+] Last record offset: {offset} (0x{offset:x})")
    if offset == 0:
        print("[!] Error: this EVTX file probably has no events")
        sys.exit(1)
    record = chunk[offset:]
    assert record[:4] == b"\x2a\x2a\x00\x00"
    return record


def get_record_id(record):
    i = int.from_bytes(record[8:16], byteorder="little")
    print("[+] Record id:", i)
    return i


def main():
    parser = argparse.ArgumentParser(description="Fix LockBit2.0 EVTX file")

    parser.add_argument("file", type=str, help="Path to evtx file")
    parser.add_argument("output", type=str, help="Where to store the resulting file")

    args = parser.parse_args()

    data = None
    with open(args.file, "rb") as f:
        data = f.read()

    print("[+] Loaded", args.file)
    chunks = get_number_of_chunks(data)
    print("[+] Number of chunks:", chunks)

    signature = b"ElfFile\x00"
    first_chunk_number = (0).to_bytes(8, byteorder="little")
    last_chunk_number = (chunks - 1).to_bytes(8, byteorder="little")

    next_record_id = get_record_id(get_last_record(get_chunk(data, chunks - 1))) + 1
    next_record_id = next_record_id.to_bytes(8, byteorder="little")
    header_size = (128).to_bytes(4, byteorder="little")
    minor_version = (1).to_bytes(2, byteorder="little")
    major_version = (3).to_bytes(2, byteorder="little")
    header_block_size = (4096).to_bytes(2, byteorder="little")
    number_of_chunks = chunks.to_bytes(2, byteorder="little")
    unk1 = b"\x00" * 76
    file_flags = (0).to_bytes(4, byteorder="little")
    crc32 = -1
    unk2 = b"\x00" * 3968

    header = (
        signature
        + first_chunk_number
        + last_chunk_number
        + next_record_id
        + header_size
        + minor_version
        + major_version
        + header_block_size
        + number_of_chunks
        + unk1
        + file_flags
    )
    crc32 = binascii.crc32(header[:120]) & 0xFFFFFFFF
    header += crc32.to_bytes(4, byteorder="little")
    header += unk2

    assert (len(header)) == 4096

    with open(args.output, "wb") as f:
        f.write(header)
        f.write(data[4096:])


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