Cyber Apocalypse 2026 — Forensics — Harvesting Severed Threads

Category: Forensics **Artifacts:** `memory.elf` (3.2 GB LiME/ELF core), `dev_disk.img` (1 GiB), `capture.pcapng` (58 packets) **Flag:** `HTB{v0l4t1l3_1uk52_d3crypt10n_w1th_k3rn3l_k3yr1ng_4nd_w1r3gu4rd_3xf1l_brrr_brrr_brrr}`

---

1. Challenge

> Our infiltration team successfully breached House Vaultrune's development pipeline, extracting a volatile memory snapshot and a vow-encrypted drive directly from their covert scriptoriums. The Black Heir's clerks are secretly engineering half-written, malicious mandates meant to quietly automate the city's ruin. Carve through the dead memory and break the encrypted codex to expose the offensive weapons they are building before they can be deployed.

The flag is split into **three parts**, each protected by a different mechanism:

Part	Storage	Protection
`part_1`	SQLCipher database `/tmp/serpent.db`	AES-256-CBC, key = Argon2id(boot_id)
`part_2`	Linux kernel keyring (`keyring:part_2@serpent`)	ChaCha20-Poly1305, same Argon2id key
`part_3`	PDF exfiltrated over a WireGuard tunnel	AES-256-GCM inside ChaCha20-Poly1305 (WG)

Everything hangs off one root artifact: the **Serpent-XTS master key of a header-less LUKS2 volume**, which only exists in the memory dump.

---

2. Initial triage

```
file capture.pcapng dev_disk.img memory.elf
# capture.pcapng: pcapng capture file - version 1.0
# dev_disk.img:   data
# memory.elf:     ELF 64-bit LSB core file, x86-64  (LiME raw-ELF
memory image)
```

2.1 The capture

```
tshark -r capture.pcapng -q -z io,phs
#   udp  frames:53
#     wg  frames:47   <-- 47 WireGuard frames
```

One WireGuard session, `192.168.56.101:51820 → 192.168.56.1:51829`, one handshake pair and ~27 KB of transport data. mDNS leaks the victim hostname: `dev-seal-forge-02.local`.

2.2 The disk

`dev_disk.img` has **no LUKS/BitLocker/VeraCrypt header** — the first 4 MiB are literally zero:

```
python3 - <<'EOF'
import mmap
mm = mmap.mmap(open('dev_disk.img','rb').fileno(), 0,
access=mmap.ACCESS_READ)
step, runs, cur = 65536, [], None
for off in range(0, len(mm), step):
    nz = mm[off:off+step].count(0) != step
    if cur is None or cur[2] != nz:
        if cur: runs.append(cur)
        cur = [off, off+step, nz]
    else: cur[1] = off+step
runs.append(cur)
for a,b,nz in runs: print(f"{a:>12} - {b:>12} nonzero={nz}")
EOF
```

```
           0 -      4194304  nonzero=False   <-- wiped / absent header
area
     4194304 -     21823488  nonzero=True
```

```
   138412032 -    138477568  nonzero=True    64 KiB island
   406847488 -    406913024  nonzero=True    64 KiB island
   675282944 -    675348480  nonzero=True    64 KiB island
   943718400 -    943783936  nonzero=True    64 KiB island
```

The image is **sparse**: only blocks actually written by the filesystem hold ciphertext. The four 64 KiB islands sit at `4 MiB + 128 MiB × {1,3,5,7}` — textbook **ext4 backup superblocks** (`sparse_super`, 4 KiB blocks ⇒ 128 MiB block groups, backups in odd groups 1,3,5,7). That pins the encrypted **payload offset to 4 MiB (`0x400000`)** and tells us the inner filesystem is ext4 with 4 KiB blocks.

---

3. Reconstructing the crime scene from memory

`volatility3` is useless here — the kernel is a *fictional* `7.0.0-22-generic` (Ubuntu, May 2026), so no ISF symbol table exists. Everything below is done with plain string/structure carving.

```
LC_ALL=C grep -a -o -E 'COMMAND=[ -~]{0,200}' memory.elf | sort -u
```

```
COMMAND=/usr/sbin/cryptsetup open --key-file /run/media/dev5812/
dev_usb/luks_keyfile \
        --header /run/media/dev5812/dev_usb/dev_header.img ./
dev_disk.img dev_volume
COMMAND=/usr/bin/mount /dev/mapper/dev_volume dev_mnt/
COMMAND=/home/dev5812/dev_mnt/pyz/exfil
COMMAND=/usr/bin/sdmem                      <-- anti-forensics: wipe
free RAM
COMMAND=/usr/sbin/ip link del enp12s01      <-- WireGuard iface
disguised as ethernet
COMMAND=/usr/bin/su
```

So the volume is **LUKS2 with a detached header and a keyfile, both on a USB stick we do not have**. `dmsetup`/udev remnants confirm the mapping:

```
LC_ALL=C grep -a -o -E '/dev/mapper/[a-zA-Z0-9_.-]+' memory.elf |
sort -u
# /dev/mapper/dev_volume
```

```
DM_UUID=CRYPT-LUKS2-fee4d3439d49470d831500fb0e3101a0-dev_volume
ID_FS_UUID=80b415ab-7282-47b9-a63f-9701575b2ff0       <-- ext4 UUID
*inside* the container
```

The terminal scrollback (VTE) also survived:

```
dev5812@dev-seal-forge-02:~$ sudo ./dev_mnt/pyz/exfil
[sudo: authenticate] Password: **********************
[*] reading file into mutable buffer ...
[*] purging page cache ...
[*] creating wireguard interface ...
[*] encrypting & wiping plaintext from memory ...
[*] purging page cache again ...
sent 27525 encrypted bytes to ('10.0.0.1', 9999)
[+] done
```

(As a bonus, `grumpy-cryo-duck-spray` is floating in a heap — exactly the 22 characters masked as `**********`: the sudo password.)

Since neither header nor keyfile is available, the only way in is the **dm-crypt master key** still resident in kernel memory.

---

4. Carving the LUKS2 master key out of RAM

4.1 A verifiable known-plaintext oracle

We know the inner ext4 UUID `80b415ab-7282-47b9-a63f-9701575b2ff0`. In an ext4 superblock `s_uuid` lives at offset `0x68`, i.e. filesystem offset `1024 + 104 = 1128`. XTS block 6 of the sector covers bytes `1120..1136`, so the *first 8 UUID bytes* land at byte 8 of that block:

```
disk offset of that ciphertext block = 4194304 (payload offset) + 1120
expected plaintext[8:16] == 80 b4 15 ab 72 82 47 b9
```

That is an 8-byte oracle — a false-positive rate of 2^{-64}. For every candidate offset in the dump we take 64 bytes as `key1‖key2`, compute the XTS tweak, decrypt one block and compare.

4.2 Alignment trap

The first sweep found nothing. Reason: in a LiME **ELF** dump the program-header file offsets are *not* page aligned:

```
python3 - <<'EOF'
import struct
d=open('memory.elf','rb').read(0x2000)
ph=struct.unpack_from('<Q',d,0x20)[0];
n=struct.unpack_from('<H',d,0x38)[0]
for i in range(n):
    o=ph+i*56
    t,=struct.unpack_from('<I',d,o)
    off,va,pa,fsz,msz,al=struct.unpack_from('<QQQQQQ',d,o+8)
    print(t, off, hex(pa), fsz, "off%8=",off%8)
EOF
# every PT_LOAD has p_offset % 8 == 4
```

All physical pages are shifted by 4 bytes in the file, so an 8-byte-strided scan starting at 0 misses *every* 8-aligned kernel object. Fix: stride 4 (or 1).

4.3 Wrong cipher

Still nothing, even at byte granularity, for AES-XTS (32/64-byte keys) and AES-CBC. Notably `aes-xts-plain64` never appears in kernel memory — but this does:

```
python3 - <<'EOF'
import mmap
mm=mmap.mmap(open('memory.elf','rb').fileno(),0,access=mmap.ACCESS_REA
D)
i=mm.find(b'serpent-xts-plain64'); print(i)
EOF
# 92017917  -> a kmalloc-32 slab object next to module section names
```

`serpent-xts-plain64` is `struct crypt_config->cipher_string`. **The volume uses Serpent-XTS, not AES.** Nearby, the dm-crypt key reference and the keyring description also survive:

```
logon:cryptsetup:fee4d343-9d49-470d-8315-00fb0e3101a0-d0
cryptsetup:fee4d343-9d49-470d-8315-00fb0e3101a0-d0
```

4.4 The Serpent-XTS key finder

`nettle` provides a spec-conformant Serpent (verified against the official vector `key=80…00, pt=0…0 → A223AA1288463C0E2BE38EBD825616C0`).

```
/* sfind.c — compile: clang -O3 sfind.c -I/opt/homebrew/include -L/
opt/homebrew/lib -lnettle -o sfind */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <pthread.h>
#include <nettle/serpent.h>

static const unsigned char UUID0_8[8]={0x80,0xb4,0x15,0xab,0x72,0x82,0
x47,0xb9};
static unsigned char CT[64];              /* fs offsets 1088..1152
(XTS blocks 4..7) */
#define B(n) (CT+((n)-4)*16)
static const unsigned char *MEM; static size_t MEMLEN; static int
NTH=12,STRIDE=4;

static inline void gf_dbl(unsigned char *t){                /*
multiply tweak by alpha */
    unsigned char c=t[15]>>7;
    for(int i=15;i>0;i--) t[i]=(unsigned char)((t[i]<<1)|(t[i-1]>>7));
    t[0]=(unsigned char)((t[0]<<1)^(c?0x87:0));
}
typedef struct{size_t start,end;}job;

static void *worker(void*arg){
    job*j=(job*)arg; struct serpent_ctx c1,c2;
    unsigned char t[16],tw[16],x[16],pt[16];
    for(size_t off=j->start; off+64<=j->end; off+=STRIDE){
        const unsigned char*k=MEM+off;
        for(int v=0;v<2;v++){                       /* 512-bit and
256-bit XTS keys */
            int klen=v?32:64, half=klen/2;
            serpent_set_key(&c2,half,k+half);       /* tweak key */
            serpent_set_key(&c1,half,k);            /* data key  */
            for(int hyp=0;hyp<2;hyp++){             /* 512-byte and
4096-byte data units */
                memset(t,0,16);
```

```
                int nd; if(hyp==0){ t[0]=2; nd=6; } else { nd=70; }
                serpent_encrypt(&c2,16,tw,t);
                for(int i=0;i<nd;i++) gf_dbl(tw);
                for(int i=0;i<16;i++) x[i]=B(6)[i]^tw[i];
                serpent_decrypt(&c1,16,pt,x);
                for(int i=0;i<16;i++) pt[i]^=tw[i];
                if(!memcmp(pt+8,UUID0_8,8)){
                    printf("HIT off=%zu klen=%d hyp=%d key=",off,klen,
hyp);
                    for(int i=0;i<klen;i++) printf("%02x",k[i]);
                    printf("\n"); fflush(stdout);
                }
            }
        }
    }
    return NULL;
}
int main(int argc,char**argv){
    long ctoff=atol(argv[3]);
    if(argc>4) NTH=atoi(argv[4]);
    if(argc>5) STRIDE=atoi(argv[5]);
    int fd=open(argv[2],O_RDONLY); pread(fd,CT,64,ctoff); close(fd);
    int mf=open(argv[1],O_RDONLY); struct stat st; fstat(mf,&st); MEML
EN=st.st_size;
    MEM=mmap(NULL,MEMLEN,PROT_READ,MAP_SHARED,mf,0);
    pthread_t th[64]; job jb[64]; size_t chunk=MEMLEN/NTH;
    for(int i=0;i<NTH;i++){ jb[i].start=((size_t)i*chunk)&~3UL;
        jb[i].end=(i==NTH-1)?MEMLEN:(size_t)(i+1)*chunk+64;
        pthread_create(&th[i],NULL,worker,&jb[i]); }
    for(int i=0;i<NTH;i++) pthread_join(th[i],NULL);
    return 0;
}
```

```
./sfind memory.elf dev_disk.img $((4194304+1088)) 12 4
# HIT off=30543868 klen=64 hyp=1
#
key=6d092b4dcb45c0141e5306b7e8ee39ceecb8cb4b4b75f6f4e1b1f8f2d74773a0
#
d4ce7acf39d2197edc70fb453728b1713ed52e396c50217e99299a6797d350df
```

Master key (512-bit Serpent-XTS)

```
6d092b4dcb45c0141e5306b7e8ee39ceecb8cb4b4b75f6f4e1b1f8f2d74773a0
d4ce7acf39d2197edc70fb453728b1713ed52e396c50217e99299a6797d350df
```

4.5 Nailing the geometry

ext4 directory blocks of the mounted volume are still in page cache, which gives a byte-exact reference. Searching for the ext4 dirent pattern `\x0c\x00\x01\x02.\x00\x00\x00` recovers the volume's **root directory block** at memory offset `1262587876`:

```
ino=2     ft=2  .
ino=2     ft=2  ..
ino=11    ft=2  pyz
ino=8161  ft=2  secure_cb
```

Comparing that reference block against decryptions:

- data unit 4096 B → only the first **2048** bytes match
- data unit **2048 B**, IV step **4** (plain64 counts 512-byte sectors) → **all 4096 bytes match**

So the container is `serpent-xts-plain64`, `--sector-size 2048`, payload offset 4 MiB.

```c
/* dec2.c — XTS decryptor (same gf_dbl / serpent as above) */
static void dec_unit(uint64_t sec, unsigned char *buf){
    unsigned char t[16],tw[16],x[16];
    memset(t,0,16);
    for(int i=0;i<8;i++) t[i]=(unsigned char)(sec>>(8*i));   /*
plain64 IV */
    serpent_encrypt(&c2,16,tw,t);
    for(int b=0;b<SEC/16;b++){
        unsigned char *p=buf+b*16;
        for(int i=0;i<16;i++) x[i]=p[i]^tw[i];
        serpent_decrypt(&c1,16,p,x);
        for(int i=0;i<16;i++) p[i]^=tw[i];
        gf_dbl(tw);
    }
}
/* main loop: pread 4 MiB at a time from offset 4194304, sec += 4 per
2048-byte unit */
```

```
./dec2 dev_disk.img plain.img 4194304 <KEY> 2048 4
file plain.img
# plain.img: Linux rev 1.0 ext4 filesystem data,
```

```
#           UUID=80b415ab-7282-47b9-a63f-9701575b2ff0 (extents)
(64bit)
```

```
debugfs -R "ls -l /secure_cb" plain.img
#  8161  40755  .
#     2  40755  ..
#    30 100770  serpent_source.zip   13597811
#    31 100644  Cargo.lock             107876
#    32 100644  Cargo.toml                719
#    33 100755  serpent_db             36282528
#    34  40755  src
debugfs -R "rdump / /tmp/vol" plain.img
```

---

5. `part_1` — the SQLCipher database

`/tmp/vol/secure_cb/src/main.rs` is a Rust "boot-bound secure codebook":

```
const DB_PATH:      &str   = "/tmp/serpent.db";
const BOOT_ID_PATH: &str   = "/proc/sys/kernel/random/boot_id";
const SALT:         &[u8]  = b"serpent_secure_salt_2026";

fn derive_key(boot_id: &str) -> Result<SecureKey> {
    let params = Params::new(262144, 4, 4, Some(32))?;           //
256 MiB, t=4, p=4
    let argon2 = Argon2::new(Algorithm::Argon2id, Version::V0x13, para
ms);
    let mut key_bytes = [0u8; 32];
    argon2.hash_password_into(boot_id.as_bytes(), SALT, &mut key_bytes
)?;
    Ok(SecureKey(Zeroizing::new(key_bytes)))
}
```

The key is bound to the machine's boot id — which is in the dump:

```
LC_ALL=C grep -a -o -E '_BOOT_ID=[0-9a-f]{32}' memory.elf | sort -u
# _BOOT_ID=710d1eb09a774c9ca148a8dd002d8755   ->  710d1eb0-9a77-4c9c-
a148-a8dd002d8755
```

```
printf '710d1eb0-9a77-4c9c-a148-a8dd002d8755' \
  | argon2 'serpent_secure_salt_2026' -id -t 4 -m 18 -p 4 -l 32 -r
# 4af1975878abc3db67aa68d510f848d277f2854fb5cf9ac92906c23a769e46e4
```

Locating the encrypted DB pages in RAM

`/tmp/serpent.db` never touched a disk we own, but its page cache is in the dump. SQLCipher 4 page layout is `ciphertext ‖ IV(16) ‖ HMAC(64)` with `reserve = 80`, so for a page-1 candidate we can decrypt the first CBC block with the raw key (IV taken from offset 4016) and check the plaintext SQLite header fields: `page_size = 0x1000`, `reserved = 0x50`, `0x40 0x20 0x20`.

```
/* findsqlc.c core */
AES_decrypt(p+16, pt, &dk);                    /* p = candidate 4096-
byte page */
for (int i=0;i<16;i++) pt[i] ^= p[4016+i];     /* CBC: xor stored IV
*/
if (pt[0]==0x10 && pt[1]==0x00 && pt[4]==0x50 &&
    pt[5]==0x40 && pt[6]==0x20 && pt[7]==0x20) report(offset);
```

```
./findsqlc memory.elf
4af1975878abc3db67aa68d510f848d277f2854fb5cf9ac92906c23a769e46e4 12 1
# SQLCIPHER PAGE1 @908525540 hdr=10000101504020208d6555a400000003
#   salt=b677a08047b28ebfb1443eb9b6ced9df       -> db is 3 pages long
```

Three consecutive pages, decrypt each with AES-256-CBC (IV at `page[4016:4032]`), re-attach the `SQLite format 3\0` magic that SQLCipher replaced with the salt:

```
from cryptography.hazmat.primitives.ciphers
import Cipher, algorithms, modes
import mmap
key  = bytes.fromhex('4af1975878abc3db67aa68d510f848d277f2854fb5cf9ac9
2906c23a769e46e4')
mm   = mmap.mmap(open('memory.elf','rb').fileno(), 0, access=mmap.ACCE
SS_READ)
base, PS, RES = 908525540, 4096, 80
out = bytearray()
for k in range(1, 4):
    page = mm[base+(k-1)*PS : base+k*PS]
    iv   = page[PS-RES : PS-RES+16]
    ct   = page[16:PS-RES] if k == 1 else page[0:PS-RES]
    pt   = (lambda d: d.update(ct)+d.finalize())(Cipher(algorithms.AES
```

```
(key), modes.CBC(iv)).decryptor())
    out += (b'SQLite format 3\x00' + pt if k == 1 else pt) + b'\x00'*R
ES
open('serpent_plain.db','wb').write(out)
```

```
sqlite3 serpent_plain.db 'select * from records;'
# part_1|HTB{v0l4t1l3_1uk52_d3crypt10n_
```

```
part_1 = HTB{v0l4t1l3_1uk52_d3crypt10n_
```

---

6. `part_2` — the kernel keyring

`main.rs` also stores records in the **kernel user keyring**, ChaCha20-Poly1305-encrypted with the same Argon2id key and base64-encoded as `nonce(12) ‖ ciphertext ‖ tag(16)`:

```
let cipher = ChaCha20Poly1305::new((&*key.0).into());
OsRng.fill_bytes(&mut nonce_bytes);
let ciphertext = cipher.encrypt(nonce, value.as_bytes())?;
let encoded = general_purpose::STANDARD.encode([&nonce_bytes[..], &cip
hertext[..]].concat());
entry.set_password(&encoded)?;                          //
keyring_core::Entry::new("serpent", &name)
```

The key description survives verbatim:

```
LC_ALL=C grep -a -o -E 'keyring:[ -~]{0,40}' memory.elf | sort -u |
grep serpent
# keyring:part_2@serpent
```

Sweep every base64 run in the dump and try to authenticate it:

```
import mmap, re, base64
from Crypto.Cipher import ChaCha20_Poly1305
key = bytes.fromhex('4af1975878abc3db67aa68d510f848d277f2854fb5cf9ac92
906c23a769e46e4')
mm  = mmap.mmap(open('memory.elf','rb').fileno(), 0, access=mmap.ACCES
S_READ)
```

```
for m in re.finditer(rb'[A-Za-z0-9+/]{40,400}={0,2}', mm):
    s = m.group(0)
    for L in range(len(s), 39, -1):
        if L % 4: continue
        try: raw = base64.b64decode(s[:L], validate=True)
        except Exception: continue
        if len(raw) < 29: continue
        try:
            pt = ChaCha20_Poly1305.new(key=key, nonce=raw[:12]) \
                                  .decrypt_and_verify(raw[12:-16],
raw[-16:])
        except Exception: continue
        print(m.start(), pt); break
```

```
30545308  b'w1th_k3rn3l_k3yr1ng_4nd_'
```

The payload itself, at memory offset `30545308`:

```
vFq7uwx/
2Zet+LsG3dpjbe1cJdfbykcWbitZs22pUysEUQREdJTNkc6jDDnYOHEHhSDFQw==
```

```
part_2 = w1th_k3rn3l_k3yr1ng_4nd_
```

---

7. `part_3` — decrypting the WireGuard exfiltration

7.1 Recovering the exfil tool from the page cache

`/dev_mnt/pyz/exfil` is a PyInstaller one-file binary. Its **CArchive cookie** is still in RAM:

```
import mmap, struct
mm = mmap.mmap(open('memory.elf','rb').fileno(), 0, access=mmap.ACCESS
_READ)
c  = mm.find(b'MEI\x0c\x0b\x0a\x0b\x0e')                    #
259222522
ln, toc, toclen, pyvers = struct.unpack('>IIII', mm[c+8:c+24])
start = c + 88 - ln                                          #
243286717
```

```
# ...walk the TOC: 88 entries, incl.  s exfil  off=24715 dlen=3274
ulen=6450
```

The archive body is *not* physically contiguous, so instead of carving by offset we scan the whole dump for zlib streams and keep the large ones:

```
import mmap, re, zlib
mm = mmap.mmap(open('memory.elf','rb').fileno(), 0, access=mmap.ACCESS
_READ)
for m in re.finditer(rb'\x78[\x01\x5e\x9c\xda]', mm):
    d = zlib.decompressobj()
    try: res = d.decompress(mm[m.start():m.start()+6000])
    except Exception: continue
    if len(res) >= 1500:
        open(f'/tmp/zs/{m.start()}_{len(res)}.bin','wb').write(res)
```

One blob is the marshalled `exfil.py` code object; `strings` on it yields all constants:

```
enp12s01
GPXmuw+xS8WjAzjPvkECvLGd2iSRwcjFP+OWPbNzsUY=
WG_CLIENT_PRIVATE_KEY
10.0.0.2/24                                          WG_CLIENT_ADDRESS
34P8Lh/ROgOdv7ciFVw1BCfBlvn2H0TBuDpCd8FZaHE=       WG_SERVER_PUBLIC_KEY
192.168.56.1                                         WG_SERVER_ENDPOINT
https://www.youtube.com/watch?v=oHafFDkFgeg          HARDCODED_SECRET
/root/dummy.pdf                                      FILE_PATH
10.0.0.1                                             SEND_DST
AESGCM / hashlib.sha256 / pyroute2.WireGuard / purge_file_cache /
secure_wiper
```

So: the file is read, AES-256-GCM encrypted with `sha256(HARDCODED_SECRET)`, and pushed over a TCP socket to `10.0.0.1:9999` *inside* a WireGuard tunnel — then plaintext and page cache are wiped.

Knowing the client's **static** private key is not enough to decrypt WireGuard (the ephemeral private keys are needed for the Noise chain). But the **transport session keys** are what actually encrypt the data.

7.2 Brute-forcing the Noise session keys

WireGuard transport uses ChaCha20-Poly1305 with `nonce = 0x00000000 ‖ counter_le64` and **no AAD**. Frame 14 is a keepalive from the victim (`counter = 0`,

empty payload, 16-byte tag) — the cheapest possible oracle: one ChaCha20 block plus a Poly1305 over 16 zero bytes per candidate.

```
/* wgkey.c — clang -O3 wgkey.c -lsodium */
if (crypto_aead_chacha20poly1305_ietf_decrypt_detached(
        NULL, NULL, NULL, 0, MAC, NULL, 0, NPUB, MEM+o) == 0) { /* 32
bytes at MEM+o is the key */ }
```

```
./wgkey memory.elf 0972afe7ff41a467aba5b60664c56997 0 12 1
# KEY @159651844 =
2d3d193f3889d53e9d1788e2fb42589d2b5390a2160c061baa6529e6da1da21c
```

That is `noise_keypair.sending.key`. In `struct noise_keypair` the receiving key follows 56 bytes later (`key[32] ‖ birthdate ‖ is_valid` + counters), and it validates against frame 16:

```
sending   (101 -> 1) :
2d3d193f3889d53e9d1788e2fb42589d2b5390a2160c061baa6529e6da1da21c
@159651844
receiving (1 -> 101) :
953c534a5e6402ff3ed260cc23383beba5dccb31adb4119429bd9c54f3e8a42e
@159651900
```

Memory around the struct:

```
159651812  ...728affff 00000000 00000000 10601a97 728affff 02000000
026cca44   <- entry / remote index
159651844  2d3d193f3889d53e9d1788e2fb42589d
2b5390a2160c061baa6529e6da1da21c   <- sending.key
159651876  a65c3c77ce000000 0100000000000000 1b00000000000000
953c534a5e6402ff <- birthdate, is_valid, ctr
159651908  3ed260cc23383beba5dccb31adb41194
29bd9c54f3e8a42e ...               <- receiving.key
```

7.3 Reassembling and decrypting

```
import struct
from Crypto.Cipher import ChaCha20_Poly1305
SEND = bytes.fromhex('2d3d193f3889d53e9d1788e2fb42589d2b5390a2160c061b
aa6529e6da1da21c')
RECV = bytes.fromhex('953c534a5e6402ff3ed260cc23383beba5dccb31adb41194
```

```
29bd9c54f3e8a42e')
segs = {}
for line in open('wgpkts.tsv'):
# tshark -T fields -e ip.src -e udp.payload
    n, src, payhex = line.rstrip('\n').split('\t')
    pay = bytes.fromhex(payhex)
    if len(pay) < 16 or pay[0] != 4: continue        # keep type-4
transport data
    counter = struct.unpack('<Q', pay[8:16])[0]
    key   = SEND if src == '192.168.56.101' else RECV
    nonce = b'\x00'*4 + struct.pack('<Q', counter)
    pt = ChaCha20_Poly1305.new(key=key, nonce=nonce) \
                          .decrypt_and_verify(pay[16:-16], pay[-16:])
    if not pt or pt[9] != 6: continue                # inner IPv4 /
TCP only
    ihl = (pt[0] & 0xf) * 4
    ip  = pt[:struct.unpack('>H', pt[2:4])[0]]
    tcp = ip[ihl:]
    seq = struct.unpack('>I', tcp[4:8])[0]
    data = tcp[(tcp[12] >> 4) * 4:]
    if data and struct.unpack('>H', tcp[2:4])[0] == 9999:
        segs[seq] = data
ks, out = sorted(segs), bytearray()
for k in ks: out[k-ks[0]:] = segs[k]
open('exfil_stream.bin','wb').write(bytes(out))      # 27529 bytes
```

The stream is `uint32be length ‖ nonce(12) ‖ AES-GCM ciphertext ‖ tag(16)`:

```
import hashlib, struct
from Crypto.Cipher import AES
raw     = open('exfil_stream.bin','rb').read()
ln      = struct.unpack('>I', raw[:4])[0]            # 27525  <-
matches "sent 27525 encrypted bytes"
payload = raw[4:4+ln]
key     = hashlib.sha256(b"https://www.youtube.com/watch?
v=oHafFDkFgeg").digest()
pt      = AES.new(key, AES.MODE_GCM, nonce=payload[:12]) \
             .decrypt_and_verify(payload[12:-16], payload[-16:])
open('dummy.pdf','wb').write(pt)
# %PDF-1.7, 27497 bytes
```

```
pdftotext -layout dummy.pdf - | tail -20
```

```
              HOUSE VAULTRUNE
 *** SIMULATED EXFILTRATION PAYLOAD - DO NOT ARREST COURIER ***
```

```
...
The capital's procedural defenses remain completely vulnerable. ...

             part_3: w1r3gu4rd_3xf1l_brrr_brrr_brrr!!}
```

```
part_3 = w1r3gu4rd_3xf1l_brrr_brrr_brrr!!}
```

---

8. Flag

```
part_1  HTB{v0l4t1l3_1uk52_d3crypt10n_
part_2  w1th_k3rn3l_k3yr1ng_4nd_
part_3  w1r3gu4rd_3xf1l_brrr_brrr_brrr!!}
```

```
HTB{v0l4t1l3_1uk52_d3crypt10n_w1th_k3rn3l_k3yr1ng_4nd_w1r3gu4rd_3xf1l_brrr_brrr_br
```

---

9. Bonus: the "offensive weapons"

Unallocated space of the decrypted volume still holds a **deleted LKM source** (`full_chain.c`) — the "half-written malicious mandate" from the briefing. It resolves WireGuard symbols via a `kallsyms_lookup_name` kprobe, injects a rogue `exfil_peer` into `wg->peer_list` with a hard-coded capture-server public key, routes a target IP through it, ships the flag file out with `kernel_sendmsg`, then removes the allowed-IPs entry while deliberately leaving the peer alive "as a forensic artifact":

```
strings -n 8 plain.img | grep -E 'exfil_peer|flag_buf|send_flag'
```

```
static char *flag_path     = "/root/dummy.pdf";
static char *wg_iface      = "wg0";
static char *target_ip_str = "10.0.0.100";
static int   capture_port  = 51828;
static const u8 capture_server_pubkey[32] = { /*
9YfryRJzmjRAUTUubsFKjtyeCdRWOcoTQ1B5KwctRg0= */ };
/* CTF path: memory dump + pcap -> find exfil_peer in wg->peer_list
```

```
 *           -> extract noise_keypair ChaCha20 keys -> decrypt
WireGuard UDP -> recover flag. */
```

---

10. Lessons / takeaways

1. **A wiped LUKS header is not the end.** As long as the mapping is live, `crypt_config` keeps the raw master key. A single 8-byte known-plaintext (the inner `ID_FS_UUID` from udev) turns 3.2 GB of RAM into a tractable key search.
2. **Never assume AES.** `serpent-xts-plain64` sitting in a `kmalloc-32` slab was the whole reason the first two sweeps failed. Grep for the *cipher string*, not for the algorithm you expect.
3. **LiME ELF dumps are not page aligned.** All `PT_LOAD` offsets here were `≡ 4 (mod 8)`; scanning at stride 8 from offset 0 silently misses every aligned kernel structure.
4. **Verify geometry against page cache.** A single surviving ext4 directory block gave a 4096-byte ground-truth that distinguished a 2048-byte XTS data unit from a 4096-byte one.
5. **Boot-bound keys are only as secret as the boot id** — `_BOOT_ID=` in the journal defeats the entire Argon2id/SQLCipher design.
6. **WireGuard is forward-secret, but session keys are just bytes in RAM.** You do not need the Noise handshake: a 16-byte Poly1305 tag from a keepalive is a perfect oracle for locating `noise_keypair.sending.key`, and the receiving key sits 56 bytes further on.
7. **Anti-forensics (`sdmem`, `ip link del`, page-cache purges) is incomplete by nature.** Terminal scrollback, journald records, dentry slabs, keyring payloads and PyInstaller TOCs all survived.