WeChat macOS 4.1.x Database Decryption — Process Report

Agent: Claude Opus 4.7 xHigh (Claude Code)
Date of work: April 2026
Target: `/Applications/WeChat.app` on macOS Tahoe 26 / Apple Silicon M3 Pro
Outcome: ✅ All-but-one of the ~dozens of encrypted SQLite databases decrypted, ~hundreds of thousands of pages, 0 bad pages. Tens of GB archive preserved (encrypted + decrypted copies).

Note: this document is provided for educational purposes only. Claude Code may make mistake when executing this plan, and could result in your data being transmitted, without your knowledge, to Anthropic; it could also corrupt your WeChat installation and existing databases. To use this file, simply attach it in a Claude Code session, and create a new folder with a copy of your WeChat `.app` file to get started.

---

1. Objective

Produce an offline, browseable archive of the user's personal WeChat chat history for long-term preservation. Deliverable: the on-disk encrypted SQLite databases decrypted to plaintext, with enough supporting tooling to answer "give me the conversation with X."

Two questions posed at the start:

1. How does WeChat macOS encrypt its chat databases?
2. How can the user extract their own chat records for archival?

---

2. Privacy constraints

The operator imposed a hard rule: **no raw or decrypted WeChat data may appear in the Claude/Anthropic conversation transcript.** Every script writes to local files; every verification uses aggregate signals only (file sizes, row counts, exit codes, HMAC pass/fail); all confirmation of correctness is done by the operator visually on their own machine.

This shaped the entire approach. Most decryption tutorials happily `cat` the DB key and sample messages. We built pipelines that keep the sensitive artifacts (32-byte keys, plaintext DB bytes) on disk behind `chmod 600`, and surface only metadata to the operator.

Future agents doing this kind of work: **establish the privacy envelope before the first tool call.** Decide what's allowed in the transcript vs what must stay on disk. It changes how you design helper scripts.

---

3. Target system at a glance

- macOS Tahoe 26.x / darwin 25.x, Apple Silicon M3 Pro (6P + 6E cores, 36 GB unified memory).
- WeChat.app version 4.1.28 at snapshot time; auto-updated to **4.1.107** mid-session via Sparkle. Version drift broke some later attach attempts — see §7.
- User data directory: `~/Library/Containers/com.tencent.xinWeChat/Data/Documents/xwechat_files/<wxid>/` (WeChat 4.x uses this *new* path, not the older `Application Support/2.0b4.0.9/<wxid>/` path that still appears alongside it for legacy reasons).
- Active account: `wxid_<redacted>`, tens of GB of data (a few GB of DBs + the bulk in attachments + some cache).
- Legacy account: `<redacted-hex-id>` under the old path, small (tens of MB), out of scope.

---

4. Encryption architecture (what we learned)

4.1 Stack

WeChat 4.x uses **WCDB** (Tencent's open-source fork of SQLCipher, github.com/Tencent/wcdb), **statically linked into `wechat.dylib`** (~300 MB main library) with stripped symbols. All the SQLCipher PRAGMA strings are present in the binary (`cipher_page_size`, `cipher_use_hmac`, `kdf_iter`, `cipher_hmac_pgno`), but no exported C symbol for `sqlite3_key` et al. Dynamic imports confirm:

- `_CCKeyDerivationPBKDF` (CommonCrypto)
- `_CCHmacInit/Update/Final` (CommonCrypto)
- `_CCCryptorCreate/Update/Final/Release` (CommonCrypto)

WeChat uses **CommonCrypto** for AES-CBC and HMAC primitives. This is the hook surface that ultimately worked (§6).

4.2 Per-DB encryption parameters

Each encrypted DB file is a SQLCipher-v4-format blob with these fixed parameters (verified by working decryption):

Parameter	Value
Cipher	AES-256-CBC
HMAC	HMAC-SHA512, 64-byte MAC per page
Page size	4096
Reserved bytes per page	80 (16-byte IV + 64-byte HMAC)
Salt location	First 16 bytes of file (page 1 only)
KDF for mac_key	PBKDF2-HMAC-SHA512, 2 iterations, salt XOR 0x3a
KDF for enc_key	**NOT used at runtime** (see next)

4.3 The key-management surprise

WCDB in WeChat 4.x does **not** derive the `enc_key` from a user password via PBKDF2 at runtime. Instead:

- Each DB gets its own random 32-byte `enc_key`, generated once at first creation.
- Keys are stored encrypted on disk (Keychain-wrapped; `key_info.dat` at `.../app_data/login/<wxid>/key_info.dat` plays a role).
- At startup, WeChat decrypts these keys and holds them in memory.
- To decrypt a page: WCDB passes the raw 32-byte `enc_key` directly into `CCCryptorCreate(kCCDecrypt, kCCAlgorithmAES, kCCOptionsPKCS7Padding=?, enc_key, 32, iv, ...)`. No KDF.
- The `mac_key` is derived from `enc_key` via `PBKDF2(enc_key, salt XOR 0x3a, iters=2, out=32, SHA-512)` on every page HMAC computation.

Critical for future attempts: chatlog's darwin V4 decryptor assumes a password-based path (PBKDF2 with 256,000 iterations). That assumption is stale for WeChat 4.x macOS. A captured password would not validate even if we had one — because there isn't one in the flow. The thing in memory that matters is the *already-derived* 32-byte enc_key, and it's unique per DB.

4.4 Per-DB key separation

On this install we captured **several dozen unique 32-byte enc_keys**. Mapping them against the DB files in the archive: **all-but-one matched exactly one captured key each** (different DB = different key). The one unmatched DB (`message_resource.db`) simply hadn't been opened during our capture window, so its key wasn't in memory yet.

This means a decryption pipeline must:

1. Capture many (key, DB) pairs, not just one magic key.
2. Validate each captured key against each DB's page-1 HMAC to build a keymap.
3. Decrypt each DB with its specific key.

4.5 Per-column compression (secondary gotcha)

WCDB applies **per-column zstd compression** on TEXT columns, flagged row-by-row by an auxiliary `WCDB_CT_<column_name>` integer column. If `WCDB_CT_message_content` is non-zero, the bytes in `message_content` are zstd-compressed and must be decompressed before UTF-8 decode.

Without this decompression step, reading message bodies returns garbled Chinese.

---

5. What did NOT work (catalogued for future reference)

Every one of these ate hours. Documenting so the next run doesn't repeat them.

5.1 `chatlog` (github.com/sjzar/chatlog) — didn't work on WeChat 4.1.x

- v0.0.31 at the time. Installed cleanly via `go install`.
- `chatlog key` requires SIP disabled; reports `"SIP is enabled"` and refuses to scan.
- After disabling SIP: `chatlog key` ran to completion but every scan returned `"no valid key found"`.
- Root cause analysis revealed chatlog's anchor pattern (`\x20 fts5(%\x00`) with offsets `[+16, −80, +64]` is stale for WeChat 4.1.x — the memory layout around those SQL template buffers has shifted.
- Even when we widened the anchor-offset search to ±8192 B with chatlog's exact v4 algorithm (256k-iter PBKDF2 of the candidate as password), no match. Because the memory doesn't hold a *password* — it holds a derived enc_key. chatlog was validating under wrong assumptions.
- **Chatlog issues #232, #279, #317, #321** all document this same failure on recent WeChat 4.x builds. No upstream fix.

5.2 `wechat-dump-rs` — can't verify integrity

- GitHub returns **HTTP 451** (Unavailable for Legal Reasons).

- Tencent filed a sweeping DMCA takedown in Jan 2026 across ~4,195 WeChat-extraction repos; `0xlane/wechat-dump-rs` was caught in it. Not on crates.io either.
- Any fork we'd pull would be from an unknown mirror — supply-chain risk not worth taking given the user's privacy posture.
- **Not used.**

5.3 Bulk memory scanning with SQLCipher-v3/v4/WCDB hypotheses

- Wrote a custom Python scanner (`wcdb_key_scanner.py`) that tests every 32-byte aligned window in a memory dump as a candidate enc_key across 3 cipher configs (v3 SHA-1, v4 SHA-512, WCDB SHA-256).
- Ran against full-style core dumps of `WeChat` (~7 GB total, ~hundreds of MB of RW segments), `WeChatAppEx` (similar scale), and `ILinkServiceHost` (a few MB).
- **Never hit.** The enc_key IS in WeChat's RAM at some moment — but either (a) it's stored split/obfuscated, (b) the byte layout doesn't survive a stride-8 scan, or (c) a full dump doesn't capture the exact region where an alive enc_key currently lives (buffers get reused).
- Also tried an *anchor-based* scanner (`wcdb_key_scanner_v2.py`) using chatlog's fts5 pattern with widened offsets. Still no hit.
- Conclusion: **linear memory scanning for WeChat 4.1.x enc_keys is not reliable.** The keys exist in RAM; finding them without runtime instrumentation is the hard problem.

5.4 `DYLD_INSERT_LIBRARIES` dylib injection

- Compiled a small dylib (`pbkdf2_hook.c`) using `DYLD_INTERPOSE` on `CCKeyDerivationPBKDF` , ad-hoc signed.
- Even with SIP fully disabled, **AMFI (Apple Mobile File Integrity) still enforces** that hardened-runtime binaries refuse `DYLD_INSERT_LIBRARIES` unless the inserted library has a matching Team ID (which ad-hoc signing does not).
- Attempted through `sudo lldb` with `settings set target.env-vars DYLD_INSERT_LIBRARIES=...` → env var stripped silently at exec time.
- Workarounds exist (`amfi_get_out_of_my_way=1` boot-arg, or full binary re-signing without `--options=runtime`), but both are deeply invasive and we had alternatives.
- **Not used in the final flow.**

5.5 CCKeyDerivationPBKDF breakpoint

- Hypothesis: if WCDB ever calls PBKDF2 to derive enc_keys, capture the arguments.
- Implemented with an lldb Python breakpoint callback (`lldb_wcdb_capture.py`).
- Captured **a couple dozen PBKDF2 calls** with rounds=256000, 32-byte passwords, 16-byte salts — matching chatlog's v4 algorithmic signature.

- But: none of the captured salts matched **any** on-disk DB salt. Across the ~100k+ files in the live WeChat data directory, zero matches.
- These PBKDF2 calls turned out to be for **authentication / session key derivation**, not DB encryption. Chat DB enc_keys in WeChat 4.1.x are *not* derived via CCKeyDerivationPBKDF at all — they're loaded from encrypted storage (see §4.3).
- **Dead end, but instructive.**

5.6 `sudo lldb launch` trap

- Launching WeChat via `sudo lldb /Applications/WeChat.app/Contents/MacOS/WeChat` **runs WeChat as root**.
- Root WeChat uses a different sandbox container: `/var/root/Library/Containers/com.tencent.xinWeChat/` — empty, fresh install, no user data.
- Our PBKDF2 captures from that session were authenticating a *fresh root account*, not the operator's actual account. Painful to notice because the process tree looked identical.
- **Always attach to a normally-launched (user-owned) WeChat**, not a debugger-launched one, when you need real data in context.

5.7 Process-recycling chaos

- WeChat 4.x spawns many children (WeChatAppEx, multiple helper processes, ILinkServiceHost, renderer processes). PIDs change between runs.
- Sparkle auto-update ran mid-session (4.1.28 → 4.1.107), restarted all child processes, invalidated our existing memory dumps.
- `pkill -f WeChat` is dangerous — it matches *any* command line containing "WeChat", including `sudo lldb ... WeChat`, killing lldb itself.
- Use `pkill -x` (exact process name match) or `pkill -i '^WeChat'` (regex anchored to start) to avoid self-kill.

---

6. What worked (the final path)

Clean sequence that produced the decryption:

6.1 Snapshot the encrypted archive

```
rsync -a --exclude='cache/' --exclude='temp/' \

~/Library/Containers/com.tencent.xinWeChat/Data/Documents/xwechat_files/<wxi
```

```
d>/ \
  ~/Documents/wechat-history/archive/$(date +%Y-%m-%d)/
```

Takes ~4 min for ~32 GB on an SSD. We did this while WeChat was live; WCDB's atomic-page design makes this safe.

6.2 Disable SIP

- Shut down → Recovery (power-hold on Apple Silicon → Options → Continue) → Utilities → Terminal → `csrutil disable` → reboot.
- SIP off is required for `task_for_pid` on a hardened-runtime target. Without it lldb cannot attach to WeChat at all on macOS 15+.
- Re-enable at end (§7.3).

6.3 Attach lldb via `--waitfor` and install a CCCryptorCreate breakpoint

This was the key insight. We don't need to derive keys from first principles — we just watch WCDB *use* them. The hook point is `CCCryptorCreate`, because WCDB calls it with the raw 32-byte enc_key as argument 4 every time it decrypts a page.

Files:

- **`lldb_cccryptor_capture.py`** — Python breakpoint callback. On every CCCryptorCreate with `op=kCCDecrypt (1)` and `keyLen ∈ [8,64]`, reads args from registers (`x0=op, x1=alg, x3=key, x4=keyLen, x5=iv`), copies the key bytes + IV out of the target process, appends a record to `~/.chatlog/wcdb-cccryptor.bin` (chmod 600). The callback returns `False` so lldb auto-continues — WeChat keeps running, just slightly slower under the breakpoint.
- **`lldb_waitfor_commands.txt`** — lldb command script:

  ```
  command script import .../lldb_cccryptor_capture.py
  process attach --name WeChat --waitfor
  install_cccryptor_bps
  continue
  ```

Procedure:

1. Fully kill all WeChat processes (`pkill -9 -i '^WeChat'`, `pkill -9 -i '^(wxocr|wxplayer|wxutility)$'`, verify empty).
2. Clear old captures: `rm -f ~/.chatlog/wcdb-cccryptor.bin`.
3. Start lldb in wait-for mode: `sudo lldb -s lldb_waitfor_commands.txt`. This blocks.
4. In a second terminal: `open -a WeChat` (launches as the user, with user's real container).

5. lldb catches the exec at `_dyld_start`, installs the breakpoint, resumes.
6. User scans QR on phone to log in. Clicks through a few chats.

This captures **tens of thousands of CCCryptorCreate records in ~5 minutes** of live use (startup + a few chat opens), collapsing to **several dozen unique 32-byte keys**.

6.4 Validate captured keys against every DB

Script: **`map_keys_to_dbs.py`**. For each captured key and each encrypted DB in the archive, compute the SQLCipher-v4 HMAC of page 1 using that key's derived `mac_key`. If it matches the stored HMAC at offset `[4032:4096]`, we have the right key for this DB.

Result: **all-but-one DB mapped**. The one unmapped DB (`message_resource.db`) simply hadn't been opened during the capture window. Writes a `keymap.txt` (`<relative-path>\t<hex-key>` per line, chmod 600).

6.5 Decrypt each DB with its key

Script: **`wcdb_decrypt.py`**. Pure-Python using `cryptography`'s AES-CBC. Page layout for WeChat 4.x:

- **Page 1:** `salt(16) | ciphertext(4000) | iv(16) | hmac-sha512(64)`
- **Page N>1:** `ciphertext(4016) | iv(16) | hmac-sha512(64)`
- **HMAC input:** `page_bytes[16 or 0 : pageSize-reserve+IVSize] || uint32_LE(page_number)`

For each page: verify HMAC → AES-CBC-decrypt ciphertext with that page's IV → write to output. Page 1 output is prefixed with the canonical `SQLite format 3\x00` 16-byte header because SQLCipher overwrites the first 16 bytes with the salt on disk.

Result: **hundreds of thousands of pages decrypted, 0 bad pages.** `sqlite3` opens every output file cleanly. Contact table confirmed with a reasonable row count.

6.6 Decompress WCDB per-column compression at read time

When reading `message_content` and similar TEXT columns: check the paired `WCDB_CT_<column>` integer. If non-zero, the bytes are zstd-compressed. Use `zstandard.ZstdDecompressor().decompress()` before UTF-8 decode. Falls back to zlib for edge cases.

---

7. Tooling & artifact map

All paths are under `~/Documents/wechat-history/` unless noted.

File	Purpose
`lldb_cccryptor_capture.py`	lldb Python module: CCCryptorCreate breakpoint + callback
`lldb_waitfor_commands.txt`	lldb command script for waitfor-attach flow
`lldb_wcdb_capture.py`	Older: CCKeyDerivationPBKDF breakpoint (kept for reference; didn't yield DB keys but did reveal auth-derivation pattern)
`lldb_capture_commands.txt`, `lldb_cccryptor_commands.txt`	Earlier lldb scripts — superseded by the waitfor version
`pbkdf2_hook.c`, `pbkdf2_hook.dylib`	DYLD_INTERPOSE dylib — built but AMFI blocks it. Kept for reference.
`map_keys_to_dbs.py`	Validates each captured key against each encrypted DB's page-1 HMAC; emits `keymap.txt`
`wcdb_decrypt.py`	Per-DB AES-CBC-decrypt loop; uses keymap; writes plaintext SQLite files
`wcdb_key_scanner.py`, `wcdb_key_scanner_v2.py`	Failed memory scanners (kept for diagnostic reference)
`validate_captured_key.py`, `decrypt_key_info.py`, `decrypt_key_info_v2.py`	Explorations of the `key_info.dat` decrypt path — not needed in final flow but documents the disk-wrapping blob shape
`archive/<date>/`	Rsync of encrypted DBs + media attachments (~32 GB)
`decrypted/`	Plaintext SQLite outputs (~3 GB)
`wcax-full.core`, `wmain-full.core`, `wmain-mod.core`, `wcax.core`, `wilink-mod.core`	Process memory dumps — no longer needed, safe to delete (many GB reclaimable)
`~/.chatlog/wcdb-cccryptor.bin`	Raw CCCryptorCreate capture records (tens of thousands total, a few hundred KB). Contains several dozen unique keys. **Sensitive.**
`~/.chatlog/wcdb-cccryptor-status.txt`	Human-readable per-call log (algorithm + key length only, no key bytes). Not sensitive.
`~/.chatlog/keymap.txt`	`<db-relative-path>\t<hex-key>` per line. **Extremely sensitive.**

8. Reproduction (condensed)

For anyone redoing this on a similar WeChat 4.1.x macOS install:

```
# 0. Baseline: SIP must be off. See §6.2.

# 1. Snapshot encrypted data (safe while WeChat is live)
SRC=~/Library/Containers/com.tencent.xinWeChat/Data/Documents/xwechat_files
# find your wxid dir: ls "$SRC"
WXID=wxid_yourid_here
DST=~/Documents/wechat-archive/$(date +%Y-%m-%d)
mkdir -p "$DST"
rsync -a --exclude='cache/' --exclude='temp/' "$SRC/$WXID/" "$DST/"

# 2. Kill WeChat cleanly
sudo pkill -9 -i '^WeChat'
sudo pkill -9 -i '^(wxocr|wxplayer|wxutility)$'
rm -f ~/.chatlog/wcdb-cccryptor.bin

# 3. Start lldb in wait-for mode (needs lldb_waitfor_commands.txt +
lldb_cccryptor_capture.py)
sudo lldb -s /path/to/lldb_waitfor_commands.txt &
# (terminal will block; that's the point)

# 4. Launch WeChat normally (from Dock or `open -a WeChat`). lldb catches
exec.
# Log in via QR. Click a few chats. Wait 3-5 min for the capture to
stabilize.

# 5. Save the capture and detach
sudo chown $(whoami) ~/.chatlog/wcdb-cccryptor.bin
# In lldb: `process detach`, then `quit`

# 6. Build keymap
python3 map_keys_to_dbs.py \
  --records ~/.chatlog/wcdb-cccryptor.bin \
  --in-dir "$DST/db_storage" \
  --out ~/.chatlog/keymap.txt

# 7. Decrypt everything
python3 wcdb_decrypt.py \
  --keymap ~/.chatlog/keymap.txt \
  --in-dir "$DST/db_storage" \
  --out-dir "$DST/decrypted"
```

```
# 8. Re-enable SIP (Recovery → csrutil enable → reboot)
```

Typical timing on an M3 Pro:

Step	Wall time
rsync tens of GB encrypted data	~4 min
SIP disable reboot cycle	~5 min
lldb + WeChat capture (QR, login, a few chats)	~10 min
keymap build	~5 s
decrypt 3 GB of DBs	~90 s
SIP re-enable reboot cycle	~5 min
Total hands-on	**~25 min if you know the path**

The failed approaches documented in §5 represent the *learning* — if you follow §6/§8 directly, you skip all of them.

---

9. Notes for future AI agents

9.1 Don't trust community tooling without checking upstream issues

Before committing to a tool (chatlog, wechat-dump-rs, pywxdump, WeChatMsg), check:

- Upstream GitHub issues for your exact WeChat build version.
- Whether the repo is still accessible (Tencent's DMCA sweep in Jan 2026 took down ~4,195 tools).
- Whether the tool's algorithmic assumptions match the current WeChat version. Older tools assume SQLCipher-v3 on macOS; WeChat 4.x switched to v4-ish parameters with a different key-provenance path.

If a tool can't find a key in 10 minutes with reasonable configuration, it's probably the wrong tool for this version — don't throw days at it. Pivot.

9.2 Prefer runtime instrumentation over memory scanning

Memory scans (anchored or brute-force) assume a predictable byte layout. WCDB/WeChat changes this between versions. lldb breakpoints on `CCCryptorCreate` (or the equivalent

dynamic import of whatever crypto primitive WCDB uses) are stable across versions — as long as WCDB keeps using CommonCrypto, which it has since at least WeChat 3.x.

Always look at `otool -L` and `nm -u` on the main app binary first. The dynamic imports tell you the instrumentation surface. If the binary imports `_CCCryptorCreate`, that's your best hook point and you don't need to find static WCDB symbols.

9.3 Use `--waitfor` instead of racing startup

`lldb process attach --name X --waitfor` is race-free — it catches the process at `_dyld_start`, before any app code has run. Essential for catching whatever happens in early startup (key decryption, DB open, first syscall).

9.4 Never `sudo lldb` a launch; attach instead

`sudo lldb /path/to/app` makes the app run as root, which for sandboxed apps means a completely different container (`/var/root/Library/Containers/<bundle>`) with no user data. You'll capture a pristine fresh-install session that's useless.

Correct pattern: user launches the app normally → `sudo lldb -p <pid>` attaches with root debugger privileges, but the target process remains under the user's UID and sees the user's data.

Or for race-sensitive captures: `sudo lldb -s waitfor-commands.txt`, then the user does `open -a AppName`. lldb's waitfor catches the user-owned process.

9.5 AMFI ≠ SIP

On macOS 14+, disabling SIP is necessary but not sufficient for full DYLD_INSERT_LIBRARIES injection into hardened-runtime binaries. AMFI enforces library-validation separately. The usable bypasses are:

- **Ad-hoc re-signing the target binary** to strip `--options=runtime`. Most invasive; can break Keychain/container access due to Team-ID change.
- **Boot-arg `amfi_get_out_of_my_way=1`.** Requires another reboot and leaves the system in a weaker security state.
- **Running in lldb** — interpose sections are processed at startup and work fine under the debugger, which is what we used de-facto via our breakpoint callback (no DYLD_INSERT needed).

For this kind of extraction, skip DYLD_INSERT entirely. The lldb breakpoint approach is cleaner and doesn't require re-signing.

9.6 Pay attention to per-DB key separation

WeChat 4.x uses a distinct 32-byte enc_key per DB file. A working pipeline must:

1. Collect many keys (10s to 100s).
2. Validate each against each DB's page-1 HMAC.
3. Build a `<db-path, key>` mapping.
4. Decrypt per-DB.

Tooling that tries to find "the key" will fail even if it works correctly — there is no single key.

9.7 Know the per-column compression format

WCDB compresses TEXT columns with zstd, flagged per-row by an auxiliary integer column `WCDB_CT_<column>`. Row `c` is compressed iff `WCDB_CT_c != 0`. This trips up anyone reading the decrypted DB directly with `sqlite3` and wondering why Chinese text is garbled.

Detection: check `sqlite_master` for any `WCDB_CT_*` column names on tables of interest. If present, apply zstd-decompression gating for every read.

9.8 Respect the operator's privacy envelope

The hardest engineering constraint on this job wasn't technical — it was the ground rule that no chat content could enter the Claude/Anthropic transcript. It changed a lot:

- Verification scripts write to local files instead of printing.
- Permission checks go to `chmod 600` before writing sensitive outputs.
- Scripts take arguments like `--out <path>` instead of printing keys.
- The agent surfaces aggregate signals (file size, row count, HMAC pass/fail) instead of content.
- When a manual step requires the operator to see a secret (e.g., inspecting a 32-byte key), the script writes it to a local file and the operator inspects it in their own terminal — never through the agent's tool output.

Set this envelope **before** the first tool call. Once leaked, secrets can't be redacted from a transcript.

9.9 Don't fight moving targets

Mid-session we hit:

- A WeChat auto-update (4.1.28 → 4.1.107) that restarted all child processes.
- Child-process recycling (ILinkServiceHost died and respawned under a new PID after relaunch).

- An accidental `sudo pkill -f WeChat` that killed lldb because lldb's command line contained "WeChat".

Lesson: when something reshuffles PIDs mid-run, *verify the current state* before continuing. `pgrep -x WeChat` (exact), `ps -axo pid,ppid,user,command` (tree with user), and paying attention to which user a process runs as (`/var/root/...` container path vs `/Users/<user>/...` is a dead giveaway).

9.10 Capture more than you need, decrypt in bulk

The CCCryptorCreate hook is cheap to run (auto-continuing breakpoints are microseconds of per-call overhead). Let it run for 5+ minutes of user activity — the user clicks through chats, triggers DB opens, and we catch more and more keys. Then build the keymap in one shot.

Conversely: don't try to be precise about *which* moment the key fires. Collect everything, filter server-side with the HMAC validator. This is far more robust than engineering a single surgical breakpoint on "the" key-init call.

Appendix A — Operator's final state

- Encrypted archive: `~/Documents/wechat-history/archive/<date>/` (tens of GB)
- Decrypted DBs: `~/Documents/wechat-history/decrypted/` (a few GB)
- Keys + keymap: `~/.chatlog/wcdb-cccryptor.bin` , `~/.chatlog/keymap.txt`
- Analysis workspace: `~/Documents/<analysis-workspace>/` (chat transcripts, JSONL, summaries)
- SIP: **TO RE-ENABLE** — user should boot to Recovery → `csrutil enable` → reboot. Do not leave SIP disabled indefinitely.
- Process memory dumps (`*.core` , many GB total): safe to delete now.

Appendix B — References

- WCDB source (public): github.com/Tencent/wcdb
- SQLCipher v4 format spec: www.zetetic.net/sqlcipher/sqlcipher-api/
- chatlog upstream (still accessible at time of writing): github.com/sjzar/chatlog
- Tencent's Jan-2026 DMCA takedown record: github.com/github/dmca/blob/master/2026/01/2026-01-08-tencent.md
- macOS codesigning & hardened runtime: `man codesign` , Apple's "Hardened Runtime" developer docs

- CommonCrypto API: Apple Developer — `<CommonCrypto/CommonKeyDerivation.h>`, `<CommonCrypto/CommonCryptor.h>`