0-DAY NO CVE HIGH PUBLIC

SQLite WAL page-size mismatch leaks a pager tail

SQLite 3.53.4 still short-reads a WAL frame into a full pager page with no tail memset. Recovery accepts a WAL page size that is a smaller valid power of two than the database. Recycled pcache then returns prior page bytes through SQL. Labbed on 3.53.4. Not RCE. No CVE yet.

NAME SQLite WAL page-size mismatch leaks a pager tail	TYPE 0-day analysis
CVE n/a	CVE RISK high
DISCLOSURE STATUS public	VENDOR SQLite (Hwaci)
AFFECTED SQLite 3.53.4 (b09c88c, SOURCE_ID 2026-07-24). Leftover of later trunk WAL page-size compare / sqlite3WalReadFrame tail memset. Local untrusted .db plus sibling -wal. Default library (WAL compiled in).	PUBLISHED 09 Oct 2026
UPDATED 09 Oct 2026	TAGS 0-day, sqlite, wal, info-leak, cwe-908, cwe-200, local

TL;DR

Lab files and proof-of-concept here.

01 The WAL said 512. The pager believed 4096.

I am @abraxas_null. The proof of concept is on GitHub: **abraxas/sqlite-wal-pagesize** (loopback client). The lab stack is `lab/` : Dockerfile, docker-compose.yml, run.sh, poc.py. Authorized lab only. It talks to loopback.

This is **SQLite 3.53.4** (`b09c88c14082339b66c7b7158d609a771e64ca69`). Fossil SOURCE_ID `2026-07-24 19:02:57 bf7c7f30031888f4e796e429ab3978879485813aaca6f641c7b33e4e09459bcc` . No CVE yet. CWE-908 / CWE-200. 7.5 High (`CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`). Local. Untrusted `.db` plus sibling `-wal` . Default library. Not RCE.

I asked SQLite for a 2000-byte blob that did not exist on disk. It handed me 2000 bytes anyway. The first 496 were `Q` from a 512-byte WAL frame. The rest was a witness string I had written into a different table, still sitting in a recycled pager slot. The WAL header said the page was 512 bytes. The database header said 4096. Recovery shrugged. The read path copied 512 bytes into a 4096-byte buffer and did not touch the tail. SQL is an excellent hex dump when you let it be one.

02 After HTTP servers, a library that eats files

I had just finished unpublished Crit/High hunts of Caddy, lighttpd, OpenLiteSpeed, and PHP. HTTP servers argue about headers. SQLite argues about pages. It is in phones, browsers, desktop apps, backup tools, and every language's "just open this file" wrapper. sqlite.org/security.html is unusually honest: untrusted SQL and

untrusted database files are first-class inputs. A hostile `.db` plus a sibling `-wal` is not a stretch. It is the documented victim model.

I only wanted Critical or High. Local is in-scope. Medium stays on the floor unless it chains. A finding on this hunt needed a leftover of a named later-trunk fix, a crash or leak on untrusted SQL or an untrusted DB file, or a privilege control the C API still advertises. `load_extension` is off on default open. I am not filing "you can load a `.so` if you turn that on."

03 Pinning 3.53.4

Latest GA. Not an RC. Not the draft. sqlite.org still pointed at 3.53.4. 3.54.0 was a draft with a planned date. Drafts move. GA is what ships in distros and amalgamation zips.

Canonical source is sqlite.org/src. GitHub sqlite/sqlite is a Fossil mirror. The tag `version-3.53.4` lands on `b09c88c14082339b66c7b7158d609a771e64ca69`. `VERSION` says `3.53.4`. `sqlite3_sourceid()` says the Fossil line above. That is the pin. If a leftover of an August trunk check-in still fires here, it is not a stale minor.

04 The skip list is half the hunt

Before you look for leftovers, you write down what is already public so you do not "discover" a CVE with a press release. SQLite's list lives at sqlite.org/cves.html. On 3.53.4 the interesting ones were already closed:

- CVE-2026-11822 / CVE-2026-11824 FTS5 heap overflow when `SQLITE_DBCONFIG_DEFENSIVE` is off. Duplicate CVEs. Fixed 3.53.2. `fts5LeafRead` rejects `szLeaf<4`.
- CVE-2026-50812 session NPD on a malformed changeset. Optional `-DSQLITE_ENABLE_SESSION`.

- CVE-2025-3277 / CVE-2025-29087 `concat_ws()` integer overflow. Fixed 3.49.1.
- CVE-2025-29088 LOOKASIDE OOB. Fixed 3.49.1.
- CVE-2025-7709 FTS5 corrupt-index integer overflow. Fixed 3.50.3.
- CVE-2025-70873 zipfile CLI OOB. CLI extra.
- CVE-2024-0232 JSON UAF. Fixed 3.43.2.
- WAL-reset corruption (Tailscale race). Fixed 3.51.3 / 3.53.0. Salt check is in this pin.
- Ada Logics `sqlite3_rsync` stack smash (`readPow2` `x>16`). In tree.
- A 2026 cluster of "critical SQLite CVEs" that SQLite and JFrog both treated as LLM slop. I left those on the floor.

Older FTS3 / printf / NULL CVEs are years before this pin. Third-party JDBC and application bugs are other trees.

05 Twenty surfaces, one leftover in this post

I split the tree. Parser and VDBE. Btree and pager. WAL. FTS3. FTS5. JSON. vtab / ATTACH / pragma. trusted_schema and authorizer. CLI and zipfile. `sqlite3_rsync`. Session and RBU. deserialize / memdb. Triggers, views, window. rtree / geopoly. WASM. 3.54 patch siblings. load_extension and VFS. C API. recover and misc extras. Schema side effects.

Most of that is SQLite doing what 3.53.4 said it would do.

`load_extension` stays off. JSON UAF class is closed. WAL-reset salt check is in. shm map sizes are `i64`. `readonly_shm` `szPage==0` is rejected before checksum. Inverse WAL (larger than the pager) is `SQLITE_CORRUPT` on checkpoint, not a heap overflow. I am not filing those.

Default core still has crash paths on untrusted SQL. COLLATE depth skips `SQLITE_LIMIT_EXPR_DEPTH`. CTE attach skips the view-depth cap. VALUES UNION ALL can blow the error-path destructor. Those are other posts. This one is the WAL.

The leftover that survived on the WAL surface is not a new idea. Winter reported it. Trunk fixed it. 3.53.4 did not pick the fix up. That is the whole genre.

06 Winter got there first. The GA did not.

Forum: 2026-08-17T15:54:41Z. Status Resolved on trunk. Reporter verified 3.53.4. Check-in comment, paraphrased: when the WAL page size and the database page size disagree, fully initialize the page that is read.

Trunk `f9b4ee3695` / branch-3.53 `4fde87b2a0` is 2026-08-16. This GA is 2026-07-24. Three weeks is a long time in SQLite years and a short time in distro years. Latest GA still has the short-read.

I did not "discover" the mismatch. I verified it is still the read path on the pin people actually ship, and I labbed the pcache tail through ordinary SQL with a witness the process had just written.

07 Two gaps, one SELECT

Two independent misses still combine on this tree.

Recovery never compares WAL size to DB size

`walIndexRecover` reads WAL header byte 8 as `szPage` and accepts any power of two in range:

```c
szPage = sqlite3Get4byte(&aBuf[8]);
if( (magic&0xFFFFFFFE)≠WAL_MAGIC
 || szPage&(szPage-1)
 || szPage>SQLITE_MAX_PAGE_SIZE
 || szPage<512
){
  goto finished;
}
pWal→szPage = szPage;
```

`wal.c` around 1446. No compare against pager or btree page size. A 512-byte WAL next to a 4096-byte database is a valid recovery as far as this predicate is concerned. Commit frames then store that size into the wal-index header.

Pager page size comes from the `.db` header, independently. `btree.c` decodes bytes 16-17. `sqlite3PagerSetPagesize` sizes `pPager→pageSize` and the pcache slots to that value. `pagerOpenWalIfPresent` opens the sibling `-wal` during `sqlite3PagerSharedLock` before btree finishes page-1 init. WAL recovery therefore runs with WAL header `szPage` while the pager already uses the `.db` page size.

Opening a WAL-mode file automatically uses WAL. The victim does not have to type `PRAGMA journal_mode=WAL`. Database header byte 19 equal to 2 is enough.

The read path short-fills a full pager buffer

`readDbPage` always asks for the pager page size:

```c
rc = sqlite3WalReadFrame(pPager→pWal, iFrame, pPager→pageSize, pPg→pData);
```

`pager.c` around 3082. `sqlite3WalReadFrame` on this pin:

```c
int sqlite3WalReadFrame(
  Wal *pWal, u32 iRead, int nOut, u8 *pOut
){
  int sz;
  i64 iOffset;
  sz = pWal->hdr.szPage;
  sz = (sz&0xfe00) + ((sz&0x0001)<<16);
  iOffset = walFrameOffset(iRead, sz) + WAL_FRAME_HDRSIZE;
  return sqlite3OsRead(pWal->pWalFd, pOut, (nOut>sz ? sz : nOut), iOffset);
}
```

`wal.c` around 3658. Exact `(nOut>sz ? sz : nOut)`. No `memset` of `pOut[sz .. nOut)`. WAL `sz` 512, pager `nOut` 4096: 512 bytes written, 3584 bytes stale.

`unixRead` zeros only the unread tail of the requested `amt`. `amt` is already `min(nOut,sz)`, so the remaining `nOut-sz` pager bytes are never touched. `readDbPage` converts `SQLITE_IOERR_SHORT_READ` to OK only on the non-WAL branch. A complete 512-byte WAL frame returns `SQLITE_OK` with the 3584-byte tail still whatever the slot was.

Trunk inserts:

```c
if( nOut>sz ){
  memset(pOut+sz, 0, nOut-sz);
  nOut = sz;
}
```

That hunk is absent here.

The inverse mismatch is not this leftover. Checkpoint refuses `walPagesize(pWal)≠nBuf`. Write-path `walFrames` does the same. Smaller WAL, read path, uninitialized tail. That is the bug.

08 Why the tail is visible through SQL

Page-cache slots are recycled without zeroing. `pcache1AllocPage` reuses `pCache->pFree` or a heap page that was not `calloc`'d. `pcache1FetchStage2` recycles an unpinned LRU page and only reallocates if `szAlloc` differs. Same-size recycle keeps prior page bytes. `pager_acquire` `memset`s the no-content / beyond-eof path. WAL hits skip that zero.

Btree treats the full pager page as valid. Table-leaf `maxLocal` is `usableSize-35`. On a 4096-byte database page that is 4061 bytes. A 2000-byte blob is fully local. `btreeParseCellPtr` takes the payload varint as `nLocal` when it is `⩽ pPage->maxLocal`. `SQLITE_CellSizeCk` is off unless you compiled `SQLITE_ENABLE_OVERSIZE_CELL_CHECK`. Even with that flag, a cell that starts in the first 512 bytes and whose local payload fits in the 4096-byte usable region passes `btreeCellSizeCheck`.

`accessPayload` copies `nLocal` bytes from `pPayload` inside the page buffer. The only bound is `usableSize` (the database page), not WAL `sz`. A table-leaf blob whose declared length is larger than the bytes actually present in the 512-byte frame is therefore satisfied from the uninitialized tail. `SELECT`, `length()`, `substr()`, `hex()` surface it. You do not need a debugger. You need SQL.

Winter's public report matches this path: WAL header 512, DB 4096, one commit frame for a leaf whose blob is long, cache primed by prior queries. I wanted a witness I had put in the process myself, not "nonzero tail." Nonzero could be anything. A repeating token from a sibling table is an argument.

Do not upgrade this to RCE. Pcache layout puts `PgHdr` after the page buffer. The disclosed tail is prior page content or fresh malloc residue, not the live `PgHdr` pointers of the current slot. Pointer-ish bytes in a recycled *new* heap page are possible. I did not demonstrate control of them. Winter privately mentioned an FTS5 heap overflow as ASLR material. That is a different leftover on a different extra. This post is the WAL. Not RCE.

09 The image is the amalgamation

Hub does not ship "SQLite 3.53.4 as a service." The lab builds it. Debian bookworm-slim, canonical `sqlite-src-3530400.zip` from sqlite.org, `./configure`, `sqlite3.c` compiled `-O2 -g -fPIC -shared` into `/usr/local/lib/libsqlite3.so`. `poc.py` is ctypes against that `.so`. It refuses to proceed unless `sqlite3_libversion()` is `3.53.4` and `sqlite3_sourceid()` is the Fossil line above. If you point this at distro libsqlite3 and the SOURCE_ID is different, that is a different pin.

Compose project `sqlite-wal-pagesize`. Loopback `127.0.0.1:18830`. The published port is an HTTP waiter so `run.sh` knows the container is up. The leftover is not HTTP. The leftover is `sqlite3_open` on a pair of files under `/work`.

10 The lab

Create a 4096-page WAL-mode database. `PRAGMA auto_vacuum=NONE`. Table `t(b BLOB)` with one dummy row, so it has a root page. Table `prime(b BLOB)` with 48 rows of 4000-byte witness. Checkpoint TRUNCATE so the WAL is empty. Record `t`'s rootpage (2) and `page_count` (51). Close. Unlink the sidecars.

Write a new `-wal`: header page size 512, native magic, version 3007000, salt `ABRAXAS!`, checksums that match `walChecksumBytes`. One commit frame for page 2. Frame body is a table-leaf (`0x0D`) whose single cell claims a 2000-byte blob of `Q`. The cell is larger than 512 bytes. Only the prefix that fits is stored in the frame. The rest of the claim lives in pager space that the WAL never wrote.

Reopen. `PRAGMA cache_size=16`. `SELECT` the prime table so pcache fills with witness pages. Then `SELECT b FROM t`.

PLAIN TEXT Copy

```
select-blob len=2000 in-frame-nonzero=False tail-len=1504 tail-hex=4553530a53514c
SUCCESS sqlite-wal-pagesize how=wal-shortread-pcache-tail len=2000 rootpage=2 SQL
```

`4553530a` is `ESS\n`. That is the end of `WITNESS\n` repeating through the tail. Bytes past the 512-byte frame are the prime table. SQL returned them as the blob I asked for.

Checksum probe first: open a real 512-page WAL-mode DB, read the header SQLite wrote, confirm the Python port of `walChecksumBytes` matches. If the checksum is wrong, recovery treats the WAL as empty and you have labbed a checksum, not a leftover. Magic on this host was `0x377f0682`. Native. Fine.

Replay:

PLAIN TEXT Copy

```
cd lab && ./run.sh
```

Or from a fresh clone:

PLAIN TEXT Copy

```
git clone https://github.com/abraxas/sqlite-wal-pagesize
cd sqlite-wal-pagesize/lab
./run.sh
```

Bind it to loopback. The loopback client is `lab/run.sh` with a banner on it. It builds SQLite from source. Give it a few minutes.

11 Wrong turns, in the order I made them

Treating forum Resolved as "fixed in 3.53.4." Resolved means trunk. The reporter already said 3.53.4 was still hot. Latest GA is the pin. Trunk dates after SOURCE_ID 2026-07-24 are leftovers until a release picks them up. 3.54.0 was still a draft. Pinning the draft is how you "prove" a fix that nobody has.

Pinning GitHub master because the Fossil mirror is inconvenient. GitHub is a mirror. Canonical is sqlite.org/src. SOURCE_ID is the pin string the library will print. If `sqlite3_sourceid()` does not match, you are not on 3.53.4.

Re-filing WAL-reset, shm-map overflow, or `readonly_shm` szPage=0. Closed on this pin. The skip list exists so the WAL hunt does not turn into a greatest-hits album.

Scoring the inverse mismatch as a heap overflow. WAL larger than the pager is `SQLITE_CORRUPT` on checkpoint. `walFrames` rejects it on write. The leftover is the smaller-WAL read path. Half a mismatch is how leftovers look when the other half already has a guard.

Claiming RCE from recycled pointers. `PgHdr` sits after the page buffer. The tail is prior page content. An FTS5 overrun on a compile extra is a different finding. I am not going to chain them on paper and call it remote code execution. Info leak. High. Stop.

Expecting a SELECT on a cold cache to leak the witness. If pcache has never held the prime pages, the tail is zeros or malloc residue and you FAIL `tail-zero`. Prime first. `cache_size=16` and 48×4000-byte rows is enough churn that the next 4096-byte slot still has the token. That is the same shape as Winter priming with prior queries. It is not heap grooming. It is "the cache is a cache."

Checksum endian. SQLite writes `WAL_MAGIC | SQLITE_BIGENDIAN` and checksums natively. Recovery's native flag is the magic LSB against `SQLITE_BIGENDIAN`. I probed a real WAL SQLite had just written and compared. The first mapping was right on this little-endian host. If you invert it you get a WAL recovery ignores, and then you write a FAIL about checksums while the short-read sits there untested.

SQLITE_CORRUPT rc=11. This was the one that ate time. An early leaf was a cell SQLite would not parse. `SELECT` returned `rc=11` malformed / corrupt. That is btree doing its job on a bad layout: wrong payload varint, overflow pointer into nowhere, `nSize` past `usableSize`, page flag that `decodeFlags` rejects. It is not the leftover. The leftover needs a cell btree believes.

The cell that works is a table-leaf `0x0D` whose payload claims 2000 bytes of blob. Table-leaf `maxLocal` is `usableSize-35`. On a 4096-byte database page, 2000 is fully local. No overflow page number. The cell starts at offset 10 in the 512-byte frame. The blob starts a few bytes later. 2000 bytes of local payload therefore run from inside the WAL frame, across the 512-byte boundary, into the 3584-byte pager tail. `accessPayload` copies the whole thing. `sqlite3_column_blob` returns 2000 bytes. Bytes past the frame are recycled pcache. Until the leaf claimed that 2000-byte local payload across the 512-byte frame, the experiment was a corrupt-page generator. After it, the last line was SUCCESS.

A 4000-byte blob is still under `maxLeaf` on 4096 and would also be local. I used 2000 so the copy was long enough to be obviously past 512 and short enough that `pc+nSize` stayed inside `usableSize`. Winter's public note used 3000. Same idea. The wrong turn was not the number. The wrong turn was a cell btree would not treat as local payload on the *database* page size.

Putting the oversized cell in the `.db` instead of the `-wal`. Then you are writing a 4096-byte page, the pager reads 4096 bytes, the tail is on disk, and there is no short-read. The mismatch has to be WAL header 512, pager 4096, frame body 512, claim 2000.

Calling this remote because CVSS AV:N. Untrusted files travel. The reach is still local: the victim process opens the files. There is no SQLite listener in this lab. I am not going to pretend `sqlite3_open` is a network daemon.

12 What an attacker can do

Ship a WAL-mode `.db` plus a sibling `-wal` whose header page size is a smaller valid power of two than the database. Recovery accepts it. `sqlite3WalReadFrame` copies only the WAL frame size into a pager buffer sized from the `.db`. A table-leaf cell that claims a payload larger than that frame is satisfied from the recycled pcache tail.

The victim opens the files. Desktop app, backup restore, "open this database," a language binding, a browser origin that still hands SQLite a blob. No SQL injection required after that. SQL on that leaf returns leftover pager and heap bytes from prior pages in the same process: other tables, previous queries, whatever the cache still held.

Not RCE. Not a crash. Not a file write. Info leak of uninitialized page tail through SQL. Confidentiality of process memory that SQLite already touched, C:H I:N A:N, local files, UI none. The inverse mismatch is `SQLITE_CORRUPT`. Single matching page sizes are not this leftover. Default `load_extension` stays off.

I am not going to print a `hex(substr(b,500,80))` you can paste at someone else's laptop. The missing compare in `walIndexRecover` and the missing `memset` in `sqlite3WalReadFrame` are the useful part. If you own the process, run the proof of concept against loopback.

13 The fix

Compare WAL header page size to the database page size during recovery. When they disagree, refuse the WAL. And when `nOut>sz`, fully initialize the pager page:

C Copy

```c
if( nOut>sz ){
  memset(pOut+sz, 0, nOut-sz);
  nOut = sz;
}
```

That hunk is on trunk after this GA. It is not in 3.53.4. Until a release ships it, a crafted 512-byte WAL next to a 4096-byte database still turns `SELECT` into a read of whatever pcache had.

Do not take "the database looks fine" as a reason to skip the sibling `-wal`. Recovery will read it. The page size in the WAL header is not a comment. On this pin it is the

size of the read.

14 References

- Proof of concept: abraxas/sqlite-wal-pagesize · sqlite-wal-pagesize-Abraxas-Labs.py
- Lab: `lab/` · Dockerfile · docker-compose.yml · run.sh · poc.py
- @abraxas_null · github.com/abraxas · abraxaslabs.tech · abraxas.null@proton.me
- CWE-908 · CWE-200 · 7.5 High
- sqlite.org/src tag version-3.53.4 (`b09c88c14082339b66c7b7158d609a771e64ca69`)
- `src/wal.c` `sqlite3WalReadFrame`, `walIndexRecover`
- `src/pager.c` `readDbPage`
- Forum 2026-08-17T15:54:41Z (Winter)

Independent security research by abraxas.

Analysis of 0-days, n-days, malware, and the leftovers that still matter.

GitHub **@abraxas_null** **abraxas.null@proton.me** **abraxaslabs.tech**

Independent research. Private analysis by arrangement. No ads. abraxas.null@proton.me · For defensive awareness and authorized testing only.

© 2026 Abraxas Labs