CONTRACT WOLF

Security Assessment

UNIFIEDX DAPP

Verified on 08/16/2026

SUMMARY

Project	CHAIN	METHODOLOGY
UNIFIEDX	Binance Smart Chain	Manual & Automatic Analysis

FILES	DELIVERY	TYPE
Repository	07/26/2026	Dapp Audit

16	1	4	5	5	1	16
Total Findings	Critical	Major	Medium	Minor	Informational	Resolved

🟥 1 Critical	An exposure that can affect the dapp's functions in several events that can risk and disrupt the code
🟧 4 Major	An opening & exposure to manipulate the code in an unwanted manner
🟨 5 Medium	An opening that could affect the outcome in executing the code in a specific situation
⬛ 5 Minor	An opening but doesn't have an impact on the functionality of the code
🟦 1 Informational	An opening that consists information but will not risk or affect the code
🟩 16 Resolved	ContractWolf's findings has been acknowledged & resolved by the project

STATUS ✔️**AUDIT PASSED**

TABLE OF CONTENTS | UNIFIEDX

DISCLAIMER | UNIFIEDX

ContractWolf audits and reports should not be considered as a form of project's "Advertisement" and does not cover any interaction and assessment from "Project Code" to "External Code"

ContractWolf does not provide any warranty on its released report and should not be used as a decision to invest into audited projects.

ContractWolf provides a transparent report to all its "Clients" and to its "Clients Participants" and will not claim any guarantee of bug-free code within its **DAPP**.

ContractWolf's presence is to analyze, audit and assess the Client's Dapp to find any underlying risk and to eliminate any logic and flow errors within its code.

Each company or project should be liable to its security flaws and functionalities.

SCOPE OF WORK | UNIFIEDX

UNIFIEDX's team has agreed and provided us with the files that need to be tested. The scope of audit is the main dapp.

The goal of this engagement is to identify if there is a possibility of security flaws in the implementation of dapp and its systems.

ContractWolf will be focusing on dapp issues and functionalities along with the project claims from smart contract to their website, whitepaper, repository which has been provided by **UNIFIEDX**.

AUDITING APPROACH | UNIFIEDX

Every line of code along with its functionalities will undergo manual review to check for security issues, quality of logic and dapp scope of inheritance. The manual review will be done by our team that will document any issues that they discovered.

METHODOLOGY

The auditing process follows a routine series of steps :

1. Code review that includes the following :
   - Review of the specifications, sources and instructions provided to ContractWolf to make sure we understand the size, scope and functionality of the DAPP.
   - Manual review of code. Our team will have a process of reading the code line-by-line with the intention of identifying potential vulnerabilities, underlying and hidden security flaws.

2. Testing and automated analysis that includes :
   - Testing the DAPP's function with common test cases and scenarios to ensure that it returns the expected results.

3. Best practices and ethical review. The team will review the dapp with the aim to improve efficiency, effectiveness, clarifications, maintainability, security and control within the dapp.

4. Recommendations to help the project take steps to eliminate or minimize threats and secure the dapp.

TOKEN DETAILS | UNIFIEDX

the unified infrastructure layer for digital finance, connecting trading, cross-chain liquidity, on-chain intelligence, and Web3 creation in one seamless ecosystem.

Token Name	Symbol	Decimal	Total Supply	Chain
0	0	0	-	Binance Smart Chain

SOURCE

Source	*Sent Via local-files*

FINDINGS | UNIFIEDX

16	1	4	5	5	1	16
Total Findings	Critical	Major	Medium	Minor	Informational	Resolved

This report has been prepared to state the issues and vulnerabilities for UNIFIEDX Dapp through this audit. The goal of this report findings is to identify specifically and fix any underlying issues and errors

ID	Title	Severity	Status
DCW-010	Address Draining	Critical	Resolved
DCW-016	Sensitive Data Exposure	Major	Resolved
DCW-011	Insecure API Usage	Major	Resolved
DCW-017	Request Limit	Major	Resolved
DCW-003	Missing HTTP Headers	Major	Resolved
DCW-018	Overflow or Precision Loss	Medium	Resolved
DCW-019	Unintended Behavior	Medium	Resolved
DCW-008	Gas Griefing	Medium	Resolved

DCW-012	Error Handling	Medium	Resolved
DCW-014	Lack of Input Validation	Medium	Resolved
DCW-013	Memory Leak	Minor	Resolved
DCW-019	Unintended Behavior	Minor	Resolved
DCW-015	Potential Backdoor	Minor	Resolved
DCW-015	Potential Backdoor	Minor	Resolved
DCW-011	Insecure API Usage	Minor	Resolved
DCW-007	Framework Version	Informational	Resolved

PENETRATION ATTACKS | UNIFIEDX

Dapp Weakness Classification and Test Cases

ID	Description	Status
DCW-001	Malware Scan	Passed
DCW-002	Phishing	Passed
DCW-003	Missing HTTP Headers	Passed
DCW-004	Valid SSL Certificate	Passed
DCW-005	Firewalls(Drop & Deny)	Passed
DCW-006	Potential SQL Injection	Passed
DCW-007	Framework Version	Passed
DCW-008	Gas Griefing	Passed
DCW-009	Address Approval	Passed
DCW-010	Address Draining	Passed
DCW-011	Insecure API Usage	Passed
DCW-012	Error Handling	Passed
DCW-013	Memory Leak	Passed
DCW-014	Lack of Input Validation	Passed
DCW-015	Potential Backdoor	Passed
DCW-016	Sensitive Data Exposure	Passed
DCW-017	Request Limit	Passed
DCW-018	Overflow or Precision Loss	Passed
DCW-019	Unintended Behavior	Passed

FIXES & RECOMMENDATION

DCW-010 | Address Draining

app/components/otc-workspace.tsx

The OTC screen decides which contract is the UNIFIEDX escrow by reading the otc value out of the page URL, and that value overrides the escrow the user actually deployed. The only check applied is isAddress(), which proves the string is a well-formed address but not that it is the audited escrow. That unverified address is then used as the ERC-20 approval spender, as the destination of createOrder({value: sell}), and - on the taker side - the approval amount itself is read from the unknown contract's own previewFill() return value.

Impact is critical. Anyone can publish a link such as /exchange?otc=56.0xATTACKER and the dapp will render the attacker's fake orders, ask the victim to approve the attacker's contract, and send real funds to it. A hostile previewFill() that returns the maximum uint256 produces an unlimited allowance, so the attacker can drain the approved token afterwards at will. Because the whole flow runs on the genuine UNIFIEDX domain, certificate and interface, every warning sign a user is trained to look for still looks correct.

```
// app/components/otc-workspace.tsx
function deploymentFromUrl(){
  const value=new URLSearchParams(window.location.search).get("otc");
  if(!value)return null;
  const [chain,contract]=value.split(".");
  // isAddress() proves the string is an address, not that it is OUR escrow
  if(Number(chain)!==OTC_NETWORK.numericId||!isAddress(contract))return null;
  return {chainId:Number(chain),contract,deployedAt:0,deployedBy:"shared"};
}

// L83 - the URL value wins over the user's own deployment
setDeployment(deploymentFromUrl()??deploymentFromStorage());

// L171 - maker approves and funds the injected address
if(allowance<sell){ await (await token.approve(target.contract,sell)).wait() }
await escrow.createOrder(...,isNativeAsset(sellAsset.address)?{value:sell}:{});

// L197 - taker reads the approval amount FROM the untrusted contract
const payment=BigInt(await escrow.previewFill(order.id,amount));
await (await paymentToken.approve(deployment.contract,payment)).wait();
```

Recommendation:

Pin the escrow to a deployment allowlist compiled into the bundle, and verify the deployed runtime bytecode against the hash of the audited artifact before any approval or transfer is offered to the wallet. The share link must carry an order id only, never a contract address. If per-user escrow deployment is a deliberate feature it cannot coexist with link sharing - in that case drop the otc parameter and require the address to be pasted into a confirmation dialog that shows the bytecode comparison result first.

```
// One audited deployment per chain, compiled into the bundle.
const ESCROW_ALLOWLIST: Record<number,string> = {
  56: "0xYOUR_AUDITED_DEPLOYMENT",
};
const ESCROW_RUNTIME_HASH = "0x<keccak256 of audited runtime bytecode>";

async function resolveEscrow(candidate: string, chainId: number,
                             provider: JsonRpcProvider): Promise<string> {
  const pinned = ESCROW_ALLOWLIST[chainId];
  if (!pinned) throw new Error("UNIFIEDX OTC is not available on this network.");

  // Only the pinned address may ever be a spender or a value recipient.
  if (candidate.toLowerCase() !== pinned.toLowerCase())
    throw new Error("This link points at a contract UNIFIEDX does not " +
                    "recognise. Do not approve or send funds to it.");

  // Defence in depth: confirm the chain hosts the audited code.
  const code = await provider.getCode(pinned);
  if (code === "0x" || keccak256(code) !== ESCROW_RUNTIME_HASH)
    throw new Error("Escrow bytecode does not match the audited release.");
  return pinned;
}

// The URL may now only carry an order id, never a contract address.
function orderIdFromUrl(): bigint | null {
  const raw = new URLSearchParams(window.location.search).get("otc");
  return raw && /^\d{1,20}$/.test(raw) ? BigInt(raw) : null;
}
```

DCW-016 | Sensitive Data Exposure

worker/index.ts

GET /api/operations?account=0x... returns every tracked cross-chain operation for the supplied wallet - counterparty address, amounts, both chains, token symbols, transaction hashes and timestamps - with no session or signature check. The account value is taken straight from the query string, so it says whose data to return rather than proving who is asking. The helper compounds this by joining its WHERE clauses with OR instead of AND, so the account filter is not an authorisation boundary even in principle.

Because wallet addresses are public, this lets anyone walk a list of addresses and rebuild who bridged what, when, and to whom. For a platform that markets AML and compliance tooling this is a data-protection exposure as much as a security one.

```
// worker/index.ts
async function selectOperations(env:Env,ids:string[],account:string){
  const clauses:string[]=[];const bindings:string[]=[];
  if(ids.length){clauses.push(`id IN (${ids.map(()=>"?").join(",")})`);
                 bindings.push(...ids)}
  if(account){clauses.push("account = ?");bindings.push(account)}
  if(!clauses.length)return [] as OperationRow[];
  // (id IN (...)) OR (account = ?) -- either clause alone satisfies the query
  const rows=await env.DB.prepare(`SELECT * FROM tracked_operations
    WHERE ${clauses.map(clause=>`(${clause})`).join(" OR ")}
    ORDER BY created_at DESC LIMIT 50`).bind(...bindings).all<OperationRow>();
  return rows.results??[];
}

// L64 - account arrives from the query string, unauthenticated
const account=clean(url.searchParams.get("account"),180);
const wallet=account&&fieldValue.test(account)?account:"";
let rows=await selectOperations(env,ids,wallet);
```

Recommendation:

Bind every read to a proven wallet. Require a short-lived SIWE-style signature, recover the signing address server-side, and scope the query to it. Make ownership a mandatory AND predicate so an id filter narrows the result set instead of widening it, and store account lower-cased at write time so the comparison stays stable.

```
// 1. Prove wallet control before returning anything.
const proof = request.headers.get("x-unifiedx-wallet-proof") ?? "";
const owner = await verifyWalletSession(env, proof); // recovers + checks nonce
if (!owner) return operationJson({ error: "Wallet signature required." }, 401);

// 2. Ownership is mandatory, never one alternative among several.
async function selectOperations(env: Env, ids: string[], owner: string) {
  const clauses = ["account = ?"];
  const bindings: string[] = [owner.toLowerCase()];
  if (ids.length) {
    clauses.push(`id IN (${ids.map(() => "?").join(",")})`);
    bindings.push(...ids);
  }
  const rows = await env.DB.prepare(
    `SELECT * FROM tracked_operations
       WHERE ${clauses.join(" AND ")}
       ORDER BY created_at DESC LIMIT 50`,
  ).bind(...bindings).all<OperationRow>();
  return rows.results ?? [];
}

// 3. On insert, trust the recovered owner - not the request body.
const account = owner.toLowerCase();
```

DCW-011 | Insecure API Usage

worker/index.ts

The deduplication key for a tracked operation is built entirely from public blockchain data (tx:<chainId>:<txHash>), and a matching key triggers an unconditional UPDATE with no check that the caller owns the record. A transaction hash is visible to everyone the moment it is mined.

An attacker who watches a bridge transaction can POST the same hash and rewrite that record's account, recipient, amounts and chain names - moving the victim's operation into their own history. The update also regenerates worker_token, so every in-flight reconciliation job for that operation fails its token check and stops, silently freezing the victim's transfer status.

```
// worker/index.ts
const providerKey=taskId?`task:${taskId}`:`tx:${sourceChainKey}:${txHash}`;
const existing=await env.DB.prepare(
  "SELECT id FROM tracked_operations WHERE provider_key=? LIMIT 1"
).bind(providerKey).first<{id:string}>();

const id=existing?.id??crypto.randomUUID();
const workerToken=crypto.randomUUID();   // rotated on EVERY call

if(existing){
  // no ownership predicate -- anyone who knows the tx hash can rewrite this
  await env.DB.prepare(`UPDATE tracked_operations SET route_id=?,worker_token=?,
    account=?,recipient=?,tx_hash=?,task_id=?,bridge=?,kind=?,...
    WHERE id=?`).bind(...).run();
}

// L130 - in-flight jobs die once worker_token changes
if(!row||!token||token!==row.worker_token)
  return operationJson({error:"Tracking job not found."},404);
```

Recommendation:

Make ownership part of the write. Recover the caller's wallet, refuse the update when the existing row belongs to someone else, and preserve the existing worker_token so in-flight reconciliation jobs survive. The account column should come from the recovered owner rather than from the request body.

```
const owner = await verifyWalletSession(env, request);
if (!owner) return operationJson({ error: "Wallet signature required." }, 401);

const existing = await env.DB.prepare(
  `SELECT id, account, worker_token FROM tracked_operations
     WHERE provider_key=? LIMIT 1`,
).bind(providerKey).first<{
  id: string; account: string; worker_token: string;
}>();

if (existing && existing.account.toLowerCase() !== owner.toLowerCase()) {
  // Someone else already tracks this hash. Do not touch their row.
  return operationJson({ error: "This transaction is already tracked." }, 409);
}

// Keep the existing token so queued reconciliation jobs stay valid.
const workerToken = existing?.worker_token ?? crypto.randomUUID();
const account = owner.toLowerCase();   // never from the request body
```

DCW-017 | Request Limit

lib/upstream-rate-limit.ts, app/api/intelligence/screen/route.ts, app/api/tokens/route.ts

The shared limiter keys only on the provider name, so every user draws from the same budget - one caller can exhaust the LI.FI quota and deny quoting, bridging and status tracking to everyone. The Dexlab policy defaults to one request per second, which a single client saturates continuously. The only per-client limiter lives in a module-level Map inside a Worker isolate, so it resets whenever an isolate is recycled, and it falls back to the client-supplied x-forwarded-for header, which is trivially rotated.

Several routes also amplify: one /api/tokens search fans out to roughly seventeen upstream calls, and /api/transaction-status accepts twenty-five hashes and issues fifty RPC calls. Combined, this makes a cheap and effective denial-of-service against the whole platform.

```
// lib/upstream-rate-limit.ts (keyed on provider, not on caller)
export async function reserveRateLimit(db:D1Database,policy:RatePolicy){
  const row=await db.prepare(`
    INSERT INTO upstream_rate_limits(provider,...) VALUES(?,?,1,0,?)
    ON CONFLICT(provider) DO UPDATE SET ...
  `).bind(policy.provider,now,now,threshold,threshold).first<LimitRow>();
  if(row.request_count>policy.limit){ /* everyone shares this budget */ }
}

// app/api/intelligence/screen/route.ts  (per-isolate, spoofable key)
const visitorChecks = new Map<string, number>();
function rateLimit(request: Request) {
  const visitor = request.headers.get("cf-connecting-ip")
    ?? request.headers.get("x-forwarded-for")?.split(",")[0]?.trim()
    ?? "local";                          // client-controlled fallback
  if (now - (visitorChecks.get(visitor) ?? 0) < 5_000) return false;
}
```

Recommendation:

Add a per-identity limiter in front of the shared upstream budget, keyed on cf-connecting-ip - which Cloudflare sets and the client cannot forge - and persist it in D1 or a Durable Object rather than isolate memory. Also cap the fan-out itself: reduce the token search to three resolveToken calls, cache resolved metadata by chain:address, and add Cloudflare edge rate-limiting rules on /api/* as an immediate stopgap.

```
export async function reserveClientLimit(
  db: D1Database, request: Request,
  route: string, limit: number, windowSeconds: number,
) {
  // cf-connecting-ip is set by the edge; never trust x-forwarded-for here.
  const ip = request.headers.get("cf-connecting-ip");
  if (!ip) throw new UpstreamRateLimitError(route, windowSeconds);

  const key = `${route}:${ip}`;
  const now = Date.now();
  const threshold = now - windowSeconds * 1000;

  const row = await db.prepare(`
    INSERT INTO client_rate_limits(key,window_started_at,request_count,updated_at)
    VALUES(?,?,1,?)
    ON CONFLICT(key) DO UPDATE SET
      window_started_at=CASE WHEN client_rate_limits.window_started_at<=?
        THEN excluded.window_started_at ELSE client_rate_limits.window_started_at END,
      request_count=CASE WHEN client_rate_limits.window_started_at<=?
        THEN 1 ELSE client_rate_limits.request_count+1 END,
      updated_at=excluded.updated_at
    RETURNING window_started_at,request_count
  `).bind(key, now, now, threshold, threshold).first<{
    window_started_at: number; request_count: number;
  }>();

  if (!row || row.request_count > limit) {
    const retryAfter = Math.max(1, Math.ceil(
      (row!.window_started_at + windowSeconds * 1000 - now) / 1000));
    throw new UpstreamRateLimitError(route, retryAfter);
  }
}

// Apply before any upstream call on every fan-out route:
await reserveClientLimit(env.DB, request, "tokens-search", 10, 60);
```

DCW-003 | Missing HTTP Headers

next.config.ts, worker/index.ts

nextConfig is an empty object and the Worker returns responses without adding headers, so the application ships with no Content-Security-Policy, no frame-ancestors or X-Frame-Options, no HSTS, no X-Content-Type-Options and no Permissions-Policy.

For a wallet-connected dapp the missing frame-ancestors matters most: the whole exchange can be framed by a hostile page with the approval and swap controls overlaid, so a user thinks they are clicking something harmless while actually driving a real wallet prompt. The absent CSP means any injected script - from a compromised dependency, or from the third-party token metadata rendered across the token picker - runs with full access to the page.

```
// next.config.ts - the entire file
import type { NextConfig } from "next";

const nextConfig: NextConfig = {
  /* config options here */      // no headers() entry at all
};

export default nextConfig;

// worker/index.ts - responses are returned unmodified
const worker = {
  async fetch(request: Request, env: Env, ctx: ExecutionContext) {
    // ... route dispatch ...
    return handler.fetch(request, env, ctx);   // no security headers added
  },
};
```

Recommendation:

Set the headers in the Worker so they cover static assets, API responses and rendered pages uniformly. The app ships an inline bootstrap script in app/layout.tsx, so give it a hash or nonce rather than falling back to 'unsafe-inline'. Roll the policy out in report-only mode first, because the wallet SDKs inject their own frames and workers and connect-src will need widening once observed in practice.

```
const SECURITY_HEADERS: Record<string, string> = {
  "content-security-policy": [
    "default-src 'self'",
    "base-uri 'none'",
    "object-src 'none'",
    "frame-ancestors 'none'",
    "form-action 'self'",
    "script-src 'self' 'sha256-<hash of platformThemeScript>'",
    "style-src 'self' 'unsafe-inline'",
    "img-src 'self' data: https://raw.githubusercontent.com",
    "connect-src 'self' https://li.quest https://*.kyberswap.com " +
      "https://api.dexscreener.com https://*.dexlab.space https://api.trmlabs.com",
    "upgrade-insecure-requests",
  ].join("; "),
  "strict-transport-security": "max-age=31536000; includeSubDomains; preload",
  "x-content-type-options": "nosniff",
  "referrer-policy": "strict-origin-when-cross-origin",
  "permissions-policy": "camera=(), microphone=(), geolocation=(), payment=()",
  "cross-origin-opener-policy": "same-origin",
};

const worker = {
  async fetch(request: Request, env: Env, ctx: ExecutionContext) {
    const response = await route(request, env, ctx);   // existing dispatch
    const headers = new Headers(response.headers);
    for (const [key, value] of Object.entries(SECURITY_HEADERS))
      headers.set(key, value);
    return new Response(response.body, {
      status: response.status,
      statusText: response.statusText,
      headers,
    });
  },
};
```

DCW-018 | Overflow or Precision Loss

app/components/otc-workspace.tsx

displayAmount() is a presentation helper - it runs the value through a JavaScript float and formats it with toLocaleString() - yet its output is fed straight back into parseUnits() as if it were a machine value. Grouping separators are on by default, so an order with 1,500 tokens remaining produces "1,500.5", and parseUnits throws on the comma.

This is a live functional break, not only a precision concern: because the same helper supplies the fill input's default value, no OTC order of 1,000 units or more can be filled at all. Below that threshold the six-decimal rounding can push the parsed amount above remainingSell, so previewFill reverts and "fill the whole order" fails for roughly half of all non-round balances. The float conversion also loses precision beyond 2^53.

```
// app/components/otc-workspace.tsx
function displayAmount(value:bigint,decimals=18){
  const result=Number(formatUnits(value,decimals));   // float precision loss
  return Number.isFinite(result)
    ? result.toLocaleString("en-US",{maximumFractionDigits:6})  // adds commas
    : "—";
}

// L196 - the human-readable string is parsed back into a uint256
const amount=parseUnits(
  fillAmounts[String(order.id)] || displayAmount(order.remainingSell,sellDecimals),
  sellDecimals,
);

// L266 - the input's default value comes from the same helper,
// so this fires even when the user types nothing
value={fillAmounts[String(order.id)]??displayAmount(order.remainingSell,sellDecimals)}
```

Recommendation:

Keep exact values in bigint and never round-trip an amount through the display layer. Format for humans only at render time, and use remainingSell directly when the user has not typed anything. Show the formatted maximum as the input's placeholder instead of its value.

```
// Display only - never parsed back.
function displayAmount(value: bigint, decimals = 18) {
  const whole = formatUnits(value, decimals);
  const [int, frac = ""] = whole.split(".");
  const grouped = BigInt(int).toLocaleString("en-US");
  const shown = frac.replace(/0+$/, "").slice(0, 6);
  return shown ? `${grouped}.${shown}` : grouped;
}

// Exact - the value actually sent to the contract.
const rawFillAmount = (order: OtcOrder, decimals: number): bigint => {
  const typed = fillAmounts[String(order.id)]?.trim();
  if (!typed) return order.remainingSell;        // exact max, no formatting
  const parsed = parseUnits(typed.replace(/,/g, ""), decimals);
  if (parsed <= 0n) throw new Error("Enter an amount greater than zero.");
  if (parsed > order.remainingSell)
    throw new Error(`Only ${displayAmount(order.remainingSell, decimals)} `
                  + `remains on this order.`);
  return parsed;
};

const amount = rawFillAmount(order, sellDecimals);
```

DCW-019 | Unintended Behavior

contracts/UnifiedXOtcEscrow.sol

previewFill rounds the taker's payment up and fillOrder then subtracts exactly what was paid, so the running total stays correct - a fully consumed order always pays the whole buyAmount. The gap is the terminal state. Whenever remainingSell is larger than remainingBuy in raw units, a taker can pick a fill size that forces the ceiling to equal the entire remainingBuy, leaving the order Open with remainingBuy at zero and sell tokens still escrowed. From there previewFill returns zero and fillOrder never rejects a zero payment, so the residue is claimable for free.

The maker is not underpaid, and the leak is bounded at roughly one smallest-unit of the buy token, which is dust on an 18-decimal pair. It matters as a broken invariant and becomes material only when the buy token has few decimals and high unit value.

```
// contracts/UnifiedXOtcEscrow.sol
function previewFill(uint256 orderId, uint256 sellFillAmount)
    public view returns (uint256 buyPayment) {
    Order storage order = orders[orderId];
    if (order.state != OrderState.Open || sellFillAmount == 0
        || sellFillAmount > order.remainingSell) revert InvalidOrder();
    buyPayment = Math.mulDiv(sellFillAmount, order.remainingBuy,
                             order.remainingSell, Math.Rounding.Ceil);
}

// L142 - nothing rejects buyPayment == 0
buyPayment = previewFill(orderId, sellFillAmount);
uint256 nextSell = order.remainingSell - sellFillAmount;
uint256 nextBuy  = order.remainingBuy  - buyPayment;   // can reach 0 early
order.remainingSell = nextSell;
order.remainingBuy  = nextBuy;
if (nextSell == 0) order.state = OrderState.Filled;

// remainingSell 1000, remainingBuy 100:
// fill 999 -> ceil(999*100/1000) = 100 = the entire remainingBuy
// order stays Open with 1 sell unit priced at nothing
```

Recommendation:

Reject zero-payment fills outright and close off the state where one leg is empty while the other is still funded, so no unpriced residue can exist. Add a regression test asserting that a 999-of-1000 fill against a 100-unit buy amount reverts.

```
error ZeroPayment();
error DustRemainder();

function fillOrder(uint256 orderId, uint256 sellFillAmount, uint256 maxBuyPayment)
    external payable whenNotPaused nonReentrant returns (uint256 buyPayment)
{
    Order storage order = orders[orderId];
    // ... existing state / expiry / taker checks ...

    buyPayment = previewFill(orderId, sellFillAmount);
    if (buyPayment == 0) revert ZeroPayment();          // never settle for free
    if (buyPayment > maxBuyPayment) revert ExcessivePayment();

    uint256 nextSell = order.remainingSell - sellFillAmount;
    uint256 nextBuy  = order.remainingBuy  - buyPayment;

    // A priced order cannot have one leg empty and the other funded.
    if (nextBuy == 0 && nextSell != 0) revert DustRemainder();

    order.remainingSell = nextSell;
    order.remainingBuy  = nextBuy;
    if (nextSell == 0) order.state = OrderState.Filled;
    // ... existing transfers ...
}
```

DCW-008 | Gas Griefing

contracts/UnifiedXOtcEscrow.sol

Every native transfer is a push that forwards all remaining gas, and any failure reverts the whole call. A maker contract whose receive() reverts - or simply burns the gas it is given - makes fillOrder revert for every taker, so the order sits unfillable in the book while takers pay gas to discover it. Because the revert can be made conditional, a maker can advertise an attractive price and then deny specific takers.

The more serious case is cancelOrder and reclaimExpired, which use the same push. If the maker cannot receive native value the escrowed funds cannot leave by any path, and the contract deliberately has no admin rescue, so the funds are locked permanently.

```
// contracts/UnifiedXOtcEscrow.sol
function _sendNative(address recipient, uint256 amount) private {
    // all remaining gas forwarded; any failure kills the whole transaction
    (bool success,) = payable(recipient).call{value: amount}("");
    if (!success) revert NativeTransferFailed();
}

// L149 - a hostile maker can make every fill revert
if (order.buyToken == address(0)) {
    if (msg.value != buyPayment) revert InvalidNativeValue();
    _sendNative(order.maker, buyPayment);
}

// L179 / L191 - and the same push blocks the only two exit paths
_sendAsset(order.sellToken, order.maker, refund);   // cancelOrder
_sendAsset(order.sellToken, order.maker, refund);   // reclaimExpired
```

Recommendation:

Escrow the credit and let recipients pull it. Settlement stays atomic, but an uncooperative recipient becomes their own problem rather than everyone's. Keep the withdrawal function outside whenNotPaused so a pause can never trap funds.

```
mapping(address account => uint256 amount) public nativeCredits;

event NativeCredited(address indexed account, uint256 amount);
event NativeWithdrawn(address indexed account, uint256 amount);

function _sendNative(address recipient, uint256 amount) private {
    if (amount == 0) return;
    // Bounded gas: enough for an EOA or a simple receive(), not for reentry.
    (bool success,) = payable(recipient).call{value: amount, gas: 8_000}("");
    if (!success) {
        // Never revert the settlement - credit and let them pull.
        nativeCredits[recipient] += amount;
        emit NativeCredited(recipient, amount);
    }
}

function withdrawNative() external nonReentrant {
    uint256 amount = nativeCredits[msg.sender];
    if (amount == 0) revert InvalidAmount();
    nativeCredits[msg.sender] = 0;            // effects before interaction
    (bool success,) = payable(msg.sender).call{value: amount}("");
    if (!success) revert NativeTransferFailed();
    emit NativeWithdrawn(msg.sender, amount);
}
```

DCW-012 | Error Handling

lib/lifi-api-handlers.ts, lib/kyber-api-handler.ts, lib/dexlab-api-handler.ts, worker/index.ts

Seven handlers share the pattern cause instanceof Error ? cause.message : "...", returning whatever the runtime produced directly to the client. Worker fetch failures, D1 errors and URL-parse errors carry internal hostnames, binding names and query fragments.

In worker/index.ts the raw message is written into status_detail, which publicOperation then serves to any caller - so an internal error becomes durably stored and readable, and combined with the unauthenticated operations endpoint it is readable by anyone. This hands an attacker a free map of the backend while giving users error text they cannot act on.

```
// lib/lifi-api-handlers.ts -
}catch(cause){
  if(cause instanceof UpstreamRateLimitError)return rateLimitResponse(cause);
  // raw runtime message goes straight to the client
  return json({error:cause instanceof Error?cause.message
                    :"LI.FI proxy request failed."},502)
}

// worker/index.ts  the message is PERSISTED and re-served
await env.DB.prepare(
  `UPDATE tracked_operations SET status_detail=?,next_check_at=?,
     last_checked_at=?,updated_at=? WHERE id=?`
).bind(cause instanceof Error?cause.message
       :"Tracking provider is temporarily unavailable",next,...,id).run();
```

Recommendation:

Log the detail server-side and return a stable, curated message plus a short reference id the user can quote to support. Apply the same treatment where the message is persisted, so a fixed string is stored rather than the runtime text.

```
function failure(cause: unknown, fallback: string, status = 502) {
  const reference = crypto.randomUUID().slice(0, 8);
  // Full detail goes to the Workers log, never to the response body.
  console.error(`[${reference}]`, cause);
  if (cause instanceof UpstreamRateLimitError) return rateLimitResponse(cause);
  return json({ error: fallback, reference }, status);
}

// Every handler now surfaces only the curated message:
} catch (cause) {
  return failure(cause, "LI.FI is temporarily unavailable. Please try again.");
}

// worker/index.ts - persist a fixed string, not cause.message
await env.DB.prepare(
  `UPDATE tracked_operations SET status_detail=?,next_check_at=?,
     last_checked_at=?,updated_at=? WHERE id=?`,
).bind("Tracking provider is temporarily unavailable", next,
       Date.now(), Date.now(), id).run();
```

DCW-014 | Lack of Input Validation

app/components/otc-workspace.tsx

Custom tokens are rehydrated from localStorage with only the address checked - decimals, symbol and name are accepted exactly as stored. When an order references a token that is not in the local catalogue, the code silently falls back to 18 decimals.

decimals is the exponent in every parseUnits call in this component, so a wrong value is a wrong amount by orders of magnitude: an order denominated in a 6-decimal token parsed as 18 decimals is off by a factor of 10^12. Any script with page access, or simply a stale entry left behind after a catalogue change, silently changes what the user signs.

```
// app/components/otc-workspace.tsx
useEffect(()=>{
  try{
    const stored=JSON.parse(localStorage.getItem(CUSTOM_TOKEN_KEY)??"[]") as Token[];
    // only the address is validated; decimals/symbol/name are trusted as stored
    setCustomAssets(stored.filter(token=>/^0x[a-fA-F0-9]{40}$/.test(token.address)));
  }catch{setCustomAssets([])}
},[]);

// L195 - an unknown token silently becomes 18 decimals
const sellDecimals=assetFor(order.sellToken)?.decimals??18;
const amount=parseUnits(..., sellDecimals);   // wrong exponent = wrong amount
```

Recommendation:

Validate the whole token shape on read, and resolve unknown decimals from the token contract instead of guessing. A token that cannot report standard ERC-20 decimals should be refused rather than defaulted.

```
function isValidToken(value: unknown): value is Token {
  if (!value || typeof value !== "object") return false;
  const t = value as Partial<Token>;
  return typeof t.address === "string" && /^0x[a-fA-F0-9]{40}$/.test(t.address)
    && typeof t.symbol === "string" && /^[\w.$-]{1,16}$/.test(t.symbol)
    && typeof t.decimals === "number" && Number.isInteger(t.decimals)
    && t.decimals >= 0 && t.decimals <= 36;
}

const stored = JSON.parse(localStorage.getItem(CUSTOM_TOKEN_KEY) ?? "[]");
setCustomAssets(Array.isArray(stored) ? stored.filter(isValidToken) : []);

// Never guess an exponent - read it from the token contract.
async function decimalsFor(address: string, provider: JsonRpcProvider) {
  const known = assetFor(address)?.decimals;
  if (known !== undefined) return known;
  if (isNativeAsset(address)) return 18;
  const raw = await new Contract(
    address, ["function decimals() view returns (uint8)"], provider,
  ).decimals();
  const value = Number(raw);
  if (!Number.isInteger(value) || value < 0 || value > 36)
    throw new Error("This token does not report standard ERC-20 decimals.");
  return value;
}
```

DCW-013 | Memory Leak

app/api/intelligence/screen/route.ts, worker/index.ts, app/api/market/route.ts

visitorChecks only prunes once it already holds more than 500 entries, and even then removes only entries older than sixty seconds, so traffic from many addresses grows it without bound in a warm isolate. In the Worker, Promise.race([reconciliation, sleep(3500)]) leaves the losing timer running with its closure retained, and each queued operation holds a timer of up to twenty seconds. The market route's module-level caches are never evicted.

Impact is degradation rather than compromise - isolates accumulate memory and are recycled more often - but the same recycling is what makes the in-memory rate limiter ineffective.

```
// app/api/intelligence/screen/route.ts
const visitorChecks = new Map<string, number>();

// L24 - only prunes AFTER exceeding 500, and only entries older than 60s
if (visitorChecks.size > 500) {
  for (const [key, timestamp] of visitorChecks) {
    if (now - timestamp > 60_000) visitorChecks.delete(key);
  }
}

// worker/index.ts  the losing timer is never cleared
const sleep=(ms:number)=>new Promise(resolve=>setTimeout(resolve,ms));
await Promise.race([reconciliation,sleep(3500)]);

// app/api/market/route.ts - never evicted
const responseCache = new Map<string,{expires:number;staleUntil:number;payload:unknown}>();
const pendingRequests = new Map<string,Promise<unknown>>();
```

Recommendation:

Prune on every write and cap the map hard, and make the race timeout cancellable so the loser is collected. For operation reconciliation prefer a Durable Object alarm over chained setTimeout - it survives isolate eviction, which the current design does not.

```
// 1. Prune on every write, and cap hard.
function rateLimit(request: Request) {
  const now = Date.now();
  for (const [key, ts] of visitorChecks)
    if (now - ts > 60_000) visitorChecks.delete(key);
  if (visitorChecks.size > 5_000) visitorChecks.clear();  // never grow unbounded
  // ... existing per-visitor check ...
}

// 2. Make the timeout cancellable so the loser is collected.
function timeout(ms: number) {
  let id: ReturnType<typeof setTimeout>;
  const promise = new Promise<void>((resolve) => { id = setTimeout(resolve, ms) });
  return { promise, cancel: () => clearTimeout(id) };
}

const guard = timeout(3500);
await Promise.race([reconciliation, guard.promise]).finally(guard.cancel);
```

DCW-019 | Unintended Behavior

contracts/UnifiedXOtcEscrow.sol, app/components/otc-workspace.tsx

The client reads previewFill and then submits fillOrder as two separate calls, so another taker's partial fill in between makes the quoted payment stale on arrival. There is also no deadline, so a transaction that sits in the mempool can execute much later than intended.

The blast radius is small - ceiling rounding only ever moves the price in the taker's favour, and the native path's strict equality check reverts rather than overcharging - but for ERC-20 payment the taker signs a transaction whose final cost is not fixed at signing time.

```
// app/components/otc-workspace.tsx
// two separate calls: the quote can go stale between them
const payment=BigInt(await escrow.previewFill(order.id,amount));
if(isNativeAsset(order.buyToken))
  await (await escrow.fillOrder(order.id,amount,{value:payment})).wait();
else{
  await (await paymentToken.approve(deployment.contract,payment)).wait();
  await (await escrow.fillOrder(order.id,amount)).wait();
}

// contracts/UnifiedXOtcEscrow.sol
// no maxBuyPayment, no deadline: taker intent is assumed, not enforced
function fillOrder(uint256 orderId, uint256 sellFillAmount)
    external payable whenNotPaused nonReentrant returns (uint256 buyPayment)
```

Recommendation:

Enforce the taker's intent on-chain rather than assuming it. Add maxBuyPayment and deadline parameters, and pass the quoted payment through from the client so a stale quote reverts instead of settling at an unexpected price.

```
error ExcessivePayment();
error DeadlinePassed();

function fillOrder(
    uint256 orderId,
    uint256 sellFillAmount,
    uint256 maxBuyPayment,
    uint64  deadline
) external payable whenNotPaused nonReentrant returns (uint256 buyPayment) {
    if (block.timestamp > deadline) revert DeadlinePassed();

    Order storage order = orders[orderId];
    // ... existing state / expiry / taker checks ...

    buyPayment = previewFill(orderId, sellFillAmount);
    if (buyPayment > maxBuyPayment) revert ExcessivePayment();
    // ... existing settlement ...
}

// client: pass the quote through, with a small tolerance and a deadline
const quoted = BigInt(await escrow.previewFill(order.id, amount));
const deadline = Math.floor(Date.now() / 1000) + 300;
await escrow.fillOrder(order.id, amount, quoted, deadline,
  isNativeAsset(order.buyToken) ? { value: quoted } : {});
```

DCW-015 | Potential Backdoor

contracts/UnifiedXOtcEscrow.sol

No backdoor is present. The contract has no admin withdrawal, no upgrade proxy, no delegatecall and no privileged transfer path - maker funds can leave escrow only through a fill, a maker cancellation or expiry reclamation, exactly as documented. That design is implemented correctly.

The one residual control is that pause() blocks createOrder and fillOrder with no time bound and no on-chain path to force an unpause. cancelOrder and reclaimExpired are correctly left unpausable so makers can always exit, which limits the exposure to takers being unable to fill and makers having to wait out expiry.

```
// contracts/UnifiedXOtcEscrow.sol
function pause() external onlyOwner { _pause(); }
function unpause() external onlyOwner { _unpause(); }

// createOrder and fillOrder are gated ...
function createOrder(...) external payable whenNotPaused nonReentrant
function fillOrder(...)   external payable whenNotPaused nonReentrant

// ... but the exit paths correctly are not, so makers can always withdraw
function cancelOrder(uint256 orderId)    external nonReentrant
function reclaimExpired(uint256 orderId) external nonReentrant
```

Recommendation:

Move ownership to a timelocked multisig and give the pause an automatic expiry so it cannot become permanent through key loss or compromise. Disclose the owner key's custody arrangement in the user-facing documentation.

```
uint64 public constant MAX_PAUSE = 7 days;
uint64 public pausedUntil;

error PauseWindowExpired();

function pause() external onlyOwner {
    pausedUntil = uint64(block.timestamp) + MAX_PAUSE;
    _pause();
}

function unpause() external onlyOwner {
    pausedUntil = 0;
    _unpause();
}

// Anyone may lift a stale pause, so it can never become permanent.
function expirePause() external {
    if (pausedUntil == 0 || block.timestamp < pausedUntil)
        revert PauseWindowExpired();
    pausedUntil = 0;
    _unpause();
}

// Deploy with ownership held by a timelocked multisig:
// constructor() Ownable(TIMELOCK_MULTISIG) {}
```

DCW-015 | Potential Backdoor

lib/dexlab-api-handler.ts, app/components/token-creator.tsx

Both the client state and the server payload default to keeping mint authority - the server line body.mintAuthority !== false means an omitted or malformed field resolves to true. Tokens created through the builder are therefore inflatable by their creator unless the user opens the collapsed advanced panel and unchecks the box.

For a public token launcher this is the rug-enabling default: most users will ship a token whose supply they can silently expand, and holders have no way to tell from the UI that this happened.

```
// lib/dexlab-api-handler.ts
const payload:Record<string,unknown>={
  payer, name, symbol, decimals, mintSupply,
  mintToPubkey:payer,
  mintAuthority:body.mintAuthority!==false,   // omitted field -> TRUE
  freezeAuthority:body.freezeAuthority===true,
  program,
};

// app/components/token-creator.tsx - client default also retains it
const [mintAuthority,setMintAuthority]=useState(true);

// L96 - and the control is hidden behind a collapsed panel
{advanced&&<div className="token-advanced"> ... </div>}
```

Recommendation:

Default to revoking mint authority and make retention an explicit opt-in. Fail closed on the server so a malformed request cannot silently enable it, and surface the choice on the main form rather than behind a disclosure toggle.

```
// lib/dexlab-api-handler.ts - fail closed
const payload: Record<string, unknown> = {
  payer, name, symbol, decimals, mintSupply,
  mintToPubkey: payer,
  mintAuthority: body.mintAuthority === true,     // omitted -> FALSE
  freezeAuthority: body.freezeAuthority === true,
  program,
};

// app/components/token-creator.tsx - safe default
const [mintAuthority, setMintAuthority] = useState(false);

// Surface the risk on the main form, not inside the advanced panel:
<label className="token-authority-warning">
  <input type="checkbox" checked={mintAuthority}
         onChange={e => setMintAuthority(e.target.checked)} />
  <span>
    <b>Keep mint authority</b>
    <small>
      Leaving this on lets your wallet mint unlimited additional supply
      later. Most holders treat this as a rug-pull risk. Off is recommended.
    </small>
  </span>
</label>
```

DCW-011 | Insecure API Usage

app/api/limit-order/route.ts

The cancellation call sets an Origin header of https://kyberswap.com on a server-to-server request, deliberately defeating an upstream control that exists to bind cancellations to KyberSwap's own front end.

It works today, but it makes the integration fragile and misrepresents the caller to a third-party API: KyberSwap can tighten this check at any time, which would break limit-order cancellation for every UNIFIEDX user with no warning and no fallback.

```
// app/api/limit-order/route.ts
if(action==="cancel"){
  const signature=String(body.signature??"");
  if(!signaturePattern.test(signature))
    return json({error:"Invalid cancellation signature."},400);

  return kyber("/write/api/v1/orders/cancel",{
    method:"POST",
    headers:{
      "content-type":"application/json",
      "origin":"https://kyberswap.com",   // spoofs the upstream origin check
    },
    body:JSON.stringify({chainId,maker,orderIds,signature}),
  });
}
```

Recommendation:

Request a proper integrator credential from KyberSwap and send that instead of a forged Origin. Until one is issued, send the real origin and handle the rejection explicitly so users get a clear message and a documented manual fallback rather than a silent failure.

```
// Identify honestly with an issued integrator credential.
const KYBER_HEADERS = {
  "accept": "application/json",
  "x-client-id": "UNIFIEDX",
  "user-agent": "UNIFIEDX/2.0 (+https://unifiedx.io)",
  ...(env.KYBERSWAP_INTEGRATOR_KEY
      ? { "x-integrator-key": env.KYBERSWAP_INTEGRATOR_KEY }
      : {}),
};

if (action === "cancel") {
  const response = await kyber("/write/api/v1/orders/cancel", {
    method: "POST",
    headers: { ...KYBER_HEADERS, "content-type": "application/json" },
    body: JSON.stringify({ chainId, maker, orderIds, signature }),
  });

  if (response.status === 403) {
    return json({
      error: "Cancellation through UNIFIEDX is temporarily unavailable. "
           + "You can cancel this order directly on kyberswap.com.",
    }, 502);
  }
  return response;
}
```

DCW-007 | Framework Version

package.json

All frameworks are on current majors - Next 16.2.6, React 19.2.6, ethers 6.17.0, OpenZeppelin Contracts 5.6.1, solc 0.8.36 and Wrangler 4.92.0 - and pragma solidity ^0.8.24 gives built-in arithmetic overflow protection. No version-specific vulnerability was identified.

The observation is about pinning practice: runtime dependencies use caret ranges while devDependencies are pinned exactly, which is the wrong way round. Since @lifi/sdk, @reown/appkit and the wallet adapters all handle transaction data, a compromised minor release of any of them would reach production without review.

```
// package.json - runtime deps float, dev deps are pinned
"dependencies": {
  "@lifi/sdk": "^4.3.0",              // <- floats
  "@reown/appkit": "^1.8.23",         // <- floats, handles wallet connections
  "@reown/appkit-adapter-ethers": "^1.8.23",
  "ethers": "^6.17.0",                // <- floats, signs transactions
  "next": "16.2.6",
  "react": "19.2.6"
},
"devDependencies": {
  "@openzeppelin/contracts": "^5.6.1",
  "typescript": "5.9.3",              // <- pinned exactly
  "vite": "8.0.13",
  "wrangler": "4.92.0"
}
```

Recommendation:

Pin runtime dependencies exactly and let a bot propose upgrades so every bump is reviewed, and add a dependency audit gate to CI. This review did not resolve the dependency tree, so no statement is made here about advisories in transitive packages - the CI gate is what will surface those.

```
// package.json - pin everything that ships to users
"dependencies": {
  "@lifi/sdk": "4.3.0",
  "@reown/appkit": "1.8.23",
  "@reown/appkit-adapter-ethers": "1.8.23",
  "ethers": "6.17.0",
  "next": "16.2.6",
  "react": "19.2.6"
},
"scripts": {
  "audit": "npm audit --audit-level=high"
}

# .github/workflows/ci.yml - fail the build on a known high severity advisory
- name: Dependency audit
  run: npm ci && npm audit --audit-level=high

# Commit the lockfile and let Dependabot/Renovate open reviewable PRs
# instead of letting caret ranges float silently into production.
```

AUDIT COMMENTS | UNIFIEDX

Dapp audit comment for a non-technical perspective

🟢 Unified has fixed/mitigated all of the issues and is MARKED safe to interact with using any EVM wallet (08-16-2026)

🟢 Smart contract has no backdoors - no admin withdrawal, upgrade or privileged transfer path

🟢 DAPP cannot drain balances

CONTRACTWOLF

Blockchain Security - Smart Contract Audits