Zellic

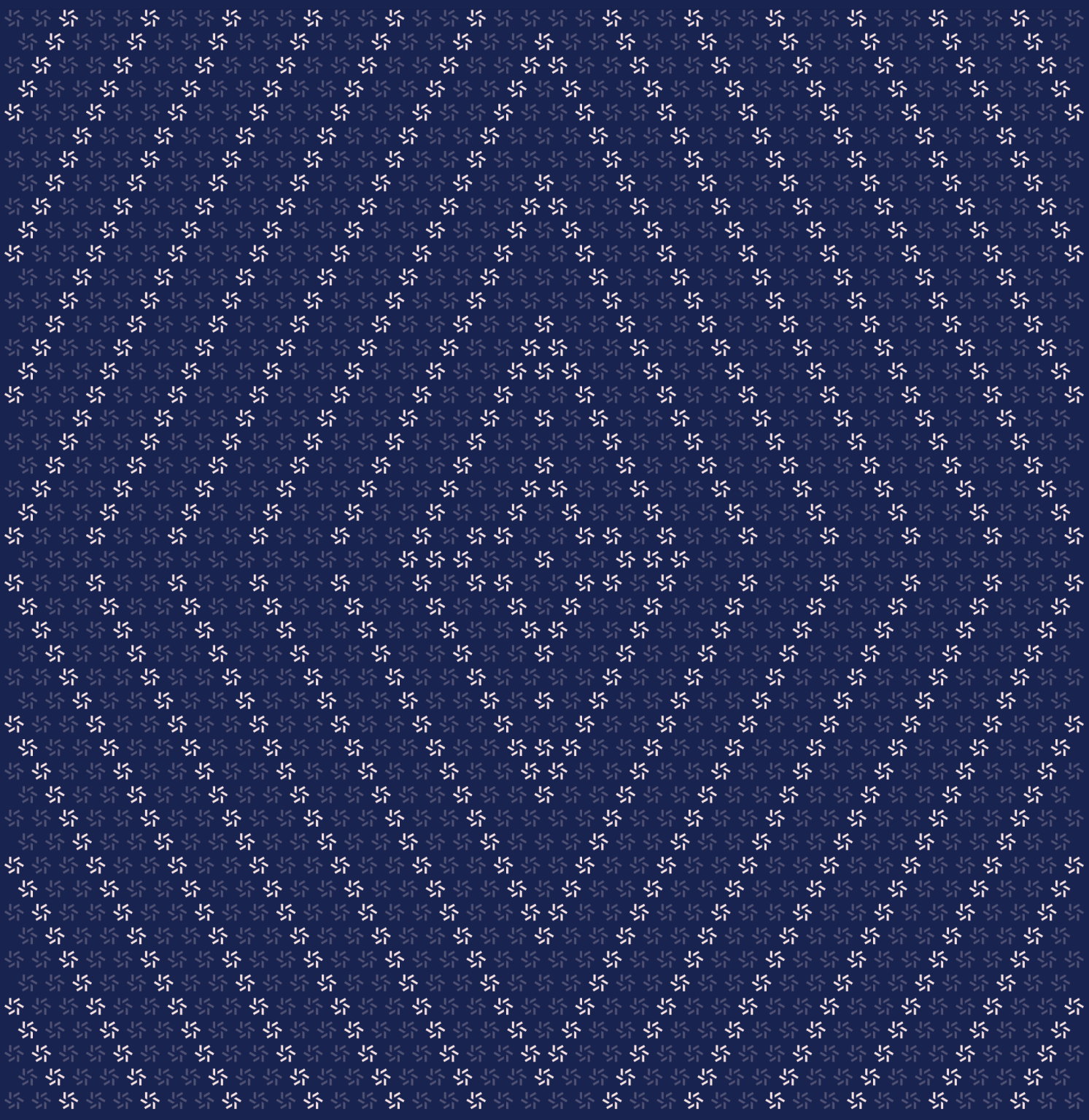
Prepared for
Franklyn Wang
Scott Becker
Liquid

Prepared by
Michael Guarino
Moritz Schneider
Zellic

August 28, 2025

LiquidMax

Web Application Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Liquid from July 28th to August 27th, 2025. During this engagement, Zellic reviewed LiquidMax's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the referral payout system secure?
- Is the key route secure?
- Does the Swift application securely handle secrets?
- Does the Swift application contain any front-end issues that can expose customer data?
- Does the backend server securely handle authentication and authorization?
- Does the backend server contain any injection or logical vulnerabilities?
- Could an attacker abuse the Privy recovery site to recover an account that they do not own?
- Could an attacker abuse the Privy recovery site to leak information about another user?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Leaderboard backend
- Access codes
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped LiquidMax contracts, we discovered 11 findings. No critical issues were found. Three findings were of high impact, four were of medium impact, three were of

low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Liquid in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	4
Low	3
Informational	1

2. Introduction

2.1. About LiquidMax

Liquid contributed the following description of LiquidMax:

> Liquid is a mobile interface for trading spot tokens, perpetual futures and for using vaults.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and

Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

LiquidMax Contracts

Types	Swift, Python, TypeScript
Platform	web
Target	hyperliquid-swift-app
Repository	https://github.com/LiquidMax-dev/hyperliquid-swift-app ↗
Version	`ab673f324ee40deff9961bb18609047b2acb6748`
Programs	`Hyperliquid/**/*.swift (and associated configuration files)`
Target	liquidmax_server
Repository	https://github.com/LiquidMax-dev/liquidmax_server ↗
Version	`552b31a0507ba1baf2ef38b33f84eeababdf7406`

Programs	`**.py (excluding liquidserver/routes/leaderboard.py)`
	`**.ts`
	`prisma/**`

Target	privy-recovery-site
Repository	https://github.com/LiquidMax-dev/privy-recovery-site
Version	`788289af0782a7973fcd43c629639ea3a035d413`
Programs	`src/**`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of seven person-weeks. The assessment was conducted by two consultants over the course of four calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Michael Guarino
Engineer
michael@zellic.io

Moritz Schneider
Engineer
moritz@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 28, 2025	Kick-off call
July 28, 2025	Start of primary review period
August 27, 2025	End of primary review period

3. Detailed Findings

3.1. Race-condition referral claiming

Target	liquidmax_server		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The server exposes an API endpoint `/api/referral/claim` to claim rewards for referrals. The route handler calls the function `claim_referral_rewards`, which fetches the unclaimed reward entries, sums them up, and then calls `exchange.usd_transfer(total_amount, wallet_address)` with the `wallet_address` of the user with the total unclaimed reward using `exchange` set up with the private key `REFERRER_PRIVATE_KEY`.

The backend is run in modal, which automatically spins up multiple instances of the backend. This means that multiple requests could be processed concurrently. If a user were to spam this API endpoint, then multiple transfers could be triggered for the same reward, meaning the user would receive more money than they should.

This is because there is a delay between fetching the reward amount and marking the rewards as paid out, creating a race condition.

```
unclaimed_rewards = await prisma_client.referralreward.find_many(
    where={"receiverId": user.id, "claimed": False}
)

#[...]

total_amount = sum(float(reward.amount) for reward in unclaimed_rewards)

#[...]

new_acct_key = os.environ.get("REFERRER_PRIVATE_KEY", "")
_, _, _, exchange, _ = await setup_local_account(new_acct_key,
    REFERRAL_ADDRESS, True)
#[...]
try:
    usd_transfer_response = await exchange.usd_transfer(total_amount,
    wallet_address)
    #[...]
#[...]
```

```
# Update all rewards as claimed
for reward in unclaimed_rewards:
    await prisma_client.referralreward.update(
        where={"id": reward.id},
        data={"claimed": True, "claimedAt": datetime.now()},
    )
```

Impact

An attacker could drain the funds of the `REFERRER_PRIVATE_KEY` wallet.

Recommendations

The code should be changed to make sure rewards are not processed multiple times.

This could be done by using transactions to mark the rewards as processed and paying out the rewards after the database transaction completes.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit `26c7eb5d`↗.

3.2. ClickHouse SQL injection

Target	liquidmax_server		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The data API constructs ClickHouse SQL by directly interpolating user-controlled input into f-strings without parameterization. In particular,

- `/api/data/pnl` embeds `wallet_address` and `mainnet` directly into the query string.
- `/api/data/volume_candles` and `/api/data/candlesticks` embed `symbol` (and derived interval fragments) directly into helper query builders `one_day_line_query` and `short_term_query`.

Here are some examples from the implementation:

```
# /api/data/pnl
query = f"SELECT * FROM pnl WHERE wallet = '{wallet_address}' AND isMainnet =
    {mainnet} AND time >= now() - 86400 ORDER BY time ASC"

# In one_day_line_query
return f"""
    ...
    FROM default.perps_mids_millis
    WHERE symbol = '{symbol}'
    AND toDateTime64(time, 3) >= toDateTime64({start_time/1000}, 3)
    AND toDateTime64(time, 3) <= toDateTime64({end_time/1000}, 3)
    ...
"""

# In short_term_query
WHERE symbol = '{symbol}'
```

Because `wallet_address` and `symbol` are taken directly from request parameters, an attacker can inject quotes and SQL operators to alter the `WHERE` clause or append additional statements. For example, a crafted wallet parameter like `abc' OR 1=1 --` would bypass the intended per-wallet filter and return all rows; a crafted symbol could `UNION` data from other tables or trigger expensive operations if the database user has sufficient privileges.

The root cause is the lack of query-parameter binding/whitelisting for string inputs. The code

should use parameterized queries supported by the ClickHouse client and strictly validate/whitelist allowable `symbol` values and formats for `wallet_address` to prevent injection.

Impact

This could

- bypass filters (e.g., force `wallet` predicates to always true) to read PNL across all users;
- `UNION` or otherwise manipulate the query to exfiltrate data from other tables accessible to the ClickHouse user; or
- trigger expensive scans/aggregations to degrade service availability and increase costs (denial of service at the query layer).

Recommendations

Use parameterized queries with bound variables provided by the ClickHouse client instead of f-string interpolation for `wallet_address`, `symbol`, and time bounds.

Remediation

This issue has been acknowledged by Liquid, and fixes were implemented in the following commits:

- `3651839b` ↗
- `f4317d9f` ↗
- `c2d765b3` ↗
- `d7f3a09b` ↗
- `c99f5f32` ↗

3.3. Privy identity token decoded without signature verification

Target	liquidserver/core/auth.py		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `get_wallet_and_provider_from_privy_identity_token` function defined within liquidserver/core/auth.py is designed to extract a user's wallet address, log-in provider, provider user ID, and email address from a Privy identity token. The `get_wallet_from_privy_identity_token` function wraps `get_wallet_and_provider_from_privy_identity_token` and returns just the wallet address. Then, `get_wallet_from_privy_identity_token` is called in the `sign_in` route at `/api/signin`, defined within liquidserver/routes/auth.py.

The `get_wallet_and_provider_from_privy_identity_token` contains a try-except block, which falls back to decoding without signature verification if the initial decode errors at any point.

```
async def get_wallet_and_provider_from_privy_identity_token(identity_token:
    str) -> Optional[Tuple[str, LoginProvider, str, Optional[str]]]:
    # [...]
    try:
        # Get the signing key from the JWKS
        signing_key = jwks_client.get_signing_key_from_jwt(identity_token)

        # Decode and verify the token
        decoded = jwt.decode(
            identity_token,
            signing_key.key,  # PyJWKClient handles the JWK to key conversion
            algorithms=["ES256"],  # Privy uses ES256
            audience=privy_app_id,
            issuer="privy.io"
        )
        logger.info("Successfully verified and decoded token with JWK")
    except Exception as e:
        logger.warning(f"JWK verification failed: {str(e)}")
        logger.warning("Falling back to decoding without verification")
        # Fallback to decoding without verification
        decoded = jwt.decode(
            identity_token,
            options={"verify_signature": False},
```

```
            audience=privy_app_id,
            issuer="privy.io"
        )
        logger.info("Successfully decoded token (signature verification
    skipped)")

    # Get the subject (Privy user ID)
    privy_user_id = decoded.get('sub')
    # [...]
```

As a failure to decode the token in the try will always return an error and revert to the verificationless fallback, any token, regardless of signature, will always decode properly.

Impact

Any token regardless of signature validity will always be decoded as if it was signed by Privy. This leads to an authentication bypass as an attacker can forge a signature for any wallet address. Any endpoints that are callable with just a resulting JWT token are able to be called by an attacker as a victim user.

Recommendations

Do not trust JWTs decoded without signature verification.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit 38d39f4a ↗.

3.4. Lack of comprehensive tests

Target			
Category	Code Maturity	**Severity**	High
Likelihood	N/A	**Impact**	High

Description

Although tests exist and cover portions of the codebase, large parts of the codebase do not have sufficient test coverage. Areas of the backend server which are currently lacking in test coverage include important API routes, core business logic, database models, microservices, and the node server. Areas of the Swift application which are currently lacking in test coverage include UI and integration testing.

The most important areas where test coverage should be improved are:

Backend server

API Routes

The following API routes currently lack comprehensive testing. They should be thoroughly tested to ensure both positive and negative validation of expected functionality. Cases should additionally be expanded to cover security and edge case testing, including areas such as input validation, rate limiting, authentication bypass attempts, race conditions, and data privacy where applicable.

- auth.py
- wallet.py
- charts.py
- generate.py
- leaderboard.py
- onramp.py
- referrals.py
- trades.py
- trending.py
- unit_bridge.py
- voice.py
- access_codes.py

Core Business Logic

The following areas of core business logic currently lack comprehensive testing. They should be thoroughly tested to ensure both positive and negative validation of expected functionality. Additional edge case testing should be implemented as applicable.

- core/auth.py
- core/encryption.py
- core/error_handlers.py
- core/exchange.py
- core/feature_flags.py
- core/notifications.py
- core/services.py

Database Models

Database models currently lack validation tests. Positive and negative test cases should be implemented to validate the functionality of each model.

Microservices

Many microservices are missing tests as well. At a minimum, microservices which enable or interact with any sensitive data or functionality should be tested for positive and negative validation as well as applicable edge cases which could impact security.

Node Server

The Node server component should be tested to the same level as the standard Liquid server. All important Node server routes, middlewares, services, models, and utilities should have the same level of comprehensive testing performed in addition to security impacting edge case testing.

Swift Application

The Swift Application could also benefit from improved testing as there are multiple areas which are currently untested. The impacted areas are:

- UI interactions (ensuring UI functionality behaves as expected in all cases)
- Authentication flows (ensuring that positive and negative authentication cases are behaving as expected, and authentication mechanisms cannot be bypassed in any way)
- Deep linking (ensuring that deep linking functionality behaves as expected and additionally cannot be abused by an attacker)
- Widget and push notifications

Impact

The existence of comprehensive testing can catch regressions in functionality or security quickly, as a failing test case due to a new change will indicate what went wrong and where. Thorough automated testing is more reliable and consistent than relying on manual human testing alone, and issues are much more likely to be detected before they can ever hit production.

Recommendations

Implement test cases for each identified area which is currently lacking. Ensure that both positive and negative cases are covered. Expand test cases to cover security-relevant edge cases wherever applicable.

Remediation

This issue has been acknowledged by Liquid, and fixes were implemented in the following commits:

- 7730856b ↗
- 8ba36d3c ↗

3.5. Missing username-length check in update route

Target	liquidmax_server		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The length check that is present in the onboarding route `/api/onboard` is missing in the update route `/api/update_username`.

The username length check in the onboarding route:

```
# Validate username length
if len(request.username.strip()) < 3:
    return OnboardResponse(
        success=False,
        message="Username too short",
        user=None,
        error="Username must be at least 3 characters long",
    )
```

Impact

Users can change their username to be less than three characters. This is inconsistent with the onboarding logic. There is no security impact.

Recommendations

Add the length check to the update route.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit df5dba71 ↗.

3.6. Limited server-side request forgery in image generation

Target	liquidmax_server (nodeserver)		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The image-generation endpoint accepts user-controlled URLs that the server will fetch, enabling server-side request forgery (SSRF) and potential resource-exhaustion attacks.

The `POST /image/generate/news` route passes the request body directly to `ImageGeneratorService.generateNewsCard`:

```
// Image routes
.post("/image/generate/pnl", imageRoutes.generatePNL)
.post("/image/generate/news", imageRoutes.generateNews)
```

Inside the service, if `author.imageUrl` is provided, the server downloads that URL:

```
// Fetch and process author image if provided
const authorImageBuffer = author.imageUrl
  ? await fetchAndProcessImage(author.imageUrl)
  : null;
```

The fetch helper in nodeserver/utils/image.ts performs a raw HTTPS `GET` to the supplied URL with no hostname allowlist, scheme/port validation, redirect limits, or size/time limits:

```
export function fetchAndProcessImage(url: string): Promise<Buffer | null> {
  return new Promise((resolve) => {
    https
      .get(url, (response) => {
        const chunks: Buffer[] = [];
        response.on("data", (chunk: Buffer) => chunks.push(chunk));
        response.on("end", async () => {
          try {
            const buffer = Buffer.concat(chunks);
            const processedBuffer = await sharp(buffer)
              .resize(80, 80, { fit: "cover" })
              .toBuffer();
```

```
            resolve(processedBuffer);
```

Because `imageUrl` is attacker-controlled, an attacker can cause the server to initiate requests to arbitrary HTTPS endpoints reachable from the server.

The exposed python route `/api/generate/news` defined in liquidserver/routes/generate.py directly passes the request body to the Elysia server.

```
@router.post("/api/generate/news", response_model=ImageResponse)
   async def generate_news(request: NewsGenerateRequest):
       """Generate a news card image."""
       try:
           async with httpx.AsyncClient() as client:
               response = await client.post(
                   f"{ELYSIA_URL}/image/generate/news",
                   json=request.dict(exclude_none=True)
               )
               result = response.json()
               return result
```

While this helper uses `https.get` (limiting it to HTTPS and `GET` requests), it could still lead to resource exhaustion.

Impact

This could lead to blind SSRF to arbitrary HTTPS endpoints from the server's network context as well as potential access to internal services exposed over HTTPS with valid certificates (there is unlikely to be any and it is also limited to `GET` requests).

It could also lead to a denial of service via large responses (unbounded buffering) or slow responses (no time-out) and possible excessive ingress costs.

Recommendations

We recommend the following:

- Enforce a strict allowlist of approved image hostnames/CDNs; reject IP literals and internal/reserved ranges.
- Set tight network controls: connect/read time-outs, maximum response size, and limit/disable redirects.
- Verify `Content-Type` and `Content-Length` before download.
- Add authentication and rate limiting to `/image/generate/news` to reduce abuse potential.

Alternatively, removing image generation from the server entirely would also solve this issue.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit a80a54a0 ↗.

3.7. SVG injection into `sharp`

Target	liquidmax_server (nodeserver)		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

User-controlled strings are interpolated directly into an SVG template in nodeserver/services/ImageGeneratorService.ts that is rasterized by `sharp` without XML/SVG escaping. Fields `content`, `author.name`, `author.platform`, and `user` are inserted into `<text>` / `<tspan>` nodes as raw text, enabling SVG injection.

```
// Create SVG with text
const svgText = Buffer.from(`
  <svg width="${this.NEWS_CARD_WIDTH}" height="${
  this.NEWS_CARD_HEIGHT
}" xmlns="http://www.w3.org/2000/svg">
    <style> ... </style>
    ${
      authorImageBuffer
        ? `
    <text x="${this.PADDING + 100}" y="${
            this.PADDING + 10
          }" class="author">${author.name}</text>
    <text x="${this.PADDING + 100}" y="${
            this.PADDING + 50
          }" class="platform">from ${author.platform || "Liquid"}</text>
    `
        : `
    <text x="${this.PADDING}" y="50" class="author">${author.name}</text>
    <text x="${this.PADDING}" y="80" class="platform">from ${
            author.platform || "Liquid"
          }</text>
    `
    }
`}
```

This is reachable via an API endpoint, similar to Finding 3.6. ↗.

Impact

This may lead to arbitrary SVG injection server-side or a denial of service via expensive SVGs, causing high CPU/memory usage while `sharp`/libvips/librsvg renders the SVG.

We experimented with different payloads and were able to take up to three seconds of full CPU usage per request before `sharp` would abort.

The library used by sharp (librsvg) only supports a subset of SVG and did not allow including external files by default. We could not turn this into an arbitrary file read.

Recommendations

We recommend the following:

- Escape all user-provided strings before embedding into SVG (replace `& < > " '`).
- Remove image processing from the server if possible.
- Isolate image-processing code to a different server to separate concerns and limit potential damages in case of compromise.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit a80a54a0 ↗.

3.8. No validation for MoonPay signature

Target	liquidmax_server		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The backend server contains a MoonPay integration. As part of MoonPay's security protections, sensitive information such as wallet addresses or emails must be signed when displayed in the MoonPay widget. This protection helps prevent third parties from misusing the autofill feature, which could be abused for phishing-type attacks against an unsuspecting user. The backend server does indeed use signed parameters; however, data is signed via the `/api/v1/moonpay/signature` route based on user input. Although the data is signed server-side, it signs arbitrary user data, which defeats the purpose of a signature as an attacker can sign anything. This bypasses MoonPay's built-in mechanism to prevent abuse.

Impact

A user is able to sign arbitrary MoonPay parameters, allowing them to craft a malicious phishing URL to cause an unsuspecting user to potentially send funds to unexpected addresses.

Recommendations

Only sign validated or expected data that is controlled by the server rather than allowing a user to sign anything they want.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit `758e9a87` ↗.

3.9. ClickHouse credentials in Git

Target	liquidmax_server		
Category	Code Maturity	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The repository contains database connection credentials directly embedded in code at liquidserver/microservices/constants.py.

Hardcoding secrets in source exposes them to all collaborators, CI systems, build artifacts, and backups. Even if removed later, secrets remain retrievable from Git history. Compromise of the repository or any leaked artifact could provide immediate read/write access to the ClickHouse instance.

In this deployment, the ClickHouse dataset consists of tweets, reactions, and other public information, which reduces confidentiality risk; however, integrity and availability risks remain.

Impact

This limits confidentiality, as the stored data (tweets, reactions, PNL) is public. Regarding integrity, unauthorized writes/deletes can corrupt or skew analytics, user-facing metrics, and product behavior. Moreover, in terms of availability, expensive or destructive queries can degrade performance or cause outages (which have a cost and reliability impact).

Recommendations

Remove credentials from the source code. Load secrets at runtime from modal.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit `fa6868fc` ↗.

3.10. Neon postgres credentials in Git

Target	liquidmax_server		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The repository contains database connection credentials directly embedded in code at LiquidMax-dev-liquidmax_server/recent_access_codes.py.

Hardcoding secrets in source exposes them to all collaborators, CI systems, build artifacts, and backups. Even if removed later, secrets remain retrievable from Git history. Compromise of the repository or any leaked artifact could provide immediate read/write access to the postgres neon database.

The database does not contain passwords or private keys but other sensitive data such as the referrals structure.

We have not verified if the credential is for read access only.

Impact

If the credential is for read-only access, it could read sensitive data such as referral structure.

If the credential is for write access, it could drain the referral wallet by creating a fake `ReferralReward` entry.

Recommendations

Remove credentials from source code. Load secrets at runtime from modal.

Additionally, rotate the credential.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit `fbb2f6e4` ↗.

3.11. Insufficient validation of `wallet_address` in `/api/bridge_unit` backend route

Target	ContractNameHere		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `wallet_address` is not sufficiently validated and could potentially allow path traversal in the HyperUnit request.

```
async with httpx.AsyncClient() as client:
                wallet_address = request.wallet_address
                url =
    f"{HYPERUNIT_API_BASE}/gen/{src_chain}/{dst_chain}/{asset}/{wallet_address}"
                print(url)
                logger.info(f"Calling HyperUnit API: {url}")
```

This could be used to bypass the checks for `valid_chains` and `valid_assets` by supplying an address like `../../../X/Y/Z/0x...`.

Impact

Since the HyperUnit API is public, there is no real impact here at the time of writing. However, this code pattern should be avoided as it could lead to severe bugs when used with private APIs.

As an example, if this was a privileged endpoint and there were other privileged endpoints on the same origin, this bug would allow an attacker to change the path of the request to another endpoint.

Recommendations

Validate the `wallet_address`.

Remediation

This issue has been acknowledged by Liquid, and a fix was implemented in commit 08d237cc ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Unused and outdated code and comments

We identified a significant amount of dead code throughout the codebase; we recommend removing these unused sections to improve maintainability and reduce potential confusion.

Here are several examples:

- In LiquidMax-dev-privy-recovery-site, LiquidMax-dev-privy-recovery-site/src/app/lib/services/PrivyService.ts, there are the unused methods `createKeyQuorum` (L25–L76), `getWalletDetails` (L81–L101), `assignKeyQuorumToWallet` (L106–L147), and `exportWallet` (L174–L201).
- In LiquidMax-dev-liquidmax_server, liquidserver/example_utils.py, there are the unused methods `unsafe_sign_hash` (L37–L38), `signHash` (L40–L56), `sign_message` (L87–L132), `signTransaction` alias (L147–L149), and `setup_multi_sig_wallets` (L233–L250). Methods like `sign_typed_data_server`, `sign_transaction`, and setup helpers are used.
- In LiquidMax-dev-liquidmax_server, liquidserver/routes/wallet.py, there is the unused route (L164–L183).

A number of comments in the codebase also are outdated, are misleading, or do not accurately reflect the current implementation; these should be reviewed and updated to match actual behavior.

Here are several examples:

- In LiquidMax-dev-liquidmax_server, liquidserver/routes/trending.py, there is incorrect module/docstring context (L24–L30 says “referral-related”), while the endpoint is for the trending tokens feature.
- In LiquidMax-dev-liquidmax_server, liquidserver/routes/unit_bridge.py, there are misleading docstrings (L28–L34 and route doc at L41–L44 mention referrals — should be unit bridge).
- In LiquidMax-dev-liquidmax_server, liquidserver/microservices/data_populator/cleanup_data.py, there is an incorrect copied function docstring (L18–L20).
- In LiquidMax-dev-liquidmax_server, liquidserver/app.py, the `create_fastapi_app()` docstring is outdated — it lists routes that do not exist and is missing routes that do (L212–L258).

4.2. Insufficient input validation in Elysia

Several endpoints accept request input (query, body) without enforcing runtime validation. While TypeScript annotations on handler arguments improve editor tooling, they do not validate or coerce untrusted runtime data; incoming JSON and query strings can have arbitrary shapes and types. Relying solely on TypeScript types creates a gap between compile-time expectations and runtime values, increasing the risk of type confusion, logic errors, and inconsistent behavior.

Elysia provides native, fast runtime validation via TypeBox schemas. Using these schemas ensures only well-formed data reaches business logic. See Elysia's validation guide ↗.

Without input validation, there is the following:

- An increased attack surface from unvalidated input (e.g., unexpected types/structures that bypass guards and trigger edge paths)
- Hard-to-reproduce production bugs from implicit coercions and unexpected shapes
- Reduced reliability and observability (without consistent validation errors, debugging malformed requests is harder)

4.3. Critical components should be split from non-critical ones

To minimize the risk of vulnerabilities in places where they would have high impact (in this case, where the private keys of the users get sent to), routes that do not need the private key and add significant attack surface should be moved to another server.

Specifically, the image-generation code should be moved away from the main server as it adds a lot of surface and risk.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: LiquidMax server

The LiquidMax server is composed of several components.

Databases

There are multiple databases, the main Prisma database, a database for the chart data, and a ClickHouse database for PNL and trading volume.

Modal is used to host these components in the cloud.

Main app

The main component (`liquidmax_server`) is a FastAPI Python server combined with a bun Elysia app for logic that was easier to implement in JavaScript, presumably due to the available libraries.

The main app serves as the backend for the mobile application. The following functionality is exposed on the server:

Authentication

The app allows for sign-in/sign-up using `privy`. The `wallet_address` that is stored in a JWT and signed by `privy` is used to associate a user to a wallet.

After sign-in/sign-up, the server returns a JWT signed by the app secret `JWT_SECRET_KEY` with `HS256`. This JWT is used in the other endpoints to get the user and an authenticated wallet address.

```
@router.post("/api/signin", response_model=SignInResponse)
@router.post("/api/refresh-token", response_model=TokenRefreshResponse)
@router.post("/api/update_username", response_model=UpdateUsernameResponse)
@router.post("/api/onboard", response_model=OnboardResponse)
@router.get("/api/user-profile", response_model=UserProfileResponse)
@router.put("/api/user", response_model=SignInResponse)
@router.post("/api/auth/check-email", response_model=CheckUserResponse)
@router.post("/api/auth/check-username", response_model=CheckUsernameResponse)
@router.post("/api/auth/check-provider-user",
```

```
    response_model=CheckProviderUserResponse)
@router.post("/api/delete-account")
```

Charts

The `/api/charts` routes forward the requests to the Elysia app and feature a cache that will cache the response for 30 seconds.

The Elysia part of the app will fetch the chart data from the ClickHouse database using `sequilize`.

```
@router.get("/api/charts", response_model=ChartResponse)
@router.post("/api/charts", response_model=ChartResponse)
```

Data

The `/api/data/*` routes serve data queried from a ClickHouse database.

```
@router.get("/api/data/pnl", response_model=List[PnlData])
@router.get("/api/data/volume_candles", response_model=List[CandleData])
@router.get("/api/data/candlesticks", response_model=List[CandleData])
```

Image

These endpoints will forward their request to the Elysia app. There, it will generate SVG that will then be rasterized using `sharp`/librsvg and return the image to the user.

```
@router.post("/api/generate/pnl", response_model=ImageResponse)
@router.post("/api/generate/news", response_model=ImageResponse)
```

News

The app features a news functionality with the possibility to react to posts. The posts and reactions are stored in the main database.

```
@router.post("/api/news/reaction_voters", response_model=VotesResponse)
@router.post("/api/news/process_like_reaction",
    response_model=ReactionResponse)
@router.post("/api/news/author_news", response_model=NewsResponse)
```

Onramp

This provides fiat-to-crypto onramping utilities via Stripe (direct onramp session), Mesh (link token generation), and MoonPay (URL signing).

```
@router.post("/api/onramp/create-session",
    response_model=CreateOnrampSessionResponse)
@router.post("/api/v1/linktoken", response_model=LinkTokenResponse)
@router.post("/api/v1/moonpay/signature",
    response_model=MoonPaySignatureResponse)
```

Orders

These are operational endpoints for bridging, placing/modifying/canceling orders, approvals, transfers, and leverage updates. Most endpoints take either the wallet from the request or from the auth token if available.

Most requests contain a `privy_signer_private_key` that is used to sign the action. After signing, the action gets sent to Hyperliquid by the server.

```
@router.post("/api/bridge_funds_to_arbitrum",
    response_model=BridgeFundsResponse)
@router.post("/api/bridge_funds_to_hyperliquid",
    response_model=BridgeFundsResponse)
@router.post("/api/withdraw_from_hyperliquid",
    response_model=WithdrawFundsResponse)
@router.post("/api/agent_perps_order", response_model=PerpsOrderResponse)
@router.post("/api/privy_perps_order", response_model=PerpsOrderResponse)
@router.post("/api/agent_spot_order", response_model=SpotOrderResponse)
@router.post("/api/agent_perps_limit_order",
    response_model=PerpsOrderResponse)
@router.post("/api/agent_perps_close", response_model=PerpsOrderResponse)
@router.post("/api/privy_perps_close", response_model=PerpsOrderResponse)
@router.post("/api/approve_builder_fee",
    response_model=ApproveBuilderFeeResponse)
@router.post("/api/approve_agent_wallet",
    response_model=ApproveAgentWalletResponse)
@router.post("/api/send_usdc", response_model=USDCTransferResponse)
@router.post("/api/close_all_spot_positions",
    response_model=CloseAllSpotPositionsResponse)
@router.post("/api/spot_perp_transfer",
    response_model=SpotToPerpTransferResponse)
@router.post("/api/agent_modify_tpsl", response_model=AgentModifyTpSlResponse)
@router.post("/api/cancel_limit_order",
```

```
    response_model=CancelLimitOrderResponse)
@router.post("/api/agent_twap_order", response_model=AgentTwapOrderResponse)
@router.post("/api/agent_cancel_twap_order",
    response_model=AgentCancelTwapOrderResponse)
@router.post("/api/update_leverage", response_model=UpdateLeverageResponse)
```

Referrals

The app implements a referral system. The referral relationship is stored in the main Prisma database using the `Referral` table. The reward from referrals is 50% of the fees and limited to 30 days. There is a `ReferralRewards` table for rewards earned through trades of referred users.

Users accumulate rewards and can pay them out using the `/api/referral/claim` endpoint. They will get paid out from a wallet controlled by the client, and the private key is passed by an environment variable called `REFERRER_PRIVATE_KEY`. The request is signed on the server. The endpoint uses the `wallet_address` of the user from the session token.

According to the client, the rewards wallet is manually funded and contains limited assets, which helps minimize damages in the event of unauthorized access or misuse.

```
@router.post("/api/referral/claim", response_model=ClaimRewardsResponse)
@router.get("/api/referral/stats", response_model=ReferralStatsResponse)
```

Trades

This is the route to get user trades and stats that are stored in the Prisma database.

```
@router.get("/api/trades", response_model=TradeHistoryResponse)
@router.get("/api/trades/stats", response_model=TradeStatisticsResponse)
```

Trending

These are the routes that return trending tokens. The test route returns a static list, while the non-test one fetches the trending tokens from the database, ordered by volume ratio (volume now compared to 24h ago). The database is populated by a cloud function `get_trending_tickers()` every three minutes.

```
@router.get("/api/trending/tokens_test",
    response_model=TrendingTokensResponse)
@router.get("/api/trending/tokens", response_model=TrendingTokensResponse)
```

Unit bridge

This endpoint returns a deposit address using HyperUnit; see the HyperUnit deposit documentation ↗.

```
@router.post("/api/bridge_unit", response_model=DepositAddressResponse)
```

Vault

This endpoint moves funds to and from the Hyperliquid fund. Similarly to order routes, it takes the `privy_signer_private_key`, signs the action, and sends it to the Hyperliquid API.

```
@router.post("/api/vaults/vault_transfer",
    response_model=TransferVaultResponse)
```

Voice

This endpoint connects to Pipecat voice service and daily room. It uses an environment secret `PIPECAT_CLOUD_API_KEY`. It does not seem to be used by the front end; instead, the front end uses https://prometheus-prod-liquidmax-voice-assistant-fastapi-app.modal.run/connect.

```
@router.post("/api/voice/connect", response_model=VoiceConnectResponse)
    #unused?
```

Wallet

The API endpoint `/api/wallet/authenticate` forwards to the Elysia server endpoint `/authenticate`. The route is used to get the `privySignerPrivateKey` from a Privy `accessToken`. This route is unused; the functionality to get the token seems to be implemented in the Swift app.

Both `balance` and `multi-balance` forward to Elysia and perform balance checks for the `wallet_address` using Infura RPC.

```
@router.get("/api/wallet/elysia-health", response_model=HealthResponse)
    #unused
@router.get("/api/wallet/balance/{wallet_address}",
    response_model=SingleBalanceResponse)
@router.get("/api/wallet/multi-balance/{wallet_address}",
    response_model=MultiBalanceResponse)
@router.post("/api/wallet/authenticate", response_model=AuthenticateResponse)
    #unused
```

Out of scope

For completeness, these are the routes that we were asked to not review.

```
@router.post("/api/access-code/validate",
    response_model=ValidateAccessCodeResponse)
@router.post("/api/leaderboard/2048/update")
@router.get("/api/leaderboard/2048", response_model=LeaderboardResponse)
@router.get("/api/leaderboard/2048/my-score")
@router.get("/api/leaderboard/2048/score/{wallet_address}")
```

Microservices and cloud functions

In addition to the main app, there are multiple microservices (modal functions).

liquidserver/microservices/btc_price_notifications.py

- **Function `check_crypto_milestones`:** Runs every five seconds. Uses Hyperliquid info API to get BTC and PUMP price and sends out notification via `OneSignal` on certain conditions.
- **Function `test_notification`:** For testing.

liquidserver/microservices/truth_social.py

- **Function `monitor_trump_posts`:** Gets trump truth posts from https://trumpstruth.org/feed and stores them in the main database. Then adds between five and 15 likes and dislikes. Uses an LLM to figure out if a tweet is relevant to markets. If it is relevant, it will send out a notification using `OneSignal`. Runs every five minutes.
- **Function `backfill_trump_truth_posts`:** Backfills older tweets into database.

liquidserver/microservices/trending_tokens.py

- **Function `get_trending_tickers`:** Fetches info from Hyperliquid info API and stores it in the database. Will find trending tokens (tokens that had their volumes increase the most relatively) and store them in another table in a database. Runs every three minutes.

liquidserver/microservices/rss_feeds.py

- **Function `summarize_rss_post`:** Uses OpenRouter to use an LLM (GPT-4o) to classify if it is relevant to markets or not and why.

- **Function `monitor_rss_feeds`:** Runs every 15 seconds, fetches two RSS feeds, uses an LLM to classify relevance, stores new tweets in database, and adds 5–15 dislikes/likes.

liquidserver/microservices/new_listings.py

- **Function `new_listings`:** Runs every 15 seconds, gets info from Hyperliquid, and sends out notifications when listings are removed/added and when leverage is updated.

liquidserver/microservices/mini_charts.py

- **Function `update_charts`:** Runs every 15 minutes. Calls /nodeserver/scripts/charts.ts. This script will update all candles using data from Hyperliquid. The data is added to the charts database using `sequelize`.

liquidserver/microservices/market_signals.py

- **Function `monitor_market`:** Runs every five minutes. Monitors market for big market moves, uses Prisma database to stort big moves, and sends out notifications on big moves with cooldown.

liquidserver/microservices/market_signal_prices.py

- **Function `update_prices`:** Runs every five minutes. Runs the prices.ts script with bun. Updates candle data.

liquidserver/microservices/deposit_monitor.py

- **Function `monitor_deposits`:** Runs continously as a server. Starts monitoring.ts with bun. Watches for transfers of all user wallets and sends out notifications when a transfer occurs (over a minimum amount).

liquidserver/microservices/data_populator/monitor_pnl_tokyo.py

- **Function `monitor_pnl`:** Runs every five minutes. Calculates PNL for all users and stores it in ClickHouse.
- **Function `drop_pnl_older_than_two_days`:** Runs every 12 hours and deletes PNL older than two days.

liquidserver/microservices/data_populator/monitor_mids_tokyo.py

- **Function `monitor_mids`:** Runs every five minutes and inserts `perps_mids_millis` into ClickHouse.

liquidserver/microservices/data_populator/monitor_candles_tokyo.py

- **Function `drop_data_older_than_45_days`:** Runs every midnight. Drops all candlestick records older than 45 days from the `perps_full_candles` table in ClickHouse.
- **Function `refresh_new_candles`:** Runs every 15 minutes. Inserts new `perps_full_candles` into ClickHouse database for each coin.
- **Function `full_dump`:** Is manually triggered. For historical backfill.

liquidserver/microservices/data_populator/cleanup_data.py

- **Function `drop_mids_older_than_two_days`:** Runs every day. Deletes old `perps_mids_millis` data from ClickHouse.

liquidserver/microservices/big_moves.py

- **Function `big_moves`:** Runs every 30 seconds. Sends out notifications on big moves.

liquidserver/microservices/alt_big_moves.py

- **Function `big_moves`:** Runs every 30 seconds. Sends out notifications on big moves. Has different criteria for big moves for different tiers of coins.

Other

For brevity, we are omitting the descriptions of

- multiple cloud functions in /benchmark.py, standalone_clickhouse_client.py, standalone_spot_orders.py, standalone_agent_perps_benchmark.py, and standalone_parallel_agent_perps_benchmark.py for benchmarking
- standalone_twap_orders.py and tests/proxy.py for testing
- `data_populator` scripts that are deduplicated (`tokyo`)
- access_code_reseeder.py (`scheduled_reseed_access_codes`, `manual_reseed_access_codes`, `get_access_code_stats`, `cleanup_old_access_codes`) — access codes were removed from the application so we omit them here

Invariants

- JWTs must match the expected structure
- Privy tokens must be verified using ES256 with the correct audience and issuer
- Users must successfully authenticate with Privy to log in to their accounts
- Usernames, wallet addresses, and referral codes must be unique
- Access codes can only be used once
- Each user can be referred by a maximum of one person
- Referral rewards last 14 days

Test coverage

Cases covered

- Data API tests (volume candles/candlesticks API positive and negative testing)
- News tests (positive and negative testing for news and "like" reactions)
- Order management tests (positive and negative testing for a variety of cases around orders)
- Service account tests (service account address validation and private key existence)
- Vault transfer tests (positive and negative test cases for transferring to/from vaults)
- Candlestick query tests (tests functionality around querying and plotting data from ClickHouse)
- User state testing
- Spot value calculation testing
- Deposit monitoring testing

Cases not covered

- Several API routes are untested entirely (auth.py, charts.py, generate.py, leaderboard.py, onramp.py, referrals.py, trades.py, trending.py, unit_bridge.py, wallet.py, access_codes.py)
- Several areas of core business logic are lacking in test coverage (core/auth.py, core/encryption.py, core/error_handlers.py, core/exchange.py, core/feature_flags.py, core/notifications, core/services.py)
- Database models are mostly untested
- Most microservices are untested
- Nodeserver functionality is largely untested
- Integration tests (real database operations, cross-service interactions, external API integrations, full API workflows)
- Security tests (input validation, authentication/authorization, rate limiting, data privacy, race conditions)
- Configuration/environment tests (environment variables, service availability)

Attack surface

- Authentication and authorization mechanisms.
- API routes accepting user input properly validating and sanitizing input where needed.
- HTTP servers properly validating HTTP headers, query parameters, `POST` bodies, and so forth to ensure that data is properly parsed and parser discrepancies cannot be abused.
- Supply-chain security, ensuring that only secure dependencies are in use.
- Secure secret storage, ensuring that secrets are not insecurely stored in places like source code.
- Race conditions, ensuring that timing/orders of requests cannot be abused to bypass security mechanisms.

5.2. Component: Swift app

Description

The Swift application is a mobile application front end for LiquidMax written in Swift. This is the main user interface for the LiquidMax project, allowing users access to multiple views and utilities. The main views of the application are a portfolio page allowing users to track their portfolios; a news feed page that displays news in real time, which could impact markets and explains why; a vault page that allows users to move funds to or from their vault; and a referrals page that allows users to refer others and cash out their rewards for doing so. The application also includes a settings page to allow a user to configure their application or username and additionally includes buttons for functionality to log out or delete their account or external links to the project's wallet recovery site or legal/privacy documents.

The application also includes deep links for navigating directly to certain views via a custom `buyhype://` URL.

New users onboard and connect an external wallet or create a new one via Privy, which also handles authentication.

Biometrics are used to facilitate secure access in the event of unauthorized access directly to a user's mobile device.

A secure enclave is used to store sensitive data with an added layer of security.

Invariants

- Biometric authentication must be performed to access the application.
- Privy authentication must be performed to access a user's account.
- Data displayed in the application views must not be modifiable by an outside source.

Test coverage

Cases covered

- Financial logic (basic operations, precision handling, price validation, currency formatting, and edge cases).
- API integration (ensuring successful/expected API requests/responses).
- Error handling (ensuring errors are properly handled in the event of insufficient funds, invalid input, or API failure responses).

Cases not covered

- User interface functionality performing as expected.
- Authentication mechanisms performing properly.
- Deep link behavior.

Attack surface

- Attacker-controllable deep links must not be used to perform automatic or unexpected actions for a user.
- Secrets must be securely stored such that they are not made available to untrusted sources.
- Local authentication must not be bypassable.

5.3. Component: Recovery site

Description

The recovery site is a small web application that allows a user to recover their wallet via Privy if they lost access. The site is written in TypeScript with Next.js, using React for the front end and Privy for authentication. Upon authenticating with Privy, a user can export their wallet.

Invariants

- A user must successfully authenticate with Privy to access their wallet recovery.

Test coverage

Cases covered

- The recovery site does not currently cover any test cases.

Cases not covered

- Testing proper authentication flows.

- Ensuring front-end views behave as expected.

Attack surface

- Supply chain security (not using any insecure dependencies).
- Privy token verification.
- Secret storage (securely storing secrets like `PRIVY_APP_SECRET` outside of source code).

6. Assessment Results

During our assessment on the scoped LiquidMax contracts, we discovered 11 findings. No critical issues were found. Three findings were of high impact, four were of medium impact, three were of low impact, and the remaining finding was informational in nature.

At the time of our assessment, portions of the LiquidMax codebase appeared somewhat disorganized, contributing to a less-than-optimal security posture. Particularly, in the backend server repository, although code was clear, concise, and mostly well-written, there were several instances in which bugs existed and instances of code that were redundant or incorrectly commented. Although many of the aforementioned occurrences are not critical issues (or even security issues in every instance), their existence gives a combined impression that additional care should be given during the development process to ensure that any new code is adequately reviewed before being merged. This could include an increased emphasis on peer reviews as well as improved test cases and automation pipelines to catch issues early.

Outside of the backend server, the Swift application and Privy recovery site components appear to be in a much healthier state of security. Once issues identified via this audit are remediated and test cases plus automation are expanded, we believe that LiquidMax will be in a position of strong security posture.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.