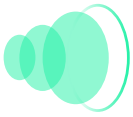
CODESPECT

Hyperwave Stablecoin Oracle and Off-Chain Bot

SECURITY ASSESSMENT REPORT

4 July, 2025

Prepared for Hyperwave



Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the security assessment conducted by CODESPECT for the smart contracts of Hyperwave. Hyperwave is a liquid token that accrues yield from the Hyperliquidity Provider Vault (HLP). hwHLP lets users profit from market making and liquidations on Hypercore, while retaining their liquid assets for DeFi on HyperEVM and beyond.

This audit focuses on the following components of the Hyperware system:

a) Redstone oracle integration used for obtaining pricing necessary for USDe/USDC and USDT0/USDC deposits and withdrawals

b) BoringVault Atomic Solver and Queue contracts.

c) Python off-chain bot handling Atomic Solver settlements.

The audit was performed using:

a) Manual analysis of the codebase.

CODESPECT found twenty three points of attention, one classified as `High`, three classified as `Medium`, three classified as `Low`, twelve classified as `Informational`, and four classified as `Best Practices`. All of the issues are summarised in Table 2.

Organization of the document is as follows:

- **Section 5** summarizes the audit.
- **Section 6** describes the functionality of the code in scope.
- **Section 7** presents the issues.
- **Section 8** discusses the documentation provided by the client for this audit.
- **Section 9** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
High	0	1	0
Medium	0	3	0
Low	0	3	0
Informational	0	11	1
Best Practices	0	3	1
Total	**0**	**21**	**2**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	Hyperwave
Type of Project	Liquid token protocol
Duration of Engagement	11 Days
Duration of Fix Review Phase	2 Days
Draft Report	July 1, 2025
Final Report	July 4, 2025
Repository (boring vault)	boring-vault
Repository (off-chain bot)	hlp-internal-be
Commit (Audit - stablecoin oracle)	ce21e49b7a7be06c3a96c3c3ea6982c32e3ff224
Commit (Audit - off-chain bot)	34fd78ae73c542ec3ca2fa34b499a28736acfa7e
Commit (Audit - queue and solver contracts)	cff94e7093a688578d13279748cdaa0d2b308293
Commit (Final - stablecoin oracle)	cf79b25a83903b2c8f0af38c7047b7bc95174c3a
Commit (Final - off-chain bot)	d1a507cdf5ccd5b67bc98c7eb921701de4cd3efb
Commit (Final - queue and solver contracts)	cf79b25a83903b2c8f0af38c7047b7bc95174c3a
Documentation Assessment	High
Test Suite Assessment	Low
Auditors	Talfao, JecikPo, 0xMrjory, Shaflow01

Table 3: Summary of the Audit

5.1 Scope - Audited Files

5.2 Contracts files:

	Contract	LoC
1	AtomicSolverV3.sol	118
2	AtomicQueue.sol	253
3	RedstoneStablecoinRateProvider.sol	137
	Total	**673**

5.3 Bot files:

	File	LoC
1	boring_vault.py	228
2	accountant.py	59
3	atomic_request.py	56
4	atomic_solver_v3.py	57
5	atomic_request.py	21
6	atomic_queue.py	21
7	rate_provider.py	13
8	__init__.py	9
	Total	**464**

5.4 Findings Overview

	Finding	Severity	Update
1	Insufficient result validation in HyperLiqud SDK response	High	Fixed
2	Incorrect price validity check	Medium	Fixed
3	Incorrectly assumes that the USDE/USDC rate will not exceed 1 most of the time	Medium	Fixed
4	Potential misreporting of vault balance	Medium	Fixed
5	Insufficient offchain filtering for withdrawal requests	Low	Fixed
6	Mix of sync/async causes latency in withdrawal processing	Low	Fixed
7	The multi-chain `solve_withdrawal` logic is too tightly coupled	Low	Fixed
8	Additional check for `isAtomicRequestValid(...)`	Info	Fixed
9	Check user's offer balance across requests to prevent unnecessary reverts	Info	Fixed
10	Improve front-running mitigations for `solve(...)`	Info	Fixed
11	Missing Database Error Handling	Info	Acknowledged
12	Missing alert on unfavourable exchange rate update	Info	Fixed
13	Non-unique addresses may cause balance miscalculation	Info	Fixed
14	Note Error	Info	Fixed
15	Exponential calculations may suffer from precision loss	Info	Fixed
16	Validity checks after price type conversion may fail to throw the expected error	Info	Fixed
17	The on-chain exchange rate update failure will not terminate the WithdrawalSolver task	Info	Fixed
18	Some calls within the WithdrawalSolver task lack a retry mechanism	Info	Fixed
19	`update_exchange_rate_all_chains` may result in partial exchange rate updates	Info	Fixed
20	Consider applying parameter validation	Best Practices	Fixed
21	Consider escaping data in `send_mg` function	Best Practices	Fixed
22	Consider refactoring textual SQL to ORM queries	Best Practices	Acknowledged
23	Missing oracle best-practice checks	Best Practices	Fixed

6 Hyperwave Update Information

The following subsections provide an overview of the three main system components of the audit scope:

- **Boring Vault Off-Chain Bot**
- **Atomic Solver and Queue contracts**
- **Redstone Stablecoin Oracle**

6.1 Boring Vault Off-Chain Bot

The `boring_vault.py` file defines the `BoringVaultService` class, which acts as the main service layer for interacting with the `AtomicSolverV3` contract.

The following points describe the Boring Vault Bot operations on high level:

- **Centralises Solver Logic:** It encapsulates all the business logic related to the Boring Vault and Solver contracts, including exchange rate calculations, and handling withdrawal requests.
- **Interacts with Contracts:** It communicates with various smart contracts (like ERC20 tokens, rate providers, accountants, and solvers) to manage assets and execute vault operations.
- **Coordinates Off-chain and On-chain Data:** It combines on-chain contract data with off-chain configuration and state.

The Bot provides the following functions:

- **Initialization:** On creation, it loads configuration from `base_config`, including vault addresses, token lists, and contract addresses. It sets up contract instances for the vault, tokens, rate providers, and other components needed for operations.
- **Asset Management:** Methods like `get_token_balances(...)` aggregate the balances of all tokens managed by the vault, both on-chain (vault contract) and in associated multisig wallets. It normalizes these balances using either on-chain oracles or rate providers to get the total asset value in a common unit.
- **Exchange Rate Calculation:** It calculates the current exchange rate for vault shares based on total assets and total supply, applying configurable buffers. The exchange rate for the boring vault is updated on-chain via the accountant contract.
- **Withdrawal Handling:** It manages withdrawal requests, which are stored on-chain (from the `AtomicQueue` contract). It validates requests, calculates how much can be withdrawn, and executes withdrawals using the `AtomicSolverV3` contract. It ensures the vault has enough balance to fulfill requests and sends alerts via Telegram if not.
- **Utility Operations:** It provides helper methods for transferring small leftover balances (“dust”) from the solver back to the vault. It logs key actions and balances for monitoring and debugging.

In essence, `boring_vault.py` is the operational core for the Boring Vault off-chain bot, it is responsible for updating the exchange ratio for the hwHLP and for solving withdrawal requests.

6.2 Atomic Solver and Queue contracts

The `AtomicQueue` contract is a smart contract that manages user withdrawal requests from Hyperwave. Users can submit "atomic requests" specifying which ERC20 token they want to offer (e.g., vault shares) and which token they want in return (e.g., USDe, USDT0 and USDhl), along with the amount, minimum price, and deadline. The contract stores these requests, validates them, and allows for batch processing (solving) of multiple user requests at once. It includes features for pausing/unpausing the contract, updating requests, and checking if requests are valid.

The `AtomicSolverV3` contract acts as an external "solver" that processes the withdrawal requests stored in the `AtomicQueue`. It implements logic to fulfil these requests by redeeming vault shares for underlying assets and distributing them to users. The solver ensures that all constraints (like minimum received amounts and maximum assets spent) which are supplied from the off-chain bot. It interacts with the queue to execute the batch, handles asset transfers, and can also rescue tokens if needed. The contract is designed to be called by authorised parties (The Boring Vault Off-Chain Solver Bot).

In summary:

- `AtomicQueue` collects and manages user withdrawal/swap requests.
- `AtomicSolverV3` processes these requests in batches, executing the actual asset transfers according to user specifications and system rules.

Together, they enable efficient, atomic, and flexible withdrawals or swaps in a multi-asset Hyperwave system.

6.3 Redstone Stablecoin Oracle

The RedstoneStablecoinRateProvider contract is an on-chain oracle adapter that calculates and provides the exchange rate between two stablecoins using Redstone-compatible price feeds.

The following points list high-level functionality of the oracle adapter:

- **Price Calculation:** The contract fetches the USD price of both the base and quote assets from their respective Redstone price feeds. It then computes the exchange rate between the quote and base assets as QUOTE/USD ÷ BASE/USD, adjusting for decimals and capping the rate within configured bounds.
- **Staleness and Safety Checks:** It ensures that the price data from both feeds is recent (not stale) and that the sequencer (for L2 chains) is up and running, with a grace period after downtime to prevent manipulation or mass liquidations.
- **Bounds and Limits:** The contract enforces lower and upper bounds on the computed rate to prevent extreme or erroneous values, which helps protect users and the protocol from oracle manipulation or feed errors.
- **Configurability:** Administrators can update the lower bound, maximum rate, and sequencer downtime parameters to adapt to changing market or infrastructure conditions.

7 Issues

7.1 [High] Insufficient result validation in HyperLiqud SDK response

File(s): `hypercore.py`

Description: The program uses the Hyperliquid Python SDK to request on-chain data, but the returned data is not properly validated. In the Hyperliquid Python SDK, the functions ultimately construct corresponding POST requests to the Hyperliquid API and then return the response data.

Below snippet from `Hyperliquid API`

```
def post(self, url_path: str, payload: Any = None) -> Any:
    payload = payload or {}
    url = self.base_url + url_path
    response = self.session.post(url, json=payload)
    self._handle_exception(response)
    try:
        return response.json()
    except ValueError:
        return {"error": f"Could not parse JSON: {response.text}"}

def _handle_exception(self, response):
    status_code = response.status_code
    if status_code < 400:
        return
    if 400 <= status_code < 500:
        try:
            err = json.loads(response.text)
        except JSONDecodeError:
            raise ClientError(status_code, None, response.text, None, response.headers)
        if err is None:
            raise ClientError(status_code, None, response.text, None, response.headers)
        error_data = err.get("data")
        raise ClientError(status_code, err["code"], err["msg"], response.headers, error_data)
    raise ServerError(status_code, response.text)
```

After receiving the response, the `_handle_exception` function is called to handle the status code. If the status code is less than 400 and the response fails to parse as JSON, it will return an error message in JSON format.

In this case, since HyperCore does not filter or validate this result, it may mistakenly assume that the returned data is valid and proceed with parsing and execution.

Impact: If data retrieval fails, the program may continue executing, which can result in incorrect or failed exchange rate updates. For example, when calling `get_spot_balances`, if `spot_user_state` returns an error JSON response, the function will not treat it as a failure. Instead, it will return an empty list, and subsequent operations will continue executing. This may lead to incorrect exchange rate calculations.

Recommendation(s): It is recommended to first check if the `error` field exists in the response data. If it does, throw an exception to terminate the task.

Status: Fixed

Update from Hyperwave: Fixed in PR-22.

7.2 [Medium] Incorrect price validity check

File(s): RedstoneStablecoinRateProvider.sol

Description: When checking the price validity of the quote token, the code mistakenly uses MAX_TIME_FROM_LAST_UPDATE_BASE_FEED instead of MAX_TIME_FROM_LAST_UPDATE_QUOTE_FEED.

```
function getRate() public view returns (uint256 rate) {
     //...
    (, int256 _quoteRate,, uint256 lastUpdatedAtQuote,) = PRICE_FEED_QuoteFeed.latestRoundData();

    if (
        lastUpdatedAtQuote > block.timestamp
            || block.timestamp - lastUpdatedAtQuote > MAX_TIME_FROM_LAST_UPDATE_BASE_FEED
    ) {
        revert MaxTimeFromLastUpdatePassed(block.timestamp, lastUpdatedAtQuote);
    }
    //...
}
```

Impact: This causes the quote token price validity period to deviate from the expected duration.

Recommendation(s): Use MAX_TIME_FROM_LAST_UPDATE_QUOTE_FEED to validate the freshness of the quote token price.

Status: Fixed

Update from Hyperwave: Fixed in PR-6.

7.3 [Medium] Incorrectly assumes that the USDE/USDC rate will not exceed 1 most of the time

File(s): RedstoneStablecoinRateProvider.sol

Description: The getRate() function obtains the price from the Redstone price feed and then calculates the rate, but both the price of USDE and the resulting rate are capped at 1.

```
function _calculateRate(uint256 baseRate, uint256 quoteRate) internal view returns (uint256) {
    //...
    uint256 oneQuoteRate = 10 ** quoteRateDecimals;
    uint256 minQuoteRate = quoteRate > oneQuoteRate ? oneQuoteRate : quoteRate;

    uint256 rate = (minQuoteRate * SCALE) / baseRate;

    return rate > ONE ? ONE : rate;
}
```

The team's response was:

> In some cases, the price of USDE/USDC can be slightly higher than 1. We don't want to mint more HLP than necessary, especially since we know that prices will revert to 1 at some point.

However, based on actual observations of the Redstone oracle, the price of USDE remains around 1.0000 to 1.0001 for extended periods, while the price of USDC stays around 0.9997 to 1.0000. For example, the USDE price from June 2nd to June 9th consistently stayed above 1. This means that the USDE/USDC ratio should be greater than 1 under stable conditions, not equal to 1 as the team expects. As a result, the exchange rate is underestimated under normal conditions.

Impact: Most of the time, the exchange rate will be lower than the actual rate.

Recommendation(s): Adjust the maximum cap slightly higher.

Status: Fixed

Update from Hyperwave: Fixed in PR-6 by adding MAX_RATE, which can be changed by the governor.

7.4 [Medium] Potential misreporting of vault balance

File(s): `hypercore.py`

Description: The bot, when calculating the new exchange ratio, takes into account all balances across the ecosystem. One of these balances is the multi-sig balance held in the HLP vault, which is fetched via an API call and returned by the following function:

```
def get_vault_equities(self, address: str) -> Dict[str, any]:
    request_data = {"type": "userVaultEquities", "user": address}
    equities = self.exchange.info.post("/info", request_data)
    if not equities or len(equities) == 0:
        return {
            "vaultAddress": base_config.hyperliquid.vault_address,
            "equity": "0",
            "lockedUntilTimestamp": 0,
        }
    return equities[0]
```

This function returns only the first element of the `equities` array. The array itself consists of dictionaries, each representing vault deposits associated with a given wallet.

Although the protocol team has stated that deposits will only be made to the HLP vault, a mistake (e.g., a deposit to a different vault) or an external actor depositing on someone else's behalf (in the context of Hyperliquid's closed system) could alter the content of the array. As a result, an incorrect vault balance may be returned, potentially skewing the accounting logic.

Note: The CODESPECT team cannot confirm whether depositing on someone else's behalf is possible. Based on the available documentation, this is not allowed or supported, which reduces the severity of the issue.

Impact: Incorrect accounting of vault balances may lead to an inaccurate exchange ratio calculation, ultimately devaluing user shares.

Recommendation(s): Filter and return only the equity associated with the specific HLP vault address to ensure accurate balance reporting.

Status: Fixed

Update from Hyperwave: Fixed in 0179e8db97501b7b0f29d628ad6df1769432f3cd.

7.5 [Low] Insufficient offchain filtering for withdrawal requests

File(s): `boring_vault.py`

Description: Before processing withdrawal requests, the bot performs offchain filtering to minimize the number of RPC calls. One of these filters involves verifying the validity of a request using the `AtomicQueue`, specifically via the `isAtomicRequestValid(...)` function.

However, the current implementation of the filtering only checks whether a request has been marked as solved or not. This is insufficient, as it does not reliably indicate whether the request has been *properly* solved.

The filtering logic should be improved by introducing the following checks:

- **`offerAmount > 0`**: This ensures that the request has actually been filled;
- **Deadline not exceeded**: Requests whose deadlines have passed should be skipped to avoid unnecessary RPC calls;
- **`offer` token equals the `boringVault` address**: If the `offer` token does not match the expected vault address, the withdrawal will revert. This check is currently missing from the `isAtomicRequestValid(...)` validation;

Impact: Failure to implement these checks may lead to unnecessary latency when solving withdrawal requests and could cause transaction reverts.

Recommendation(s): Enhance the offchain filtering logic to include:

- A check for `offerAmount > 0`;
- Validation that the deadline has not been exceeded;
- Confirmation that the `offer` token matches the `boringVault` address;

Status: Fixed

Update from Hyperwave: Fixed in 4ca30a19ba0e76c77ceff2c3ce7084133018b189.

7.6 [Low] Mix of sync/async causes latency in withdrawal processing

File(s): boring_vault.py

Description: In the solve_atomic_requests() function, the following line is used to send a Telegram alert:

```
asyncio.run(
  self.telegram_bot.send_msg(
      base_config.telegram.group_chat_id,
      f"Boring Vault {self.boring_vault_address} has not enough balance to solve withdrawal requests for
      ↪ {want_contract.symbol()}: "
      f"Required: {want_contract.from_wei(minimum_assets_out)} {want_contract.symbol()} "
      f"Vault balance: {want_contract.from_wei(boring_vault_balance)} {want_contract.symbol()}",
  )
)
```

This implementation mixes asynchronous and synchronous programming, which introduces several concerns:

- asyncio.run() creates a new event loop and blocks the current thread until completion.
- It adds unnecessary overhead just for a single async call.
- If the Telegram API call is slow or fails, it can block or slow down the entire withdrawal request handling.

Impact: Potential performance degradation or blocking of the withdrawal processing flow due to external API delays.

Recommendation(s):

- Offload the Telegram call using a background thread or queue-based notification mechanism.
- Alternatively, refactor the logic to be fully synchronous with proper try/except handling to isolate potential Telegram errors from the main logic.

Status: Fixed

Update from Hyperwave: Fixed in 41378fa9b9fbee9b334ee0e27163e060dfa8f0cb by creating new function for sending Telegram message synchronously.

7.7 [Low] The multi-chain `solve_withdrawal` logic is too tightly coupled

File(s): `boring_vault.py`

Description: Currently, the multi-chain `solve_withdrawal` logic is too tightly coupled. A single-node failure on any one chain can easily cause the entire task to fail, preventing order fulfillment across all chains.

For example:

- `solve_withdrawal` relies on all oracles and rate providers across all chains functioning correctly, and on prices not being outdated. If the oracle for any asset on any chain fails to meet the required conditions, the task will terminate, preventing order execution on all chains. This is because calculating and update the exchange rate involves calling the oracles and rate providers for all assets across all chains. A single oracle failure can prevent order fulfillment across all chains. For instance, if a rateProvider on one chain is unable to provide a valid price due to stale price data from its source, orders on other chains will also fail to be fulfilled.
- During the `solve_atomic_requests` process, when fulfilling orders on each chain, if an RPC exception, network instability, or transaction failure occurs while calling a contract on any chain, the process will immediately terminate instead of skipping and continuing to fulfill orders on the remaining chains. This causes partial execution.

```
def solve_atomic_requests(
    self, chain_id: str, offer: str, want: str, requests: List[AtomicRequest], rate_by_token: Dict[str, int]
):
    // ...
    want_contract.approve_if_needed(solver_account, vault_config.atomic_solver_v3_address, minimum_assets_out)
    self.atomic_solver_v3.redeem_solve(
        chain_id=chain_id,
        atomic_queue_address=vault_config.atomic_queue_address,
        teller_address=vault_config.teller_address,
        offer=offer,
        want=want,
        users=users,
        minimum_assets_out=minimum_assets_out,
        max_assets=max_assets,
    )
    self.transfer_dust_from_solver_to_vault(chain_id)
```

Impact: The strong coupling makes the entire system vulnerable to single points of failure.

Recommendation(s): It is recommended to decouple order fulfillment across different chains, allowing partial fulfillment in some cases. Additionally, providing fallback oracles or alternative pricing methods can help reduce the impact when the primary oracle is down.

Status: Fixed

Update from Hyperwave: Fixed in PR-22.

7.8 [Info] Additional check for `isAtomicRequestValid(...)`

File(s): `AtomicQueue.sol`

Description: The `isAtomicRequestValid(...)` function is used to validate user-submitted withdrawal requests. It performs several checks, such as verifying the user's balance and ensuring they have approved the Queue contract to spend their `offer` tokens.

However, the function does **not** verify whether the `offer` token is the Boring Vault share token. This check is crucial and should be added to ensure complete validation redundancy, as recommended for offchain filtering.

Impact: Invalid requests may delay the processing of valid ones, since this function plays a critical role in the bot's request validation logic.

Recommendation(s): Add a check to confirm that the `offer` token is the Boring Vault share token.

Status: Fixed

Update from Hyperwave: Fixed in d3d7f57116f1c366062daf6ddd762e18d9df12de.

7.9 [Info] Check user's offer balance across requests to prevent unnecessary reverts

File(s): `boring_vault.py`

Description: Users create atomic requests via `safeUpdateAtomicRequest(...)`, which verifies whether the user has sufficient `offer` balance and allowance to the `AtomicQueue` contract. This is intended to ensure only valid requests are created and that the user has enough Boring Vault shares.

Later, the Hyperwave solver bot checks request validity again via `isAtomicRequestValid(...)`, which performs similar validations.

However, these checks are insufficient and allow for a specific edge case that may cause the bot to raise an error and delay solving of otherwise valid requests. Here's an example:

- A user holds `10` shares;
- The user submits two requests: A (`want = USDe`) and B (`want = USDT`), both with `offerAmount = 10`;
- Both requests pass the balance and allowance checks at submission time and during bot validation;
- The bot includes both requests in its solving plan;
- The batch for request A is solved successfully;
- When attempting to solve the batch containing request B, the user's share balance is already depleted (0), causing the transaction to revert;

As a result, the bot encounters a failure when executing the second batch, leading to unnecessary delays in processing valid requests.

Impact: Potential failure of the bot and delay in solving requests due to inconsistent request accounting.

Recommendation(s): Consider adding logic on the bot side to track cumulative user balances across all pending requests, and only include those that can be fully covered.

Status: Fixed

Update from Hyperwave: Fixed in 5eedc610bb1d0e9df64e366067ccbc297cc3f9c0.

7.10 [Info] Improve front-running mitigations for `solve(...)`

File(s): `AtomicQueue.sol`

Description: The current design of `AtomicQueue` is prone to front-running attacks, as acknowledged in the `@dev` comment of the `solve(...)` function: _"It is very likely `solve` TXs will be front run if broadcasted to public mem pools, so solvers should use private mem pools."_

When `solve` is called, an attacker may update their withdrawal request in the same block to either:

- Gain a more favourable price ratio, or;
- Cause the entire batch to revert by exceeding the `maxAssets` limit supplied by the solver bot;

This introduces a potential attack surface that could disrupt the batch execution or unfairly benefit malicious actors.

To mitigate this, the `AtomicQueue` design should be adjusted to include a delay mechanism. For example, a request must exist in the system for a minimum duration (e.g., X seconds) before it becomes solvable.

Impact: Front-running of atomic request updates could lead to unfair price execution for a user or complete transaction failure due to batch reverts.

Recommendation(s): Consider enforce a rule that requests can only be solved after a certain time period has passed since their submission.

Status: Fixed

Update from Hyperwave: fixed in 1503d53e5517df71602468a30984860ddd3064e4 (contract side) and 5eedc610bb1d0e9df64e366067ccbc297cc3f9c0 (off-chain bot side)

7.11 [Info] Missing Database Error Handling

File(s): `atomic_request_repo.py`

Description: The `AtomicRequestRepo` class lacks error handling for database operations. All database calls (`get_by_id(...)`, `get_by_is_solved(...)`, `create(...)`) can throw SQL exceptions that will propagate unhandled to the calling code.

Impact:

- **Silent failures** - `get_by_id(...)` returns `None` for both "not found" and "database error" scenarios;
- **Poor debugging** - no logging of database errors;

Recommendation(s): Add comprehensive error handling with proper exception categorization:

```
def create(self, request: AtomicRequest) -> AtomicRequest:
    try:
        with Session(self.db_engine) as session:
            session.add(request)
            session.commit()
            session.refresh(request)
            return request

    except IntegrityError as e:
        logger.error(f"Constraint violation creating request: {e}")
        if "duplicate key" in str(e).lower():
            raise ValueError("Request already exists")
        raise ValueError("Request violates business rules")

    except OperationalError as e:
        logger.error(f"Database connection error: {e}")
        raise RuntimeError("Database temporarily unavailable")

    except Exception as e:
        logger.error(f"Unexpected database error: {e}")
        raise
```

> **Note**: This code is provided as a reference example for the types of errors that should be handled.

Status: Acknowledged

Update from Hyperwave: Acknowledge. Silent failures will be handled in domain logic (e.g. BoringVaultService in boring_vault.py). For logging, there's a global exception handler (FastAPI/celery built-in) which will catch and log the error, so we want to keep current version to make the code of AtomicRequestRepo short.

7.12 [Info] Missing alert on unfavourable exchange rate update

File(s): `accountant.py`

Description: The `boring_vault` bot updates the exchange rate in the Accountant contract. The contract checks if the new rate is within the allowed bounds. If not, it pauses the vault:

```
function updateExchangeRate(uint96 newExchangeRate) external requiresAuth {
  // ...
  uint64 currentTime = uint64(block.timestamp);
  uint256 currentExchangeRate = state.exchangeRate;
  uint256 currentTotalShares = vault.totalSupply();
  if (
    currentTime < state.lastUpdateTimestamp + state.minimumUpdateDelayInSeconds
        || newExchangeRate > currentExchangeRate.mulDivDown(state.allowedExchangeRateChangeUpper, 1e4)
        || newExchangeRate < currentExchangeRate.mulDivDown(state.allowedExchangeRateChangeLower, 1e4)
  ) {
      state.isPaused = true;
  // ...
```

As the rate is updated multiple times daily, an out-of-bound value may pause the vault unnecessarily. The bot should notify the protocol team (e.g., via Telegram) before applying such a value, so it can be reviewed and, if needed, updated manually.

Impact: An invalid exchange rate can pause the vault, potentially causing unexpected liquidations in integrated protocols that use the vault's liquid HLP.

Recommendation(s): Notify the protocol team of unfavourable updates and skip the update until reviewed. However, the protocol team should ensure the rate stays fresh.

Status: Fixed

Update from Hyperwave: Fixed in 7ff95c02b0cb1b147bb1f8a4ae89314e001d982a and 5328945b3866847bfed2b224e5c7d6086a017143.

7.13 [Info] Non-unique addresses may cause balance miscalculation

File(s): `boring_vault.py`

Description: During the calculation of token balances, the following loops are executed:

```
# ...
for token_address in base_config.boring_vault.all_token_addresses:
    token_contract = self.token_contract_by_address[token_address]
    balance = token_contract.balance_of(self.boring_vault_address)
    token_balances[token_address] = token_balances.get(token_address, 0) + balance

# Multisig balances on HyperCore
for ms in base_config.boring_vault.multisig_addresses:
    base_token_decimals = self.base_token_contract.decimals()
    # ...
```

The token and multisig addresses are defined in `all_token_addresses` and `multisig_addresses`, respectively—both populated via a configuration file maintained by the protocol team.

Currently, these structures are lists (`array` type), which may allow accidental duplication of entries. Since each token or multisig address should be processed only once, it would be a best practice to use a `set` instead. This ensures uniqueness and prevents redundant computation.

Impact: If any address is duplicated in the configuration, it may lead to incorrect accounting due to repeated balance aggregation.

Recommendation(s): Change the data structures `all_token_addresses` and `multisig_addresses` from lists to sets to enforce uniqueness automatically.

Status: Fixed

Update from Hyperwave: Fixed in d8da0f599a898a4484e22dbb3652f496f671580b.

7.14 [Info] Note Error

File(s): `RedstoneStablecoinRateProvider.sol`

Description: During the assignment in the `constructor` function, there is an error in the note. The note states that the Default Lower Bound is 5 bps, but it should actually be 9995 bps.

```
constructor(...) {
    //...
    // Default Lower Bound is 5 bps
    lowerBound = 10 ** RATE_DECIMALS * 9995 / 10_000;
    //...
}
```

Impact: Incorrect comments may mislead developers and cause difficulties in code maintenance.

Recommendation(s): Change 5 bps to 9995 bps

Status: Fixed

Update from Hyperwave: Fixed in PR-6.

7.15 [Info] Exponential calculations may suffer from precision loss

File(s): boring_vault.py

Description: When performing exponential calculations, if token_contract decimals is less than base_token_contract decimals, the exponent will become negative. Exponentiating 10 with a negative number will result in a value less than 1, and direct computation may lead to precision loss.

```
def calc_withdrawable_rate_by_token(self, chain_id: str, token_address: str, exchange_rate_lower_bound: int) -> int:
    // ...
    return int(
        Decimal(exchange_rate_lower_bound)
        * Decimal(10 ** (token_contract.decimals() - base_token_contract.decimals()))
        / rate_vs_base
    )

def normalize_balances_by_oracle(self, chain_id: str, token_balances: Dict[str, Decimal]) -> Decimal:
    // ...
        total_assets += (
            balance * rate_vs_base * Decimal(10 ** (base_token_contract.decimals() - token_contract.decimals()))
        )

    return total_assets

def normalize_balances_by_rate_provider(self, chain_id: str, token_balances: Dict[str, Decimal]) -> Decimal:
    // ...
        total_assets += balance * Decimal(rate) / Decimal(10 ** (2 * decimals - base_token_contract.decimals()))

    return total_assets

def get_oracle_rate_vs_base_upper_bound(self, chain_id: str, token_address: str) -> Decimal:
    // ...
    rate_vs_base = max(
        Decimal(oracle_rate)
        / Decimal(base_token_oracle_rate)
        * Decimal(10 ** (base_token_oracle.decimals() - oracle.decimals())),
        Decimal(1),
    )
    logger.info(f"Oracle rate upper bound vs base for {token_contract.symbol()}: {rate_vs_base}")

    return rate_vs_base
```

Impact: Performing operations directly between a negative number and 10 may lead to some precision loss.

Recommendation(s): It is recommended to convert to ‘Decimal‘ before performing exponential calculations. For example change the following statment:

```
int(
        Decimal(exchange_rate_lower_bound)
           * Decimal(10 ** (token_contract.decimals() - base_token_contract.decimals()))
        / rate_vs_base
    )
```

To the following:

```
int(
        Decimal(exchange_rate_lower_bound)
           * (Decimal(10) ** Decimal(token_contract.decimals() - base_token_contract.decimals()))
        / rate_vs_base
    )
```

Status: Fixed

Update from Hyperwave: Fixed in PR-22.

7.16 [Info] Some calls within the WithdrawalSolver task lack a retry mechanism

File(s): `boring_vault.py`

Description: The WithdrawalSolver task involves many operations that rely on on-chain data retrieval and calls via underlying RPC, as well as database connections. If these operations temporarily fail due to transient network instability or other issues, the program often throws exceptions directly without an appropriate retry mechanism with backoff delay. This will cause the WithdrawalSolver task to terminate immediately.

For example, in the `approve_if_needed` function, it attempts to send the transaction only once, and if it fails, it directly raises an exception without retrying.

```
def approve_if_needed(self, owner: LocalAccount, spender: str, amount: int) -> Optional[str]:
    // ...
    signed_tx = self.w3.eth.account.sign_transaction(tx, owner.key)
    tx_hash = self.w3.eth.send_raw_transaction(signed_tx.raw_transaction)
    tx_hash_str = add_0x_prefix(tx_hash.hex())
    logger.info(f"Transaction sent: {tx_hash_str}")

    # Wait for transaction receipt (success)
    receipt = self.w3.eth.wait_for_transaction_receipt(tx_hash)
    logger.debug(f"Transaction mined: {receipt}")
    ...
}
```

Impact: The WithdrawalSolver task has a relatively long scheduling interval—approximately one hour. As a result, if the task execution fails due to transient instability, it may lead to withdrawal delays and increase the processing load during the next execution.

Recommendation(s): It is recommended to implement an appropriate delayed retry mechanism for failures in database and on-chain RPC calls.

Status: Fixed

Update from Hyperwave: Fixed in PR-22 by adding auto retry to Celery tasks.

7.17 [Info] `update_exchange_rate_all_chains` may result in partial exchange rate updates

File(s): `boring_vault.py`

Description: The `ExchangeRateTask`, which is responsible for updating the on-chain `exchange_rate`, calls the `update_exchange_rate_all_chains` function. This function attempts to update the exchange rate for each chain.

However, if any one of the chains encounters an issue during the process, it will cause the program to throw an exception and stop execution, resulting in only partial exchange rate updates.

For example, if the exchange rates for two chains have already been updated but the third chain's contract is paused and throws an exception, the task will immediately stop. The exchange rates for the first two chains have been updated, but the remaining chains will not be updated.

```
def update_exchange_rate_all_chains(self):
    """Update the exchange rate of the vault across all chains."""
    for chain_id, new_rate in self.get_exchange_rate_upper_bound_by_chain().items():
        self.accountant.update_exchange_rate(chain_id, new_rate)
    }

def update_exchange_rate(self, chain_id: str, new_rate: int) -> str:
    # Check current state
    current_state = self.get_accountant_state(chain_id)
    if current_state.is_paused:
        self.telegram_bot.try_send_msg(
            chat_id=base_config.telegram.group_chat_id,
            text=f"Exchange rate update failed on {chain_id}: Accountant is paused.",
        )
        raise ExchangeRateError(f"Accountant is paused on {chain_id}")
    if current_state.last_update_timestamp + current_state.minimum_update_delay_in_seconds > now_seconds():
        logger.info(f"Accountant last update timestamp is too recent on {chain_id}, skipping exchange rate update.")
        self.telegram_bot.try_send_msg(
            chat_id=base_config.telegram.group_chat_id,
            text=f"Exchange rate update skipped on {chain_id}: Last update too recent.",
        )
        raise ExchangeRateError(
            f"Last update timestamp is too recent on {chain_id}, skipping exchange rate update."
        )
    //...
}
```

Impact: This issue has a low impact because `solve_withdrawal_queue_all_chains` currently calls `try_update_exchange_rate_all_chains` to update the exchange rates again. However, if, according to the comments, `solve_withdrawal_queue_all_chains` is designed to terminate upon failure to update the exchange rate, then this issue needs to be taken into consideration.

Recommendation(s): It is recommended not to let the exchange rate update failure of a single chain affect the updates of subsequent chains.

Status: Fixed

Update from Hyperwave: Fixed in PR-22.

7.18 [Info] The on-chain exchange rate update failure will not terminate the WithdrawalSolver task

File(s): boring_vault.py

Description: When executing the WithdrawalSolver task, it will first attempt to update the on-chain exchange rate. According to the comments, if the on-chain exchange rate update fails, it is expected to throw an error and stop the task.

```
def solve_withdrawal_queue_all_chains(self):
    """Solve a withdrawal requests."""
    # 1. Update exchange rates for all chains
    # If updaing exchange rate fails, it will raise an exception and stop the process.
    self.try_update_exchange_rate_all_chains()
    // ...
}
```

However, upon inspecting the `try_update_exchange_rate` function, it can be seen that when an error occurs during the on-chain update, the function simply returns instead of propagating the exception.

```
def try_update_exchange_rate(self, chain_id: str, new_rate: int) -> str:
    """Try to update the exchange rate, catching any errors."""
    try:
        return self.update_exchange_rate(chain_id, new_rate)
    except Exception:
        return ""
```

Impact: The task continues to execute silently even after the on-chain exchange rate update fails, rather than stopping.This does not match the expected behavior.

Recommendation(s): It is recommended that call `update_exchange_rate` function raises an exception instead of `try_update_exchange_rate`. Or modify the comments to match the intended behavior.

Status: Fixed

Update from Hyperwave: Fixed in PR-22.

7.19 [Info] Validity checks after price type conversion may fail to throw the expected error

File(s): RedstoneStablecoinRateProvider.sol

Description: After fetching the price from the oracle, the price is first converted from `int256` to `uint256`, and only then is the `price > 0` check performed.

```
function getRate() public view returns (uint256 rate) {
    //...
    rate = _calculateRate(_baseRate.toUint256(), _quoteRate.toUint256());

    _rateCheck(rate);
}

function _calculateRate(uint256 baseRate, uint256 quoteRate) internal view returns (uint256) {
    require(baseRate > 0, "Base rate must be greater than 0");
    require(quoteRate > 0, "Quote rate must be greater than 0");
    //...
}
```

Under the current integration with the Redstone oracle, this step poses no issue because Redstone includes an internal `price > 0` check when publishing prices.

However, in the future, the project may reuse with Chainlink. Chainlink may return a negative price in the event of an oracle failure. As a result, converting from `int256` to `uint256` before performing the `price > 0` check could directly trigger a `SafeCastOverflowedIntToUint` error, instead of the intended custom or expected error.

Impact: The protocol fails to throw the expected error in the event of an abnormal price.

Recommendation(s): It is recommended to check `price > 0` before performing the type conversion.

Status: Fixed

Update from Hyperwave: Fixed in PR-6.

7.20 [Best Practice] Consider applying parameter validation

File(s): atomic_queue.py

Description: The function lacks proper input validation and error handling for blockchain interactions. Specifically, `request.offer` and `request.user_address` are passed to the smart contract without verifying they are valid checksummed Ethereum addresses.

Impact: Invalid address formats may cause contract calls to fail.

Recommendation(s): Implement address validation using `Web3.is_checksum_address()` for both `request.offer` and `request.user_address` before contract interaction.

Status: Fixed

Update from Hyperwave: Fixed in 7b5fd1148af2fc90b76af58918300968a668eab2.

7.21 [Best Practice] Consider escaping data in `send_mg` function

File(s): boring_vault.py

Description: Dynamic content sent to Telegram lacks proper output encoding, creating potential security risks when data is transmitted to external services. While the current risk is mitigated because the protocol team controls the data sources (smart contracts and internal calculations).

Impact: Implementing proper output encoding represents a security best practice that prevents potential injection vulnerabilities.

Recommendation(s): Implement HTML escaping for all dynamic content before transmission to Telegram using `html.escape()`. For example: `html.escape(str(self.boring_vault_address))`

Status: Fixed

Update from Hyperwave: Fixed in 361d3ffb8a1dfb0f5517fc21b2f5ff385bc68dfd.

7.22 [Best Practice] Consider refactoring textual SQL to ORM queries

File(s): `atomic_request_repo.py`

Description: The `atomic_request_repo.py` file inconsistently combines ORM and raw SQL usage. This practice is generally discouraged, as it can lead to reduced readability, maintainability, and consistency across the codebase. Whenever possible, ORM should be preferred over raw SQL.

An example of raw SQL usage is found in the `get_by_is_solved(...)` function:

```
def get_by_is_solved(self, is_solved: bool) -> List[AtomicRequest]:
  stmt = text(
      """
      SELECT * FROM atomic_requests
      WHERE is_solved = :is_solved
      """
  )
  stmt = stmt.bindparams(bindparam("is_solved"))
  with Session(self.db_engine) as session:
      result = session.exec(
          statement=stmt,
          params={"is_solved": is_solved},
      )
      rows = result.all()
      return [AtomicRequest(**dict(row._mapping)) for r
```

Impact: There is no direct security impact. However, this is noted as a best-practice finding.

Recommendation(s): Refactor the function to use ORM queries instead of raw SQL.

Status: Acknowledged

Update from Hyperwave: Acknowledged but will keep current version, respect author's coding style.

7.23 [Best Practice] Missing oracle best-practice checks

File(s): `oracle.py`

Description: The bot interacts directly with on-chain oracles to fetch price data used in various calculations. The current implementation of `oracle.py` includes a check for data staleness.

However, if the protocol team decides to switch from the current Redstone oracle to a provider like Chainlink, it is considered best practice to also verify that the returned `answer` is greater than `0`.

Additionally, the current implementation does not consider sequencer downtime, which is relevant in some Layer 2 environments and should be checked to ensure oracle responses are valid and trustworthy.

Impact: These are best-practice recommendations and do not currently introduce a direct vulnerability.

Recommendation(s):

- Check that the returned oracle `answer` is greater than `0`;
- Include a check for sequencer downtime;

Status: Fixed

Update from Hyperwave: Fixed in 83ea2a2c2a1c207ecc166449fc4fab73afef035a. For sequencer downtime checking, currently there's no sequencer status checker for HyperEVM. Will update when we have it.

8 Evaluation of Provided Documentation

The Hyperwave documentation was primarily delivered through **NatSpec** comments within both the Solidity contracts and the off-chain bot. These comments were helpful in understanding the overall functionality of the protocol and were particularly effective in areas where they explained specific design decisions.

That said, the documentation could benefit from more detailed function descriptions and a formal external specification to further improve clarity and developer usability. Nevertheless, the Hyperwave team remained consistently available and responsive, promptly addressing all questions and concerns raised by **CODESPECT** during the audit process.

9 Test Suite Evaluation

Not applicable in this case, as the evaluation of the test suite was not performed due to the nature, duration, and scope of the audit.

9.1 Compilation Output

```
> forge compile
[] Compiling 352 files with Solc 0.8.21
[] Compiling 137 files with Solc 0.8.23
[] Solc 0.8.23 finished in 5.64s
Compiler run successful!
```

9.2 Tests Output

```
> forge test
Ran 1 test for test/RedstoneStablecoinRateProvider.t.sol:RedstoneStablecoinOracleTestUSDe
[PASS] testGetRateUSDe() (gas: 25688)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 614.40ms (202.26µs CPU time)

Ran 1 test for test/RedstoneStablecoinRateProvider.t.sol:RedstoneStablecoinOracleTestUSDT
[PASS] testGetRateUSDT() (gas: 25711)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 606.18ms (174.82µs CPU time)

Ran 2 tests for test/RedstoneStablecoinRateProvider.t.sol:RedstoneStablecoinOracleTestSetToOne
[PASS] testGetRateUSDT() (gas: 24208)
[PASS] testGetRateUSDe() (gas: 24207)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 1.12s (257.97µs CPU time)

// The deployment tests and non-scope components are not included
```

9.3 Notes on the Test Suite

The provided test suite included coverage for the Boring Vault and its related components, though not all of them were within the scope of this audit. As shown in the output above, the suite includes tests related to the stablecoin oracle. These tests offered basic coverage of oracle feed usage but were built under the assumption that the rate cannot exceed 1, which may not hold in all scenarios.

A notable area for improvement is the development of a comprehensive `pytest` suite dedicated to testing the offchain bot. While the Hyperwave team supplied a script for environment setup, a dedicated testing suite was not included. Functionality in this area was verified manually or through predefined scenarios by the Hyperwave team. A robust automated test suite would significantly enhance overall protocol coverage, especially given the critical role the offchain bot plays in the system's operation.