hashlock.

Security Audit

Diffuse Protocol 2nd (DeFi)

December 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

The Diffuse Protocol team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

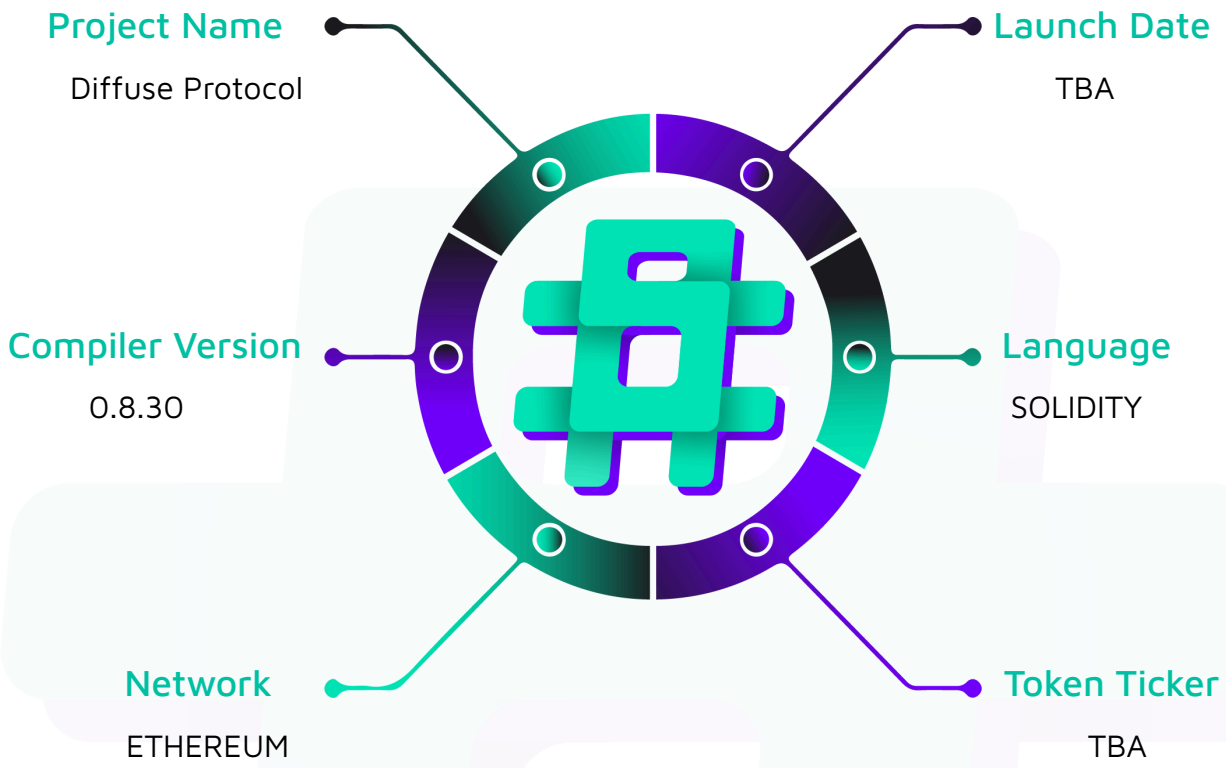
Diffuse Prime is a non-custodial protocol that enables hedge funds to amplify yields by up to 3x through overcollateralized loans funded by Diffuse Prime LPs.

LP capital is deployed into hedge fund DeFi strategies with lending-market-grade security, enhanced by verifiable risk assessment and an automated capital protection layer.

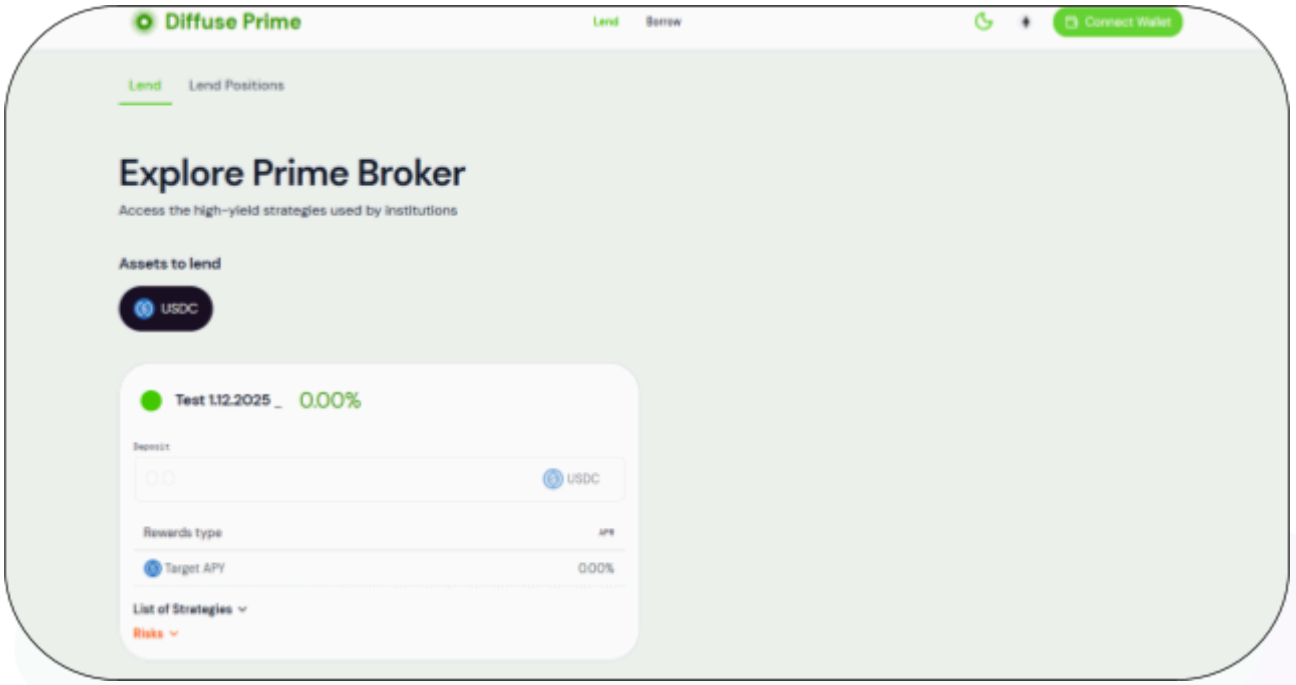
Diffuse Protocol powers Diffuse Prime with a trustless, multichain infrastructure built on TEE, zkTLS Oracles, Collateral Abstraction, and zkMessaging, providing cryptographically verifiable trust and security.

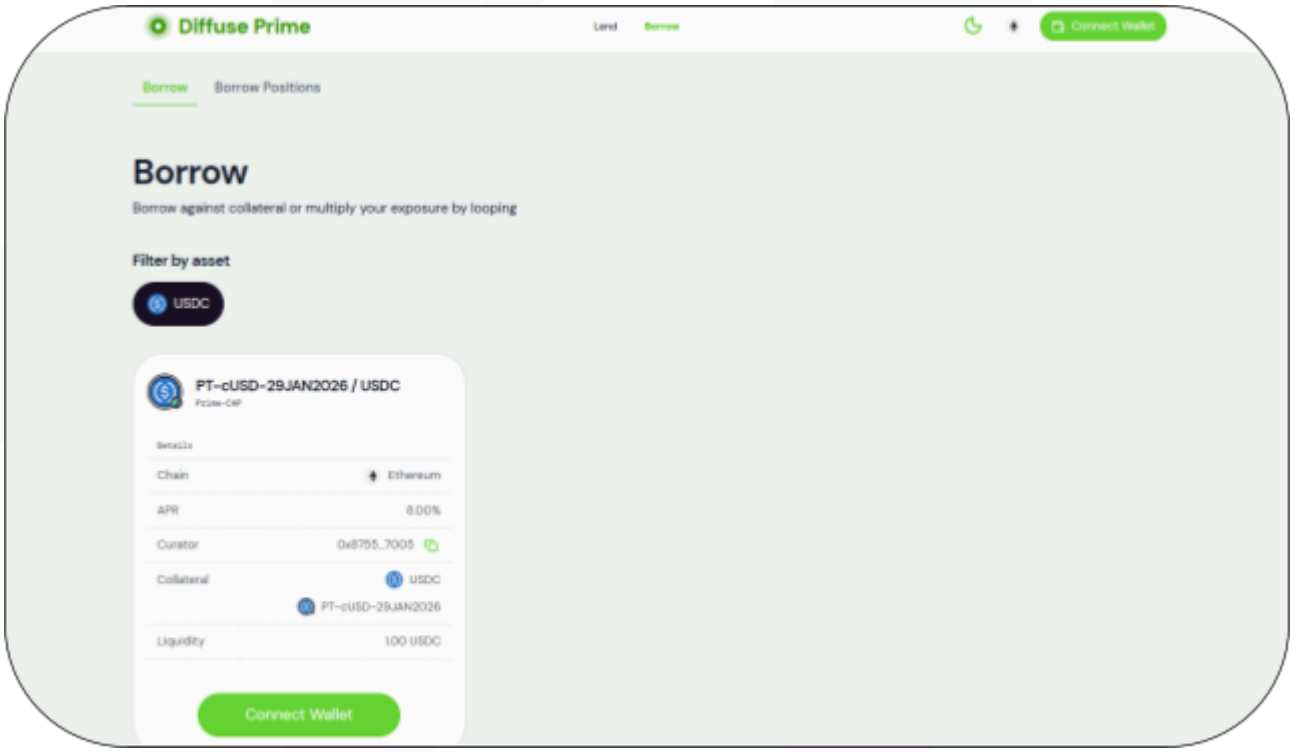
Project Name: Diffuse Protocol
Project Type: Defi
Compiler Version: 0.8.30
Website: https://ap.prime.diffuse.fi
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the Diffuse Protocol project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Diffuse Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**September, 2025**
Contract 1	AegisMintAdapter.sol
Contract 2	AegisRedeemAdapter.sol
Contract 3	AegisRedeemAdapterLogic.sol
Contract 4	IAegisRedeemAdapter.sol
Audited GitHub Commit Hash	ddcd3f50ba065c2acff0bd0be3b1bc4b69c5478a
Fix Review GitHub Commit Hash	bd1cfd4f93a1aa50cacee1e691e888a285b805b7

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

3 Medium severity vulnerabilities

5 Low severity vulnerabilities

1 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
AegisMintAdapter.sol - Provides a one-way minting adapter for the Aegis protocol integration - Enables instant minting of Aegis tokens (yUSD) using collateral tokens (TOKEN_IN) - Handles synchronous minting operations without requiring async continuation flows - Manages token approvals and interactions with the Aegis MINTER contract - Processes mint orders with parameters including: - yUSD amount to mint - Slippage-adjusted amounts - Expiry timestamps - Nonces for replay protection - Additional data and signatures - Transfers collateral tokens to Aegis and receives yUSD tokens in return - Completes minting operations atomically in a single transaction	**Contract achieves this functionality.**
AegisRedeemAdapter.sol - Manages asynchronous redemption operations for converting yUSD back to collateral tokens - Implements a multi-instance architecture using proxy clones for load distribution - Handles the lifecycle of redeem instances:	- The first deployed instance (index 0) is automatically treated as active even if never whitelisted by Aegis - No mechanism exists for users to cancel and reclaim funds from pending or perpetually unfulfillable

- Deployment of new instance clones via deterministic cloning - **Whitelisting coordination with Aegis protocol** (owner must call `setWhitelistedInstanceLastIndex` after Aegis whitelists instances) - Round-robin instance selection for load balancing - Tracks position-specific data, including: - Locked yUSD amounts per vault/position - Instance assignments for each position - Lock status to prevent concurrent operations - Implements a two-phase redemption flow: - `_buy`: Initial phase that locks yUSD amount for a position - `_continueSwap`: Continuation phase that submits redeem requests and handles completion - Manages instance availability and selection through `findNextAvailableInstanceIndex()` - Handles async redeem request states (PENDING, APPROVED, REJECTED, WITHDRAWN) - Provides refund mechanisms for rejected or withdrawn requests - Caches instance indices to maintain consistency across async operations	redeem requests - The adapter can only expand the whitelisted instance range (`setWhitelistedInstanceLastIndex` requires `index > whitelistedInstanceLastIndex`) and has no way to remove or skip bad instances once activated - `_buy` accepts and locks TOKEN_IN without confirming that any redeem instance is deployed or whitelisted - `_continueSwap` recomputes the instance index and requires exact match, causing reverts when round-robin pointer moves between calls
AegisRedeemAdapterLogic.sol	- `finishRedeem` transfers

- Implements the core logic contract that is cloned into multiple instances - Manages individual redeem request lifecycle per instance - Tracks single redeem request per instance at a time (via `requestId`) - Implements availability flag (`notAvailable`) to prevent concurrent requests - Handles interaction with Aegis protocol's redeem request system: - Submits redeem requests to Aegis MINTER contract - Queries request status from Aegis - Processes approved requests by transferring collateral tokens - Handles refunds for rejected/withdrawn requests - Validates redeem request parameters including: - yUSD amounts - Collateral amounts and slippage-adjusted amounts - Expiry timestamps and nonces - Request signatures - Transfers tokens between instance and adapter: - Receives yUSD from adapter for redemption - Transfers collateral tokens back to adapter upon approval - Refunds yUSD back to adapter for failed requests - Ensures only the AegisRedeemAdapter can call instance functions via access control	the entire `TOKEN_OUT` balance of the instance, including any idle or accidentally sent tokens - `refundYusdToAdapter` unconditionally reverts when instance's yUSD balance is even 1 wei below expected amount

Code Quality

This audit scope involves the smart contracts of the Diffuse Protocol project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Diffuse Protocol project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] AegisRedeemAdapter#findNextAvailableInstanceIndex - First instance auto-activated without whitelist confirmation

Description

AegisRedeemAdapter computes its active instance set as `[0..whitelistedInstanceLastIndex]`, with whitelistedInstanceLastIndex defaulting to zero. As soon as the first clone is deployed, index 0 is treated as active even if the owner never called setWhitelistedInstanceLastIndex. This bypasses the intended off-chain whitelist confirmation step and lets the adapter route redeem requests through an unapproved instance. Calls into Aegis can then revert due to missing whitelist/authorization, halting the async flow.

Vulnerability Details

whitelistedInstanceLastIndex is a zero-initialized uint256 and never gated by an "activated" flag. `findNextAvailableInstanceIndex` derives `activeInstanceCount = whitelistedInstanceLastIndex + 1`, so a single deployed instance makes the active set size 1 by default. Because `AegisRedeemAdapterLogic.available` returns true initially (notAvailable is false), the loop deterministically returns index 0 once `instances.length == 1`. `_continueSwap` then requires that index to match user input and forwards to requestRedeem, assuming Aegis already whitelisted the clone, but the contract cannot enforce that prerequisite. If Aegis rejects the request (e.g., address not whitelisted), the revert occurs after `TOKEN_IN` is locked, leaving the position stuck until manual intervention.

```
// src/adapters/Aegis/AegisRedeemAdapter.sol:77-95
uint256 _lastUsedInstanceIndex = lastUsedInstanceIndex;
uint256 activeInstanceCount = whitelistedInstanceLastIndex + 1;

while (tries < activeInstanceCount) {
```

```
    tries++;
    _lastUsedInstanceIndex++;
    if (_lastUsedInstanceIndex > whitelistedInstanceLastIndex) {
        _lastUsedInstanceIndex = 0;
    }
    if (IAegisRedeemAdapterLogic(instances[_lastUsedInstanceIndex]).available()) {
        return _lastUsedInstanceIndex; // index 0 is eligible by default
    }
}
```

Impact

Any first deployed instance is automatically treated as active, so a vault can lock `TOKEN_IN` in `_buy` and immediately route the redeem request through an instance that Aegis has not yet whitelisted; Aegis-side rejection or revert blocks continueSwap, leaving the position in a pending state with user funds held and no onchain mechanism to retry with a valid instance until operators intervene.

Recommendation

Add an explicit activation flag and require it before selecting instances; initialize whitelistedInstanceLastIndex only when the owner sets the active range.

Status

Resolved

[M-02] AegisRedeemAdapter#continueSwap - No user cancel path for pending or unfulfillable redeems

Description

Async redeem requests lock `TOKEN_IN` in the adapter/instance, but there is no function for a vault/user to cancel and reclaim funds once a request is pending or repeatedly rejected. `_continueSwap` only finishes on approval or refunds for retries; otherwise it loops forever waiting on Aegis without clearing locks or returning funds. A request that stays PENDING or is perpetually unfulfillable leaves the yUSD balance and amountInForPosition stuck with no escape hatch.

Vulnerability Details

`_buy` sets `amountInForPosition[vault][positionId] = yUsdAmount` and only clears it on the approved path of _continueSwap, leaving the value set for all retry or pending states. In `_continueSwap`, the pending catch-all path simply returns false and leaves instanceLocked/amountInForPosition intact, so the vault cannot reclaim yUSD even after repeated calls.

Even when refundYusdToAdapter succeeds on `REJECTED`/`WITHDRAWN`, the adapter keeps amountInForPosition nonzero and does not transfer the refunded `yUSD` back to the vault, forcing infinite retries. Although IAegis exposes `withdrawRedeemRequest(string)`, the adapter never calls it, so there is no on-chain way for a user to cancel a stuck request.

Any request that never reaches `APPROVED` or a successful refund traps user funds indefinitely in the adapter/instance, blocking capital and `UX`.

```
try instance.finishRedeem() {
    ... amountInForPosition[vault][positionId] = 0;
} catch {
    try instance.refundYusdToAdapter() {
        instanceLocked[vault][positionId] = false;
        // amountInForPosition remains set; yUSD not returned to vault
```

```
        return false;
    } catch {
        return false; // pending forever, locks remain
    }
}
```

Impact

Stuck or unfulfillable redeem requests permanently lock the user's `yUSD` in the adapter/instance, preventing withdrawal or reuse of funds and forcing endless retries; a single stalled request can indefinitely immobilize capital and degrade adapter availability, exposing users to operational loss of liquidity and denied withdrawals.

Recommendation

Add a user-triggered cancel path that (a) calls `IAegis.withdrawRedeemRequest(requestId)` for pending requests, (b) forces a refund from the instance, and (c) clears locks and returns `yUSD` to the vault by zeroing `amountInForPosition` and transferring the balance.

Status

Acknowledged

[M-03] AegisRedeemAdapter#setWhitelistedInstanceLastIndex - No dewhitelist path traps invalid instances

Description

The adapter can only expand `whitelistedInstanceLastIndex` and has no way to remove or skip a bad instance once it is activated. All indices up to the last value are always treated as eligible, even if Aegis never whitelisted those clones or later revokes them. `findNextAvailableInstanceIndex` keeps selecting an invalid clone because available() is just a busy flag and stays true. Redeem attempts then revert forever with no on-chain mechanism to roll back or bypass the bad index.

Vulnerability Details

Instance activation is gated solely by the monotonically increasing whitelistedInstanceLastIndex, so the active set is implicitly `[0..whitelistedInstanceLastIndex]` with no removal path. `findNextAvailableInstanceIndex` iterates that range and only checks available(), which is !notAvailable inside each clone and unrelated to the Aegis whitelist. If a clone in that range is not accepted by Aegis, it still returns true for `available`, becomes the next target, and `_continueSwap` calls its requestRedeem.

`AegisRedeemAdapterLogic.requestRedeem` immediately invokes `IAegis(AEGIS_MINTER).requestRedeem` with no pre-flight whitelist or try/catch. When Aegis rejects the call, the transaction reverts, leaving the clone marked available and `lastUsedInstanceIndex` unchanged, so the next attempt hits the same bad index again. With no way to dewhitelist or skip, the adapter remains stuck until a fresh deployment.

```
function test_DewhitelistMissing_DoS() external {
    adapter.deployInstance(bytes32("1")); // instance[0] already exists
    adapter.setWhitelistedInstanceLastIndex(1); // cannot go back later

    yusd.mint(1000 ether);
    yusd.approve(address(adapter), type(uint256).max);
    bytes memory data = _encodeDataForInstance(1, "req_bad");
```

```
    vm.expectRevert(); // Aegis mock rejects unwhitelisted instance[1]
    adapter.continueSwap(POSITION_ID, 0, data);

    vm.expectRevert(); // still routes to the same bad index on retry
    adapter.continueSwap(POSITION_ID, 0, data);
}
```

Impact

Once an invalid or revoked clone is included in `[0..whitelistedInstanceLastIndex]`, every redeem attempt is routed to it and reverts before progressing, effectively halting all new redemptions. Operators cannot lower `whitelistedInstanceLastIndex` or mark the clone unavailable on-chain, so the adapter remains unusable until redeployed and reconfigured, blocking user withdrawals and disrupting vault liquidity.

Recommendation

Provide an admin path to exclude indices or shrink the active range. For example, add `mapping(uint256 => bool)` disabledInstance and check it alongside `available` in `findNextAvailableInstanceIndex`, plus allow the owner to lower `whitelistedInstanceLastIndex` or permanently disable a clone. Alternatively, track an explicit allowedInstances set from Aegis and select only those indices; revert selection when the index is not whitelisted.

Status

Resolved

Low

[L-01] Adapter#claimBatchCall - Timelock allows arbitrary token drain

Description

Adapter.sol's 3-day timelock is not limited to upgrades: once elapsed, the owner can execute arbitrary external calls via `claimBatchCall`. This bypasses the withdrawUncommonToken blocklist and allows direct transfer or approval calls on `TOKEN_IN/TOKEN_OUT`. Funds held for pending Aegis redemptions can be siphoned by the owner after the delay, turning the timelock into a weak trust anchor rather than a safeguard.

Recommendation

Restrict claimBatchCall to a curated allowlist of target contracts/functions or explicitly forbid token transfer/approval calls for `TOKEN_IN` and `TOKEN_OUT` even when invoked through batched execution. Alternatively, scope timelock-executed calls to upgrade/maintenance actions only (e.g., via fixed calldata templates) to prevent discretionary fund movements.

Status

Acknowledged

[L-02] AegisRedeemAdapter#_buy - Deposits Locked When No Instance Exists

Description

`_buy` accepts `TOKEN_IN`, locks the amount for a (vault, positionId), and returns `finished = false` without confirming that any redeem instance is deployed or whitelisted. If `instances.length == 0` or no instance is available, users can deposit but cannot obtain continuation data, leaving their funds idle until an instance is later added. There is no retry or cancel path in this state.

Recommendation

Gate `_buy` so it reverts when no instance is deployable or whitelisted, e.g., require `instances.length > 0 && whitelistedInstanceLastIndex < instances.length` or invoke `findNextAvailableInstanceIndex` before locking funds. Only lock `TOKEN_IN` after an available instance is confirmed.

Status

Resolved

[L-03] AegisRedeemAdapterLogic#finishRedeem - Stranded TOKEN_OUT Can Be Swept Into Redeem Payouts

Description

finishRedeem transfers the entire `TOKEN_OUT` balance of the instance to the adapter, assuming all tokens belong to the current request. Any idle or accidentally sent `TOKEN_OUT` sitting on the instance is swept alongside the approved redeem amount. This lets an unrelated requester claim stray tokens that were never intended for their redemption. The issue persists whenever donations, dust, or leftovers remain on the instance.

Recommendation

Limit the transferred amount to the specific request value (e.g., `request.order.slippageAdjustedAmount`) and leave excess tokens untouched, or add a dedicated sweep function with access controls. Alternatively, document the behavior explicitly and ensure instances never hold unrelated `TOKEN_OUT` by design.

Status

Acknowledged

[L-04] AegisRedeemAdapter#_continueSwap - Strict instance index check causes race reverts

Description

getInfoForPosition encodes `instanceIndex = findNextAvailableInstanceIndex` into the user data, but `_continueSwap` recomputes the index and requires an exact match before proceeding. Because lastUsedInstanceIndex is shared across all positions, any other redemption that runs first will shift the next index and make previously valid data revert with InstanceIndexMismatch. Users and keepers must re-query and re-sign whenever the round-robin pointer moves, creating unnecessary timing failures without improving safety.

Recommendation

Stop recomputing the index at `_continueSwap` time; honor the `_instanceIndex` provided in the calldata as long as it is within bounds and the instance reports `available`. Alternatively, cache the chosen index when locking amountIn and validate against that stored value instead of `findNextAvailableInstanceIndex`. This preserves determinism for already-signed payloads and avoids race-induced reverts.

Status

Acknowledged

[L-05] AegisRedeemAdapterLogic#refundYusdToAdapter - Refund reverts on minor deficit locking instance

Description

`refundYusdToAdapter` unconditionally reverts with `TokenInRefundAmountMismatch` when the instance's `yUSD` balance is even 1 wei below `request.order.yusdAmount`. Any deflationary token behavior, fee-on-transfer, or rounding-induced dust makes the refund path unusable. The revert leaves `notAvailable` true and `requestId` set, permanently blocking the instance and the associated position.

Recommendation

Add tolerance or recovery handling so refunds can proceed when the shortfall is negligible, or at least clear state to avoid locking the instance. Example: accept `balance + tolerance >= expected`, or upon shortfall emit an event and reset `notAvailable`/`requestId` while routing the deficit to treasury settlement.

Status

Acknowledged

QA

[Q-01] AegisRedeemAdapter#_continueSwap - Redundant instance lock assignment

Description

`instanceLocked[vault][positionId]` is set to true immediately after the data length check and then set to true again later in `_continueSwap`. No branch flips it back between these writes, so the second store is redundant. This unnecessary state write adds cost and hints at possible copy/paste error in the locking logic..

Recommendation

Remove the duplicate assignment and keep a single lock write before the instance transfer.

Status

Resolved

Centralisation

The Diffuse Protocol project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the Diffuse Protocol project seems to have a sound and well-tested code base,now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au