FAILSAFE

Prepared for
Aegis

Prepared by
ret2basic.eth
jesjupyter
FailSafe

9 December 2025

Aegis

Smart Contract Audit Report

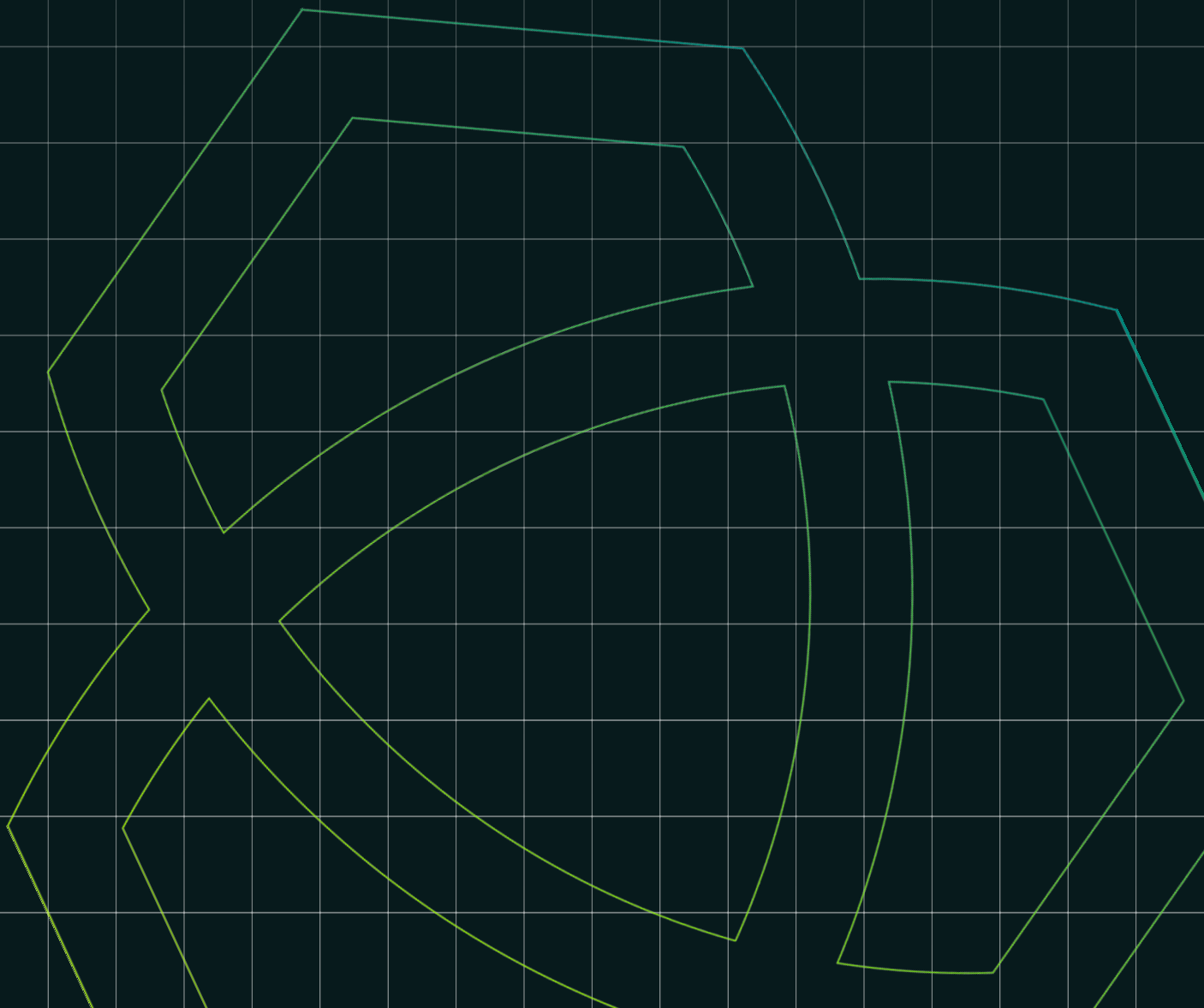

Table of Contents

Executive Summary

FailSafe was enlisted by the Aegis project to conduct an elite security review of its smart contracts spanning multiple blockchain platforms. Our team of seasoned auditors engaged in a rigorous analysis of the system's architecture, code, and operational procedures. This comprehensive audit aimed at identifying vulnerabilities and providing actionable recommendations to enhance the security and reliability of the project's smart contract infrastructure. Through meticulous scrutiny, FailSafe leveraged its expertise to ensure that Aegis's systems are robust and resilient against potential threats.

During the audit, several notable vulnerabilities were uncovered, highlighting areas of concern that could impact the project's security posture. Among the critical issues identified were non-atomic bridge burn processes that could lead to irrecoverable loss of funds if transactions are directed to blacklisted or zero addresses. Additionally, a significant flaw was discovered allowing blacklisted spenders to bypass restrictions, potentially undermining the blacklist's purpose of thwarting malicious activities. High-severity issues included missing defensive checks that could obstruct transaction processes. Medium-severity findings pointed to potential inflation vulnerabilities and inaccurate fee deductions affecting user transactions. These findings underscore the importance of addressing both immediate threats and systemic vulnerabilities to safeguard the integrity of the Aegis project.

Project Details

Project	Aegis
Website	https://aegis.im/
Repository	https://github.com/Aegis-im/aegis-contracts/tree/feature/jusd
Blockchain	Multichain
Audit Type	Smart Contract Audit Report
Initial Commit	2912ff432ddfbfdcb6d2f75f0fd8caaf96069b3e
Final Commit	e25105753f00f5e3bfb478e074822681a365e2e3
Timeline	25 November 2025 - 2 December 2025
	Final Report: 9 December 2025

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We will employ a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involves:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involves an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) Business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

We will perform functional testing using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This includes:

1. Functional Testing: Writing comprehensive tests to cover various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

If deemed necessary based on the complexity and criticality of the smart contract(s), we will perform fuzzing and invariant testing to identify vulnerabilities that might not be caught through conventional methods. This includes:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

Our audit will thoroughly cover a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

A thorough description of the issue, highlighting the potential impact on the system.

1. The location within the codebase where the issue is found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. We will suggest improvements in the overall system architecture or design, if relevant.
6. Wherever applicable, we'll include a PoC to demonstrate issue severity, aiding effective mitigation.

Report Generation

1. Document all findings, including identified vulnerabilities, their severity, and potential impact.
2. Provide clear and actionable recommendations for addressing security issues.

Remediation Support

1. Collaborate with the project's development team to address and remediate identified vulnerabilities.
2. Review and validate code changes and security fixes.

Final Assessment

Re-evaluate the project's security posture after remediation efforts to ensure vulnerabilities have been adequately addressed.

In-scope

- AegisChainlinkOracleV2.sol
- AegisChainlinkOracleV3.sol
- AegisConfig.sol
- AegisMinting.sol
- AegisMintingJUSD.sol
- AegisOracle.sol
- AegisOracleJUSD.sol
- AegisRewards.sol
- AegisRewardsManual.sol
- JUSD.sol
- JUSDMintBurnOFTAdapter.sol
- JUSDOFT.sol
- TimelockController.sol
- YUSD.sol
- YUSDMintBurnOFTAdapter.sol
- YUSDOFT.sol
- ClaimRewardsLib.sol
- OrderLib.sol
- sJUSD.sol
- sJUSDOFT.sol
- sJUSDOFTAdapter.sol
- sJUSDSilo.sol
- sYUSD.sol
- sYUSDOFT.sol
- sYUSDOFTAdapter.sol
- sYUSDSilo.sol

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
Critical	2	-	-	-	2
High	1	-	-	-	1
Medium	4	-	2	-	2
Low	3	-	3	-	-
Info	4	-	4	-	-
Total	14	0	9	0	5

#	Findings	Severity	Status
1	Blacklisted Spender Can Bypass Restrictions via transferFrom	Critical	Resolved
2	Non-Atomic Bridge Burn: Destination Blacklist/Zero Causes Irrecoverable Loss	Critical	Resolved
3	JUSDMintBurnOFTAdapter.sol: Missing address(0) Handling in _credit()	High	Resolved
4	OFT Dust Loss in Mint/Burn Adapters	Medium	Resolved
5	Pre-collateralized Mint Limit Compounds With Supply	Medium	Acknowledged
6	Preview Functions Ignore Instant-Unstake Fee	Medium	Acknowledged
7	sJUSD Unstake Accepts Silo Address, Burning User Assets	Medium	Resolved
8	Claim Underflow on Over-Allocated ID Causes Revert/DoS	Low	Acknowledged
9	Cross-Chain Events Omit Provenance Fields	Low	Acknowledged
10	Oracle V2 Masks Missing Price by Returning 1e36	Low	Acknowledged
11	Oracle Timestamp Overflow	Info	Acknowledged
12	Slippage Check Ignores Mint Fee, Users Receive Less Than Minimum	Info	Acknowledged
13	Stale Oracle De-peg Arbitrage on Redemption	Info	Acknowledged
14	Zero-Amount Claims Permanently Flag IDs as Claimed	Info	Acknowledged

Finding 1: Blacklisted Spender Can Bypass Restrictions via transferFrom

Severity: Critical

Status: Resolved

Description:

The `JUSD` and `YUSD` contracts implement a blacklist mechanism by overriding the `_update` function. This function checks if the `from` (sender) or `to` (recipient) address is blacklisted before allowing any token movement (mint, burn, or transfer).

However, the standard ERC20 `transferFrom` function (used when a spender moves tokens on behalf of an owner) operates as follows:

1. Calls `_spendAllowance(from, spender, value)`.
2. Calls `_transfer(from, to, value)`, which calls `_update(from, to, value)`.

In this flow, `_update` validates `from` (the token owner) and `to` (the recipient). It **does not** validate `msg.sender` (the spender). Consequently, if a blacklisted address has a pre-existing allowance from a non-blacklisted user, they can successfully execute `transferFrom` to move funds, effectively bypassing the blacklist restriction intended to freeze their activity.

This also applies to `burnFrom` (inherited from `ERC20Burnable`), where a blacklisted spender could burn tokens belonging to another user.

Impact:

Critical. The blacklist is a primary security control to freeze malicious actors (e.g., hackers, sanctioned entities). This bypass allows a blacklisted entity to continue operating by draining funds from any accounts that have approved them (e.g., via a compromised dApp or phishing attack), defeating the purpose of the freeze.

Source:

- `contracts/JUSD.sol`: `_update` override (lines ~54-62)
- `contracts/YUSD.sol`: `_update` override (lines ~54-62)
- `reference/ERC20.sol`: `transferFrom` (lines ~138-143) and `_update` (lines ~160-185)

Code:

```
1. Attacker (Blacklisted) calls `JUSD.transferFrom(Victim, Attacker, 1000)`.
2. `transferFrom` calls `_spendAllowance(Victim, Attacker, 1000)`. (No blacklist check in base ERC20).
3. `transferFrom` calls `_update(Victim, Attacker, 1000)`.
4. `_update` checks `isBlackListed[Victim]` (False) and `isBlackListed[Attacker]` (True).
    - *Correction*: If the attacker is the *recipient* (`to`), `_update` **will** catch it.
    - *Refined Scenario*: Attacker calls `JUSD.transferFrom(Victim, Accomplice, 1000)`.
5. `_update` checks `isBlackListed[Victim]` (False) and `isBlackListed[Accomplice]` (False).
6. Transfer succeeds. The Blacklisted Attacker successfully initiated the transfer.
```

Proof of Concept:

Append the following code to `test/16_jusd.spec.ts`:

```
  describe('#blacklistBypass', () => {
    it('allows blacklisted spender to drain allowance via transferFrom', async () => {
      const [owner, victim, spender, recipient] = await ethers.getSigners()

      const contract = await ethers.deployContract('JUSD', [owner.address])
      await contract.setMinter(owner.address)

      const initialBalance = ethers.parseEther('100')
      await contract.mint(victim.address, initialBalance)

      const allowance = ethers.parseEther('10')
      await contract.connect(victim).approve(spender.address, allowance)

      await contract.addBlackList(spender.address)

      await expect(contract.connect(spender).transferFrom(victim.address, recipient.address, allowance
    )).
        to.emit(contract, 'Transfer').
        withArgs(victim.address, recipient.address, allowance)

      await expect(contract.balanceOf(recipient.address)).eventually.to.equal(allowance)
      await expect(contract.balanceOf(victim.address)).eventually.to.equal(initialBalance - allowance)
    })
  })
```

Run PoC with `npx hardhat test test/16_jusd.spec.ts`.

Remediation:

Override the `_spendAllowance` function in both `JUSD` and `YUSD` to explicitly check if the `spender` is blacklisted. This covers both `transferFrom` and `burnFrom`.

```
function _spendAllowance(address owner, address spender, uint256 value) internal virtual override {
    if (isBlackListed[spender]) {
        revert Blacklisted(spender);
    }
    super._spendAllowance(owner, spender, value);
}
```

Alternatively, override `transferFrom` and `burnFrom` individually, but `_spendAllowance` is cleaner.

Finding 2: Non-Atomic Bridge Burn: Destination Blacklist/Zero Causes Irrecoverable Loss

Severity: Critical

Status: Resolved

Description:

The `JUSDMintBurnOFTAdapter` and `YUSDMintBurnOFTAdapter` contracts implement cross-chain bridging by burning tokens on the source chain and minting them on the destination chain.

The process is **non-atomic**:

1. **Source Chain**: `_debit` is called. Tokens are transferred from the user to the adapter, then burned via `aegisMinting.burnForCrossChain`. The transaction finalizes on the source chain.
2. **LayerZero**: The message is relayed to the destination chain.
3. **Destination Chain**: `_credit` is called. It attempts to mint tokens to the recipient via `aegisMinting.mintForCrossChain`.

If the destination transaction fails (reverts), the message is stuck. While LayerZero allows retrying failed messages, some failures are **permanent**:

- **Blacklisted Recipient**: If the recipient address is blacklisted on the destination chain, `JUSD.mint` (called by `mintForCrossChain`) will revert. Since the blacklist status is unlikely to change, the message cannot be processed.
- **Zero Address**: If the recipient is `address(0)`, `JUSD.mint` will revert. (Note: Standard OFT logic redirects `address(0)` to `0xdead`, but these adapters override `_credit` and miss that check - see separate finding).

In these cases, the user's funds are burned on the source chain, but they never receive them on the destination. There is no mechanism to refund the user on the source chain because the source transaction is already final.

Impact:

Critical. Irrecoverable loss of funds. If a user bridges to a blacklisted address (or one that becomes blacklisted before the message arrives) or address(0), their assets are destroyed with no recourse.

Source:

- `contracts/JUSDMintBurnOFTAdapter.sol`: `_debit` (lines ~33-47) and `_credit` (lines ~56-63)
- `contracts/YUSDMintBurnOFTAdapter.sol`: `_debit` (lines ~33-47) and `_credit` (lines ~56-63)

Code:

```
**Vulnerable Flow:**

1. User calls `send()` on Source Chain.
2. `_debit` burns tokens. Source TX succeeds.
3. LayerZero relays message.
4. Destination `_credit` calls `aegisMinting.mintForCrossChain(BlacklistedUser, amount)`.
5. `JUSD` reverts because `BlacklistedUser` is blacklisted.
6. LayerZero message fails. User has no tokens on either chain.
```

Remediation:

1. **Zero Address**: Implement the standard OFT check in `_credit` to redirect `address(0)` to a burn address (e.g., `0xdead`) or the adapter itself, ensuring the message succeeds (even if funds are effectively lost/burned, the state is consistent).
2. **Blacklist/General Failure**: Implement a "fallback" or "escrow" mechanism in `_credit`.
   - Wrap the `mintForCrossChain` call in a `try/catch` block.
   - If minting fails (e.g., due to blacklist), mint the tokens to the **adapter contract** or a designated **escrow address** instead of the user.
   - Provide a function for the user (or admin) to rescue/claim these tokens later (e.g., after being removed from the blacklist or providing a new address).

```solidity
function _credit(address _to, uint256 _amountLD, uint32) internal override returns (uint256) {
    if (_to == address(0)) _to = address(0xdead); // Fix zero address issue

    try aegisMinting.mintForCrossChain(_to, _amountLD) {
        // Success
    } catch {
        // Fallback: Mint to adapter/escrow to prevent message blocking
        aegisMinting.mintForCrossChain(address(this), _amountLD);
        emit FailedMintSavedToEscrow(_to, _amountLD);
    }
    return _amountLD;
}
```

Finding 3: JUSDMintBurnOFTAdapter.sol: Missing address(0) Handling in _credit()

Severity: High

Status: Resolved

Description:

LayerZero's canonical mint/burn implementation (`OFT.sol`) performs a defensive check inside `_credit` to redirect messages destined for `address(0)` to `address(0xdead)` so that `_mint` never reverts: https://github.com/LayerZero-Labs/LayerZero-v2/blob/ab9b083410b9359285a5756807e1b6145d4711a7/packages/layerzero-v2/evm/oapp/contracts/oft/OFT.sol#L83.

The default `OFTAdapter` skips this guard because it merely unlocks escrowed tokens; a zero-address transfer there just reverts before any supply change happens.

`JUSDMintBurnOFTAdapter` inherits from `OFTAdapter`, but its `_credit` function does not unlock tokens—it mints fresh JUSD via `aegisMinting.mintForCrossChain()`. Because it follows the mint/burn pattern, it needs the same zero-address handling as `OFT.sol`, yet the check is missing.

If a cross-chain message is sent with `_to` as `address(0)`, `aegisMinting.mintForCrossChain()` will call `JUSD.mint(address(0), amount)`, which will revert (standard ERC20 behavior): https://github.com/OpenZeppelin/openzeppelin-contracts/blob/03666bd5653762fc983f31cdad2e704c27397bdf/contracts/
L217.

Impact:

This causes the LayerZero lzReceive transaction to fail. Depending on the LayerZero configuration (blocking vs. non-blocking), this could block the channel for subsequent messages or simply leave the message permanently failed/stuck, effectively burning the user's funds on the source chain without minting them on the destination.

Source:

contracts/JUSDMintBurnOFTAdapter.sol, _credit()

Code:

```
// JUSDMintBurnOFTAdapter.sol
function _credit(address _to, uint256 _amountLD, uint32)
  internal
  override
  returns (uint256 amountReceivedLD)
{
  // Call AegisMinting directly to mint tokens to the recipient
  aegisMinting.mintForCrossChain(_to, _amountLD); // <--- BUG: Reverts if _to is address(0)
```

```
9
10    return _amountLD;
11  }
```

Remediation:

Add the standard OFT safety check:

```
1  if (_to == address(0x0)) _to = address(0xdead);
```

Finding 4: OFT Dust Loss in Mint/Burn Adapters

Severity: Medium

Status: Resolved

Description:

The `JUSDMintBurnOFTAdapter` and `YUSDMintBurnOFTAdapter` contracts override the `_debit` function to handle cross-chain transfers by burning tokens on the source chain. However, these implementations fail to remove "dust" (amounts smaller than the shared decimal resolution) before burning the tokens.

The LayerZero OFT standard uses a shared decimal system (typically 6 decimals) for cross-chain compatibility. When converting from a local decimal system (e.g., 18 decimals) to shared decimals, the least significant digits ("dust") are truncated. The standard `OFT` and `OFTAdapter` implementations handle this by calling `_removeDust` (via `_debitView`) to ensure that only the amount that can be credited on the destination chain is debited from the user.

In `JUSDMintBurnOFTAdapter.sol`:

```
function _debit(address _from, uint256 _amountLD, uint256, uint32)
  internal
  override
  returns (uint256 amountSentLD, uint256 amountReceivedLD)
{
  // Transfer tokens from user to this contract first
  IERC20(this.token()).safeTransferFrom(_from, address(this), _amountLD);

  // Approve AegisMinting to spend our tokens
  IERC20(this.token()).approve(address(aegisMinting), _amountLD);

  // Call AegisMinting to burn tokens from this contract (this contract must be a cross-chain
  operator)
  aegisMinting.burnForCrossChain(address(this), _amountLD);

  return (_amountLD, _amountLD);
}
```

The function takes `_amountLD` (the full amount requested by the user) and burns it entirely. It then returns `_amountLD` as the amount sent/received. Subsequently, `OFTCore._send` calls `_buildMsgAndOptions`, which calls `_toSD(_amountLD)`. The `_toSD` function truncates the dust.

Other OFT implementations in the codebase are **not** vulnerable:

- **`JUSDOFT` and `sJUSDOFT`**: These inherit from `OFT` and do not override `_debit`. The standard `OFT._debit` implementation calls `_debitView`, which internally calls `_removeDust` to clean the amount before burning.
- **`sJUSDOFTAdapter`**: This inherits from `OFTAdapter` and does not override `_debit`. The standard `OFTAdapter._debit` implementation also calls `_debitView` to remove dust before locking tokens.

Only the custom adapters (`JUSDMintBurnOFTAdapter` and `YUSDMintBurnOFTAdapter`) that manually override `_debit` to integrate with `AegisMinting` are affected.

Impact:

Medium. Users suffer a definite and permanent loss of funds due to incorrect accounting logic. The adapter burns the full amount (including dust) on the source chain, but the destination chain only credits the truncated amount (due to shared decimal conversion).

For example, if a user sends `1.000000000000000123` JUSD (18 decimals):

1. The adapter burns `1.000000000000000123` JUSD.
2. The message encodes `1000000` (6 decimals).
3. The destination receives `1.000000000000000000` JUSD.
4. The user loses `0.000000000000000123` JUSD (123 wei).

This violates the protocol's accounting integrity. In competitive audits (e.g., Sherlock, Code4rena), this is classified as Medium severity because it constitutes a guaranteed leak of value from the user, regardless of the amount's size. Standard OFT implementations explicitly prevent this by removing dust before the debit operation.

Remediation:

Update `_debit` in both `JUSDMintBurnOFTAdapter` and `YUSDMintBurnOFTAdapter` to remove dust before processing the transfer. Use `_debitView` or call `_removeDust` explicitly.

```
function _debit(address _from, uint256 _amountLD, uint256 _minAmountLD, uint32 _dstEid)
  internal
  override
  returns (uint256 amountSentLD, uint256 amountReceivedLD)
{
  // Calculate amounts with dust removed
  (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD, _dstEid);

  // Transfer tokens from user to this contract first
  IERC20(this.token()).safeTransferFrom(_from, address(this), amountSentLD);

  // Approve AegisMinting to spend our tokens
  IERC20(this.token()).approve(address(aegisMinting), amountSentLD);

  // Call AegisMinting to burn tokens from this contract
  aegisMinting.burnForCrossChain(address(this), amountSentLD);

  return (amountSentLD, amountReceivedLD);
}
```

Finding 5: Pre-collateralized Mint Limit Compounds With Supply

Severity: Medium

Status: Acknowledged

Description:

The _checkPreCollateralizedMintLimit guard in contracts/AegisMintingJUSD.sol derives the allowed amount from jusd.totalSupply() at the moment of each mint. Because totalSupply already includes the amounts minted earlier in the same period, every successful pre-collateralized mint increases the ceiling for the next one. This lets the privileged minter chain multiple transactions within a single period and exceed the intended percentage cap. In the extreme case where maxPeriodAmountBps is 10,000 (100%), the function never reverts and the minter can mint indefinitely.

Impact:

preCollateralizedMinter can mint more JUSD than governance intended for a period, effectively inflating supply until the token loses value. With maxPeriodAmountBps >= 10,000, the check never fires and the minter has truly unlimited power.

Source:

contracts/AegisMintingJUSD.sol, _checkPreCollateralizedMintLimit()

Code:

```
uint256 totalSupply = jusd.totalSupply();
if (
  currentPeriodEndTime >= block.timestamp &&
  limits.currentPeriodTotalAmount + yusdAmount > (totalSupply * limits.maxPeriodAmountBps) / MAX_BPS
) {
  revert LimitReached();
}
```

Proof of Concept:

Scenario: 10% Limit Bypass

Assume:

- **Initial Supply**: 1,000,000 JUSD
- **Limit**: 10% (`maxPeriodAmountBps` = 1000)

1. **Transaction 1**: Attacker mints **100,000 JUSD** (10% of 1M).

- Check: `100,000 <= 10% of 1,000,000`. **Pass**.
- New Supply: 1,100,000 JUSD.
- Used Limit: 100,000.

2. **Transaction 2**: Attacker attempts to mint more.
   - New Limit Calculation: 10% of **1,100,000** = 110,000.
   - Remaining Allowance: 110,000 (New Limit) - 100,000 (Used) = **10,000**.
   - Attacker mints **10,000 JUSD**. **Pass**.
   - New Supply: 1,110,000 JUSD.
3. **Transaction 3**: Attacker attempts to mint more.
   - New Limit Calculation: 10% of **1,110,000** = 111,000.
   - Remaining Allowance: 111,000 - 110,000 = **1,000**.
   - Attacker mints **1,000 JUSD**. **Pass**.

Scenario: Infinite Mint (100% Limit)

If `maxPeriodAmountBps` is set to 10,000 (100%), the check becomes ineffective:

1. Supply = 1M. Limit = 1M. Attacker mints 1M.
2. New Supply = 2M. Limit = 2M. Used = 1M. Remaining = 1M.
3. Attacker mints 1M.
4. This can be repeated indefinitely within the same period.

Remediation:

Snapshot the circulating supply at the start of each period (store it alongside currentPeriodStartTime) and compute the allowed budget from that fixed value. Alternatively, express maxPeriodAmount as an absolute quantity rather than a dynamic percentage of the live supply.

Finding 6: Preview Functions Ignore Instant-Unstake Fee

Severity: Medium

Status: Acknowledged

Description:

`sYUSD` and `sJUSD` override `withdraw`/`redeem` to charge an `INSTANT_UNSTAKING_FEE` whenever `cooldownDuration > 0`. The actual implementation (`sYUSD.sol`, lines ~210-290; `sJUSD.sol`, same region) deducts the fee and only transfers `assets - fee` to the user (fee goes to `INSURANCE_FUND`).

However, neither contract overrides `previewWithdraw` nor `previewRedeem`, so they inherit OpenZeppelin's default `ERC4626` previews, which simply call `_convertToShares` and `_convertToAssets` and **do not apply any fee**. Consequently:

- `previewWithdraw(assets)` tells integrators they must burn `X` shares to receive `assets`, but the actual `withdraw` call returns `assets - fee` to the receiver.
- `previewRedeem(shares)` promises `Y` assets, but the user only receives `Y * (1 - feeBP/10_000)` when cooldown is enabled.

Any protocol relying on previews for slippage checks or front-end quotes will be misled as soon as the instant-unstake fee is turned on.

Impact:

Medium. Integrators, routing contracts, or front-ends that call the standard ERC-4626 preview functions can misprice withdrawals by up to INSTANT_UNSTAKING_FEE (configurable up to 100%). Users can receive less than quoted amounts, triggering unnecessary slippage reverts or silently short-changing users when fees are small.

Source:

- `contracts/sYUSD.sol`: `withdraw`/`redeem`/`_instantUnstakeAssets`/`_instantUnstakeShares` (lines ~210-390)
- `contracts/sJUSD.sol`: same functions and fee logic
- No overrides for `previewWithdraw`/`previewRedeem` exist

Code:

```
// sYUSD.sol withdraw (cooldown on)
uint256 fee = (assets * INSTANT_UNSTAKING_FEE) / 10_000;
uint256 netAssets = assets - fee;
// previews never apply this subtraction
```

Remediation:

Override the preview functions (and optionally `previewMint` / `previewDeposit` if similar fees ever apply) to reflect the same logic as `withdraw` / `redeem` when cooldown is active. Example for `previewWithdraw`:

```
function previewWithdraw(uint256 assets) public view override returns (uint256) {
    if (cooldownDuration == 0 || INSTANT_UNSTAKING_FEE == 0) {
        return super.previewWithdraw(assets);
    }

    uint256 grossAssets = (assets * 10_000) / (10_000 - INSTANT_UNSTAKING_FEE);
    return super.previewWithdraw(grossAssets);
}

function previewRedeem(uint256 shares) public view override returns (uint256) {
    uint256 grossAssets = super.previewRedeem(shares);
    if (cooldownDuration == 0 || INSTANT_UNSTAKING_FEE == 0) {
        return grossAssets;
    }
    return grossAssets - (grossAssets * INSTANT_UNSTAKING_FEE) / 10_000;
}
```

Finding 7: sJUSD Unstake Accepts Silo Address, Burning User Assets

Severity: Medium

Status: Resolved

Description:

`sJUSD.unstake(receiver)` only prevents the zero address; it allows `receiver == address(silo)` (see `contracts/sJUSD.sol`, lines ~250-280). During cooldown, all unstakeable JUSD sits inside `sJUSDSilo`. When `unstake` is called and the cooldown has elapsed, the contract:

1. Copies `underlyingAmount` into `assets`.
2. Resets `cooldownEnd` and `underlyingAmount` to zero **before** transferring anything.
3. Calls `silo.withdraw(receiver, assets)`.

`sJUSDSilo.withdraw` simply executes `_JUSD.transfer(to, amount)` with no guard against `to == address(this)` (file `contracts/sJUSDSilo.sol`, line 26). Therefore, if a user (or malicious UI) submits `receiver = address(silo)`, the transfer becomes a no-op self-transfer. The storage accounting in `sJUSD` is already cleared, so the user's share of JUSD remains forever trapped inside the silo contract with no way to map it back to them; future `unstake` calls see `underlyingAmount == 0`.

Impact:

A single bad unstake call strands the user's cooled-down JUSD. They receive nothing, cooldown state resets, and there's no mechanism to recover the specific amount because the silo simply holds undifferentiated funds. A compromised front-end could grief many users by silently substituting the silo address as the receiver.

Code:

```
// contracts/sJUSD.sol
cooldown.cooldownEnd = 0;
cooldown.underlyingAmount = 0;
silo.withdraw(receiver, assets); // succeeds even if receiver == address(silo)

// contracts/sJUSDSilo.sol
function withdraw(address to, uint256 amount) external onlyStakingVault {
    _JUSD.transfer(to, amount); // self-transfer leaves funds in silo
}
```

Remediation:

Disallow the silo as a receiver:

```
if (receiver == address(silo)) revert InvalidReceiver();
```

Finding 8: Claim Underflow on Over-Allocated ID Causes Revert/DoS

Severity: Low

Status: Acknowledged

Description:

In `AegisRewards.sol` and `AegisRewardsManual.sol`, the `claimRewards` function iterates through a list of reward IDs and amounts provided by the user (verified via signature). For each ID, it subtracts the claimed amount from the remaining reward balance:

```
_rewards[claimRequest.ids[i]].amount -= claimRequest.amounts[i];
```

Since Solidity 0.8.x enables checked arithmetic by default, this subtraction will revert if `claimRequest.amounts[i]` is greater than `_rewards[claimRequest.ids[i]].amount`.

This creates a Denial of Service (DoS) vulnerability if the off-chain signer issues signatures where the sum of allocated user rewards exceeds the total deposited reward amount for a specific ID (over-allocation). If earlier claimers drain the pool such that the remaining balance is less than a later user's signed amount, the later user's transaction will revert. They cannot claim any of their rewards (even for other valid IDs in the same batch) because the entire transaction fails.

Impact:

Users with valid, signer-approved claims are permanently blocked whenever a reward ID becomes over-allocated. Every attempt to claim that ID (even alongside other valid IDs) reverts, so an honest claimant cannot recover any funds until governance re-issues smaller signatures or tops up the pool. In AegisRewardsManual, the same revert also prevents _totalReservedRewards from decreasing, so the contract's accounting remains stuck and no new deposits can be made safely.

Source:

- `contracts/AegisRewards.sol`: `claimRewards` (line ~108)
- `contracts/AegisRewardsManual.sol`: `claimRewards` (line ~118)

Code:

```
// AegisRewards.sol
_rewards[claimRequest.ids[i]].amount -= claimRequest.amounts[i]; // Reverts on underflow

// AegisRewardsManual.sol
_rewards[claimRequest.ids[i]].amount -= claimRequest.amounts[i];
_totalReservedRewards -= claimRequest.amounts[i]; // never executes if the first line underflows
```

Remediation:

Modify the loop to cap the claimable amount at the remaining balance of the reward ID. This ensures the transaction succeeds, paying out whatever is left, rather than reverting.

```
uint256 claimAmount = claimRequest.amounts[i];
uint256 remaining = _rewards[claimRequest.ids[i]].amount;

if (claimAmount > remaining) {
    claimAmount = remaining;
}

_rewards[claimRequest.ids[i]].amount -= claimAmount;
totalAmount += claimAmount;
```

Alternatively, explicitly check `if (claimRequest.amounts[i] > _rewards[claimRequest.ids[i]].amount) continue;` to skip over-allocated claims but process others, though capping is generally more user-friendly.

Finding 9: Cross-Chain Events Omit Provenance Fields

Severity: Low

Status: Acknowledged

Description:

The `mintForCrossChain` and `burnForCrossChain` functions in `AegisMintingJUSD.sol` emit `CrossChainMint` and `CrossChainBurn` events, respectively. However, these events only log the local recipient/sender and the amount. They omit provenance data such as the **source chain ID**, **source address**, or **message nonce**.

Because the `AegisMintingJUSD` contract is decoupled from the specific bridge adapter (it only checks `onlyCrossChainOperator`), the emitted events provide no on-chain or off-chain way to link a specific mint/burn to the originating cross-chain message. This makes it impossible to audit the bridge flow or detect unauthorized mints (e.g., if the operator key is compromised) by reconciling them against source chain events.

Impact:

1. **Lack of Auditability**: It is impossible to verify off-chain that a specific mint on Chain A corresponds to a specific burn on Chain B.
2. **Monitoring Blind Spots**: Security monitoring tools cannot distinguish between legitimate bridge traffic and unauthorized mints by a compromised operator, as there is no "source chain" field to cross-reference.

Source:

- `contracts/AegisMintingJUSD.sol`: `mintForCrossChain` (lines ~430-440)
- `contracts/AegisMintingJUSD.sol`: `burnForCrossChain` (lines ~445-455)

Code:

```
function mintForCrossChain(address to, uint256 amount) external ... {
    jusd.mint(to, amount);
    emit CrossChainMint(to, amount); // Missing context
}
```

Remediation:

Update the `mintForCrossChain` and `burnForCrossChain` functions to accept and emit provenance data.

1. Update the events in `IAegisMintingEvents`:

```
event CrossChainMint(address indexed to, uint256 amount, uint32 srcChainId, bytes32 srcAddress,
    uint64 nonce);
event CrossChainBurn(address indexed from, uint256 amount, uint32 dstChainId, bytes32 dstAddress
    , uint64 nonce);
```

2. Update the function signatures in `AegisMintingJUSD` to accept these parameters from the adapter and emit them.

Finding 10: Oracle V2 Masks Missing Price by Returning 1e36

Severity: Low

Status: Acknowledged

Description:

`AegisChainlinkOracleV2` is the Morpho-compatible oracle for YUSD. Its `price()` implementation (see `contracts/AegisChainlinkOracleV2.sol`, lines ~100-135) returns `1e36` whenever `_priceData.price <= 0`:

```
if (_priceData.price <= 0) {
    // Default to 1 USD if no price set
    return 1e36;
}
```

This means that if no operator has ever set a price, or the stored price becomes negative/zero due to a bug, the oracle silently emits "1 USD" rather than signaling failure. Morpho (and any other consumer) treats this as a valid quote, so safety checks that rely on a failing oracle to halt borrowing/liquidations will continue operating on the assumption that YUSD = $1.

Unlike Chainlink feeds, there is no heartbeat or staleness guard here; `_priceData` is just an `int256`. Returning 1e36 when the variable is uninitialized (default 0) hides the fact that the oracle has never been configured.

Impact:

If operators forget to update the price before enabling markets, Morpho will think the oracle is healthy and priced at parity. In the event of an outage, the system cannot distinguish between "no data" and "$1 peg," so liquidations/mint caps won't pause automatically. This undermines circuit-breaker expectations because a critical failure degrades into a false positive.

Code:

```
function price() external view returns (uint256) {
    if (_priceData.price <= 0) {
        return 1e36; // current behaviour: hides missing price
    }
    return uint256(_priceData.price) * SCALE_FACTOR;
}
```

Remediation:

Instead of defaulting to 1, revert when `_priceData.price <= 0` or when the timestamp is older than an acceptable freshness threshold. Example:

```
if (_priceData.price <= 0) revert NoOraclePrice();
```

Finding 11: Oracle Timestamp Overflow

Severity: Info

Status: Acknowledged

Description:

Both `AegisOracle.sol` and `AegisOracleJUSD.sol` store the last price update time in a `uint32` slot:

```
struct YUSDUSDPriceData { int256 price; uint32 timestamp; }
...
_priceData.timestamp = uint32(block.timestamp);
```

Because `block.timestamp` is cast down to 32 bits, the value wraps every 4,294,967,296 seconds (roughly February 2106). After the wrap the on-chain timestamp suddenly appears **smaller** than previous values, so any consumer that enforces freshness (e.g., `require(block.timestamp - lastUpdateTimestamp < MAX_STALENESS)`) will misbehave—either rejecting perfectly fresh updates or believing an ancient update is still new.

Although the wrap is far in the future, this contract is meant to be long-lived infrastructure; the bug is already baked into storage and migrations later would require special handling.

Impact:

After ~Feb 7, 2106, the timestamp resets to zero and any freshness checks become unreliable, potentially freezing the protocol or letting stale oracle values through.

Source:

- `contracts/AegisOracle.sol`: `YUSDUSDPriceData.timestamp` and `updateYUSDPrice` lines 5-38
- `contracts/AegisOracleJUSD.sol`: `JUSDUSDPriceData.timestamp` and `updateJUSDPrice` lines 5-38

Code:

```
// contracts/AegisOracle.sol
struct YUSDUSDPriceData {
    int256 price;
    uint32 timestamp; // <--- overflows in 2106
}

function updateYUSDPrice(int256 price) external onlyOperator {
    _priceData.price = price;
    _priceData.timestamp = uint32(block.timestamp);
}
```

Remediation:

- Store timestamps as `uint256` (preferred) or at least `uint48/uint64`, which cover hundreds of thousands of years.
- If storage layout must remain compact, keep the struct but change the type and re-deploy/migrate before 2106.

Finding 12: Slippage Check Ignores Mint Fee, Users Receive Less Than Minimum

Severity: Info

Status: Acknowledged

Description:

`AegisMintingJUSD.mint` computes two values:

1. `jusdAmount = _calculateMinJUSDAmount(...)`, the pre-fee output constrained by oracle prices.
2. `(mintAmount, fee) = _calculateInsuranceFundFeeFromAmount(jusdAmount, mintFeeBP)`.

The slippage gate compares `jusdAmount` to `order.slippageAdjustedAmount`:

```
if (jusdAmount < order.slippageAdjustedAmount) revert PriceSlippage();
```

but the user actually receives `mintAmount = jusdAmount - fee`. If a taker specifies their **net** minimum via `slippageAdjustedAmount`, the transaction still succeeds as long as the gross `jusdAmount` exceeds it—even when the minted amount after fees falls below the user's threshold. For example, with `mintFeeBP = 100` (1%), a user can set `slippageAdjustedAmount = 100 ether`. If the oracle-bound `jusdAmount` resolves to `100 ether`, the check passes, yet the minted output is `99 ether`, violating the user's intent.

This is unlike the redeem path (`approveRedeemRequest`), where the slippage check is run **after** calculating `burnAmount` (post-fee).

Impact:

Users who assume slippageAdjustedAmount is the net amount can be short-changed by up to mintFeeBP/10_000%. Integrators performing quotes based on the min amount they will actually receive may unintentionally let transactions through that violate their internal guards.

Source:

contracts/AegisMintingJUSD.sol, lines ~284-307.

Code:

```
uint256 jusdAmount = _calculateMinJUSDAmount(...);
if (jusdAmount < order.slippageAdjustedAmount) revert PriceSlippage(); // ignores fee
(uint256 mintAmount, uint256 fee) = _calculateInsuranceFundFeeFromAmount(jusdAmount, mintFeeBP);
```

Remediation:

Perform the comparison against the post-fee figure:

```
(uint256 mintAmount, uint256 fee) = _calculateInsuranceFundFeeFromAmount(jusdAmount, mintFeeBP);
if (mintAmount < order.slippageAdjustedAmount) revert PriceSlippage();
```

or clarify in the Order spec that `slippageAdjustedAmount` must be pre-fee and enforce that same convention across all flows. Aligning mint and redeem behaviour (slippage after fees) is less error-prone.

Finding 13: Stale Oracle De-peg Arbitrage on Redemption

Severity: Info

Status: Acknowledged

Description:

If JUSD/YUSD de-pegs (drops) and the AegisOracle manual price is not updated (stays at $1), users can redeem cheap stablecoins for $1 worth of collateral, draining reserves. The protocol uses Math.min(Chainlink price, AegisOracle price) for redemption calculations. The AegisOracleJUSD contract allows operators to manually update the JUSD/USD price via updateJUSDPrice() with NO staleness checks, heartbeat requirements, or automatic updates. During a de-peg event where JUSD drops to $0.90 but the oracle remains at $1.00, users can: 1) Buy cheap JUSD at market ($0.90), 2) Request redemption using the stale $1.00 oracle price, 3) Receive full collateral value based on the higher price. The _calculateRedeemMinCollateralAmount function computes collateral as min(chainlinkAmount, oracleAmount, userAmount). The critical vulnerability is that AegisOracleJUSD has zero automation - it relies entirely on manual operator updates with no staleness validation.

Impact:

Attackers can extract 10-15% excess collateral per redemption cycle during de-peg events. With $10M in reserves, a coordinated attack during a de-peg could drain $1-1.5M before operators respond. Reserve depletion reduces backing for remaining JUSD holders, accelerates de-peg spiral as market participants lose confidence seeing reserves drain, creates MEV opportunity for sophisticated actors monitoring oracle lag, and disproportionately harms honest JUSD holders who cannot exit at favorable rates.

Source:

- File: AegisOracleJUSD.sol, Function: updateJUSDPrice (Lines: 35-39)
- File: AegisMintingJUSD.sol, Function: _getAssetJUSDPriceOracle (Lines: 725-741)
- File: AegisMintingJUSD.sol, Function: _calculateRedeemMinCollateralAmount (Lines: 685-707)
- File: AegisMintingJUSD.sol, Function: approveRedeemRequest (Lines: 216-244)

Code:

```
function updateJUSDPrice(int256 price) external onlyOperator {
  _priceData.price = price;
  _priceData.timestamp = uint32(block.timestamp);
  emit UpdateJUSDPrice(_priceData.price, _priceData.timestamp);
}

function _getAssetJUSDPriceOracle(address asset) internal view returns (uint256, uint8) {
  if (address(aegisOracle) == address(0)) {
    return (0, 0);
  }
```

```
    int256 jusdUSDPrice = aegisOracle.jusdUSDPrice();
    if (jusdUSDPrice <= 0) {
      return (0, 0);
    }
    uint8 jusdUSDPriceDecimals = aegisOracle.decimals();
    (uint256 assetUSDPrice, ) = _getAssetUSDPriceChainlink(asset);
    return ((assetUSDPrice * 10 ** jusdUSDPriceDecimals) / uint256(jusdUSDPrice), jusdUSDPriceDecimals);
  }
```

Proof of Concept:

1. Deploy test environment with AegisMintingJUSD and AegisOracleJUSD
2. Set oracle JUSD price to 100000000 (1.00 USD)
3. Simulate de-peg by setting up test JUSD market at 0.90 rate
4. Submit redemption request for 100,000 JUSD
5. Approve redemption as FUNDS_MANAGER
6. Measure collateral received vs expected
7. Verify ~11% excess collateral due to stale oracle
8. Repeat with oracle staleness at various time intervals to measure impact windows

Remediation:

- Implement staleness checks in _getAssetJUSDPriceOracle: require(block.timestamp - aegisOracle.lastUpdateTimestamp() <= MAX_ORACLE_DELAY) where MAX_ORACLE_DELAY = 1 hour or similar
- Add price deviation bounds: reject oracle prices that deviate >5% from Chainlink without recent update (within 15 minutes)
- Implement automatic price feeds: integrate Chainlink JUSD/USD feed if available, or use TWAP from DEX liquidity pools as backup
- Add circuit breakers: pause redemptions automatically if oracle price hasn't updated in X hours during volatile periods
- Require multiple oracle sources: use median of 3+ price feeds (Chainlink, DEX TWAP, manual oracle) to prevent single point of failure
- Implement progressive delays: redemptions during de-peg events (price <$0.98) require 24-48 hour timelock
- Add keeper automation: use Chainlink Keepers or Gelato to trigger oracle updates based on price deviation thresholds

Finding 14: Zero-Amount Claims Permanently Flag IDs as Claimed

Severity: Info

Status: Acknowledged

Description:

`AegisRewardsManual.claimRewards` (and the non-manual variant) iterate through every `(id, amount)` pair in the signed `ClaimRewardsLib.ClaimRequest`. For each entry that passes the finalized/expiry checks and has remaining balance, the function immediately sets `_addressClaimedRewards[msg.sender][id] = true` **before** checking whether `amounts[i]` is non-zero:

```
_addressClaimedRewards[_msgSender()][claimRequest.ids[i]] = true;
_rewards[claimRequest.ids[i]].amount -= claimRequest.amounts[i];
totalAmount += claimRequest.amounts[i];
```

If the signer (or buggy client) includes an ID with `amount = 0`, that ID is still marked as claimed for the caller even though no funds are transferred (since subtracting zero leaves the reward untouched and no YUSD is sent). Later, if additional rewards are deposited under the same ID, the user can never claim them because `_addressClaimedRewards[user][id]` remains true. This enables a malicious signer to DoS an address by issuing a batch claim containing their IDs with zero amounts.

Impact:

Each user's "already claimed" flag is permanent per ID. A malicious or careless reward distributor can burn a user's right to future rewards simply by signing a claim that includes the correct ID but an amount of zero. No compensation is ever paid, and the user has no on-chain recourse.

Source:

- `contracts/AegisRewardsManual.sol` lines ~103-126
- same pattern in `contracts/AegisRewards.sol`.

Code:

```
if (_addressClaimedRewards[user][id]) continue;
_addressClaimedRewards[user][id] = true; // happens even when amount == 0
_rewards[id].amount -= claimRequest.amounts[i]; // subtracts zero
```

Remediation:

Require `claimRequest.amounts[i] > 0` before setting `_addressClaimedRewards`. Either skip zero-amount entries entirely or revert the transaction to force clean inputs. Example fix:

```
uint256 amount = claimRequest.amounts[i];
if (amount == 0) continue;
_addressClaimedRewards[msg.sender][id] = true;
_rewards[id].amount -= amount;
```

Disclaimer

This security report (“Report”) is provided by FailSafe (“Tester”) for the exclusive use of the client (“Client”). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe’s prior written consent in each instance.

This Report is not, nor should it be considered, an “endorsement” or “disapproval” of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology’s proprietors, business, business model, or legal compliance.

This Report should not be used in any way to make decisions around investment or involvement with any particular project. This Report in no way provides investment advice, nor should it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe’s position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe’s goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RE-

SPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.