OpenZeppelin | security

Flare FAssets Audit

January 27, 2026

Table of Contents

Summary

Type	
Timeline	From 2025-10-13 To 2025-12-02
Languages	Solidity

Total Issues	49 (30 resolved, 3 partially resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	7 (4 resolved, 2 partially resolved)
Low Severity Issues	12 (7 resolved)
Notes & Additional Information	27 (16 resolved, 1 partially resolved)
Client Reported Issues	2 (2 resolved)

Scope

OpenZeppelin performed an audit of the flare-foundation/fassets repository at commit 9823c6.

In scope were the following files:

```
contracts
├── agentOwnerRegistry
│   └── implementation
│       ├── AgentOwnerRegistry.sol
│       └── AgentOwnerRegistryProxy.sol
├── agentVault
│   ├── implementation
│   │   ├── AgentVault.sol
│   │   └── AgentVaultFactory.sol
│   └── interfaces
│       ├── IAgentVault.sol
│       └── IAgentVaultFactory.sol
├── assetManager
│   ├── facets
│   │   ├── AgentAlwaysAllowedMintersFacet.sol
│   │   ├── AgentCollateralFacet.sol
│   │   ├── AgentInfoFacet.sol
│   │   ├── AgentPingFacet.sol
│   │   ├── AgentSettingsFacet.sol
│   │   ├── AgentVaultAndPoolSupportFacet.sol
│   │   ├── AssetManagerDiamondCutFacet.sol
│   │   ├── AvailableAgentsFacet.sol
│   │   ├── ChallengesFacet.sol
│   │   ├── CollateralReservationsFacet.sol
│   │   ├── CollateralTypesFacet.sol
│   │   ├── CoreVaultClientFacet.sol
│   │   ├── CoreVaultClientSettingsFacet.sol
│   │   ├── EmergencyPauseFacet.sol
│   │   ├── LiquidationFacet.sol
│   │   ├── MintingDefaultsFacet.sol
│   │   ├── MintingFacet.sol
│   │   ├── RedemptionConfirmationsFacet.sol
│   │   ├── RedemptionDefaultsFacet.sol
│   │   ├── RedemptionRequestsFacet.sol
│   │   ├── RedemptionTimeExtensionFacet.sol
│   │   ├── SettingsManagementFacet.sol
│   │   ├── SettingsReaderFacet.sol
│   │   ├── SystemInfoFacet.sol
│   │   ├── SystemStateManagementFacet.sol
│   │   ├── UnderlyingBalanceFacet.sol
│   │   └── UnderlyingTimekeepingFacet.sol
│   ├── implementation
│   │   ├── AssetManager.sol
```

```
│   │   ├── AssetManagerBase.sol
│   │   └── AssetManagerInit.sol
│   ├── interfaces
│   │   ├── IAgentAlwaysAllowedMinters.sol
│   │   ├── IAssetManager.sol
│   │   └── IAssetManagerEvents.sol
│   └── library
│       ├── AgentBacking.sol
│       ├── AgentCollateral.sol
│       ├── AgentPayout.sol
│       ├── Agents.sol
│       ├── AgentUpdates.sol
│       ├── CollateralTypes.sol
│       ├── Conversion.sol
│       ├── CoreVaultClient.sol
│       ├── EmergencyPause.sol
│       ├── Globals.sol
│       ├── Liquidation.sol
│       ├── LiquidationPaymentStrategy.sol
│       ├── Minting.sol
│       ├── PaymentConfirmations.sol
│       ├── PaymentReference.sol
│       ├── RedemptionDefaults.sol
│       ├── RedemptionQueueInfo.sol
│       ├── RedemptionRequests.sol
│       ├── Redemptions.sol
│       ├── SettingsInitializer.sol
│       ├── SettingsUpdater.sol
│       ├── SettingsValidators.sol
│       ├── TransactionAttestation.sol
│       ├── UnderlyingBalance.sol
│       └── UnderlyingBlockUpdater.sol
├── assetManagerController
│   ├── implementation
│   │   ├── AssetManagerController.sol
│   │   └── AssetManagerControllerProxy.sol
│   └── interfaces
│       └── IAssetManagerController.sol
├── collateralPool
│   ├── implementation
│   │   ├── CollateralPool.sol
│   │   ├── CollateralPoolFactory.sol
│   │   ├── CollateralPoolToken.sol
│   │   └── CollateralPoolTokenFactory.sol
│   └── interfaces
│       ├── ICollateralPool.sol
│       ├── ICollateralPoolFactory.sol
│       ├── ICollateralPoolToken.sol
│       └── ICollateralPoolTokenFactory.sol
├── coreVaultManager
│   ├── implementation
│   │   ├── CoreVaultManager.sol
│   │   └── CoreVaultManagerProxy.sol
│   └── interfaces
│       └── ICoreVaultManager.sol
├── diamond
```

```
│   ├── facets
│   │   ├── DiamondCutFacet.sol
│   │   ├── DiamondInit.sol
│   │   └── DiamondLoupeFacet.sol
│   ├── implementation
│   │   └── Diamond.sol
│   ├── interfaces
│   │   ├── IDiamond.sol
│   │   ├── IDiamondCut.sol
│   │   └── IDiamondLoupe.sol
│   └── library
│       └── LibDiamond.sol
├── fassetToken
│   ├── implementation
│   │   ├── CheckPointable.sol
│   │   ├── FAsset.sol
│   │   └── FAssetProxy.sol
│   ├── interfaces
│   │   ├── ICheckPointable.sol
│   │   └── IFAsset.sol
│   └── library
│       ├── CheckPointHistory.sol
│       └── CheckPointsByAddress.sol
├── fdc
│   ├── implementation
│   │   ├── FdcHubMock.sol
│   │   ├── FdcInflationConfigurationsMock.sol
│   │   ├── FdcRequestFeeConfigurationsMock.sol
│   │   └── FdcVerificationMock.sol
│   └── interfaces
│       ├── IFdcHub.sol
│       ├── IFdcInflationConfigurations.sol
│       ├── IFdcRequestFeeConfigurations.sol
│       └── IFdcVerification.sol
├── flareSmartContracts
│   ├── implementation
│   │   ├── AddressUpdatable.sol
│   │   ├── AddressUpdatableMock.sol
│   │   ├── AddressUpdaterMock.sol
│   │   ├── PriceReader.sol
│   │   └── StateConnector.sol
│   └── interfaces
│       ├── IAddressUpdatable.sol
│       └── IPriceReader.sol
├── ftso
│   ├── implementation
│   │   ├── FtsoV2PriceStore.sol
│   │   └── FtsoV2PriceStoreProxy.sol
│   └── interfaces
│       └── IFtsoV2PriceStore.sol
├── governance
│   ├── implementation
│   │   ├── Governed.sol
│   │   ├── GovernedBase.sol
│   │   ├── GovernedMock.sol
│   │   ├── GovernedProxyImplementation.sol
```

```
│   │   ├── GovernedUUPSProxyImplementation.sol
│   │   └── GovernedWithTimelockMock.sol
│   └── interfaces
│       └── IGoverned.sol
├── userInterfaces
│   ├── IAgentOwnerRegistry.sol
│   ├── IAgentPing.sol
│   ├── IAgentVault.sol
│   ├── IAssetManager.sol
│   ├── IAssetManagerController.sol
│   ├── IAssetManagerEvents.sol
│   ├── ICollateralPool.sol
│   ├── ICollateralPoolToken.sol
│   ├── ICoreVaultClient.sol
│   ├── ICoreVaultClientSettings.sol
│   ├── ICoreVaultManager.sol
│   ├── IFAsset.sol
│   ├── IRedemptionTimeExtension.sol
│   └── data
│       ├── AgentInfo.sol
│       ├── AgentSettings.sol
│       ├── AssetManagerSettings.sol
│       ├── AvailableAgentInfo.sol
│       ├── CollateralReservationInfo.sol
│       ├── CollateralType.sol
│       ├── EmergencyPause.sol
│       ├── RedemptionRequestInfo.sol
│       ├── RedemptionTicketInfo.sol
│       └── SupportedCollateral.sol
└── utils
    ├── implementation
    │   ├── CustomError.sol
    │   ├── Impermeable.sol
    │   └── Storage.sol
    └── library
        ├── Addresses.sol
        ├── CollateralTypeInt.sol
        ├── CustomErrorMock.sol
        ├── EffectiveEmergencyPause.sol
        ├── IERC5267.sol
        ├── MathUtils.sol
        ├── MerkleTree.sol
        ├── SafeMath64.sol
        ├── SafePct.sol
        └── Transfers.sol
```

System Overview

The FAsset system is a decentralized protocol designed to create synthetic assets, known as FAssets, on the Flare Network. These synthetics are 1:1 backed by real assets on other blockchains, such as Ripple (XRP) or Bitcoin (BTC). The system's primary goal is to enable the use of non-native assets in the Flare ecosystem without relying on a traditional, centralized bridge. Its trustless nature is founded on a combination of verifiable cross-chain attestations, significant over-collateralization provided by a network of agents, and crypto-economic incentives that align the interests of all participants.

The architecture is modular, with a central `AssetManager` contract per underlying chain, acting as an orchestration engine that coordinates the distinct roles of FAsset tokens, Agent Vaults, Collateral Pools, a governing Controller, and the Core Vault.

AssetManager

The `AssetManager` is the core logic and state-management hub of the entire system. It functions as a diamond proxy, delegating function calls to various facets (implementation contracts), which makes the system highly modular and upgradeable. This contract is the primary entry point for all user-initiated operations, and it orchestrates the complex interactions between all other components.

- **Minting Orchestration:** When a user wishes to create FAssets, the `AssetManager` contract manages the process. It first selects an available Agent from a FIFO queue system, and locks a portion of that Agent's collateral at a minimum collateral ratio (set by governance and potentially enhanced by the Agent), ensuring that the collateral value exceeds the FAssets being minted. It then provides the user with the Agent's underlying payment address and a unique payment reference that identifies this specific minting request. Once the user sends their assets (e.g., XRP), the Flare Data Connector (FDC) attests to this payment, and upon successful verification of this proof, `AssetManager` mints new FAsset tokens to the user's address.
- **Redemption Management:** To redeem underlying assets, a user sends their FAssets to the `AssetManager` contract, which burns them and starts a redemption. Redemptions are processed through a FIFO queue of tickets that are created when FAssets are minted: every mint creates a ticket that represents the Agent's future obligation to

redeem those FAssets. The `AssetManager` contract takes tickets from the front of the queue and turns them into redemption requests, each assigned to the corresponding Agent. The Agent must then send the underlying asset to the user's address on the underlying chain, using a unique payment reference, and prove the payment via FDC before a deadline. If the Agent completes this payment and provides proof, their locked collateral is released. If the Agent defaults, `AssetManager` seizes the Agent's collateral and pays it directly to the redeemer, making the user whole, often with a premium.

- **Liquidation and Challenge Enforcement:** `AssetManager` enforces the system's security through two forms of liquidation. It continuously tracks the *economic health* of every Agent by comparing their collateral value against their minted FAsset value. If an Agent's collateral ratio falls below the minimum required threshold, `AssetManager` opens that position to economic liquidation. Crucially, the system also polices *protocol violations* through a challenge mechanism. Any observer can challenge an Agent by submitting cryptographic proof that the Agent has misbehaved. Proved violations include the following:
  - **Illegal Payments:** Moving underlying assets from their custody address for any reason other than a valid redemption.
  - **Double Payments:** Using the same underlying transaction to mint FAssets more than once.

A successful challenge immediately triggers a "full liquidation," where the Agent is penalized, their collateral is seized to compensate users and the challenger, and the agent is eventually removed from the system. This provides a powerful incentive for Agents to strictly follow protocol rules at all times.

FAsset

The `FAsset` contract is the ERC-20-compliant token that represents a user's 1:1 claim on an underlying asset held within the system. For every FXRP token in circulation, there is one XRP held in custody by an Agent and backed by on-chain collateral.

- **Controlled Supply:** The total supply of FAssets is strictly controlled. Only the `AssetManager` contract possesses the authority to mint new tokens (during the minting process) and burn existing ones (during redemption or liquidation). This ensures that the number of FAssets in circulation never exceeds the amount of underlying assets secured by the system.

- **Historical Checkpointing:** The contract includes `CheckPointable` functionality, which periodically saves historical token balances. This feature enables efficient and trustless lookups of past balances without requiring expensive, full on-chain snapshots, and it is critical for calculating rewards or other time-dependent distributions.

Agent Owner Registry

The `AgentOwnerRegistry` contract maintains Agent records, including whitelist status and metadata. In addition, it lets agents designate a trusted work address authorized to act on their behalf.

- **Governance and Managers**: They may whitelist or revoke agents and manage Agent metadata in `AgentOwnerRegistry`.
- **Agents**: They can set and update their own work address. They are responsible for choosing a trustworthy address that will not act maliciously.

Agents (Custodians) and AgentVaults

Agents are the custodians in the FAsset system. They are independent entities who are responsible for holding the underlying assets (e.g., XRP) in their custody on the native chain. To participate, an Agent must be whitelisted in the `AgentOwnerRegistry` contract and lock a significant amount of value on the Flare Network as collateral. This collateral is held in a dedicated `AgentVault` smart contract.

- **`AgentVault`:** Each Agent has their own `AgentVault` instance. This acts as their personal on-chain safe, holding the collateral (e.g., stablecoins, Collateral Pool Tokens, etc.) that they have committed to back FAssets. This segregates risk, ensuring that the default of one Agent does not affect the collateral of others.
- **Management of Underlying Assets:** The Agent's primary responsibility is to manage the underlying assets. When a user mints FAssets, they send funds to an address controlled by the Agent. When a user redeems, the Agent is responsible for sending the underlying assets back to the user. Agents can also pre-fund their balance by sending underlying assets to their custody address without minting FAssets. This creates a "free balance" (where `underlying balance > backed FAssets`), which allows them to perform instant self-minting later on.
- **Agent Configuration:** Agents can configure various operational parameters, including minting fees, collateral ratios (above system minimums), and the share of fees allocated to the collateral pool. These settings are timelocked to prevent sudden changes that

could affect users or the pool. Agents must also maintain minimum collateral ratios for both vault and pool collateral, and must hold a minimum amount of pool tokens to participate in minting.

- **Non-Custodial Seizure:** While an Agent can deposit and (with a time delay) withdraw collateral from their `AgentVault`, they cannot prevent `AssetManager` from seizing it. In a liquidation or redemption default, the `AssetManager` contract is authorized to unilaterally pull collateral from the vault to satisfy the Agent's debt, ensuring that users are made whole.
- **Withdrawal Controls:** Agents must announce collateral withdrawals in advance and wait for a specified time period before executing them. This announcement period prevents Agents from quickly removing collateral if challenged, ensuring that the system can respond to protocol violations.
- **Standardized Deployment:** All `AgentVault` instances are deployed through a single `AgentVaultFactory` contract, ensuring that every vault is identical and adheres to system rules. The Agent's underlying address is set at vault creation time and is immutable, providing a stable payment destination for users.

`CollateralPool`

The `CollateralPool` contract serves as a system-wide insurance fund and a second layer of security. It is funded by liquidity providers who deposit `NAT` in exchange for `CollateralPoolToken`s, which represent a share of the pool and its future earnings.

- **Systemic Backstop:** In the event an Agent defaults and their personal `AgentVault` collateral is insufficient to cover their obligations (a scenario possible during extreme market volatility), the funds of the `CollateralPool` contract are used to cover the remaining shortfall.
- **Incentivized Liquidity:** Liquidity providers are incentivized to fund the pool by earning a portion of the system's operational fees (e.g., from minting and redemptions). However, their provided capital is at risk, which encourages them to monitor the overall health of the system and the Agents within it.
- **Solvency Protection:** Under stress conditions, to exit the pool, a liquidity provider must burn not only their `CollateralPoolToken`s but also a corresponding amount of FAssets. This mechanism ensures that the pool can never be drained in a way that would leave the system under-collateralized.

Before an Agent may mint new FAssets, its vault **must hold a minimum quantity of `CollateralPoolToken`s**. This aligns incentives: Agents share in the downside risk of defaults elsewhere and therefore have a direct interest in maintaining healthy collateral ratios.

When an Agent enters liquidation due to a shortfall in the `AgentVault` contract, `AgentVault`'s `CollateralPoolToken`s can be slashed.

Core Vault

In addition to the decentralized network of Agents, the FAsset system supports a `CoreVaultManager` contract, which enables a different custody model based on a single, centrally-managed reserve. This is typically operated by an institutional-grade custodian and allows for integration with more traditional financial infrastructure. The primary advantage for Agents is the ability to transfer their underlying assets into the Core Vault, which frees up the collateral that was backing those assets. This enhances capital efficiency, allowing Agents to withdraw their freed collateral or use it to mint more FAssets.

The Core Vault functions as a pooled reserve of underlying assets. Agents can send assets to the vault through a specialized on-chain "redemption request" that, upon confirmation of the underlying transfer, releases their locked collateral. To retrieve assets, an Agent can make a formal "request for return" (which temporarily locks their collateral again) or directly redeem their own FAssets against the vault's reserves. This direct redemption mechanism is also available to regular FAsset holders, allowing them to bypass the Agent-based redemption process, provided their underlying address is approved by governance. However, this method of redemption can be slower than Agent-based redemptions.

All interactions with the Core Vault are managed through specific `AssetManager` facets, and FAsset governance can configure its operational parameters, including the ability to halt all transfers in an emergency. This model deliberately trades the system's native crypto-economic security for the operational and reputational security of a trusted, centralized custodian. The guarantee of redemption from the vault relies on the custodian's off-chain operations and reputation, as there is no on-chain collateral to seize in a default scenario.

`AssetManagerController`

The `AssetManagerController` contract is the administrative heart of the FAsset protocol, holding the highest level of authority over all other contracts. It is owned and managed by a governance entity (expected to be a secure multisig or a decentralized autonomous organization with a timelock).

- **Parameter Management:** `AssetManagerController` is responsible for setting and modifying all critical system parameters. This includes adjusting collateral ratios, setting

the duration of timelocks, defining the fees for minting and redemption, configuring payment deadlines, setting liquidation parameters, managing collateral types, and updating critical contract addresses (such as FDC verification, price readers, and factory contracts).

- **Upgrade Authority:** As the owner of the `AssetManager` contract's diamond proxy and other key contracts, `AssetManagerController` has the exclusive power to perform system upgrades. This allows governance to fix bugs, introduce new features, or replace existing facet logic.

- **Emergency Powers:** Governance can act through the controller to pause or completely halt system operations in the event of a critical security threat, an oracle failure, or other black swan events, providing a crucial circuit breaker to protect user funds. The system supports three progressively stricter pause levels:

    - `START_OPERATIONS`: Prevents starting mint, redeem, liquidation, and core vault transfer/return
    - `FULL`: Everything from `START_OPERATIONS`, plus the prevention of finishing or defaulting already started mints and redeems
    - `FULL_AND_TRANSFER`: Everything from FULL, plus the prevention of FAsset transfers

Security Model and Trust Assumptions

The FAsset system's security model is built upon a foundation of crypto-economic incentives, over-collateralization, and decentralized cross-chain attestations. While designed to minimize reliance on centralized trust, its integrity and liveness are critically dependent on the honest and timely operation of several external systems, specific smart contracts, and designated privileged actors.

The system assumes that all privileged actors will act correctly and in good faith, that external systems will operate reliably, that configurations will be accurate, and that operational dependencies will be maintained. The correct configuration, ongoing monitoring, and non-malicious behavior of these entities are paramount. Any failures in these assumptions could compromise system security,

Governance

The system relies on a powerful governance entity, which controls the `AssetManagerController`. This is implemented as a 6/15 Multisig, operating with both timelocked and non-timelocked operations. Governance is highly trusted, having the authority to:

- largely, arbitrarily define and update system-wide parameters (e.g., collateral ratios, fees, lot sizes, attestation windows, minimum threshold of `AgentVault` share in the pool required for minting, etc.)
- control the `AddressUpdater`, responsible for maintaining a mapping of contract addresses, enabling the governance to replace critical contracts, including the `AssetManagerController` itself, allowing for arbitrary upgrades
- approve new vault collateral tokens
- set crucial Flare Time Series Oracle (FTSO) parameters (e.g., max spread) and designate trusted oracle providers
- pause the system during emergencies or updates using three progressively stricter pause levels
- whitelist agents who are authorized to operate vaults and mint FAssets in the system

Most governance actions are time-locked and follow a two-step process. First, governance must initiate an action. After a specified time has passed, designated entities called **executors** can finalize the action. Some actions are also `rateLimited`, meaning they cannot be executed too frequently. However, not all governance actions are time-locked or rate-limited, giving governance the flexibility to act quickly during special circumstances such as hacking attempts.

Although most actions include a delay before executors can finalize them (protecting users from immediate, unexpected changes) there is no upper time bound on when executors must finalize these actions. It is assumed that governance will act in good faith, update parameters responsibly, and the executors will execute actions promptly once the delay has passed.

Governance should also exercise particular caution with complex, multi-step actions that cannot be executed atomically due to block gas limit. System contract updates (`WNat`, `AddressUpdater`, or `AssetManagerController`) across multiple Asset Managers may require batched transactions when there are too many managers to update in single block. Additionally, `WNat` token migration is a two-phase process: first updating the Asset Manager settings, then migrating each Collateral Pool's funds. During these operations, some Asset Manager settings or Collateral Pools may temporarily reference different contract versions. To avoid temporary desynchronization issues, it may be necessary to pause the protocol while performing such updates.

Governance Settings

The governance address, timelock duration and permitted executors are defined by a genesis-deployed contract on the Flare mainnet.

`AddressUpdater`

The `AddressUpdater` contract maintains canonical addresses for core system components, including `WNat`, the `AssetManagerController`, and itself. Only governance should be able to modify these entries. Through this mechanism, governance can perform a complete replacement of the `AssetManagerController` contract, upgrade the `WNat` contract, or reassign the `AddressUpdater` role to an arbitrary address. Correct configuration and exclusive governance control of the `AddressUpdater` is a foundational trust assumption. Any compromise or misconfiguration would allow an attacker to redirect critical contract references and subvert the entire protocol.

FTSO Oracle Providers and Trusted Oracle Providers (External Data Integrity)

The FAsset system critically depends on the `FtsoV2PriceStore` contract for accurate and timely price data, which underpins every collateralisation check, liquidation trigger, and redemption valuation. Price feeds enter the system through two complementary channels.

The first is the decentralised FTSO (Flare Time Series Oracle), where independent data providers submit prices each voting epoch. The protocol aggregates these submissions via a weighted median, and the resulting value is anchored on-chain through a Merkle proof verified against the Relay contract. The second channel consists of a smaller, governance-curated set of Trusted Oracle Providers who may submit prices during a brief window immediately after each FTSO epoch closes. Their submissions are used to compute a secondary "trusted" median that can serve as a faster or fallback reference.

Both channels carry inherent risk vectors. FTSO providers are economically incentivized through the broader Flare reward mechanism, yet low participation or delayed submissions can cause the published price to lag real market conditions—particularly dangerous during periods of rapid volatility when timely liquidation is essential. Trusted providers, by contrast, operate under governance appointment rather than protocol-level staking, meaning their direct on-chain incentive to report accurately is weaker. A compromised or negligent trusted provider could inject skewed prices that, if accepted, would mis-value collateral and expose the system to under-collateralized positions or premature liquidations.

Both FTSO and Trusted Oracle Providers can be subject to stale prices, misbehavior, manipulation, or reduced participation due to network issues. These external dependencies fall

outside the scope of this audit. As such, the correct operation of both provider sets is assumed. These dynamics make oracle integrity a first-order security concern: any systematic bias, delay, or manipulation in either price channel directly threatens the accuracy of collateral ratios and the fairness of liquidation outcomes across the entire FAsset ecosystem.

Custodians (Highly Trusted External Role)

For certain underlying assets, the system might rely on custodians. This is particularly relevant for assets like Bitcoin that require multi-sig patterns. Such custodians are "Fully Trusted", controlling funds in the Core Vault and capable of refusing payments with no recovery mechanism. This introduces a significant centralization risk and reliance on the operational security and honesty of these external entities.

Flare Data Connector (FDC) (External Data Integrity)

The FAsset protocol relies on the Flare Data Connector which falls outside the scope of this audit. It is used to attest cross-chain payment events—minting confirmations, redemption proofs, and challenge evidence all pass through FDC-signed attestations rather than cryptographic validity proofs. Correct operation of the FDC, including its resistance to collusion and censorship, is a foundational trust assumption.

Agent Honesty and Economic Incentives

While Agents are over-collateralized to mitigate individual defaults, the system assumes they will generally act honestly and remain live to process redemption requests promptly. Analysis of the economic model highlights potential scenarios where Agents could act maliciously against the Collateral Pool or exploit edge cases in liquidation mechanisms.

Collateral Pool Participants

Collateral pool token holders bear residual risk from Agent failures and under-collateralization. The system assumes that these participants understand the risk profile (including potential losses in extreme scenarios) and that pool parameters set by agents and governance are sufficient to keep the pool solvent under stressed market conditions.

Liquidations

If the collateral ratio of an Agent's vault or collateral pool (i.e., collateral value divided by the value of the backed FAssets) falls below a governance-defined threshold, anyone can initiate a

liquidation. A liquidator burns a chosen amount of FAssets and receives an equivalent value of collateral plus a premium. This premium increases over time to further incentivize liquidators.

Under normal circumstances, liquidations are profitable. However, the protocol does not guarantee that every liquidation will be profitable (this would happen e.g. if the collateral price had fallen too low) and liquidators must independently assess whether a liquidation is in their interest. In addition, the protocol does not include any special mechanism for handling scenarios in which the peg between an FAsset and its underlying asset has been lost and treats them as always having the same price.

Core Financial Invariants and Solvency

The system assumes that its core financial invariants (e.g., 100% backing, over-collateralization) will always hold true. However, specific edge cases in the accounting logic, such as the possibility of creating unbacked redemption tickets during certain Core Vault transfers, could threaten these invariants and must be handled cautiously.

Underlying Chain Reliability and Integration

The FAsset system's security is inextricably linked to the integrity, stability, and finality of the underlying blockchains. The system's design is heavily influenced by the features of its primary supported assets, such as the reliance on XRP-native escrows. Integrating chains with different characteristics, for example, Bitcoin's volatile block times or alternative multisig schemes, introduces significant technical challenges and dependencies that must be carefully managed.

System Performance

Having multiple Agents in the system improves its overall performance and resilience. With more Agents available, minting and redemption requests can be distributed across multiple Agents, reducing bottlenecks and improving system throughput and avoiding a single point of failure.

Token Standards Compliance

The system assumes that all whitelisted vault collateral tokens comply with standard ERC-20 behavior and do not implement problematic patterns such as:

- **callback tokens** that execute code during transfers
- **double entry point tokens** that have multiple interfaces or non-standard transfer behaviors

- **rebasing tokens** that change balances automatically
- **fee-on-transfer tokens** that deduct fees during transfers

Privileged Roles

Throughout the codebase, the following privileged roles and actions were identified:

- **Governance:** As described in the previous section, governance can set and update critical system parameters, upgrade system contracts, and pause the system for a duration of its choice. Governance is also responsible for assigning and revoking all the privileged roles within the system.

- **Emergency Pause Senders:** Defined and dynamically updatable by governance, these roles can pause `AssetManagers` at three distinct levels (with progressively stricter restrictions), up to a full freeze of FAsset movements. They can select the pause duration up to governance-defined limits and are restricted in how frequently they can invoke pauses within a given time period. However, the system remains vulnerable to disruption through multiple short pauses that technically comply with the limits. While the documentation does not specify when each pause level should be used, they are expected to be activated during critical updates or in response to security incidents.

- **Executors:** Executors are responsible for finalizing actions initiated by governance. They are expected to remain active and promptly execute actions once the specified delay has passed. Furthermore, executors, along with governance, are permitted to call the `consolidateSmallTickets` and `upgradeWNatContract` functions directly, bypassing the two-step process.

- **Trusted Price Providers:** Beyond FTSO prices, governance can designate trusted addresses to submit prices for each voting round. The median of these submissions serves as a fallback price reference.

- **Agents:** Agents are expected to promptly fulfill FAsset minting and redemption requests. They have also some limited power related to their collateral pool, as they can delegate deposited WNat and claim the corresponding rewards on behalf of the pool. Additionally, agents control several configuration parameters governing their vault and pool operations, for example, minting fees, the share of fees allocated to the pool, and various collateral ratios — including the threshold below which normal pool exits are blocked, and the ratios required for minting. These parameters are assumed to be set responsibly by agents. For instance, although not strictly enforced on-chain when the Agent is

created, the pool-exit collateral ratio is assumed to be configured above the system's minimum collateral ratio.

    - **Always Allowed Minters**: Agents can specify always allowed minters who can mint from the Agent even when the Agent is not in the publicly available agents list, and when doing so, no collateral reservation fee is charged.

- **Managers**: These entities support governance by handling routine operations, including whitelisting and revoking agents, and updating Agent metadata in the `AgentOwnerRegistry` contract.

High Severity

H-01 Agent Vault Ownership Hijacking via Whitelisting Race Condition

The `setWorkAddress` function allows a whitelisted management address to bind a single "work" address that can act on its behalf. It explicitly rejects using a currently whitelisted address as the work address, to avoid ownership confusion during agent-vault creation. In that flow, the system first maps the caller to its management via `_getManagementAddress`, and only then enforces the whitelist on the resolved management in `createAgentVault`. However, this protection does not cover the case where a non-whitelisted work address later becomes whitelisted, because whitelisting does not clear legacy work→management links.

Step-by-step Attack Simulation

1. Attacker A (already whitelisted) binds victim B (not whitelisted) as A's work address via `setWorkAddress(B)`; this succeeds because B is not yet whitelisted. A can also front-run governance's upcoming whitelisting of B by monitoring the mempool and sending `setWorkAddress(B)` just before the whitelisting transaction executes, ensuring B is non-whitelisted at A's call time while locking in the work→management link.
2. Governance later whitelists B using `whitelistAndDescribeAgent`, which calls the internal `_addAddressToWhitelist` but does not validate or clear existing `workToMgmtAddress` entries. The legacy mapping persists.
3. When B creates an agent vault, ownership resolution maps `B → A` in `_getManagementAddress` and the whitelist requirement in `createAgentVault` is satisfied by A. At this point, the vault's management address is A, with B as A's work address, allowing B to add collateral while enabling A to steal it.

Consider checking and severing any work↔management link whenever whitelisting a new address.

Update: *Resolved at commit* `cd61c30`.

> *Setting the work address is now a two-step process. First, the management address calls* `setWorkAddress`*. Then, the designated work address must explicitly confirm the assignment by calling* `acceptWorkAddressAssignment`*, passing the management*

> *address as a parameter. This ensures that a work address can only be assigned if it is controlled by the intended party, preventing attackers from assigning arbitrary addresses they do not control.*

Medium Severity

M-01 Terminal Underlying Balance State Mismatch Causes Underlying Balance Invariant Break

A late payment that is blocked for a redemption which is already in default can incorrectly pass the validation logic: the amount-handling path treats the `PAYMENT_BLOCKED` exception as acceptable, which bypasses both the late-payment check and the already-defaulted check. As a result, `confirmRedemptionPayment` reaches the assertion requiring an active redemption and reverts, leaving the redemption unclosable.

Since the revert prevents updating the underlying balance to account for gas spent on the defaulted redemption, the Agent's tracked balance remains artificially inflated. This allows the Agent to withdraw excess underlying after default, driving the actual balance below `requiredUnderlyingBalance`. This withdrawal cannot be challenged, as `freeBalanceNegativeChallenge` still treats the redemption as pending. Consequently, `balanceAfterPayments` increases, preventing the challenger from liquidating the Agent.

Consider treating such redemptions as closed in the challenge logic when a late, blocked payment is provided for an already-defaulted redemption. Doing so would ensure that `freeBalanceNegativeChallenge` and `balanceAfterPayments` reflect the terminal state.

Update: *Resolved at commit `44b588f`. `_validatePayment` now uses independent `if` statements, ensuring that all checks are executed every time.*

M-02 `collateralsUnderwater` Bitmap Allows Permanent Reduction of Agent's Responsibility After Temporary Pool Underwater Event

When a liquidation begins, the `_startLiquidation` function sets the `collateralsUnderwater` bitmap of the Agent to indicate which components (the vault, the collateral pool, or both) are underwater. This bitmap can be set every time `liquidate()` is invoked, but is only reset once the liquidation ends and both the vault and the collateral pool are healthy.

Since this bitmap determines the Agent's responsibility, this behavior creates a persistent state that may no longer reflect actual risk. Specifically, when both the vault and the collateral pool are marked as underwater, the Agent's pool responsibility is halved. If the collateral pool becomes temporarily unhealthy and later recovers (for example due to new deposits improving its collateral ratio) the bitmap is not updated. It remains in the "both underwater" state until liquidation fully ends, and both components are healthy. As a result, the Agent continues to benefit from reduced responsibility even though the collateral pool has recovered, placing an unfair burden on other pool participants.

Consider updating the `collateralsUnderwater` bitmap whether either the vault or the collateral pool recovers instead of waiting for liquidation to end with both components healthy. This would allow the collateral pool to recover independently of the Agent's behavior, ensuring fairness for depositors.

Update: *Resolved at commit 049fc83.*

M-03 Missing Freshness, Sanity, and Participation Checks in FTSOV2PriceStore Allow Use of Stale or Invalid Prices

The `FTSOV2PriceStore` contract stores prices from the FTSO and trusted-providers prices together with their corresponding voting round. The `getPrice` and `getPriceFromTrustedProviders` functions return the latest available price along with the timestamp at which the voting round ends. These functions are used during liquidations and other critical protocol operations that rely on accurate and timely pricing.

However, although these functions return a timestamp, no freshness/staleness check is performed to ensure that the price being returned is recent enough. This design is compatible with the assumption that the protocol must operate under any circumstances, even if off-chain

FTSO components fail. But if an issue arises in the off-chain submission process, the system may be left with stale prices and continuing operations based on outdated pricing may be risky.

Furthermore, the struct includes a `turnoutBIPS` field that represents the weighted participation of oracles for that voting round. This metric is not stored or exposed by the contract. As a result, there is no on-chain visibility into whether a price was derived from strong oracle consensus or from minimal participation. Low turnout could indicate network issues, oracle failures, or potential manipulation, yet consumers of the price data have no way to assess this risk.

While the intention may be to avoid halting the protocol, introducing a reasonable freshness threshold and validating `turnoutBIPS` would improve safety without violating that design assumption, under the weak assumption that off-chain components may only fail temporarily but not permanently. In addition, although the FTSO is assumed to be trusted for this audit, basic sanity checks (e.g., ensuring that the price is non-zero, would further enhance the robustness at minimal cost).

Consider adding freshness, sanity, and participation checks to reduce the risk of operating the protocol based on stale, unreliable or invalid prices.

Update: *Partially Resolved at commits c759e33, 1b43d41, and 0ba59bc. Turnout is now checked against a governance-adjustable minimum, and prices are not published if turnout is too low (an event is emitted instead). Sanity checks (zero-price validation) are also enforced. However, price freshness is not enforced, and prices could become stale. The Flare team stated:*

> *FTSO prices are created in a decentralized way and can be published by anybody to FTSOV2PriceStore, so we believe that price staleness is easy to avoid by off-chain mechanisms. Just like for current block number/time, all the actors are informed that the prices should be published regularly. Moreover, we think that alllowing prices to be slightly stale is less harmful than unexpectedly blocking the protocol. If, on the other hand, the prices deviate a lot from the real value (which we cannot detect in the contract), the monitoring tools will trigger emergency pause.*

M-04 Stale Underlying Block Tracking Distorts Deadlines and Can Prevent Non-Payment Proofs

When the tracked underlying block is stale, the time delta used in `_lastPaymentBlock` can become very large. The function derives `lastUnderlyingBlock` by multiplying this delta by

`averageBlockTimeMS`, so any inaccuracy in this average is magnified. If the average block time is configured lower than reality, the calculated deadline is pushed far into the future. If configured higher, the deadline collapses and becomes too close to the current block.

As a result, a minter can gain substantially more time than configured to pay the agent because the minting deadline is overestimated. A similar effect applies to redemptions: an agent can gain extra time to pay the redeemer when the deadline is overestimated, or a redeemer may have to wait for an underestimated deadline even though the timestamp-based grace period has already passed.

Another scenario is that if the tracked underlying block is very outdated, the system requires non-payment proofs for redemption defaults to cover a window starting from that stale underlying block. For large gaps (for example, larger than the proof provider's maximum window), the redeemer or minter cannot obtain a valid proof at all. This affects both `RedemptionDefaultsFacet` and `MintingDefaultsFacet`, since they enforce `minimalBlockNumber <= firstUnderlyingBlock`.

Before allowing minting or redemption operations, consider validating that `currentUnderlyingBlock` (along with its timestamp) is not older than a reasonable bound. If the block data is stale, require callers to refresh it by providing a `ConfirmedBlockHeightExists` proof.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *Checking that the current block was recently updated could address this issue, but the real solution is off-chain. Regular block updates are essential, and blocking mints/ redeems based on timing could confuse users with unexpected errors.*

M-05 Incomplete Reset of Data When Updating Trusted Providers

In the `FtsoV2PriceStore` contract, governance can update the set of trusted addresses authorized to submit prices by calling the `setTrustedProviders` function. While the `setTrustedProviders` function correctly removes the old entries from `trustedProvidersMap` and adds the new ones, it does not clear the entries of `submittedTrustedPrices`.

As a result, if any of the old trusted providers had already submitted prices for the previous voting epoch, their data remain stored. Newly added providers can also submit prices for the same epoch, causing the median calculation of trusted prices to include submissions of both

old and new providers. Similarly, when `updateSettings` changes the `trustedDecimals`, it resets `trustedValue` and `trustedVotingRoundId` but fails to reset `numberOfSubmits`. This causes `getPriceFromTrustedProvidersWithQuality` to return a zero price with a misleading non-zero submission count, incorrectly suggesting quality metrics for a price that was just invalidated.

When setting new trusted providers, consider also clearing the `submittedTrustedPrices` entries for the previous voting round, and resetting `numberOfSubmits` to zero when decimals change.

Update: *Resolved at commits 50c1c15 and 958a064. Any submitted prices from trusted providers for the previous voting epoch are deleted when trusted providers are changed. Providers that remain trusted after the change need to resubmit.*

M-06 Incorrect Share Accounting After Pool Depletion

When users `enter` an agent's collateral pool, they receive Collateral Pool Tokens (i.e., shares) proportional to the amount of the collateral they deposited. Under normal circumstances, this ensures a fair distribution of collateral ownership within the pool.

However, issues arise when the pool has been emptied. A pool is considered empty if either:

- the `totalCollateral` is zero, or
- the total CPT supply is zero

The two conditions are not equivalent. A major liquidation event or default redemption can deplete the pool's collateral while leaving a non-zero pool token supply. In such a case, if a new user deposits collateral, the contract mints shares equal to the deposit amount. The total collateral would then only consist of the new deposit, but because old users still hold shares, they could claim a portion of this newly deposited collateral. As a result, the new depositor loses part of his deposit.

Relying on users to detect and avoid such scenarios is insufficient and introduces unnecessary risk. Moreover, once a collateral pool has been emptied, if new users avoid it because of the current problematic accounting, the pool cannot recover. Similarly, if the total CPT supply is zero but there is still collateral left in the pool, the first new depositor will end up owning not only the collateral he deposited but also the remaining old collateral.

Consider revising the share-minting and accounting logic to properly handle cases where the pool has been depleted of either the collateral or the shares.

Update: *Resolved at commit e0d2feb. However, pools reaching extreme CPT/collateral ratios (including zero on either side) are blocked from accepting deposits and become permanently irrecoverable.*

M-07 Using Stale Underlying Block Proofs Can Make Redemptions and Reservations Immediately Defaultable

The asset manager derives the last allowed payment block and timestamp for redemptions and collateral reservations from its tracked underlying state. If this state were to be updated with an old underlying block that is newer than the stored one but still far behind the real head, the computed `lastUnderlyingBlock` and `lastUnderlyingTimestamp` values can already lie in the past relative to the underlying chain, causing new redemptions or reservations to be created with an effectively expired payment window and immediately defaultable.

For simplicity, the explanation below focuses on the `lastUnderlyingBlock` / `lastUnderlyingTimestamp` calculation used for collateral reservations, which is the simpler variant. The same issue applies to the redemption path, which uses the same base formula with two additional small extension terms that only shift the deadline slightly but do not change the core behavior.

The system tracks the underlying chain using `currentUnderlyingBlock`, `currentUnderlyingBlockTimestamp`, and `currentUnderlyingBlockUpdatedAt`. These values are updated from two main proof types:

- An `IPayment` proof, where the underlying block number and timestamp from the payment's block are used
- An `IConfirmedBlockHeightExists` proof, where the block number and timestamp of the confirmed block are used

On each update, the asset manager:

- increases `currentUnderlyingBlock` if the new block number is larger than the stored one
- increases `currentUnderlyingBlockTimestamp` if the new timestamp is larger than the stored one

- sets `currentUnderlyingBlockUpdatedAt` to `block.timestamp` on the native chain at the time of the update

When creating a collateral reservation, the library computes the payment deadline using `_lastPaymentBlock`.

This function:

- computes a `timeshift` as `block.timestamp - currentUnderlyingBlockUpdatedAt` to amortize for the time that has passed from the last underlying block update
- converts `timeshift` to a `blockshift` using `averageBlockTimeMS`
- derives:
  - `lastUnderlyingBlock = currentUnderlyingBlock + blockshift + underlyingBlocksForPayment`
  - `lastUnderlyingTimestamp = currentUnderlyingBlockTimestamp + timeshift + underlyingSecondsForPayment`

This logic assumes that `currentUnderlyingBlock` and `currentUnderlyingBlockTimestamp` are close to the real underlying head at the moment of update, and that `block.timestamp - currentUnderlyingBlockUpdatedAt` is a good proxy for how far the underlying chain has moved since that block.

However, the contracts accept any proof whose block number and timestamp are merely greater than the stored values. If the asset manager's underlying state **has not been updated for a long time** and is far behind the real chain, a user can submit an `IPayment` or `IConfirmedBlockHeightExists` proof for an underlying block that is newer than the stale `currentUnderlyingBlock`, but still much older than the real underlying head. Then `currentUnderlyingBlock` and `currentUnderlyingBlockTimestamp` are updated to those stale-but-newer values and `currentUnderlyingBlockUpdatedAt` is set to the current Flare `block.timestamp`.

Immediately after that update, for a newly created reservation:

- `block.timestamp - currentUnderlyingBlockUpdatedAt` is close or equal to zero
- `timeshift` is close or equal to zero
- `blockshift` is near zero
- `lastUnderlyingBlock` ends up only `underlyingBlocksForPayment` blocks after a stale underlying block number

- `lastUnderlyingTimestamp` is only `underlyingSecondsForPayment` seconds after a stale underlying timestamp

If the real underlying chain head is already beyond this computed `lastUnderlyingBlock` / `lastUnderlyingTimestamp` at the time of the request. Then, in real time, the payment window has already elapsed when the reservation is created. The system's own deadline fields (`lastUnderlyingBlock`, `lastUnderlyingTimestamp`) thus represent a window that is already expired relative to the true chain state. So, a non-payment/default flow can be triggered as soon as the corresponding proofs are available.

As a result, Agents and users can receive significantly less time than configured, or close to zero real time, to execute redemption or reservation payments. An attacker who supplies a stale but monotonic proof can cause subsequent redemptions against a target agent to be created with already-expired effective deadlines on the underlying chain, increasing the risk of forced defaults or griefing. In the case of reservations, the user receives an already-expired time window to perform the payment and loses the reservation fee.

Consider ensuring that timekeeping updates cannot bring in significantly stale underlying blocks as the effective "current" state. Possible approaches include rejecting proofs whose underlying block number or timestamp is too far in the past relative to the native `block.timestamp` and `averageBlockTimeMS`, or basing `timeshift` on the age of the underlying block in the proof (relative to current time) rather than only on `currentUnderlyingBlockUpdatedAt`. In any case, this behavior should be clearly documented.

Update: *Partially Resolved at commits `b3bafc4` and `4c6dc7a`. It is now documented that the current underlying block must be updated regularly with fresh proofs. Additionally, automatic timestamp updates from payment proofs were removed, as they were affected by the same stale-proof issue.*

Low Severity

L-01 Agents Can Trigger Unintended Full Liquidation When Withdrawing Underlying

After a withdrawal is announced, the Agent's recorded underlying balance is not updated until a valid proof is later provided to confirm the withdrawal. During this period, the Agent can still

use their free underlying balance to self-mint. If the Agent or their worker address self-mints after announcing a withdrawal but before its confirmation, the effective underlying balance may drop below the required threshold once the withdrawal is finalized. This can immediately push the Agent into full-liquidation.

To prevent this scenario, consider disallowing self-minting using free underlying balance whenever there is an active withdrawal announcement.

Update: *Resolved at commit* `1704c32`.

L-02 Challenges and Liquidations Become Blocked When Agent Vault Is Blacklisted

Whenever a redemption defaults, the logic handles cases where the Agent vault has been blacklisted by the stable asset. It does this by using `tryPayoutFromVault` instead of `payoutFromVault`, allowing the revert to be caught and the payout to be replaced with collateral pool shares. However, when a challenger is rewarded, the function uses `payoutFromVault`, which will always revert if the Agent vault is blacklisted. The same issue applies during liquidations, where the liquidator payout reverts, blocking the entire liquidation process.

Consider using `tryPayoutFromVault` for challenges and falling back to the collateral pool if the vault payout fails. For liquidations, consider handling the blacklisting case explicitly, as otherwise, the underwater vault collateral will never be able to be released.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *It's unlikely an agent vault will be blacklisted, and all options for replacing vault liquidation payments with the pool carry serious drawbacks. Each approach - paying the full amount, capping to agent pool tokens, or burning tokens and reverting - introduces risks that may be worse than leaving the code as-is.*

L-03 Incorrect Rounding in Redemption Defaults Causes Collateral Pool Overpayment

In `RedemptionDefaults._collateralAmountForRedemption`, using `mulDivRoundUp` to compute the pool-funded portion when the vault-cap constraint is binding introduces an upward rounding bias that can overpay by up to 1 AMG per defaulted request (drawn from pool collateral), enabling profit via many small defaults.

Consider replacing the ceiling rounding with floor (`mulDiv`), or computing the shortfall directly in wei and rounding down to avoid systematic overpayment by the pool.

Update: *Resolved at commit* `397c180`.

L-04 Incorrect Failure Reason When a Late Payment to Core Vault Is Made

When validating a redemption payment, if the payment was not late, but the request had already defaulted, `_validatePayment` returns "redemption already defaulted". The system uses this to flag a potential FDC issue because a non-late redemption should not default. However, the late-payment check also includes whether the transfer is to the core vault. This causes late payments to the core vault to still return "redemption already defaulted", misleading off-chain monitoring into thinking there is an FDC issue.

Consider moving `!request.transferToCoreVault` inside the late-payment branch, mirroring how the `PAYMENT_BLOCK` status is handled.

Update: *Resolved at commit* `348aba4`. `_validatePayment` *now handles core vault transfers and normal redemptions separately, with scenario-specific revert messages on default.*

L-05 `CollateralPool` State not Updated After Destruction

The `destroy` function in the `CollateralPool` contract transfers its entire FAsset and WNat balances to the recipient. Although these balances are described as untracked in the comments, they also include the tracked collateral (`totalCollateral`) and FAsset fees (`totalFAssetFees`). However, these tracked values are not updated during destruction and continue to reflect outdated amounts. As a result, a user might mistakenly believe that the pool is still active.

Consider updating `totalCollateral` and `totalFAssetFees` when destroying the pool.

Update: *Resolved at commit* `61a224d`.

L-06 Collateral-to-Share Ratio Inflation

The `CollateralPool` contract issues collateral pool tokens (CPT) to users proportional to their deposited collateral. Artificially inflating the collateral-to-CPT supply ratio can lead to serious vulnerabilities. While the current implementation prevents the most critical attack vector (i.e., theft of new users' collateral through ratio manipulation) other related issues remain unaddressed. Specifically, the `CollateralPool` contract allows the `totalCollateral` value to arbitrarily without minting additional tokens. This can occur, for example, if the agent calls the `claimDelegationRewards` function, using a custom `_rewardManager` contract that will transfer a large collateral amount to the pool.

If this happens right before a user deposit, the new user will receive significantly fewer CPT than expected. This issue is particularly problematic if CPT have a secondary market, as it allows the agent to manipulate the exchange rate between CPT and collateral, potentially misleading depositors or impacting token pricing. Additionally, the use of a fixed minimum threshold (`1 ether`) for both collateral and CPT supply is inconsistent with their dynamic ratio.

Consider allowing users to set a minimum acceptable CPT amount when depositing and replacing the hardcoded minimum thresholds with ratio-aware conditions.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *The proposed solution would require changing the collateral pool API. Since the contracts are already deployed, we prefer not to modify the API.*

L-07 Underwater Pool Recovery Not Applied on Redemption Default

Whenever a redemption is defaulted, the redeemer is first compensated using vault collateral. If that is insufficient, the shortfall is covered by using funds from the collateral pool. If the collateral pool is underwater, a redemption default that is primarily (or even partly) paid from the Agent vault can still improve the pool's collateral ratio and potentially lift the Agent out of liquidation. However, during redemption-default handling, `endLiquidationIfHealthy` is not invoked.

Consider invoking `endLiquidationIfHealthy` immediately after processing a redemption default.

Update: *Resolved at commit* `fdd066e`.

L-08 Suboptimal Liquidation Splitting May Lead to Unprofitable Liquidations

During liquidations, the protocol computes the maximum amount for both the vault and the collateral pool (i.e., the amount that would restore each to the target, healthy collateral ratio) to avoid over-liquidations. However, the total liquidatable amount is then capped by the `maxLiquidatedAMG`, which is the max of these two values. Since this value is afterwards split between the vault and the collateral pool, it is possible that the portion for one side exceeds the actual collateral held.

Although the liquidation will not revert (since the protocol ultimately caps this amount by the available on each side), this misalignment in accounting may cause the liquidation to become non-profitable for the liquidator, even if, for example, the vault alone held enough collateral to make the action profitable. Liquidators are expected to assess profitability themselves, but if this scenario occurs frequently, they may avoid liquidations, posing a risk for the system.

Consider implementing a more precise splitting mechanism during liquidations to better handle edge cases like the one described above, thereby reducing instances in which viable liquidations become unprofitable because of the suboptimal splitting.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *The splitting is actually already solved in the method `currentLiquidationFactorBIPS`, which caps the vault/pool splitting factors to corresponding CRs and increases the other factor if needed. \ The only remaining problem is when the total (vault+pool) CR is less than total liquidation factor should be. But in this case it was our decision that we cap liquidation payouts to the total CR (the alternative would create bad debt - fassets backed with 0 collateral).*

L-09 Lack of Slippage Protection When Entering or Exiting the Collateral Pool

When a user deposits collateral to enter the pool, the protocol checks that the `tokenShare` he receives is non-zero. Similarly, upon exiting, it checks the `natShare` returned is non-zero. These checks are intended to prevent loss of collateral due to rounding. Although a minimum amount exists when entering, making inflation attacks to have minimal impact, rounding effects can still cause a user to receive slightly less than expected. This is especially relevant when exiting the pool, where no minimum output is enforced.

Consider allowing users to specify a minimum acceptable amount for shares when entering and collateral when exiting the pool, providing explicit slippage protection and reducing the impact of rounding errors.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *As part of the M6 fix, we introduced min/max price checks when entering the pool. With these safeguards, any rounding effects should be negligible, smaller than typical rounding in frontend displays. Given that the contracts are already deployed and off-chain applications depend on the current interfaces, we prefer not to change the pool API at this stage. _*

L-10 Access Control Logic Mismatch for `buyCollateralPoolTokens`

The implementation of the `AgentVault::buyCollateralPoolTokens` function conflicts with the documented and interface-level intent. The interface comment explicitly states that anyone should be able to call this method, allowing third parties to top up the agent's pool position. However, the current implementation restricts the call to the vault owner via the `onlyOwner` modifier, which prevents third-party deposits that would strengthen the agent's `CollateralPool` position.

Consider removing the `onlyOwner` modifier to allow anyone to top up the agent's pool position, aligning behavior with the stated intent.

Update: *Resolved at commit `3fd8406`. The interface comment has been updated to reflect the current implementation.*

L-11 Race condition between `confirmMintingPayment` and `mintingPaymentDefault`

The `confirmMintingPayment()` function lacks an upper bound check on the payment block number, allowing minters to submit payment proofs after the `lastUnderlyingBlock` deadline. This creates a race condition where both `confirmMintingPayment()` and `mintingPaymentDefault()` can be valid simultaneously, with transaction ordering determining the outcome. As a result, minters who pay slightly late can lose their entire underlying payment with no protocol-side recovery. An agent can obtain the proof of a late

underlying payment, then front-run by calling `mintingPaymentDefault()` to default the request and retain the underlying funds, leaving the user without minted fassets.

Consider applying an explicit upper-bound check (e.g., `proof.blockNumber <= lastUnderlyingBlock` or timestamp equivalent) in `confirmMintingPayment()`, and surfacing a clear error for late proofs to avoid user confusion. Also consider clearly documenting this behavior, including the deadline boundary conditions and the rationale for rejecting late payments, to help users understand when payments will be accepted versus when defaults can be triggered.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *We believe it's more beneficial to allow minters to confirm minting up until the point of default, rather than enforcing a hard cutoff. Even the official agent bot follows this approach: it checks for payment before initiating the default process, and if a payment exists (even if it's late) it proceeds with the confirmation instead.*

L-12 Self-mints And Top-ups Reject Payments Included In The Agent's Creation Block

In `MintingFacet::selfMint` and `UnderlyingBalanceFacet::confirmTopupPayment`, the validations use a strict inequality against the agent's creation block:

```
require(_payment.data.responseBody.blockNumber > agent.underlyingBlockAtCreation,
SelfMintPaymentTooOld());
```

This conflicts with the documented semantics of `agent.underlyingBlockAtCreation` in `Agent.State` (valid since that block, inclusive). It can incorrectly reject self-mints and top-ups included in the creation block, preventing execution and the locking of underlying assets.

Change the comparison to be inclusive (`>=` instead of `>`). This aligns with the spec ("both inclusive"), ensuring that payments included in the creation block are accepted.

Update: *Resolved at commit* `66d242a`.

Notes & Additional Information

N-01 Permission Mismatch in `consolidateSmallTickets` Breaks the Documented Permissionless Behavior

The `consolidateSmallTickets` function is documented to be callable by anybody: "Since the method just cleans the redemption queue it can be called by anybody". However, the implementation applies the `onlyImmediateGovernanceOrExecutor` modifier, restricting it to governance or authorized executors. This is a business logic inconsistency: the code does not match the stated behavior. The practical impact is reduced liveness and potential operational issues.

The function exists to prevent redemption stickiness when many sub-lot tickets are at the front of the queue (bounded by `maxRedeemedTickets`). If only governance/executors can invoke it, the system relies on them to maintain queue health. In periods where they are unavailable or do not actively maintain the queue, redemptions can be impeded (e.g., the first `maxRedeemedTickets` are < 1 lot), contradicting the intended open, self-healing mechanism. This also creates an inconsistent trust model compared to `convertDustToTicket`, which is open to anyone to improve fungibility.

Consider removing the `onlyImmediateGovernanceOrExecutor` modifier from `consolidateSmallTickets` to match its documented behavior.

Update: *Resolved at commit `2c9037a`. The documentation was updated to match the current implementation.*

N-02 Unused Event

The `CreatedTotalSupplyCache` event in `CheckPointable.sol` is never emitted.

Consider either removing the aforementioned event or emitting it when the total supply changes.

Update: *Resolved at commit `32c837c`.*

N-03 Unused Errors

Throughout the codebase, multiple instances of unused errors were identified:

- The `CollateralWithdrawalAnnounced` error in `AgentCollateralFacet.sol`
- The `FAssetNotTerminated` error in `AgentCollateralFacet.sol`
- The `NotWhitelisted` error in `AssetManagerBase.sol`
- The `ConfirmationTimeTooBig` error in `SettingsManagementFacet.sol`
- The `InvalidAddressOwnershipProof` error in `UnderlyingAddressOwnership.sol`
- The `EOAProofRequired` error in `UnderlyingAddressOwnership.sol`
- The `NoSelectorsGivenToAdd` error in `LibDiamond.sol`
- The `NotContractOwner` error in `LibDiamond.sol`
- The `FAssetTerminated` error in `FAsset.sol`

To improve the overall clarity, intentionality, and readability of the codebase, consider either using or removing any currently unused errors.

Update: *Resolved at commit* `68336ef`.

N-04 Unnecessary Data Fields in Event Emission

Throughout the codebase, multiple instances of unnecessary data fields being emitted in events were identified:

- The `emit IAssetManagerEvents.LiquidationStarted(_agent.vaultAddress(), block.timestamp);` event emission in `LiquidationFacet.sol`
- The `emit IAssetManagerEvents.FullLiquidationStarted(_agent.vaultAddress(), block.timestamp);` event emission in `Liquidation.sol`
- The `emit IAssetManagerEvents.CurrentUnderlyingBlockUpdated(state.currentUnderlyingBl state.currentUnderlyingBlockTimestamp, block.timestamp);` event emission in `UnderlyingBlockUpdater.sol`.

To improve the clarity and efficiency of the codebase, consider removing unnecessary data fields from the event emission such as `block.number` or `block.timestamp` since they are already included in the block information.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *Since the contracts are already deployed, it's preferable to keep the current API unchanged for now, even if that means leaving some unnecessary data in place.*

N-05 Duplicate Imports

In `CoreVaultClient.sol`, multiple instances of duplicate imports were identified:

- Import `import {ICoreVaultClient} from "../../userInterfaces/ICoreVaultClient.sol";` imports duplicated alias `ICoreVaultClient`
- Import `import {ICoreVaultClient} from "../../userInterfaces/ICoreVaultClient.sol";` imports duplicated alias `ICoreVaultClient`

Consider removing duplicate imports to improve the overall clarity and readability of the codebase.

Update: *Resolved at commit* `b14501f`.

N-06 Unnecessary Cast

The `address(_controller)` cast in the `SettingsManagementFacet` contract is unnecessary.

To improve the overall clarity and intent of the codebase, consider removing any unnecessary casts.

Update: *Resolved at commit* `2d138e0`.

N-07 Updating Liquidation Step Duration During Active Liquidation Breaks Reward Progression

When a liquidation occurs, the liquidators receive collateral worth more than the FAssets they burn. The bonus amount is determined by the `liquidationCollateralFactorBIPS` array, whose values increase step-by-step. Every `liquidationStepSeconds`, the protocol moves to the next entry of this array, and the liquidation reward percentage is expected to increase throughout the liquidation process.

However, if governance increases the `liquidationStepSeconds` while a liquidation is already ongoing, the calculated step may revert to a previous index in the array. This results in a lower reward percentage than expected, violating the invariant that the reward percentage should only increase during a liquidation and potentially reducing the liquidators' incentive.

Consider updating the `liquidationStepSeconds` parameter only when no liquidation is active.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *This behavior is correct though it may surprise a liquidator. We monitor ongoing processes and can pause if governance changes pose risks.*

N-08 Misleading NatSpec Documentation

The `nonTimelockedBalanceOf` function of the `ICollateralPoolToken` interface incorrectly states that it "returns the amount of account's tokens that are timelocked". The intended behavior is to return the portion of an account's tokens that are not timelocked. This documentation mismatch creates a specification ambiguity that can lead implementers or integrators to invert the semantics.

In addition, the documentation for the `exit` function of the `ICollateralPool` interface states that, because there are multiple ways to split spending transferable and non-transferable tokens, the method "also takes a parameter called `_exitType`". However, the actual function signature only accepts `_tokenShare` and returns `_natShare`. This creates a clear mismatch between the intended behavior described in the docs and the exposed interface.

Moreover, the documentation for `selfCloseExit` and `selfCloseExitTo` functions describes that they should first utilize the caller's accumulated fee share to cover redemption amounts before requiring external FAsset transfers. However, the implementation unconditionally requires the full redemption amount from the user's wallet, never actually applying their fee share to burn FAssets and reduce the transfer requirement.

Consider updating the documentation to accurately reflect the actual implementation and function signatures.

Update: *Resolved at commit* `63b0f6c`.

N-09 Dead Code

The codebase contains multiple instances of unused code elements including errors, interfaces, and functions that are never referenced or called. This dead code increases contract size, compilation time, and maintenance burden while potentially confusing developers about which components are actually active in the system.

Identified instances include the following:

- The `UpdaterState` struct and `UPDATES_STATE_POSITION` constant in the `SettingsManagementFacet` contract are unused
- The `IPriceChangeEmitter` interface is defined but never implemented or used by any contract in the system.

Consider conducting a comprehensive dead code analysis across the entire codebase and removing all unused elements. For any code that is intentionally kept for future use, clear documentation should be added explaining why the code has been retained.

Update: *Resolved at commit* `d68b3b6`.

N-10 First-epoch Trusted Price Submissions are Blocked

The `submitTrustedPrices` function allows trusted providers to submit asset prices after each voting epoch. However, the gate that prevents double submissions uses `require(lastVotingEpochIdByProvider[msg.sender] < previousVotingEpochId, AlreadySubmitted());`. When the first admissible previous epoch ID equals 0 (which is the case when the current epoch is 1), the condition becomes `0 < 0` and always fails. This makes it impossible for any trusted provider to submit trusted prices for voting round ID 0, even during the valid submission window.

Consider allowing submissions when `lastVotingEpochIdByProvider[msg.sender] == 0 && previousVotingEpochId == 0` to handle the first-epoch case, or initialize the mapping values to a sentinel value to ensure that the condition passes for first-time submissions.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *During deployment, we ensure `firstVotingRoundStartTs` is set low enough so that `previousVotingEpochId` is never zero.*

N-11 Missing Timeout Fallback for `confirmReturnFromCoreVault`

The `confirmReturnFromCoreVault` function can only be called by the Agent vault owner, with no timeout-based fallback allowing others to confirm after a period of inactivity. This is inconsistent with other confirmation functions like `confirmRedemptionPayment` that allow others to confirm after `confirmationByOthersAfterSeconds`.

Consider aligning `confirmReturnFromCoreVault` with other confirmation patterns by introducing a timeout-based fallback that allows other parties to call it after a configurable delay.

Update: *Resolved in commit 390c891.*

N-12 Confirmation Rewards Can Be Extracted Even While the Agent Is in Full Liquidation

The `confirmUnderlyingWithdrawal` function allows anyone to confirm a withdrawal after `confirmationByOthersAfterSeconds` and receive a reward from the agent's vault. The function only checks whether `msg.sender` is the agent's `ownerManagementAddress` or `workAddress`, and any other address is treated as a non-agent and receives the reward. This design allows the `confirmationByOthersRewardUSD5` amount to be extracted from the agent's vault even while the agent is in full liquidation. In particular, the agent can simply use a different address (not equal to `ownerManagementAddress` or `workAddress`) to perform the confirmation and receive the reward, removing collateral that should be preserved for liquidations.

Consider adding a check to ensure that the agent is not in full liquidation before allowing third-party confirmations to proceed.

Update: *Resolved at commit `f88012e`.*

N-13 Inaccurate Comment in Library

The comment in the `mulDiv` function of the `SafePct` library suggests that the function will not revert if an intermediate operation overflows, as long as the final result fits in a `uint256`. While the implementation does eliminate most of these cases (so-called phantom overflows), it does not cover all possible scenarios. For example, if `x = y = 2^128` and `z = 2^256 - 1`, then the expected result is `1`, but `mulDiv` will revert because both `b =`

`x%z` and `c = x%z` equal `2^128`. Thus, the computation of `b*c` will revert, because it equals `2^256`, which does not fit in a `uint256` container and Solidity versions after 0.8 revert on overflows.

Such scenarios are unlikely to occur in the current codebase, where the `z` value is expected to be relatively small (the library works as expected for any value of `z <= 2^128`). However, the `mulDiv` function is part of a library that could, in principle, be used in other contexts, where `z` is large. In such cases, unexpected reverts may occur, contrary to the behavior described in the comments.

Consider updating the comments to accurately describe the behavior of the `mulDiv` function.

Update: *Resolved at commit `b436515`. The implementation now uses OpenZeppelin's `Math.mulDiv`.*

N-14 Inefficient Contract Addresses Update Logic

The `_getContractAddress` function of the `AddressUpdatable` contract reverts if the hash of its `_nameToFind` argument is not included in the `_nameHashes` array. Since this function is used in `updateContractAddresses` and is called with `_nameToFind == "AddressUpdater"`, the `_contractNameHashes` array passed to `updateContractAddresses` must always include the name hash of the `AddressUpdater` contract, even when its address is not actually being updated.

A similar issue exists in `AssetManagerController::_updateContractAddresses`, which calls `_getContractAddress` for the `AddressUpdater`, `AssetManagerController` and `WNat` contracts. This means that even if none of these contracts is being updated, the caller must still supply their name hashes and addresses, otherwise the function will revert.

Consider allowing contract address updates without requiring the caller to include entries for contracts that are not being updated. This would reduce unnecessary gas consumption and make the functions easier and more intuitive to use.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *We acknowledge that the behavior is not optimal, but it follows the standard update mechanism used by `AddressUpdater`. Since `AddressUpdater` (and many*

> *contracts that depend on it) existed long before FAssets, we decided to retain this approach for consistency.*

N-15 Inconsistent Storage Namespace Conventions Across Contracts

The codebase uses a diamond proxy for the Asset Manager, with name-spaced storage to avoid storage collisions between facets. However, the namespace strings used to derive storage slot positions are inconsistent across the codebase, and none follow the ERC-7201 recommended formula.

Several different namespace prefixes are used:

- `fasset.AssetManager.Settings`
- `fasset.AssetManager.State`
- `fasset.AssetManager.Agent`
- `fasset.GovernedBase.GovernedState`
- `fasset.RedemptionTimeExtension.State`
- `fasset.AssetManager.UpdaterState`
- `fasset.CoreVault.State`
- `flare.diamond.AddressUpdatable.ADDRESS_STORAGE_POSITION`
- `diamond.standard.diamond.storage`
- `utils.ReentrancyGuard.ReentrancyGuardState`

ERC-7201 specifies that storage locations should be derived as `keccak256(abi.encode(uint256(keccak256("namespace.id")) - 1)) & ~bytes32(uint256(0xff))` to ensure the slot ends in 0x00, which provides a buffer against sequential slot collisions. The current implementation uses a simpler `keccak256("namespace")` approach without the subtraction and masking, and without a consistent namespace hierarchy.

Additionally, some upgradeable proxy contracts in the system use traditional linear storage layouts (with sequential storage slots starting from slot 0) instead of namespaced storage. When upgrading these contracts, developers must carefully manage storage layout to avoid collisions with existing state variables, as adding or reordering variables can corrupt existing data. While no concrete collision has been identified, the inconsistent naming conventions across namespaced contracts and the mix of storage patterns (namespaced vs. linear) increase maintenance complexity and the risk of accidental storage corruption during upgrades.

Consider adopting a unified namespace convention across all contracts using namespaced storage (e.g., `fasset.v1.<ContractName>`) and aligning with the ERC-7201 slot derivation formula for defense in depth. For contracts using linear storage, document the storage layout carefully and implement strict upgrade procedures to prevent storage collisions.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *We agree that using ERC-7201 namespacing everywhere would be preferable. However, since contracts using the current namespaces have already been deployed, changing them now would break upgrade compatibility.*

N-16 `uint16` Multiplication Overflow Can Break Future Updates

In the `setPaymentChallengeReward` function of the `SettingsManagementFacet` contract, the upper-bound check for the proportional reward uses `settings.paymentChallengeRewardBIPS * 4` where `settings.paymentChallengeRewardBIPS` is a `uint16` value. Under Solidity 0.8.x, arithmetic on fixed-size integers is checked. Thus, if `settings.paymentChallengeRewardBIPS > 16383`, the multiplication by 4 would overflow and revert before the `require` condition is even evaluated.

This makes any subsequent call to `setPaymentChallengeReward` revert unconditionally once the stored value exceeds 16383, effectively bricking further updates (including attempts to decrease the value). Since the function does not clamp `_rewardBIPS` to a safe range beforehand, governance can legitimately raise `settings.paymentChallengeRewardBIPS` above 16383 over several allowed updates (e.g., 0 → 100 → 500 → 2100 → 8500 → 34100). After that, the next invocation hits the overflow when computing the bound and cannot proceed.

This issue can occur only if the fee percentage is set to an unreasonably high value, but it is still better to mitigate it to avoid potential permanent consequences. A similar issue exists in the `setCollateralReservationFeeBips` function.

Consider casting values to 256-bit integers before performing the multiplication to prevent overflow-induced reverts.

Update: *Resolved at commit* `29c81ef`.

N-17 `getFeedIdsWithDecimals` Returns Only Trusted Decimals, Not FTSO Decimals

The `getFeedIdsWithDecimals` function returns the list of feed IDs along with their associated decimals. However, it only returns the `trustedDecimals` value stored in `latestPrices[feedId].trustedDecimals`, which is the decimal precision configured for trusted-provider submissions. The FTSO-published prices have their own decimals field (stored in `latestPrices[feedId].decimals`) which may differ from `trustedDecimals`. Callers using `getFeedIdsWithDecimals` to interpret FTSO prices instead of trusted prices could misinterpret the decimal scaling.

Consider either renaming the function to `getFeedIdsWithTrustedDecimals` for improved clarity, providing a separate getter that returns FTSO decimals, or returning both decimal values.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *The documentation for `IPricePublisher.getFeedIdsWithDecimals` already clarifies that the decimals are intended for trusted providers. Additionally, decimals for FTSO feeds aren't fixed and can vary with each submission (they are predefined off-chain). For these reasons, we prefer to keep the API as it is.*

N-18 Non-Standard Upgradeable ERC-20 Implementation

The `CollateralPoolToken` contract uses a non-standard approach for implementing an upgradeable ERC-20 token. It does this by inheriting from the regular ERC-20 contract alongside `UUPSUpgradeable` instead of using OpenZeppelin's `ERC20Upgradeable` contract that is specifically designed for upgradeable patterns.

The current implementation mixes upgradeable and non-upgradeable patterns: it inherits from the standard ERC-20 contract (which uses constructor-based initialization) but then overrides the `name` and `symbol` functions to return custom storage variables (`tokenName` and `tokenSymbol`) that are set in an `initialize` function. This creates unnecessary complexity as the contract essentially reimplements functionality that already exists in OpenZeppelin's upgradeable contracts library.

While the current implementation functions correctly, using `ERC20Upgradeable` would provide a cleaner, more maintainable solution that follows established patterns in the

ecosystem. The standard upgradeable contracts handle initialization properly through `__ERC20_init` and manage storage in a way that is explicitly designed for proxy patterns. This would reduce code complexity, improve maintainability, and align with community best practices for upgradeable token contracts.

Consider refactoring the code to use `ERC20Upgradeable` from OpenZeppelin's upgradeable contracts library. Doing so would eliminate the need for custom storage variables and manual overrides while providing the same functionality with less code and better standardization.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *Since the contracts are already deployed, we need to avoid major layout changes that could impact upgrades.*

N-19 Unbounded Cooldown Time Can Freeze all Future Rate-limited Updates

The `setMinUpdateRepeatTimeSeconds` function that controls the global update cooldown is itself protected by the `rateLimited` modifier and accepts any positive value without an upper bound or bounded-increase checks. As a result, a single call can set `settings.minUpdateRepeatTimeSeconds` to an extremely large value. This creates a self-referential lock: after setting a huge cooldown at time T, the next call to `setMinUpdateRepeatTimeSeconds` to reduce the cooldown will only be permitted after that huge duration expires. Meanwhile, the new large cooldown applies immediately to every other `rateLimited` setter, effectively preventing normal administration of the system for that duration. Unlike most other setters in this facet that constrain increases/decreases, this function has no such guardrails.

Consider either removing the `rateLimited` modifier from this specific setter to ensure it can always be adjusted promptly, protecting it with a distinct timelock or role separate from the general rate limit, or adding both a strict upper bound and a bounded-increase policy.

Update: *Resolved at commit `47dc3f7`. Two validation checks were introduced - one enforcing a strict upper bound and another constraining the maximum allowable increase relative to the minimum value.*

N-20 Missing Event Emissions for Important State Changes

Several critical, state-changing functions across the codebase fail to emit events, preventing off-chain services, monitoring tools, and users from tracking important protocol changes. This lack of transparency makes it difficult to audit system behavior, debug issues, or maintain accurate off-chain state representations.

Affected functions include the following:

- `AssetManagerController` **setting changes**: Functions that modify critical asset manager parameters and configurations do not emit corresponding events when updated.
- `SystemStateManagementFacet` `pauseMinting`, `unpauseMinting`, and `attachController` **functions**: Critical system state transitions and controller attachments occur silently without event notifications, making it impossible to track when the system enters different operational modes.
- The `setAddressUpdaterValue` function of the `AddressUpdateable` abstract contract updates critical addresses without any event emissions.

Consider systematically reviewing all state-changing functions and ensuring that they emit the appropriate events.

Update: *Partially Resolved at commit 6db5969. The Flare team stated:*

> *To `AssetManagerController` we added `AssetManagerAdded/Removed` and `EmergencyPauseSenderAdded/Removed` events. The asset manager state changes should be tracked through asset manager events.\ To `AssetManager` we added `MintingPaused(bool)` and `EmergencyPauseTotalDurationReset` events. Calls to `attachController` should be tracked through controllers `AssetManagerAdded/Removed`, because it is internal method that can only be called through controller's add/remove asset manager.\ We haven't added any event for `setAddressUpdater`, because it is standard for all `AddressUpdatable` contracts that there are no events at contract updates (throughout Flare system contracts). `AssetManager` is a bit of exception because it has `ContractChanged` event (but it isn't `AddressUpdatable`).*

N-21 Zero Address Can Be Incorrectly Reported as an Owner When Work Address Is Unset

The `isAgentVaultOwner` function returns the result of `Agents.isOwner(agent, _address)` without validating the caller-provided `_address`. The underlying check in `Agents.isOwner` considers an address an owner if it equals either `agent.ownerManagementAddress` or the "work address" returned by the agent owner registry. When the work address is not set, registries return `address(0)`. In that case, calling `isAgentVaultOwner(_agentVault, address(0))` will return `true` for any agent whose work address is unset, incorrectly treating the zero address as an owner.

While core authorization paths in the system use `Agents.requireAgentVaultOwner` with `msg.sender` and are not directly compromised by this, the incorrect result in this view helper can mislead off-chain services or other integrations that rely on `isAgentVaultOwner` for access checks, leading to access control confusion.

Consider explicitly rejecting the zero address or ensuring that a zero work address is not treated as an owner before returning.

Update: *Resolved at commit* `d8ea842`.

N-22 Typographical Error in NatSpec Comments for `FAsset::mint` and `FAsset::burn`

- The NatSpec comment for `FAsset::mint` says "Mints `_amount` od fAsset." instead of "Mints `_amount` of fAsset".
- The NatSpec comment for `FAsset::burn` says "Burns `_amount` od fAsset." instead of "Burns `_amount` of fAsset".

Consider correcting the aforementioned errors to improve the documentation quality of the codebase. While these errors do not affect the functionality of the protocol, fixing them maintains professional standards and prevents the error from propagating to auto-generated documentation and developer tooling.

Update: *Resolved at commit* `ee900c5`.

N-23 Ineffective Minimum Amount Enforcement for Core Vault Transfers

The `minimumAmountLeftBIPS` parameter is intended to ensure that agents maintain a minimum amount of minted FAssets after transferring to the core vault. However, the current implementation uses a percentage-based approach that fails to enforce an absolute minimum threshold. The `_minimumRemainingAfterTransferForCollateralAMG` function calculates the minimum required amount as a percentage of the Agent's maximum supported AMG based on their collateral. Since this is a relative percentage rather than an absolute floor, Agents can repeatedly transfer funds while always maintaining the required percentage of their decreasing balance.

This defeats the apparent purpose of maintaining operational liquidity in Agent vaults. If the intention is to ensure Agents always retain enough FAssets to handle small redemptions or maintain basic operational capacity, the percentage-based approach allows Agents to circumvent this requirement through iterative transfers that gradually drain their vaults to negligible amounts.

Consider implementing an absolute minimum threshold that cannot be transferred regardless of the agent's total balance. Alternatively, consider combining the percentage-based check with an absolute floor to ensure that Agents maintain meaningful operational reserves.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *This is by design. Setting an absolute limit on remaining underlying is difficult, and the current process—while allowing agents to fully clear their vaults over time—is considered sufficient. Faster clearing can be done using alternative strategies involving borrowing and minting.*

N-24 Weak Randomness in Core Vault Withdrawal Request IDs

When an Agent wants to withdraw collateral from the core vault, they must first call `requestReturnFromCoreVault`. This function reserves part of the Agent's collateral and assigns the request a pseudorandom ID using `the block.number`. The purpose of this randomness is to prevent the Agent from predicting the ID and attempting to finalize payment before submitting the request.

The team acknowledges that the randomness used is weak, and the comment states this is acceptable because even a single wrong guess by the Agent would trigger a full liquidation. However, if an Agent colludes with a block builder, they could ensure that their request is included in a specific block that produces the desired ID.

Consider using a stronger source of randomness, such as a VRF, to prevent request ID prediction.

Update: *Resolved at commit fba0850.*

N-25 Inconsistent Handling of Destroyed Agents Across Modifiers and Getters

The codebase inconsistently uses the `get` function which rejects destroyed agents, versus the `getAllowDestroyed` function which permits destroyed Agents, without clear documentation of which functions should operate post-destruction.

The `onlyOwner` modifier in `AgentVault` and the `onlyAgent` modifier in `CollateralPool` use `isAgentVaultOwner` which calls `getAllowDestroyed`, permitting all owner-gated operations on destroyed agents. Meanwhile, `onlyAgentVaultOwner` in `AssetManagerBase` uses `Agent.get`, rejecting destroyed agents. This permits semantically invalid operations on destroyed vaults while other functions correctly check destroyed status explicitly.

Within `AgentInfoFacet`, most functions use `getAllowDestroyed`, but `getAgentFullCollateral` and `getAgentFullPoolCollateral` call `_getFullCollateral`, which uses `Agent.get` and reverts for destroyed agents. This breaks off-chain indexers and monitoring tools that need collateral data post-destruction.

Consider standardizing the approach, using `getAllowDestroyed` for read-only getters, introducing a distinct `onlyActiveAgentOwner` modifier for functions requiring active agents, and documenting which operations are permitted post-destruction.

Update: *Resolved at commits 0eb2901 and dfeb116.*

N-26 User Can DoS Agent Destroying by Redeeming From Core Vault Without Ticket Burn

When a user redeems from the core vault, no tickets are burned, and the function assumes that an agent has already transferred the funds to the core vault and closed their tickets. This allows users to transfer to the core vault, update the available funds, and then redeem those funds, burning the corresponding FAssets.

This leaves `totalBackedAMG` unchanged while the total FAsset supply decreases. As a result, an Agent can end up with redemption tickets whose total value exceeds the total FAsset supply, making it impossible to self-close and destroy the Agent if needed.

1. The Agent has `totalBackedAMG` of 100 and the total FAsset supply is 100.
2. A user transfers 10 underlying tokens to the core vault and updates `availableBalance`.
3. The user calls `redeemFromCoreVault`, burning 10 FAssets.
4. The total FAsset supply is now 90, while the Agent still has an obligation of 100 to self-close. Self-minting does not help, as it creates additional tickets and the supply remains 10 short.
5. The Agent must wait for another Agent to open a vault, mint, and redeem, which pushes the same issue onto that Agent.

The last Agent to destroy will not be able to self close the whole position and will leave part of their collateral stuck in the system.

Consider only allowing the Agents to update the available balance for the core vault.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *For such DoS attack the user has to send XRP to core vault and then redeem the same amount of FXRP to get them back, so the initially sent XRP is lost to the user.\ On the other hand, the agent can always self-mint the FXRP amount that has to remain after transfer to core vault (possibly borrowing the required XRP), immediately transfer the rest to core vault, and then self-close the self-minted FXRP.*

N-27 Worthless CPTs Can Back AMGs When Pool Collateral Is Zero

The `agentsPoolTokensCollateralData` function calculates the value of an Agent's `CollateralPoolTokens` (CPTs) for collateral ratio purposes. When the pool's

`fullCollateral` (total wNAT) is zero, the function falls back to pricing CPTs at 1:1 with wNAT instead of computing their actual value. In the unlikely scenario where outstanding CPT supply is non-zero but the pool holds zero wNAT, the Agent's CPTs would be valued as if they were worth their face value in wNAT, even though they are effectively worthless.

This edge case is difficult to reach through normal pool operations, as entry and exit enforce minimum balance constraints. However, it becomes theoretically possible when pool collateral is paid out during liquidations or other settlement events: these payouts reduce the pool's wNAT collateral without proportionally reducing the outstanding CPT supply. If repeated payouts combined with concurrent withdrawals drive the pool's collateral to exactly zero while CPTs remain outstanding, the fallback pricing would allow an Agent to satisfy collateral requirements using CPTs that have no real backing.

Consider computing the actual CPT value as zero when `fullCollateral == 0` and `totalPoolTokens > 0`, or adding an explicit invariant check that prevents this state from occurring.

Update: *Acknowledged, not resolved. The Flare team stated:*

> *The agent's CPTs are not collateral - they are never paid out to redeemers or liquidators. They are just required to prevent liquidations caused by the agent from hurting other pool users (agent's CPTs are burned in such case to compensate for pool payout). Moreover, sufficient agent's CPTs are only required at minting and at that point high enough pool CR is required independently.*

Although this edge case does not present a direct security risk to the FAssets protocol itself, it could pose risks to external integrations. Third-party protocols that query functions such as `getAgentFullPoolCollateral` to derive CPT pricing may receive erroneous valuations in this edge case, where CPTs with no backing would be priced at 1:1 with wNAT. External developers integrating with the FAssets system should be aware of this limitation and implement additional safeguards when using these values as price oracles.

Client Reported

CR-01 De-whitelisted Agents Can Bypass Minting Restrictions via Core Vault Transfers

When an Agent is de-whitelisted through the governance mechanism, the intention is to prevent that Agent from participating in new minting operations. The system correctly blocks direct minting requests from de-whitelisted Agents through the standard minting interface. However, de-whitelisted Agents retain the ability to request transfers from the core vault. When these transfer requests are confirmed by the underlying chain verifier, they automatically initiate a new minting of FAssets corresponding to the transferred amount. This creates a bypass mechanism where de-whitelisted agents can effectively continue minting operations despite their revoked status.

This undermines the governance's ability to restrict problematic agents, as the de-whitelisting mechanism fails to fully prevent minting activities. A malicious or compromised agent that has been de-whitelisted for security or compliance reasons could continue to mint FAssets through this alternative path, potentially exposing the system to the very risks that prompted their de-whitelisting.

Consider implementing a whitelist check in the core vault transfer confirmation flow to ensure that de-whitelisted agents cannot initiate new minting operations through any mechanism.

Update: *Resolved at commit* `b3ee66a`.

CR-02 Agent Self-Close Bypasses Redemption Fee Collection

When agents perform self-closing operations to redeem their own FAssets, the protocol does not charge or distribute any redemption fees to the collateral pool. This is in contrast to normal redemptions, where users pay a fee that gets distributed to liquidity providers as compensation for their risk. This fee exemption creates an exploitable loophole for fee evasion through collusion. A user seeking to avoid redemption fees can transfer their FAssets to a cooperating Agent, who then performs a self-close operation. The Agent receives the underlying assets without any fee deduction and can transfer them back to the original user through an off-chain arrangement. This allows users to effectively bypass the redemption fee mechanism entirely.

Redemption fees serve as an important incentive mechanism for collateral pool liquidity providers, compensating them for the risk of potential liquidations. When fees are circumvented through self-closing, it undermines the economic model that attracts and retains liquidity in the system. Additionally, this creates an unfair advantage for users with Agent relationships over regular users who must pay full redemption fees.

Consider applying the standard redemption fee to self-close operations, with the fee being distributed to the collateral pool as in normal redemptions.

Update: *Resolved at commit* `5956dac`.

Conclusion

The FAsset system is a sophisticated and modular architecture for issuing decentralized, 1:1 backed synthetic assets on the Flare Network, leveraging a dual-collateral model and a network of agents to ensure solvency.

The assessment identified one high-severity issue involving a race condition that could allow agent vault ownership hijacking, alongside several medium- and low-severity vulnerabilities. Although the protocol demonstrates a robust, multi-layered security architecture with comprehensive safety mechanisms, the findings reveal a recurring need to strengthen state validation—particularly to mitigate race conditions and to ensure the freshness and correctness of off-chain data proofs.

The FAsset codebase demonstrates exceptional engineering quality through its well-structured modular architecture, comprehensive documentation, and thoughtful implementation of defense-in-depth principles. The system's multiple redundant safety mechanisms, clear separation of concerns, and extensive use of invariant checks reflect a mature approach to smart contract development. The code is notably clean and maintainable, with consistent patterns throughout and careful attention to edge cases that shows deep understanding of both the technical and economic risks inherent in the design.

The Flare team is greatly appreciated for their exceptional collaboration throughout this audit. Their deep technical expertise, comprehensive documentation, and proactive engagement in addressing the queries posed by the audit team exemplified the highest standards of professionalism. The Flare team's thorough understanding of the system's complexities and their willingness to engage in detailed technical discussions greatly enhanced the quality and depth of the review process.