Zellic

Prepared for
Luka Avbreht
Iztok Kavkler
Flare Network

Prepared by
Weipeng Lai
Kuilin Li
Zellic

November 25, 2025

Flare FAssets
Smart Contract Patch Review

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. Introduction

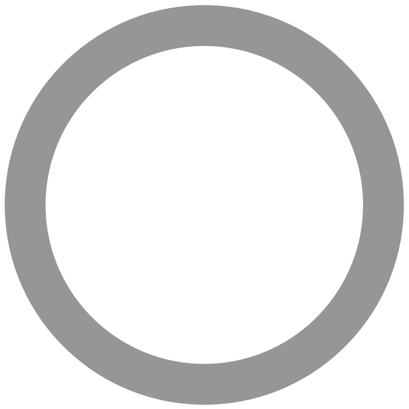
We were asked to review the patches in commit range `58023489` to `4c52db64` of Flare FAssets from November 13th to November 14th, 2025.

1.1. Results

During our assessment on the scoped Flare FAssets contracts, we discovered three findings, all of which were informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	0
⬜ Informational	3



1.2. Scope

The engagement involved a review of the following targets:

Flare FAssets Contracts

Type	Solidity
Platform	EVM-compatible

Target	Only changes between 58023489...4c52db64
Repository	https://gitlab.com/flarenetwork/fassets/fasset ↗
Version	`4c52db6438dbd2aa3a705c20494cd79af60748ab`
Programs	`contracts/assetManager/**/*` `contracts/ftso/**/*` `contracts/userInterfaces/**/*`

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Weipeng Lai
Engineer
weipeng.lai@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

1.3. 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.

2. Detailed Findings

2.1. Missing zero-amount check for closedUBA in selfClose

Target	RedemptionRequestsFacet		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

To compute the `closedWithFeeUBA` amount and ensure that it is less than or equal to the `_amountUBA` input, the `selfClose` function first scales `_amountUBA` down to `toCloseUBA` via `mulDiv`, passes it to `Redemptions.closeTickets` to obtain `closedUBA`, and then applies the inverse `mulDiv` to compute `closedWithFeeUBA`.

```
function selfClose(
    address _agentVault,
    uint256 _amountUBA
)
    external
    notEmergencyPaused
    nonReentrant
    onlyAgentVaultOwner(_agentVault)
    returns (uint256 _closedAmountUBA)
{
    // [...]
    uint256 factorDiv = SafePct.MAX_BIPS * SafePct.MAX_BIPS;
    uint256 factorMul = factorDiv - uint256(settings.redemptionFeeBIPS)
    * agent.redemptionPoolFeeShareBIPS;
    uint256 toCloseUBA = _amountUBA.mulDiv(factorMul, factorDiv);
    // close the agent's backing
    (, uint256 closedUBA) = Redemptions.closeTickets(agent,
    Conversion.convertUBAToAmg(toCloseUBA), true);
    // Calculate the `closedUBA` with added fee by reversing the operation in
    calculating `toCloseUBA`.
    // Resulting `closedWithFeeUBA` will be less than or equal to `_amountUBA`
    since `closedUBA <= toCloseUBA`
    // and the two inverse `mulDiv`s can only make value smaller due to
    rounding down.
    // Therefore, the sender should have enough fassets for the burn (the
    sender is expected to hold
    // `_amountUBA` fassets for selfClose to work).
    // Note also that `factorMul` cannot be 0, since `redemptionFeeBIPS <
    MAX_BIPS`.
    uint256 closedWithFeeUBA = closedUBA.mulDiv(factorDiv, factorMul);
    // [...]
}
```

However, for some nonzero `_amountUBA` values, the combination of the `mulDiv` scaling and `Redemptions.closeTickets` can result in `closedUBA` being zero.

Impact

When `closedUBA` is zero, `selfClose` still performs an FAsset burn and mint operation and emits a `SelfClose` event, even though no effective self-close has occurred.

Recommendations

Add a zero-amount check for `closedUBA`:

```
(, uint256 closedUBA) = Redemptions.closeTickets(agent,
    Conversion.convertUBAToAmg(toCloseUBA), true);
require(closedUBA != 0, SelfCloseOfZero());
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 404c41db ↗.

2.2. Missing zero-amount check before minting the pool fee in `selfClose`

Target	RedemptionRequestsFacet		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `selfClose` function in RedemptionRequestsFacet mints a pool fee and calls `fAssetFeeDeposited` without first checking whether the fee amount is nonzero:

```
function selfClose(
    address _agentVault,
    uint256 _amountUBA
)
    external
    notEmergencyPaused
    nonReentrant
    onlyAgentVaultOwner(_agentVault)
    returns (uint256 _closedAmountUBA)
{
    // [...]
    Globals.getFAsset().mint(address(agent.collateralPool), closedWithFeeUBA
    - closedUBA);
    agent.collateralPool.fAssetFeeDeposited(closedWithFeeUBA - closedUBA);
    // [...]
}
```

However, the pool-fee amount `closedWithFeeUBA - closedUBA` can be zero. This occurs, for example, when `agent.redemptionPoolFeeShareBIPS` is configured to zero:

```
// [...]
uint256 factorDiv = SafePct.MAX_BIPS * SafePct.MAX_BIPS;
uint256 factorMul = factorDiv - uint256(settings.redemptionFeeBIPS)
    * agent.redemptionPoolFeeShareBIPS;
// [...]
uint256 closedWithFeeUBA = closedUBA.mulDiv(factorDiv, factorMul);
```

If `agent.redemptionPoolFeeShareBIPS == 0`, then `factorMul == factorDiv`, so `closedWithFeeUBA == closedUBA` and the computed fee is zero.

Impact

When the pool-fee amount is zero, `selfClose` still executes an FAsset `mint` and an `fAssetFeeDeposited` call with a zero amount. These operations are redundant and incur unnecessary gas costs.

Recommendations

Only mint the pool fee and call `fAssetFeeDeposited` when the fee amount is nonzero:

```
if (closedWithFeeUBA > closedUBA) {
        Globals.getFAsset().mint(address(agent.collateralPool), closedWithFeeUBA
    - closedUBA);
        agent.collateralPool.fAssetFeeDeposited(closedWithFeeUBA - closedUBA);
}
```

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit 454453be ↗.

2.3. The `confirmClosedMintingPayment` feature marginally incentivizes agent misconduct during minting

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

Description

In the case where a user initiates a mint, sends the underlying tokens, and then becomes unresponsive, an honest agent will call `executeMinting` in order to finish the mint and deliver the FAsset tokens to the minter. A dishonest agent has the option of ignoring the underlying transaction, however. If enough time elapses such that no FDC proofs are available to prove the existence or nonexistence of correct underlying payment, then the agent can call `unstickMinting` to release the collateral.

When `unstickMinting` is successfully called, the agent pays a configurable premium in NAT over the price of the collateral in question, and this collateral is then sent directly to the agent. This means that the extra price the agent pays is just this premium. On the other hand, the amount of underlying tokens that were sent to the agent's underlying address is not provable. If the user did actually send the tokens, then those tokens will belong to the agent. However, since the tokens are not part of the agent's underlying balance, the agent will be unable to actually access the value of those tokens until it rotates the entire agent vault. Therefore, even though the price of the premium may economically incentivize agents to call `unstickMinting` whenever they can (and therefore never `executeMinting` on behalf of the user), this inconvenience makes that conceivably unlikely.

However, the addition of the `confirmClosedMintingPayment` function removes this inconvenience disincentive. If an agent notices that a user is not calling `executeMinting`, they can generate an FDC proof and then hold onto it until after they can call `unstickMinting`. The rest of the above proceeds as usual — except afterwards, instead of needing to rotate the entire agent vault, they can provide that proof to `confirmClosedMintingPayment` and immediately obtain the value of the tokens that were sent to their underlying address.

Impact

With this change, agents will be slightly less disincentivized to call `executeMinting` because it makes the underlying balance easier to withdraw.

The impact is informational because, over time, the agent will likely periodically rotate the agent vault and therefore be able to withdraw the amounts underneath its tracked underlying balance anyways.

Recommendations

Since `unstickMinting` is designed to be a rare path in the business logic, we recommend disallowing payment references that have gone through `unstickMinting` from being the subject of `confirmClosedMintingPayment` calls.

Remediation

This issue has been acknowledged by Flare Network, and a fix was implemented in commit `1999a0fc` ↗.

3. Patch Review

The purpose of this section is to document the core changes in the codebase made within the commit range 58023489 ↗ to 4c52db64 ↗.

3.1. Commit 26fc82f0a7

The Flare Network team describes the purpose of commit 26fc82f0 ↗ as follows:

> Confirm payment for closed minting as underlying topup.

The commit introduces a new `confirmClosedMintingPayment` function in MintingFacet:

```
function confirmClosedMintingPayment(
    IPayment.Proof calldata _payment,
    uint256 _crtId
)
    external
    notFullyEmergencyPaused
    nonReentrant
{

    CollateralReservation.Data storage crt = Minting.getCollateralReservation(
        _crtId, false);
    Agent.State storage agent = Agent.get(crt.agentVault);
    // verify transaction
    TransactionAttestation.verifyPaymentSuccess(_payment);
    // only agent can present proof once the payment is defaulted
    Agents.requireAgentVaultOwner(agent);

    require(_payment.data.responseBody.standardPaymentReference == PaymentRefe
        rence.minting(_crtId),
        InvalidMintingReference());
    require(_payment.data.responseBody.receivingAddressHash == agent.
        underlyingAddressHash,
        NotMintingAgentsAddress());
    // we do not allow payments before the underlying block at requests,
        because the payer should have guessed

    // the payment reference, which is good for nothing except attack attempts
    require(_payment.data.responseBody.blockNumber >= crt.firstUnderlyingBlock
        ,
        MintingPaymentTooOld());
    // only closed (not ACTIVE) minting payments can be confirmed in this
        way -
    // ACTIVE minting payments should be confirmed by executeMinting

    require(crt.status != CollateralReservation.Status.ACTIVE, InvalidCollater
        alReservationStatus());
    // mark payment confirmed, which also checks that it hasn't been
        confirmed already
    AssetManagerState.get().paymentConfirmations.confirmIncomingPayment(_
        payment);
    // preform underlying balance topup
    uint256 amountUBA = SafeCast.toUint256(_payment.data.responseBody.
        receivedAmount);
    UnderlyingBalance.increaseBalance(agent, amountUBA.toUint128());
    // update underlying block
    UnderlyingBlockUpdater.updateCurrentBlockForVerifiedPayment(_payment);
    // notify

    emit IAssetManagerEvents.ConfirmedClosedMintingPayment(agent.vaultAddress(
        ),
        _payment.data.requestBody.transactionId, amountUBA);
}
```

This function covers cases where the minter

- paid too late (minting is already defaulted before execution),
- paid too little (so `executeMinting` would revert), or
- paid twice.

In all of these cases, the paid amount ends up on the agent vault's underlying account, but it is not confirmed, and therefore the agent cannot withdraw it without triggering full liquidation (though they can legally withdraw it once the vault is closed).

The `confirmClosedMintingPayment` function allows the agent to confirm such payments, converting the deposited amount to the agent's free underlying.

Note that the introduction of `confirmClosedMintingPayment` slightly weakens an agent's incentive to call `executeMinting` for unresponsive minters (see Finding 2.3. ↗).

3.2. Commit 5956dacedd

The Flare Network team describes the purpose of commit 5956dace ↗ as follows:

> Pay pool share of redemption fee on self close.

This commit updates `selfClose` in RedemptionRequestsFacet so that self-closing also pays the collateral pool's share of the redemption fee, instead of only burning the backing 1:1. The function now derives a reduced backing amount `toCloseUBA`, attempts to close tickets for that amount, and then inverts the fee calculation to obtain `closedWithFeeUBA`, which is burned from the agent. The difference `closedWithFeeUBA - closedUBA` is minted to the collateral pool.

```
function selfClose(
    address _agentVault,
    uint256 _amountUBA
)
    external
    notEmergencyPaused
    nonReentrant
    onlyAgentVaultOwner(_agentVault)
    returns (uint256 _closedAmountUBA)
{
    Agent.State storage agent = Agent.get(_agentVault);
    require(_amountUBA != 0, SelfCloseOfZero());
    uint64 amountAMG = Conversion.convertUBAToAmg(_amountUBA);
    (, uint256 closedUBA) = Redemptions.closeTickets(agent, amountAMG, true);
    // burn the self-closed assets
    Redemptions.burnFAssets(msg.sender, closedUBA);
    AssetManagerSettings.Data storage settings = Globals.getSettings();
    // calculate the amount that can be closed so that there is still enough
        left for the pool fee
    uint256 factorDiv = SafePct.MAX_BIPS * SafePct.MAX_BIPS;
    uint256 factorMul = factorDiv - uint256(settings.redemptionFeeBIPS) *
        agent.redemptionPoolFeeShareBIPS;
    uint256 toCloseUBA = _amountUBA.mulDiv(factorMul, factorDiv);
    // close the agent's backing
    (, uint256 closedUBA) = Redemptions.closeTickets(agent, Conversion.
        convertUBAToAmg(toCloseUBA), true);
    // Calculate the `closedUBA` with added fee by reversing the operation
        in calculating `toCloseUBA`.
    // Resulting `closedWithFeeUBA` will be less than or equal to `_
        amountUBA` since `closedUBA <= toCloseUBA`
    // and the two inverse `mulDiv`s can only make value smaller due to
        rounding down.
    // Therefore, the sender should have enough fassets for the burn (the
        sender is expected to hold
    // _amountUBA fassets for selfClose to work).
    uint256 closedWithFeeUBA = closedUBA.mulDiv(factorDiv, factorMul);
    // Burn sender's self-closed assets plus pool fee
    Redemptions.burnFAssets(msg.sender, closedWithFeeUBA);
    // Mint pool fee to the pool (could transfer fassets from sender to the
        pool instead, but that would require
    // the sender to approve ERC20 transaction, which isn't needed by
        burning and then minting).
    // Note that agent's backing does not need to increase, because in `
        burnFAssets` the fee was burned along
    // with the closed backing amount.
    Globals.getFAsset().mint(address(agent.collateralPool), closedWithFeeUBA
        - closedUBA);
    agent.collateralPool.fAssetFeeDeposited(closedWithFeeUBA - closedUBA);
    // try to pull agent out of liquidation
    Liquidation.endLiquidationIfHealthy(agent);
    // send event
    emit IAssetManagerEvents.SelfClose(_agentVault, closedUBA);
    return closedUBA;
}
```

Overall, the change correctly routes the redemption fee's pool share during self-close and keeps the economics aligned with regular redemptions.

However, we have identified two minor issues:

1. There is a missing zero-amount check for `closedUBA` (see Finding 2.1. ↗).
2. There is a missing zero-amount check before minting the pool fee (see 2.2. ↗)

3.3. Commit c759e33b88

The Flare Network team describes the purpose of commit c759e33b ↗ as follows:

> Require min turnout in publishing ftso prices.

The commit introduces a governance-controlled parameter `minTurnoutBIPS` that specifies the minimum provider turnout required for an FTSO feed's price to be published. A dedicated setter set-

`MinTurnoutBIPS` is added, and the `updateSettings` governance function is extended to configure this parameter.

In `publishPrices`, a feed's price is now only updated if `feed.turnoutBIPS` is greater than or equal to `minTurnoutBIPS`; otherwise, the stored price is left unchanged, and only a `LowTurnoutForFeed` event is emitted.

```
function publishPrices(FeedWithProof[] calldata _proofs) external {
    uint32 votingRoundId = 0;
    require(_proofs.length == feedIds.length, WrongNumberOfProofs());
    for (uint256 i = 0; i < _proofs.length; i++) {
        // [...]

        PriceStore storage priceStore = latestPrices[feedId];
        priceStore.votingRoundId = feed.votingRoundId;
        priceStore.value = uint32(feed.value);
        priceStore.decimals = feed.decimals;

        if (feed.turnoutBIPS >= minTurnoutBIPS) {
            priceStore.votingRoundId = feed.votingRoundId;
            priceStore.value = uint32(feed.value);
            priceStore.decimals = feed.decimals;
        } else {
            emit LowTurnoutForFeed(feedId, feed.votingRoundId, feed.
                turnoutBIPS);
        }

        // [...]
    }
}
```

Overall, the commit correctly introduces a governance-controlled turnout gate on FTSO price publication. Our main concern is operational: if `minTurnoutBIPS` is configured too aggressively, some feeds may update infrequently or become effectively stalled, leading to stale prices.

3.4. Commit `a860735f08`

The Flare Network team describes the purpose of commit `a860735f` ↗ as follows:

> Penalty for defaults on core vault transfer.

The commit introduces a penalty paid in vault collateral for defaulting on the underlying payment after an agent initiates a transfer to the core vault. This penalty is intended to discourage blocking operations like liquidation via creating unpaid transfers to the core vault.

A new `transferDefaultPenaltyBIPS` setting is introduced and stored in `CoreVaultClient.State`, configured via `initCoreVaultFacet` and the newly added governance-controlled setter `setCoreVaultTransferDefaultPenaltyBIPS`, with a corresponding new added getter `getCoreVaultTransferDefaultPenaltyBIPS` to read it back.

This setting is used by the newly added `transferToCoreVaultDefaultPenalty` function in `CoreVaultClient`, which computes a penalty in vault collateral by converting the redemption amount to an equivalent collateral amount, multiplying by `transferDefaultPenaltyBIPS`, and capping the result by the vault's `maxRedemptionCollateral` for that redemption.

The computed penalty is then charged in `RedemptionDefaults.executeDefaultOrCancel` for the `transferToCoreVault` default path, where the function calls `transferToCoreVaultDefaultPenalty` and pays the resulting amount from the agent to the core vault via `AgentPayout.tryPayoutFromVault`.

Overall, the commit introduces a configurable penalty for defaulted transfers to the core vault, strengthening economic incentives against strategic defaults.

3.5. Commit `080a0191ed`

The Flare Network team describes the purpose of commit 080a0191 ↗ as follows:

> Limit fee settings to less than max bips.

This commit enforces that collateral-reservation–fee and redemption-fee parameters (in BIPS) are strictly less than `SafePct.MAX_BIPS` (i.e., < 100%) instead of allowing exactly `MAX_BIPS`. The corresponding validation logic is updated both in the setter functions (`setCollateralReservationFeeBips`, `setRedemptionFeeBips`) and in the settings initialization path (`AssetManagerInit.init` → `SettingsInitializer.validateAndSet` → `SettingsInitializer._validateSettings`).

```
function setCollateralReservationFeeBips(uint256 _value)
    external
    onlyAssetManagerController
    rateLimited
{
    AssetManagerSettings.Data storage settings = Globals.getSettings();
    // validate
    require(_value > 0, CannotBeZero());
    require(_value <= SafePct.MAX_BIPS, BipsValueTooHigh());
    require(_value < SafePct.MAX_BIPS, BipsValueTooHigh());
    require(_value <= settings.collateralReservationFeeBIPS * 4,
    FeeIncreaseTooBig());
    require(_value >= settings.collateralReservationFeeBIPS / 4,
    FeeDecreaseTooBig());
    // update
    settings.collateralReservationFeeBIPS = _value.toUint16();
    emit SettingChanged("collateralReservationFeeBIPS", _value);
}


function setRedemptionFeeBips(uint256 _value)
    external
    onlyAssetManagerController
    rateLimited
{
    AssetManagerSettings.Data storage settings = Globals.getSettings();
    // validate
    require(_value > 0, CannotBeZero());
    require(_value <= SafePct.MAX_BIPS, BipsValueTooHigh());
    require(_value < SafePct.MAX_BIPS, BipsValueTooHigh());
    require(_value <= settings.redemptionFeeBIPS * 4, FeeIncreaseTooBig());
    require(_value >= settings.redemptionFeeBIPS / 4, FeeDecreaseTooBig());
    // update
    settings.redemptionFeeBIPS = _value.toUint16();
    emit SettingChanged("redemptionFeeBIPS", _value);
}

function _validateSettings(
    AssetManagerSettings.Data memory _settings
)
    private pure
{
    // [...]
    require(_settings.collateralReservationFeeBIPS <= SafePct.MAX_BIPS,
        BipsValueTooHigh());

    require(_settings.redemptionFeeBIPS <= SafePct.MAX_BIPS, BipsValueTooHigh(
        ));
    require(_settings.collateralReservationFeeBIPS < SafePct.MAX_BIPS,
        BipsValueTooHigh());
    require(_settings.redemptionFeeBIPS < SafePct.MAX_BIPS, BipsValueTooHigh(
        ));
    // [...]
}
```

Overall, this commit tightens validation so that the collateral-reservation fee and redemption fee must be strictly less than 100%.