



cQinspect
You build, we defend.



Smart Contract Audit
FAssets V2 Update

Dec, 2024



FAsset V2 Updates Smart Contract Audit

Version: v241217

Prepared for: Flare

December 2024

Security Assessment

1. Executive Summary
2. Summary of Findings
 - 2.2 Finding where caution is advised
 - 2.3 Solved issues & recommendations
3. Scope
4. Assessment
 - Main Changes
 - Minor Changes
 - Minor changes after initial review
5. Detailed Findings

FAS-040 - Collateral pools will spontaneously break leading to financial losses

FAS-041 - Fee payment bypass by transferring small amounts of FAssets

FAS-042 - Lack of reinitialization protection in TransferFeeFacet

FAS-043 - A single compromised or rogue trusted provider can manipulate the price feed for profit

FAS-044 - Quality of trusted price feeds is not guaranteed

FAS-045 - Call to arbitrary address without respecting Check-Effects-Interactions

FAS-046 - FAssets fee distribution incentivizes agents to be close to liquidation

FAS-047 - Evil minter can lock agent collateral for free




FAS-048 - Banned redeemers targeting agents with handshake can farm premium fees

6. Disclaimer

1. Executive Summary

In **November, 2024**, Flare engaged Coinspect to perform a Smart Contract Audit of FAsset V2 Protocol. The objective of the project was to evaluate the security of new features and changes added to the smart contracts.

The **FAsset V2 Protocol** is a collateralized bridge solution that enables cross-chain native token transfers.

 Solved	 Caution Advised	 Resolution Pending
High 2	High 0	High 0
Medium 0	Medium 1	Medium 0
Low 1	Low 0	Low 0
No Risk 5	No Risk 0	No Risk 0
Total 8	Total 1	Total 0

During this assessment, Coinspect identified three high-risk, one medium-risk and one low-risk issues.

FAS-40 shows how collateral pools are broken and will not work as the FAsset token now charges a fee upon transfer. FAS-46 mentions that users are incentivized to increase their debt to get FAsset fees, ultimately lowering the protocol's global collateral ratio. Lastly, FAS-47 depicts how an evil minter is able to reserve collateral for free abusing the handshake.

FAS-48 shows a mechanism where evil redeemers could farm redemption premiums and FAS-41 warns about how users could bypass paying FAsset transfer

fees.

2. Summary of Findings

This section provides a concise overview of all the findings in the report grouped by remediation status and sorted by estimated total risk.

2.2 Finding where caution is advised

Issues with risk in this list have been addressed to some extent but not fully mitigated. Any future changes to the codebase should be carefully evaluated to avoid exacerbating these issues or increasing their probability.

Findings with a risk of *None* pose no threat, but document an implicit assumption which must be taken into account. Once acknowledged, these are considered solved.

Id	Title	Risk
FAS-048	Banned redeemers targeting agents with handshake can farm premium fees	Medium

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
FAS-040	Collateral pools will spontaneously break leading to financial losses	High
FAS-047	Evil minter can lock agent collateral for free	High
FAS-041	Fee payment bypass by transferring small amounts of FAssets	Low

FAS-042	Lack of reinitialization protection in TransferFeeFacet	None
FAS-043	A single compromised or rogue trusted provider can manipulate the price feed for profit	None
FAS-044	Quality of trusted price feeds is not guaranteed	None
FAS-045	Call to arbitrary address without respecting Check-Effects-Interactions	None
FAS-046	FAssets fee distribution incentivizes agents to be close to liquidation	None

3. Scope

The scope was set to be the repository at <https://gitlab.com/flarenetwork/fasset> at commit `fcf6ab7b5e4628c6558f7fee736b911945a1ddb0`.

Flare requested Coinspect to review two main changes related to new features added to the protocol:

- **Handshake:** agents are now allowed to reject a minting or redemption request depending on the requester's identity.
- **FAsset Token Fee On Transfer:** Each `FAsset` token collects a fee when making a transfer, which is then distributed among all agents.

Also, a set of smaller, second-priority changes were included alongside the before mentioned major features.

Coinspect focused on reviewing these changes comparing the previous' review commit, `997fac606b93f2fac0176f1243f455da44041d4f` against the new one.

4. Assessment

During this security assessment, Coinspect reviewed multiple changes added to the FAsset Protocol. These changes were separated into main and minor changes.

Main Changes

Handshake

The main changes include a new way for agents to determine whether they fulfill a redemption or minting request based on the requester's identity. This feature is called **Handshake**. Agents can require a **handsake** and will have to confirm or reject a request before it is resolved. For minting requests, once a user reserves collateral the agent should confirm or reject this request. There are three scenarios:

1. **Accept**: once the request is accepted, the last underlying payment blocks and timestamps are updated and the minter has to pay in the underlying chain.
2. **Reject**: if a request is rejected, the minter is able to recover the reservation fee.
3. **No agent response**: same scenario as 2. but triggered by the minter after a time-rejection window.

Coinspect identified that this feature allows minters to use accounts with low reputation to lock an agent's collateral on purpose by knowing in advance they will reject the request.

In terms of redemptions, the handshake allows agents to reject a redemption request. Since FAssets are burned from the redeemer once a request is created, when an agent rejects a redemption there are two possible outcomes:

1. Another agent takes over the redemption, proceeding to pay the redeemer in the underlying chain.
2. No agent takes over the redemption, and the redeemer gets paid in collateral (plus a premium) on Flare Chain. This payment is made by the agent that rejected the redemption.

FAsset Token transfer fees

The last main change adds fee-on-transfer functionality to the FAsset token. Now, when a FAsset is transferred a portion of fees are diverted to a common pool. This pool is divided across all agents using a share and epoch based mechanism. Agents with more minted FAssets get more shares thus collected fees.

Coinspect identified that this change encourages agents to reduce their collateral ratio by increasing the exposure to minted FAssets, for example, while minting against themselves. Globally, this leads to collateral ratios being reduced and introduces a system risk as agents are incentivized to be closer to liquidation.

Minor Changes

Coinspect also reviewed a set of minor changes, comprised by the following items:

- Simplified settings updates. Moved setters from `SettingsUpdater` to `SettingsManagementFacet`.
- Removed `AMEvents` library. Emit events from interface `IAssetManagerEvents`.
- Modified agent's whitelist and owner registry:
 - The governance can now assign a manager, expected to be a simpler multisig, to take over whitelisting agents. Coinspect observed that managers can also be externally owned accounts, creating a single point of failure.
 - Agents can optionally add "terms of use" URL.
 - Agent metadata items can be set separately.
- Consecutive redemption tickets to the same agent are merged.
- Minting from free underlying logic: when an agent has more than 1 lot of free underlying, they can mint without underlying payment.
- Added new methods to query the state of the redemption queue view.
- Changed FAsset termination logic on collateral pools: after a FAsset is terminated, fees are not paid.
- Added support for different recipients when exiting a pool.
- Changed the architecture for the Asset Manager Controller, deploying it behind a proxy.

A note on this last item: Coinspect identified that `authorizeUpgrade` has no access control. Access control is at `upgradeTo` and `upgradeToAndCall` level. The `authorizeUpgrade` function is meant to work as the access control function. In case of future upgrades unaware of this, removing the modifiers on `upgradeTo` and `upgradeToAndCall` will allow anyone to trigger an update.

- Made minor changes to the liquidation logic:
 - `startLiquidation` reverts if that liquidation status does not change.

- `LiquidationStarted` is sent on `startLiquidation` or first `liquidate` also if liquidation started due to CCB time expiration.
- `LiquidationPerformed` event not sent if nothing is liquidated.
- Changed the destination of the collateral reservation fee: it is now paid to the agent and their~ pool on successful minting, instead of burning it.
- Modified the Agent's info: available agent info contains liquidation status. This aims to allow quickly filtering out liquidated agents, that cannot mint.
- Self mint gets own event `SelfMint`.
- Reward claiming changed for `FTS0v2`.
- Use of `FDC` instead of `StateConnector` to make verifications.
- Added a price store contract for publishing `FTS0v2` anchor feeds and at the same time submitting trusted providers' prices.
- Converted agent vault, collateral pool and collateral pool token to upgradeable (UUPS) proxies.
- Converted `FAsset` from transparent to UUPS proxy to align with other proxy contracts.

Minor changes after initial review


Coinspect reviewed new feature that allows the governance or an allowed account to pause `FAsset` transfers. No issues related to this feature were identified.

The scope for this new feature was set to be the repository at <https://gitlab.com/flarenetwork/fasset> at commit `1edeef1b3c8472e205261c51f75eead61145aead`.

5. Detailed Findings

FAS-040

Collateral pools will spontaneously break leading to financial losses

Status Solved	Risk High
	
Resolution Fixed	Impact High Likelihood High

Location

`fasset/contracts/assetManager/implementation/CollateralPool.sol`

Description

Each interaction with the Collateral Pools will result in an FAsset accountancy discrepancy, where either less is deposited upon entering, too much debt is canceled during repayment, insufficient assets are available for user exits or fee withdrawals, or fees are not fully covered. These discrepancies lead to unexpected reverts and accountancy errors across the Collateral Pools. This arises from the system's new FAsset implementation introducing transfer fees.

For an example, consider debt repayment. When entering a Collateral Pool, if there is a difference between the pool shares and the FAssets provided, debt is minted to the user:

```
uint256 tokenShare = _collateralToTokenShare(assetData, msg.value);
uint256 fAssetShare = assetData.poolTokenSupply > 0 ?
    assetData.poolVirtualFAssetFees.mulDiv(tokenShare,
assetData.poolTokenSupply) : 0;
uint256 depositedFAsset = _enterWithFullFAssets ? fAssetShare :
Math.min(_fAssets, fAssetShare);
// transfer/mint calculated assets
if (depositedFAsset > 0) {
    require(fAsset.allowance(msg.sender, address(this)) >=
depositedFAsset,
        "f-asset allowance too small");
    _transferFAsset(msg.sender, address(this), depositedFAsset);
}
_mintFAssetFeeDebt(msg.sender, fAssetShare - depositedFAsset);
```

Then, when exiting or repaying this debt, the Collateral Pool will burn the debt in a `_fAssets` amount:

```
function payFAssetFeeDebt(uint256 _fAssets)
    external
    nonReentrant
{
    require(_fAssets != 0, "zero f-asset debt payment");
    require(_fAssets <= _fAssetFeeDebtOf[msg.sender], "debt f-asset
balance too small");
    require(fAsset.allowance(msg.sender, address(this)) >= _fAssets,
"f-asset allowance too small");
    _burnFAssetFeeDebt(msg.sender, _fAssets);
    _transferFAsset(msg.sender, address(this), _fAssets);
    // emit event
    emit Entered(msg.sender, 0, 0, _fAssets,
_fAssetFeeDebtOf[msg.sender], 0);
}
```

```
function _burnFAssetFeeDebt(address _account, uint256 _fAssets)
    internal
{
    _fAssetFeeDebtOf[_account] -= _fAssets;
    totalFAssetFeeDebt -= _fAssets;
}
```

However, the amount of FAssets received is less than the parameter's value, because fees are diverted to other recipient:

```
function _transferFAsset(
    address _from,
    address _to,
    uint256 _amount
)
```

```

internal
{
  if (_amount > 0) {
    if (_from == address(this)) {
      totalFAssetFees -= _amount;
      fAsset.safeTransfer(_to, _amount);
    } else { // if (_to == address(this)) {
      /* solhint-disable reentrancy */
      totalFAssetFees += _amount;
      fAsset.safeTransferFrom(_from, _to, _amount);
    }
  }
}
}

```

This behavior also triggers accountancy discrepancies and reversals after the following cases:

1. A collateral pool will receive fewer `additionallyRequiredFAssets` upon `selfClose`
2. Users might not be able to exit a pool because there may not be enough `FAssets` to cover the `freeFAssetFeeShare`
3. The amount of deposited `FAssets` when entering a pool will be less than what users are entitled to
4. Users might not be able to withdraw fees, for the same reason as in point 2.
5. Debt repayments will receive less `FAsset` value than the canceled debt amount

Coinspect considers this issue to have a high likelihood, as `FAssets` are now a fee-on-transfer token, enabling multiple accountancy discrepancies in every Collateral Pool. The impact is also considered high because this issue affects all Agents, decoupling their `FAsset` balance from their internal accountancy.

Coinspect observed that the tests for the Collateral Pool use a `ERC20Mock` token instead of deploying the actual `FAsset` with the fee-on-transfer functionality:

```
fAsset = await ERC20Mock.new("fBitcoin", "fBTC");
```

Recommendation

Handle effective received balances when using the new version of `FAsset`. Update those tests that use a `ERC20Mock` to represent `FAssets`, using their actual implementation.

Status

Fixed on commit 541d4617453434f202ee57b10de1254f61b330b2.

The Collateral Pool smart contract now performs FAsset transfers and internal accountancy updates considering transfer fees.

Proof of Concept

The following test shows how less FAssets are received by the Collateral Pool after a user fully repays their debt.

To make this test, a high transfer fee was set (50%) but it will be ultimately also triggered with high enough FAsset transfer amounts and lower fees.

```
FAsset Fee Debt Acc 1: 79
- Debt Repaid -
Fasset Balance Diff @ Collateral Pool: 40
FAsset Fee Debt Acc 1: 0
```

```
it("Coinspect - Fee of FAssets break CollateralPool accountancy", async
() => {
  // To amplify the impact of fees for small amounts, this test uses
  a 50% of fee.
  // However, this scenario will happen for every FAsset interaction
  with a Collateral Pool
  const agent = await Agent.createTest(context, agentOwner1,
  underlyingAgent1);
  const minter = await Minter.createTest(context, userAddress1,
  underlyingUser1, context.lotSize().muln(100));
  await agent.depositCollateralsAndMakeAvailable(toWei(1e8),
  toWei(1e8));
  mockChain.mine(10);
  await context.updateUnderlyingBlock();
  // settings
  const lotSize = context.lotSize();
  const transferFeeMillionths = await
  assetManager.transferFeeMillionths();
  const eventDecoder = new Web3EventDecoder({ fAsset: context.fAsset
  })
  // perform minting
  const lots = 10;
  const [minted] = await minter.performMinting(agent.vaultAddress,
  lots);
  const transfer1LotFee =
  lotSize.mul(transferFeeMillionths).divn(1e6);

  // transfer
  const startBalance1 = await fAsset.balanceOf(minter.address);
  const res1 = await fAsset.transfer(userAddress2, lotSize, { from:
  minter.address });
  const res2 = await fAsset.transfer(userAddress3, lotSize, { from:
  minter.address });

  const ONE_ETH = toBN("100000000000000000");
```

```

const ETH = (x: any) => ONE_ETH.mul(toBN(x));

    await context.fAsset.approve(agent.collateralPool.address, ETH(10),
{ from: userAddress2 });
    await agent.collateralPool.enter(0, true, {value: ETH(10), from:
userAddress2})

    await context.fAsset.approve(agent.collateralPool.address, ETH(1),
{ from: userAddress3 });
    await agent.collateralPool.enter(ETH(0), false, { value: ETH(100),
from: userAddress3 });

    // Account 1 tries to repay the fasset fee debt
    let ac1FassetFeeDebt = await
agent.collateralPool.fAssetFeeDebtOf(userAddress3)
    console.log(`FAsset Fee Debt Acc 1: ${Number(ac1FassetFeeDebt)}`);

    const balanceBefore = await
context.fAsset.balanceOf(agent.collateralPool.address);
    await agent.collateralPool.payFAssetFeeDebt(ac1FassetFeeDebt, {
from: userAddress3 });
    const balanceAfter = await
context.fAsset.balanceOf(agent.collateralPool.address);
    console.log("- Debt Repaid -")
    console.log(`Fasset Balance Diff @ Collateral Pool:
${Number(balanceAfter.sub(balanceBefore))}`)
    console.log(`FAsset Fee Debt Acc 1: ${Number(await
agent.collateralPool.fAssetFeeDebtOf(userAddress3))}`);
});

```


FAS-041

Fee payment bypass by transferring small amounts of FAssets

Status

Solved



Resolution

Fixed

Risk

Low



Impact

Low

Likelihood

Low

Location

fasset/contracts/fassetToken/implementation/FAsset.sol:352

Description

Adversaries are able to bypass paying fees by making FAsset transfers of low amounts.

The incentives to bypass the fee payment are related to the network congestion, the asset's price and decimals. For example, in an uncongested network adversaries will be incentivized to perform small transfers of `fBTC`. This incentive grows if the price of BTC rises (price of BTC at the time of this review, \$99,000).

This happens because the fee calculation floors down:

```
function _transferFeeAmount(uint256 _transferAmount)
    private view
    returns (uint256)
```

```
{
  uint256 feeMillionths =
  IIAssetManager(assetManager).transferFeeMillionths();
  return SafePct.mulDiv(_transferAmount, feeMillionths, 1e6);
}
```

Attackers can easily calculate the maximum amount to transfer so they do not pay any fees:

```
amt = (1e6 / currentFee) - 1
```

In other words, for lower fees the maximum transfer amount that bypasses fee payment grows. At most, the maximum amount is $1e6 - 1$, that happens when `currentFeeMillionths = 1`. Considering a FAsset of 8 decimals (fBTC) this amount represents almost 0.01 fBTC, \$990 at current market prices.

Recommendation

Make and document a fee adjustment plan to reduce the profitability of making small FAsset transfers.

Status

Fixed on commit [541d4617453434f202ee57b10de1254f61b330b2](#).

Calculation of transfer fees now rounds up.

Proof of Concept

The following test show how an user bypasses paying fees by making a small fBTC transfer.

```
it("Coinspect - Bypass paying transfer fees", async () => {
  const agent = await Agent.createTest(context, agentOwner1,
  underlyingAgent1);
  const minter = await Minter.createTest(context, userAddress1,
  underlyingUser1, context.lotSize().muln(100));
  const redeemer = await Redeemer.create(context, userAddress2,
  underlyingUser2);
  const agentInfo = await agent.getAgentInfo();
  await agent.depositCollateralsAndMakeAvailable(toWei(1e8),
  toWei(1e8));
  mockChain.mine(10);
  await context.updateUnderlyingBlock();
});
```

```

const currentEpoch = await assetManager.currentTransferFeeEpoch();
const trfSettings = await assetManager.transferFeeSettings();
// perform minting
const lots = 3;
const [minted] = await minter.performMinting(agent.vaultAddress,
lots);

// transfer just one token less than 1e6/fee to trigger the integer
division.
const transferAmount =
toBN(1e6).div(toBN(trfSettings.transferFeeMillionths)).sub(toBN(1));
const transferFee =
transferAmount.mul(toBN(trfSettings.transferFeeMillionths)).divn(1e6);

console.log(`Transfer Fee: ${Number(transferFee)}`);
console.log(`Transfer Amt: ${Number(transferAmount)}`);

const startBalanceM = await fAsset.balanceOf(minter.address);
const startBalanceR = await fAsset.balanceOf(redeemer.address);
const startBalanceAM = await
fAsset.balanceOf(assetManager.address);

const transfer = await minter.transferFAsset(redeemer.address,
transferAmount);
const endBalanceM = await fAsset.balanceOf(minter.address);
const endBalanceR = await fAsset.balanceOf(redeemer.address);
const endBalanceAM = await fAsset.balanceOf(assetManager.address);

assertWeb3Equal(endBalanceAM, startBalanceAM); // No Fassets were
sent to the AssetManager
});

```

FAS-042

Lack of reinitialization protection in TransferFeeFacet

Status

Solved



Resolution

Fixed

Risk

None



Impact

Recommendation

Likelihood

-

Location

fasset/contracts/assetManager/facets/TransferFeeFacet.sol:36

Description



The `TransferFeeFacet` includes initialization logic invoked only when adding this facet by calling `diamondCut()`. Although this function is not added to the proxy's supported selectors, anyone is able to invoke it by directly calling the `TransferFeeFacet` contract.

```
function initTransferFeeFacet(
    uint256 _transferFeeMillionths,
    uint256 _firstEpochStartTs,
    uint256 _epochDuration,
    uint256 _maxUnexpiredEpochs
)
    external
{
    LibDiamond.DiamondStorage storage ds =
    LibDiamond.diamondStorage();
```


The mentioned smart contracts now include reinitialization protection.

FAS-043

A single compromised or rogue trusted provider can manipulate the price feed for profit

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -

Location

fasset/contracts/assetManager/implementation/FtsoV2PriceStore.sol:344

Description

Adversaries compromising only a single provider are allowed to report and control the trusted price between a range, allowing them to take profits by trading tied assets.

The implementation for `FTSOV2PriceStore` calculates the median for the prices reported for a feed. When the amount of reported values is even, the average of the two values in the middle is used as the epoch's trusted price:

```
uint256 middleIndex = length / 2;  
if (length % 2 == 1) {  
    return prices[middleIndex];  
} else {
```

```
// if median is "in the middle", take the average price of the
two consecutive prices
return (prices[middleIndex - 1] + prices[middleIndex]) / 2;
}
```

However, when the amount of reported prices is odd, the returned median is in the center of the ordered list of prices. This allows an attacker compromising a single provider to report any price between `prices[middleIndex - 1]` and `prices[middleIndex + 1]`.

Consider the following scenario:

- Three trusted providers are registered, O1, O2 and O3.
- Two of them report some prices: [1000, 1200]
- Then, the remaining oracle can decide to report any price between the already reported prices (e.g. 1001, 1099, 1199, etc) and take profits elsewhere.

This issue is informational only as users of the price oracle are expected to query both the trusted and non-trusted prices and compare between the two, taking reasonable action if a considerable deviation is encountered.

Recommendation

Design and include a monitoring plan to detect anomalies in trusted providers, which will allow the Governance to quickly remove the rogue/compromised provider from the trusted list.

Status

Fixed on commit [aac7275bcb5479aa9f502074226cb0b5a99d786e](#).

A variable called `maxSpreadBIPS` was added. Trusted prices are now only updated if the spread between the two middle prices is less than the accepted value.

Proof of Concept

The following test shows how a rogue/compromised provider is able to decide the price of a feed between a range.

```
it("Coinspect - Submissions with high deviation allow a single oracle
to manipulate the price feed", async () => {
```



```

    const newTrustedProviders = [accounts[1], accounts[2],
accounts[3]];
    await priceStore.setTrustedProviders(newTrustedProviders, 2, {
from: governance });

    await time.increaseTo(startTs + 2 * votingEpochDurationSeconds); //
start of voting round 2
    const feeds0 = [];
    const feeds1 = [];
    const feeds2 = [];

    for (let i = 0; i < feedIds.length; i++) {
        feeds0.push({ id: feedIds[i], value: 10000, decimals:
feedDecimals[i] });
        feeds1.push({ id: feedIds[i], value: 10215, decimals:
feedDecimals[i] }); // Rogue provider's price
        feeds2.push({ id: feedIds[i], value: 11000, decimals:
feedDecimals[i] });
    }

    await priceStore.submitTrustedPrices(1, feeds0, { from:
newTrustedProviders[0] });
    await priceStore.submitTrustedPrices(1, feeds1, { from:
newTrustedProviders[1] }); // Rogue provider
    await priceStore.submitTrustedPrices(1, feeds2, { from:
newTrustedProviders[2] });

    await publishPrices();

    const { 0: price, 1: timestamp, 2: decimals } = await
priceStore.getPriceFromTrustedProviders("USDC");
    // price will be 10215, submitted by accounts[2] (no average is
made)
    // only one compromised oracle is able to report a price between
[Oracle min - Oracle Max]
    assertWeb3Equal(price, 10215);
});

```

FAS-044

Quality of trusted price feeds is not guaranteed

Status

Solved

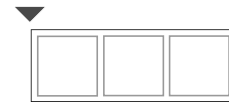


Resolution

Fixed

Risk

None



Impact

Recommendation

Likelihood

-

Location

fasset/contracts/assetManager/implementation/FtsoV2PriceStore.sol

Description

Users cannot determine the quality of a median price over time since the submission threshold might change in the future.

The new price feed allows the governance to set any `trustedProvidersThreshold` for the submission of trusted prices, even zero. In other words, users are expected to trust that all providers are not faulty at anytime. This assumption can be broken if only one trusted provider gets compromised or goes rogue.

```
function setTrustedProviders(  
    address[] calldata _trustedProviders,  
    uint8 _trustedProvidersThreshold  
)  
    external onlyGovernance
```

```

    {
        require(_trustedProviders.length >= _trustedProvidersThreshold,
            "threshold too high");
        trustedProvidersThreshold = _trustedProvidersThreshold;
        // reset all trusted providers
        for (uint256 i = 0; i < trustedProviders.length; i++) {
            trustedProvidersMap[trustedProviders[i]] = false;
        }
        // set new trusted providers
        trustedProviders = _trustedProviders;
        for (uint256 i = 0; i < _trustedProviders.length; i++) {
            trustedProvidersMap[_trustedProviders[i]] = true;
        }
    }
}

```

The threshold can be changed by the governance at anytime and stored prices are untied to that threshold. This means that the quality of a published price for an epoch where the threshold was 1 is treated the same as another one where the threshold was 20.

```

if (trustedPrices.length > 0 && trustedPrices.length >= 4 *
    trustedProvidersThreshold) {
    // calculate median price
    uint256 medianPrice = _calculateMedian(trustedPrices);
    // store the median price
    priceStore.trustedVotingRoundId = votingRoundId;
    priceStore.trustedValue = uint32(medianPrice);
    // delete submitted trusted prices
    delete submittedTrustedPrices[feedId][votingRoundId];
}

```

Recommendation

Store the threshold on each `latestPrices` feed and return it when querying trusted prices. This way, consumers will be able to determine if the current threshold is safe according to their threat model.

Status

Fixed on commit [aac7275bcb5479aa9f502074226cb0b5a99d786e](#).

A new function that returns the amount of trusted submissions alongside a price was added.

FAS-045

Call to arbitrary address without respecting Check-Effects-Interactions

Status

Solved



Resolution

Fixed

Risk

None



Impact

Recommendation

Likelihood

-

Location

fasset/contracts/assetManager/library/CollateralReservations.sol

Description

A call is made to the arbitrary `minter` address on a `CollateralReservation` when the collateral reservation is rejected or cancelled in `_rejectOrCancelCollateralReservation` without the `Checks-Effects-Interactions` pattern.

The logic is not currently affected by reentrancy issues as the callers employ the `nonReentrant` modifier. Nevertheless, the pattern is useful to avoid issues even in the case of changes in the caller's logic.

```
function _rejectOrCancelCollateralReservation(  
    CollateralReservation.Data storage crt,  
    uint64 _crtId  
)  
    private
```

```

{
    uint256 totalFee = crt.reservationFeeNatWei +
crt.executorFeeNatGWei * Conversion.GWEI;

    // guarded against reentrancy in CollateralReservationsFacet
    /* solhint-disable avoid-low-level-calls */
    //slither-disable-next-line arbitrary-send-eth
    (bool success, ) = crt.minter.call{value: totalFee, gas:
100000}("");
    /* solhint-enable avoid-low-level-calls */
    if (!success) {
        // if failed, burn the fee
        Agents.burnDirectNAT(totalFee);
    }

    // release agent's reserved collateral
    releaseCollateralReservation(crt, _crtId); // crt can't be
used after this
}

```

Recommendation

Implement the checks-effects-interaction pattern by first releasing the collateral and then making the arbitrary call.

Status

Fixed on commit [7cb3e370751883ec55543c34c41a82d2b7683275](#).

The smart contract now releases the collateral reservation before making the external call.

FAS-046

FAssets fee distribution incentivizes agents to be close to liquidation

Status

Solved

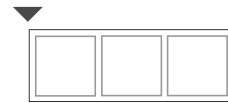


Resolution

Acknowledged

Risk

None



Impact

Recommendation

Likelihood

-

Location

fasset/contracts/assetManager/library/data/TransferFeeTracking.sol:140

Description

Agents are incentivized to increase their exposure to minted FAssets, reducing the system's total collateral ratio and increasing the risk of liquidation.

When a FAsset is transferred, a percentage representing fees is diverted and accumulated in a pool. Then, agents are able to claim those fees based on their share:

```
function claimFees(Data storage _data, address _agentVault, uint256
_maxClaimEpochs)
    internal
    returns (uint256 _claimedFees, uint256 _remainingUnclaimedEpochs)
{
    AgentData storage agent = _data.agents[_agentVault];
```

```

uint256 currentEpochNo = currentEpoch(_data);
uint256 firstUnclaimedEpoch = Math.max(agent.firstUnclaimedEpoch,
_data.firstClaimableEpoch);
uint256 claimUntilEpoch = Math.min(currentEpochNo,
firstUnclaimedEpoch + _maxClaimEpochs);
_claimedFees = 0;
for (uint256 epoch = firstUnclaimedEpoch; epoch < claimUntilEpoch;
epoch++) {
uint256 feeShare = agentFeeShare(_data, agent, epoch);
_claimedFees += feeShare;
_data.epochs[epoch].claimedFees += feeShare.toUint128();
}
agent.firstUnclaimedEpoch = claimUntilEpoch.toUint64();
_remainingUnclaimedEpochs = currentEpochNo - claimUntilEpoch;
}

```

This share is calculated considering the minting exposure at the claiming epoch:

```

function agentFeeShare(Data storage _data, AgentData storage _agent,
uint256 _epoch)
internal view
returns (uint256)
{
ClaimEpoch storage claimEpoch = _data.epochs[_epoch];
uint256 agentCumulativeMinted = epochCumulative(_data,
_agent.mintingHistory, _epoch);
uint256 totalCumulativeMinted = epochCumulative(_data,
_data.totalMintingHistory, _epoch);
assert(agentCumulativeMinted <= totalCumulativeMinted);
if (agentCumulativeMinted == 0) return 0;
return SafePct.mulDiv(claimEpoch.totalFees, agentCumulativeMinted,
totalCumulativeMinted);
}

```

This way of calculating shares drives the protocol to a risky state since agents will be minting more FAssets, increasing their shares but decreasing their collateral ratio. In case of a sudden price swing, multiple agents will go underwater and there might not be enough liquidators to liquidate all agents in time.

Recommendation

Checkpoint the agent's total collateral and use it to determine each agent's shares. This way, agents will be incentivized to increase their position in the FAsset protocol and avoids rewarding risky behavior.

Status

Acknowledged.

The Flare Team considered the recommendation of this issue and made a thorough analysis on which parameter should be used to calculate the portion of rewards:

"The choice of using agent's backing of minted FAssets to calculate the transfer fee share was deliberate.

We wanted to compensate the agents for the locked collateral, especially if there are not enough redemptions - in such a case, the agent's collateral is locked indefinitely, with no possibility to earn more minting fees.

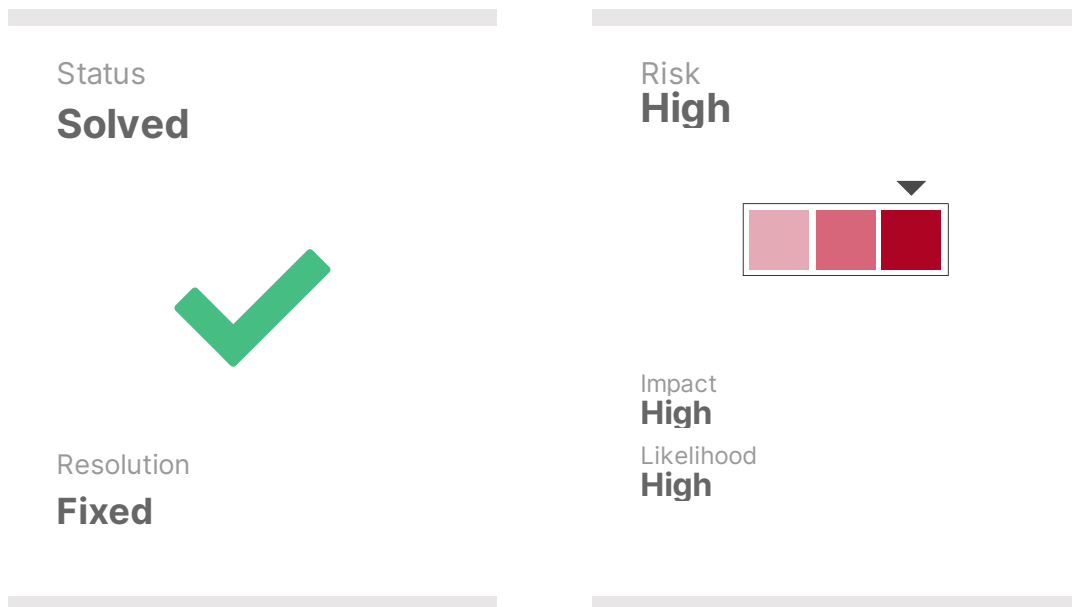
If we switched to total amount of agent's collateral for calculating the fee share, an agent could put up huge amount of collateral and set huge minting CR (e.g. 10000 instead of usual single digit value). Such an agent could not be minted against, since the collateral wouldn't be enough for even 1 lot, but would still collect the majority of transfer fees. (Another way would be to have normal minting CR but set minting fee to 100%.) Such an agent would also not be locked into the system, as they could withdraw the collateral at any time (after timelock), and they would carry no risk of liquidation. Every rational agent would soon switch to this mode of operation as it provides income with no risk, which would make minting impossible.

Moreover, we also don't believe that implemented transfer fee sharing increases the risk to the system: agents are incentivized to avoid liquidations by the threat of liquidation payment premium, which is significantly higher than the transfer fee. Possibility of liquidation is the risk for the agent - they need to set their minting CR by weighing better return on capital versus higher risk of liquidation premium payments. (Besides, minting fees were also based on the amount of backed minting, not the amount of collateral, so nothing has really changed in this regard.)"

In addition to that, Coinspect also considers that incentives should be carefully monitored as there could be a scenario where the collection of transfer fees exceeds what it's lost due to liquidation premiums. For example, a liquid market that has a high volume of daily FAsset transfers. Incentives should be carefully aligned by fine-tuning the transfer fee ratio to prevent experiencing a collateral ratio reduction via the mechanism mentioned on this issue.

FAS-047

Evil minter can lock agent collateral for free



Description

A minter with a low reputable account is able to reduce an agent's collateral availability knowing in advance that their request will be rejected.

When a user performs a collateral reservation request, the required agent's collateral to back the positions is locked until the request is resolved. In exchange, the minter pays a reservation fee. By adding the handshake, Agents are now allowed to reject a minting request, which transfers the fee back to the minter.

Minters that know that their request will be rejected can abuse the system to make minting reservation requests that are going to be denied. This way, they lock the agent's collateral for some time. When the minting is rejected, they will get the fee they paid for back, making the cost of the attack negligible. Minters can execute the attack as many time as they want, making the lock permanent.

An attacker willing to exploit this issue can force an address to be denied service by an agent that uses handshake by investigating the properties that an agent uses to decide whether to reject or accept request. It is likely most

handshake users will have certain easily achieved conditions to deny service, for example, denying service to accounts that have interacted with a sanctioned services.

Note that after the agent rejects the collateral reservation, the locked collateral is released and the minter receives the fee back. The attacker would even get an small amount of gas to execute arbitrary actions:

```
function _rejectOrCancelCollateralReservation(
    CollateralReservation.Data storage crt,
    uint64 _crtId
)
private
{
    uint256 totalFee = crt.reservationFeeNatWei +
    crt.executorFeeNatGWei * Conversion.GWEI;

    // guarded against reentrancy in CollateralReservationsFacet
    /* solhint-disable avoid-low-level-calls */
    //slither-disable-next-line arbitrary-send-eth
    (bool success, ) = crt.minter.call{value: totalFee, gas: 100000}
    ("");
    /* solhint-enable avoid-low-level-calls */
    if (!success) {
        // if failed, burn the fee
        Agents.burnDirectNAT(totalFee);
    }

    // release agent's reserved collateral
    releaseCollateralReservation(crt, _crtId); // crt can't be used
    after this
}
```

The opportunity cost of the evil minter can be defined as it follows:

```
opportunityCost = collateral reservation fee * time until rejection +
txGas * gasPrice @ reservation - 100k * gasPrice @ rejection
```

For each agent, this cost can be expressed as:

```
opportunityCost = lockedCollateral * time until rejection + (txGas +
100k) * gasPrice @ rejection
```

It can be seen that the opportunity cost for each agent enforcing a handshake is considerably higher because they:

- Get all the position's collateral locked
- They subsidize up to 100k of gas at the minter's context when performing the rejection

Coinspect considers the likelihood of this issue to be high since any minter is able to get a banned address and trigger evil collateral reservation requests.

The impact is high since the evil minter can repeat this process indefinitely as they receive the reservation fee back.

Recommendation

Rejected mintings should not return the fee to requesters.

To avoid the risk of honest minters being denied service and losing their fee, the system needs to allow agents to commit to a mint operation *before* the collateral is reserved.

The handshake would work like this:

1. Minter provides information on mint operation to agent
2. Agent commits to providing or rejecting mint operation
3. Minter pays collateral reservation fee and reserves the collateral
4. Agent needs to reject or accept according to the previous commitment

If the agent rejects, no fee is returned: the minter had the necessary information to make a correct decision. If the agent accepts, the process continues as normal.

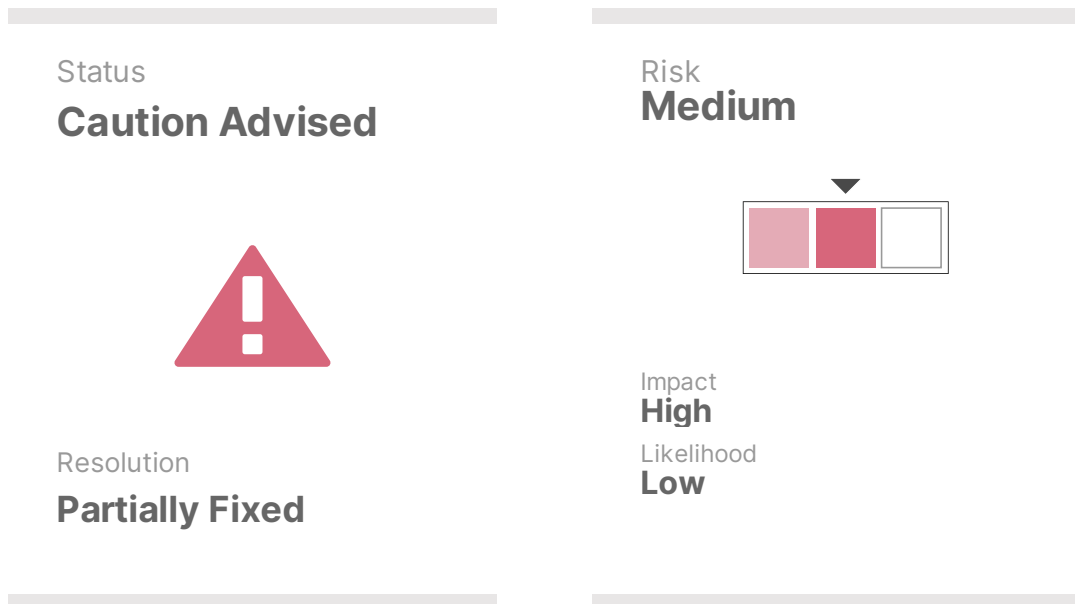
Status

Fixed on commit `9ba17bbc9d414b0ce74dbe3d7d4b68182944a2f2`.

The Flare Team implemented a fix that allows setting a burn percentage of the reservation fee after a rejection. This way, attackers will lose a portion of the reservation fee. Moreover, agents are not incentivized to continuously perform a rejection for no reason as they will not increase their utilization rate, which then translates in more transfer fees received due to FAsset transfers.

FAS-048

Banned redeemers targeting agents with handshake can farm premium fees



Description

Redeemers using a low reputable or banned address are able to target an agent that requires handshake as an attempt to receive more collateral because the system compensates them with a premium. When a user requests a redemption, if an agent rejects the request and no other agent performs the takeover after some time, the redeemer receives the equivalent in collateral plus a premium.

Similarly to **FAS-046**, a user not willing to receive any funds on the underlying chain and wanting to get collateral, might use a banned or blacklistable account. Then, by requesting a redemption with that account or exiting the pool of an agent requiring handshake, they can receive the equivalent of the burned FAssets in collateral plus a premium.

```
bool isRejection = _request.rejectionTimestamp != 0;  
_vaultCollateralWei =  
Conversion.convertAmgToTokenWei(_request.valueAMG,  
cdAgent.amgToTokenWeiPrice)
```

```

        .mulBips(isRejection ?
settings.rejectedRedemptionDefaultFactorVaultCollateralBIPS :
    settings.redemptionDefaultFactorVaultCollateralBIPS);
// calculate paid amount and max available amount from the pool
Collateral.Data memory cdPool =
AgentCollateral.poolCollateralData(_agent);
_poolWei = Conversion.convertAmgToTokenWei(_request.valueAMG,
cdPool.amgToTokenWeiPrice)
    .mulBips(isRejection ?
settings.rejectedRedemptionDefaultFactorPoolBIPS :
    settings.redemptionDefaultFactorPoolBIPS);
uint256 maxPoolWei = cdPool.maxRedemptionCollateral(_agent,
_request.valueAMG);
// if there is not enough collateral held by agent, pay more from the
pool
if (_vaultCollateralWei > maxVaultCollateralWei) {
    uint256 extraPoolAmg =
        _request.valueAMG.mulDivRoundUp(_vaultCollateralWei -
maxVaultCollateralWei, _vaultCollateralWei);
    _poolWei += Conversion.convertAmgToTokenWei(extraPoolAmg,
cdPool.amgToTokenWeiPrice);
    _vaultCollateralWei = maxVaultCollateralWei;
}

```

This operation reduces the collateral ratio of the agent that rejected the request since they are obliged to back the burned FAssets with even more collateral (backed value + premium). Also, the reason that derived into the rejection (e.g. the redeemer interacted with Tornado Cash) could be enough to stop other agents to take that request over, potentially driving the system to a critical collateral ratio if the redeemer decides to repeat this process.

Coinspect considers the likelihood to be low since it assumes that no other agent takes over the request. The impact is high as the redeemer can repeat this process multiple times, stealing premiums and reducing the agents' collateral ratio.

Recommendation

See the fix for issue [FAS-046](#). The same fix would mitigate this risk, as rejected addresses should not receive any premium.

Status

Partially Fixed.

The Flare Team stated:

```

The premium for handshake rejection is a separate setting and is
supposed to be much lower than the premium for

```

ordinary redemption default (currently it is just 0.1% in production parameters, but we may even set it to 0).
We added the option of premium to disincentivize the agents from rejecting valid redemption in order to profit from price fluctuations.

Coinspect considers this issue as partially fixed since it is a decision that requires constant monitoring to ensure that incentives are aligned. A small misalignment between several factors such as the premium values, and current market conditions could make the issue exploitable and adversaries may still take advantage from it.

6. Disclaimer

The contents of this report are provided "as is" without warranty of any kind. Coinspect is not responsible for any consequences of using the information contained herein.

This report represents a point-in-time and time-boxed evaluation conducted within a specific timeframe and scope agreed upon with the client. The assessment's findings and recommendations are based on the information, source code, and systems access provided by the client during the review period.

The assessment's findings should not be considered an exhaustive list of all potential security issues. This report does not cover out-of-scope components that may interact with the analyzed system, nor does it assess the operational security of the organization that developed and deployed the system.

This report does not imply ongoing security monitoring or guaranteeing the current security status of the assessed system. Due to the dynamic nature of information security threats, new vulnerabilities may emerge after the assessment period.

This report should not be considered an endorsement or disapproval of any project or team. It does not provide investment advice and should not be used to make investment decisions.