PHOENIX
SMART SECURITY
BY TANU GUPTA

Flare - FAsset Protocol Audit Report

Version 1.0

Tanu Gupta

November 17, 2025

Flare - FAsset Protocol Audit Report

Tanu Gupta, Code4rena

Nov 17, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

    * [M-2] Vault Collateral deprecation does not compel agent to switch to valid collateral, leading to pool-only liquidation
- Low
    * [L-1] Missing Zero Address Validation
    * [L-2] No Implementation Address Validation in Constructor of `AgentVaultFactory` (Unsafe Initialization)
- Informational
    * [I-1] Unused Custom Errors
    * [I-2] Unbounded array returned in `alwaysAllowedMintersForAgent` function leading to potential gas limit issues
- Gas
    * [G-1] Inefficient Storage Layout for Agent Metadata in `AgentOwnerRegistry` (Redundant Mappings Increase Gas Usage)

Protocol Summary

The FAsset contracts are used to mint assets on top of Flare. The system is designed to handle chains which don't have smart contract capabilities. Initially, FAsset system will support XRP native asset on XRPL. At a later date BTC, DOGE, add tokens from other blockchains will be added.

The minted FAssets are secured by collateral, which is in the form of ERC20 tokens on Flare/Songbird chain and native tokens (FLR/SGB). The collateral is locked in contracts that guarantee that minted tokens can always be redeemed for underlying assets or compensated by collateral. Underlying assets can also be transferred to Core Vault, a vault on the underlying network. When the underlying is on the Core Vault, the agent doesn't need to back it with collateral so they can mint again or decide to withdraw this collateral.

Two novel protocols, available on Flare and Songbird blockchains, enable the FAsset system to operate:

- FTSO contracts which provide decentralized price feeds for multiple tokens.
- Flare's FDC, which bridges payment data from any connected chain.

Disclaimer

I, Tanu Gupta make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by me is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

I use the Code4rena severity matrix to determine severity. See the documentation for more details.

Audit Details

The audit was performed between August 19, 2025 and September 23, 2025. The codebase was viwed in depth and tested using hardhat tests.

The findings correspond to the code at https://github.com/code-423n4/2025-08-flare.

Scope

See scope.txt

Files out of scope

See out_of_scope.txt

Roles

Role	Description
Governance (multi-sig)	controls protocol settings
Agents	provide minting and redeeming services. While Agents undergo KYC, they cannot be considered fully trusted—especially if significant potential gains could incentivize malicious behavior.

Executive Summary

The Flare Asset Manager is a critical component of the Flare network, responsible for managing collateral assets that back the minting of F-Assets. This report identifies several security issues within the Asset Manager contract from medium to low severity.

Issues found

Severity	Number of issues found
High	0
Medium	2
Low	2
Info	2
Gas	1
Total	7

Findings

Medium

[M-1] Minting Allowed With Deprecated Vault Collateral even with paused Minting before Liquidation begins

Description The `FAssets` system allows minting operations to proceed even when collateral tokens have been deprecated and when minting is globally paused. Specifically:

1. **Deprecated Collateral**: With the deprecated vault collateral, *agents/minter/executer* can still mint the stale CRTs as long as the `collateral reservation (CRT)` was created before the deprecation. There is no restriction preventing minting with such CRTs.
2. **Minting Paused State:** The `executeMinting` function can be successfully executed even when minting has been globally paused by governance, as long as the CRT was created before the pause.

Impact

1. **Deprecated Collateral Risk:** Vaults using deprecated collateral should not be able to facilitate new minting operations once the token is invalidated. Allowing this creates systemic risk as the collateral value may become unreliable.
2. **Bypassing Emergency Pauses:** The global minting pause mechanism can be bypassed if users create CRTs before the pause, undermining governance's ability to quickly halt system operations during emergencies.

Proof of Concepts

1. An agent creates a CRT before the deprecation.
2. Governance deprecates the vault collateral token.
3. Governance pauses minting globally.
4. The agent successfully executes minting using the CRT after the collateral has been deprecated.
5. The agent successfully executes minting using the CRT after minting has been paused.

Paste this code snippet in the test file to reproduce the issue.

Proof Of Code

```
it("perform minting with deprecated vault collateral token", async ()
    => {
                const currentSettings =
                    await context.assetManager.getSettings();
```

```
            const agent = await Agent.createTest(
                context,
                agentOwner1,
                underlyingAgent1
            );
            const minter = await Minter.createTest(
                context,
                minterAddress1,
                underlyingMinter1,
                context.underlyingAmount(10000)
            );
            // TRACK INITIAL BALANCES
            // make agent available
            const fullAgentCollateral = toWei(3e8);
            await agent.depositCollateralsAndMakeAvailable(
                fullAgentCollateral,
                fullAgentCollateral
            );
            //update block
            await context.updateUnderlyingBlock();
            // perform minting
            const lots = 100;
            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)
            );
            const agentInfo = await agent.checkAgentInfo({
                totalVaultCollateralWei: fullAgentCollateral,
                freeUnderlyingBalanceUBA: minted.agentFeeUBA,
                mintedUBA: minted.mintedAmountUBA.add(minted.
                    poolFeeUBA),
            });
            //Deprecate collateral token
            await context.assetManagerController.
               deprecateCollateralType(
                [context.assetManager.address],
                2,
                context.usdc.address,
                currentSettings.tokenInvalidationTimeMinSeconds,
                { from: governance }
            );
            //Check if a user can create reservation request now
            await context.updateUnderlyingBlock();
```

```
                    // perform minting
                    const newlots = 100;
                    const newcrt = await minter.reserveCollateral(
                        agent.vaultAddress,
                        newlots
                    );

                    const newtxHash = await minter.performMintingPayment(
                       newcrt);

                    await context.assetManagerController.pauseMinting(
                        [context.assetManager.address],
                        { from: governance }
                    );
                    assert.isTrue(await context.assetManager.mintingPaused
                       ());

                    const newminted = await minter.executeMinting(
                        newcrt,
                        newtxHash
                    );
                    assertWeb3Equal(
                        newminted.mintedAmountUBA,
                        context.convertLotsToUBA(newlots)
                    );
                });

solidity
```

Recommended mitigation

1. Add a check in `executeMinting` to revert if global minting is paused, regardless of when the CRT was created.

```
+ require(state.mintingPausedAt == 0, MintingPaused());
```

2. Add a check in `executeMinting` to revert if the collateral type used has been deprecated.

```
+ require(AssetManagerState.get().collateralTokens[agent.
   vaultCollateralIndex].validUntil == 0, "Vault collateral deprecated"
   );

+ require(AssetManagerState.get().collateralTokens[agent.
   poolCollateralIndex].validUntil == 0, "Pool collateral deprecated");
```

[M-2] Vault Collateral deprecation does not compel agent to switch to valid collateral, leading to pool-only liquidation

Description A vault collateral token can be deprecated by governance, but agents using that collateral are not required to switch to a valid collateral token. As a result, agents can continue operating with deprecated collateral indefinitely.

Consider a scenario where an agent gets liquidated while using a deprecated vault collateral token.

```
function currentLiquidationFactorBIPS(Agent.State storage _agent, uint256 _vaultCR, uint256 _poolCR)
        internal
        view
        returns (uint256 _c1FactorBIPS, uint256 _poolFactorBIPS)
    {
        AssetManagerSettings.Data storage settings = Globals.getSettings();
        uint256 step = _currentLiquidationStep(_agent);
        uint256 factorBIPS = settings.liquidationCollateralFactorBIPS[step];
        _c1FactorBIPS = Math.min(settings.liquidationFactorVaultCollateralBIPS[step], factorBIPS);
        CollateralTypeInt.Data storage vaultCollateral = _agent.getVaultCollateral();
        CollateralTypeInt.Data storage poolCollateral = _agent.getPoolCollateral();
        if (!vaultCollateral.isValid() && poolCollateral.isValid()) {
            // vault collateral invalid - pay everything with pool collateral
@>            _c1FactorBIPS = 0;
        } else if (vaultCollateral.isValid() && !poolCollateral.isValid()) {
            // pool collateral - pay everything with vault collateral
            _c1FactorBIPS = factorBIPS;
        }
        // never exceed CR of tokens
        if (_c1FactorBIPS > _vaultCR) {
            _c1FactorBIPS = _vaultCR;
        }
        _poolFactorBIPS = factorBIPS - _c1FactorBIPS;
        if (_poolFactorBIPS > _poolCR) {
            _poolFactorBIPS = _poolCR;
@>            _c1FactorBIPS = Math.min(factorBIPS - _poolFactorBIPS, _vaultCR);
        }
    }

    function _performLiquidation(Agent.State storage _agent, Liquidation.CRData memory _cr, uint64 _amountAMG)
        private
        returns (uint64 _liquidatedAMG, uint256 _payoutC1Wei, uint256
            _payoutPoolWei)
    {
        (uint256 vaultFactor, uint256 poolFactor) =
@>            LiquidationPaymentStrategy.currentLiquidationFactorBIPS(
    _agent, _cr.vaultCR, _cr.poolCR);
        uint256 maxLiquidatedAMG = Math.max(
            Liquidation.maxLiquidationAmountAMG(_agent, _cr.vaultCR,
                vaultFactor, Collateral.Kind.VAULT),
            Liquidation.maxLiquidationAmountAMG(_agent, _cr.poolCR,
                poolFactor, Collateral.Kind.POOL)
        );
        uint64 amountToLiquidateAMG = Math.min(maxLiquidatedAMG,
            _amountAMG).toUint64();
        (_liquidatedAMG,) = Redemptions.closeTickets(_agent,
            amountToLiquidateAMG, true);
        _payoutC1Wei =
            Conversion.convertAmgToTokenWei(uint256(_liquidatedAMG).
                mulBips(vaultFactor), _cr.amgToC1WeiPrice);
        _payoutPoolWei =
            Conversion.convertAmgToTokenWei(uint256(_liquidatedAMG).
                mulBips(poolFactor), _cr.amgToPoolWeiPrice);
    }
```

`_performLiquidation` calls `currentLiquidationFactorBIPS` to determine how much collateral to liquidate from vault vs pool. With the invalid vault collateral, the function will attempt to liquidate only from the pool collateral. Hence giving the agent an unintended advantage of avoiding liquidation of vault collateral.

If vault and the pool both are undercollateralized, then the vault's liability is reduced to half, hence again giving an unintended advantage to the agent of avoiding full liquidation.

```
function _agentResponsibilityWei(Agent.State storage _agent, uint256
    _amount) private view returns (uint256) {
        if (_agent.status == Agent.Status.FULL_LIQUIDATION || _agent.
            collateralsUnderwater == Agent.LF_VAULT) {
            return _amount;
        } else if (_agent.collateralsUnderwater == Agent.LF_POOL) {
            return 0;
        } else {
@>          return _amount / 2;
        }
    }
```

Impact Agents may strategically avoid switching to a valid collateral token during liquidation, undermining the system's design.

Proof of Concepts

1. An agent creates a CRT before the deprecation.

2. Governance deprecates the vault collateral token.
3. The agent successfully mints using the CRT.
4. Time is advanced to allow liquidation to start.
5. A liquidator starts liquidation on the agent's vault.
6. The liquidation process only utilizes pool collateral, leaving the deprecated vault collateral untouched.

Paste this code snippet in the test file to reproduce the issue.

Proof Of Code

```
it("check if the agent get is not incentivized to not switch deprecated
    collateral", async () => {
    const currentSettings = await context.assetManager.getSettings();
    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);
    await agent.depositCollateralsAndMakeAvailable(
        fullAgentCollateral,
        fullAgentCollateral
    );
    //update block
    await context.updateUnderlyingBlock();
    // perform minting
    const lots = 100;
    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));
    //Deprecate collateral token
    await context.assetManagerController.deprecateCollateralType(
        [context.assetManager.address],
        2,
        context.usdc.address,
        currentSettings.tokenInvalidationTimeMinSeconds,
        { from: governance }
    );

    const collateralType = await context.assetManager.getCollateralType
       (
        2,
        context.usdc.address
    );

    assertWeb3Equal(
        collateralType.validUntil,
        (await time.latest()).add(
            toBN(currentSettings.tokenInvalidationTimeMinSeconds)
        )
    );
    // Should not be able to start liquidation before time passes
    await expectRevert.custom(
        context.assetManager.startLiquidation(agent.agentVault.address,
            {
            from: liquidator.address,
        }),
        "LiquidationNotStarted",
        []
    );
    //Wait until you can switch vault collateral token
    await time.deterministicIncrease(
        currentSettings.tokenInvalidationTimeMinSeconds
    );
    // liquidator "buys" f-assets
    await context.fAsset.transfer(liquidator.address, minted.
       mintedAmountUBA, {
        from: minter.address,
    });
    const tx = await context.assetManager.startLiquidation(
        agent.agentVault.address,
        {
            from: liquidator.address,
        }
    );
    expectEvent(tx, "LiquidationStarted");

    //Perform liquidation
    const liquidateMaxUBA = minted.mintedAmountUBA.divn(lots);

    await liquidator.liquidate(agent, liquidateMaxUBA);

    const res = await context.assetManager.liquidate(
        agent.agentVault.address,
        liquidateMaxUBA,
        {
```

```
            from: liquidator.address,
        }
    );
    let amountPaidFromVault;
    if (res.logs[1].event === "LiquidationPerformed") {
        amountPaidFromVault = (
            res.logs[1].args as any
        ).paidVaultCollateralWei.toString();
    }

    assertWeb3Equal(amountPaidFromVault, 0);
});
```

Recommended mitigation In case the collateral token becomes invalid, liquidation should still require payments from the vault based on the same `vaultFactor` logic, rather than shifting the entire burden to the pool. This ensures agents remain properly incentivized to update their collateral token when governance decisions invalidate one.

Low

[L-1] Missing Zero Address Validation

Description The `setManager` function of `AgentOwnerRegistry` allows governance to update the manager address without validating the input.

```
function setManager(address _manager) external onlyGovernance {
    manager = _manager;
    emit ManagerChanged(_manager);
}
```

If the zero address is mistakenly passed, the manager variable would be set to address(0), potentially locking out all manager-only functionalities.

Impact Setting manager to the zero address could break critical logic or render parts of the contract unusable

Recommended mitigation

```
 function setManager(address _manager) external onlyGovernance {
+    require(_manager != address(0), "Invalid manager address");
     manager = _manager;
     emit ManagerChanged(_manager);
}
```

[L-2] No Implementation Address Validation in Constructor of `AgentVaultFactory` (Unsafe Initialization)

Description In the constructor, the implementation address is assigned directly without any validation:

```
constructor(address _implementation) {
    implementation = _implementation;
}
```

If a zero address or an unintended contract is passed as `_implementation`, future proxy deployments will fail or point to an incorrect implementation.

Impact Deployments could create bricked proxies that cannot function correctly.

Recommended mitigation

```
constructor(address _implementation) {
+    require(_implementation != address(0), "Invalid implementation
    address");
+    require(_implementation.code.length > 0, "Implementation address
    has no code");
     implementation = _implementation;
}
```

Informational

[I-1] Unused Custom Errors

Description Several custom errors are declared in the contract but never actually used in any function logic.

```
//AgentCollateralFacet.sol
error FAssetNotTerminated()
```

Impact Unnecessary bytecode size, potential confusion for future maintainers

Recommended mitigation

```
//AgentCollateralFacet.sol
- error FAssetNotTerminated()
```

[I-2] Unbounded array returned in `alwaysAllowedMintersForAgent` function leading to potential gas limit issues

Description The `alwaysAllowedMintersForAgent` function returns an unbounded array of addresses:

```
function alwaysAllowedMintersForAgent(address _agentVault) external
    view returns (address[] memory) {
        Agent.State storage agent = Agent.get(_agentVault);
        return agent.alwaysAllowedMinters.values();
    }
```

Impact

1. **Denial of Service:** If the alwaysAllowedMinters set grows too large, calling this function may consistently run out of gas, making it unusable for legitimate users and frontends.
2. **High Gas Costs:** Even if it doesn't run out of gas, returning a large array can be very expensive in terms of gas, leading to high costs for users trying to access this data.

Recommended mitigation Implement pagination or size limits to prevent unbounded gas consumption.

```
 function alwaysAllowedMintersForAgent(address _agentVault, uint256
    offset, uint256 limit) external view returns (address[] memory) {
        Agent.State storage agent = Agent.get(_agentVault);
-        return agent.alwaysAllowedMinters.values();
+        uint256 total = agent.alwaysAllowedMinters.length();
+        if (offset >= total) {
+            return new address[](0);
+        }
+        uint256 end = offset + limit;
+        if (end > total) {
+            end = total;
+        }
+        address[] memory result = new address[](end - offset);
+        for (uint256 i = offset; i < end; i++) {
+            result[i - offset] = agent.alwaysAllowedMinters.at(i);
+        }
+        return result;
    }
```

Gas

[G-1] Inefficient Storage Layout for Agent Metadata in `AgentOwnerRegistry` (Redundant Mappings Increase Gas Usage)

Description The contract maintains multiple separate mappings for related agent metadata:

```
mapping(address => string) private agentName;
mapping(address => string) private agentDescription;
mapping(address => string) private agentIconUrl;
mapping(address => string) private agentTouUrl;
```

Each mapping stores data keyed by the same address but in separate storage slots. This design leads to higher gas consumption due to multiple SSTORE and SLOAD operations when updating or retrieving an agent's details. It also makes the code less maintainable and increases storage fragmentation.

Impact Significant gas overhead and poor storage design when managing agent metadata, leading to increased operational costs and complexity.

Recommended mitigation

```
- mapping(address => string) private agentName;
- mapping(address => string) private agentDescription;
- mapping(address => string) private agentIconUrl;
- mapping(address => string) private agentTouUrl;
+ struct Agent {
+     string name;
+     string description;
+     string iconUrl;
+     string touUrl;
+ }

+ mapping(address => Agent) private agents;
```