coinspect

You build, we defend.

flare

Smart Contract Audit

FAssets Update

Sept, 2024

FAsset Update
Smart Contract Audit

Version: v240910 Prepared for: Flare September 2024

Security Assessment

1. Executive Summary
2. Summary of Findings
    2.3 Solved issues & recommendations
3. Scope
4. Assessment
    4.1 Security assumptions
    4.2 Decentralization
    4.3 Testing
    4.4 Code quality
5. Detailed Findings

FAS-38 - Rogue agents can delay underlying chain payments by inflating deadlines

FAS-39 - Agents can bypass emergency pause by redeeming against themselves

6. Disclaimer

1. Executive Summary

In August 2024, Flare engaged Coinspect to review a set of updates and the introduction of new features for the **FAsset Protocol** smart contracts.

The **FAsset Protocol** is a collateralized bridge solution that enables cross-chain native token transfers. The objective of the project was to evaluate the security of those changes and how they integrate with the previously reviewed system.

✓ Solved	Caution Advised	✗ Resolution Pending
High 0	High 0	High 0
Medium 0	Medium 0	Medium 0
Low 1	Low 0	Low 0
No Risk 1	No Risk 0	No Risk 0
Total **2**	Total **0**	Total **0**

During this assessment, Coinspect identified two issues, a medium-risk and a low-risk issue. `FAS-38` shows how rogue agents can inflate payment times in the underlying chain and `FAS-39` illustrates a scenario where agents can still make redemptions during an emergency pause.

2. Summary of Findings

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

2.3 Solved issues & recommendations

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

Id	Title	Risk
FAS-39	Agents can bypass emergency pause by redeeming against themselves	Low
FAS-38	Rogue agents can delay underlying chain payments by inflating deadlines	None

3. Scope

The source code review of `Flare FAssets` started on August 19, 2024, and was conducted on the `fasset-v2-audit-4` branch of the git repository located at https://gitlab.com/flarenetwork/fasset as of commit `997fac606b93f2fac0176f1243f455da44041d4f`.

The previous report of the **FAsset Protocol v240529** covers the review of the protocol's migration to the Diamond Proxy infrastructure.

4. Assessment

The new code includes several changes and features from the previous review:

- A new facet allowing anyone to ping an agent to check if they are live. This functionality allows users to emit an event pinging a specific agent, who then responds to the liveliness check.
- Congested scenarios where multiple redemptions are requested to a single agent, extend the payment deadline on the underlying chain. This gives agents more time to complete all payments when they are experiencing a high demand.
- The collateral pool token address is now included as part of the data retrieved when querying an Agent's information as well as during their creation.
- An emergency pause feature restricting mint, redeem and liquidate was added. This is only callable by the governance and previously allowed accounts. The governance has no restrictions on the maximum pause time, whereas the duration of an emergency triggered by an allowed account is capped with a maximum value.
- In the context of redemption failures, `finishRedemptionWithoutPayment` does not delete redemption request anymore since there is no certainty if proofs have expired. The redemption is set as defaulted at the end of the call. Also, for payment confirmations `_cleanupPaymentVerification` was removed and `_recordPaymentVerification` does not cleanup payment hashes older than 14 days. The Flare Team stated as an additional precaution measure they decided to leave proofs of payments in the database without cleaning them up.
- Collateral Pools, Agent Vaults and Collateral Pool Tokens now include an initialize method. In spite of providing factories for each one of those contracts, it is strongly recommended to document the risks of not making their deployments along with an atomic initialization (e.g. if the factory is bypassed, adversaries could initialize those contracts first).

Regarding the **FAsset Token**:

- Migrated to an upgradeable architecture
- Vote power functionality was removed since FTSOv2 no longer consumes it
- The Flare Team stated that FAsset token will include permit capabilities, however, Coinspect identified that no permit feature is implemented.

4.1 Security assumptions

Coinspect considers that the following aspects should be respected across the life expectancy of the protocol:

- It is possible to create storage collisions across multiple FAsset versions on purpose. This could be weaponized to rollback the terminate status of a FAsset Token. New versions of this token should preserve its storage layout.
- Future **FAsset** implementations should always include the `assetManager()` public variable/function respecting the first implementation's slot. Without this, the access control mechanism of the **FAssetProxy** could not work as expected.

4.2 Decentralization

The new emergency pause feature allows either the governance or a specific set of allowed accounts to temporarily pause core functionalities of the protocol. Besides the governance, these sole accounts should have a secure structure to prevent take-overs (e.g. multisig with a timelock).

Additionally, since the FAsset Token is now upgradeable, users should be aware of upcoming new changes. Upgrades to this token should be properly announced and executed with a time-lock so users have enough time to react to these changes.

4.3 Testing

Coinspect identified that the new features and functionalities were included into the testing suite. Thanks to this, testing adversarial scenarios was quicker requiring less setup steps.

4.4 Code quality

The newly introduced code includes relevant comments and NatSpec, also respecting the protocol's code style. Because of this, the code was easy to read and understand.

5. Detailed Findings

FAS-38

Rogue agents can delay underlying chain payments by inflating deadlines

Status	Risk
Solved	**None**
	Impact
	Recommendation
Resolution	Likelihood
Acknowledged	–

Location

```
./contracts/assetManager/library/RedemptionRequests.sol:73
```

Description

Rogue agents can create multiple low-amount redemption tickets against themselves to inflate their payment deadlines by abusing from the new time extension mechanism.

This new feature that intends to increase the time-window for congested/demanded agents could be weaponized against the protocol by malicious agents, by triggering multiple redemptions directly using `redeemFromAgent()`:

```
    /**
     * Create a redemption from a single agent. Used in self-close exit
from the collateral pool.
     * Note: only collateral pool can call this method.
     */
    function redeemFromAgent(
        address _agentVault,
        address _receiver,
        uint256 _amountUBA,
        string memory _receiverUnderlyingAddress,
        address payable _executor
    )
        external payable
        notEmergencyPaused
    {
        RedemptionRequests.redeemFromAgent(_agentVault, _receiver,
_amountUBA, _receiverUnderlyingAddress, _executor);
    }
```

```
    function redeemFromAgent(
        address _agentVault,
        address _redeemer,
        uint256 _amountUBA,
        string memory _receiverUnderlyingAddress,
        address payable _executor
    )
        internal
    {
        Agent.State storage agent = Agent.get(_agentVault);
        Agents.requireCollateralPool(agent);
        require(_amountUBA != 0, "redemption of 0");
        // close redemption tickets
        uint64 amountAMG = Conversion.convertUBAToAmg(_amountUBA);
        (uint64 closedAMG, uint256 closedUBA) =
Redemptions.closeTickets(agent, amountAMG, false);
        // create redemption request
        AgentRedemptionData memory redemption =
AgentRedemptionData(_agentVault, closedAMG);
        _createRedemptionRequest(redemption, _redeemer,
_receiverUnderlyingAddress, true,
            _executor, (msg.value / Conversion.GWEI).toUint64());
        // burn the closed assets
        Redemptions.burnFAssets(msg.sender, closedUBA);
    }
```

This function creates a redemption request, which contains the last underlying block and timestamp at which the agent can make the payment on the underlying chain.

```
(request.lastUnderlyingBlock, request.lastUnderlyingTimestamp) =
_lastPaymentBlock(_data.agentVault);
```

Both last block and timestamp depend on a timeshift that is extended when the agent is experiencing high demand:

```
        // timeshift amortizes for the time that passed from the last
underlying block update;
        // it also adds redemption time extension when there are many
redemption requests in short time
        uint64 timeshift = block.timestamp.toUint64() -
state.currentUnderlyingBlockUpdatedAt
            +
RedemptionTimeExtension.extendTimeForRedemption(_agentVault);
```

```
    function extendTimeForRedemption(address _agentVault)
        internal
        returns (uint64)
    {
        State storage state = getState();
        AgentTimeExtensionData storage agentData =
state.agents[_agentVault];
        uint64 timestamp = block.timestamp.toUint64();
        uint64 accumulatedTimestamp = agentData.extendedTimestamp +
state.redemptionPaymentExtensionSeconds;
        agentData.extendedTimestamp =
SafeMath64.max64(accumulatedTimestamp, timestamp);
        return agentData.extendedTimestamp - timestamp;
    }
```

By creating multiple subsequent redemptions to the same agent, the `extendedTimestamp` grows, inflating the `timeshift` as a consequence, which also increases both `lastUnderlyingBlock` and `lastUnderlyingTimestamp`.

This process is not profitable for redemption requests made through `redeem()`, because a minimum redemption size is enforced as redeemers provide the `lots` to burn, instead of the `amountUBA`.

The likelihood of this issue is considered high since agents could inflate the payment time with no further efforts. Regarding the impact, it is considered to be medium as it depends on external conditions such as the value of `redemptionPaymentExtensionSeconds` and market conditions in order to make this attack profitable.

Recommendation

To reduce the profitability of the payment time inflation, enforce a minimum amount to redeem in `redeemFromAgent()`.

Status

Acknowledged.

The Flare Team stated that this path is not directly exploiteable since every redemption request is handled by Collateral Pools that only call `redeemFromAgent` if the amount to redeem is greater than one lot. However, Coinspect decided to leave this issue as informational since adding new paths in the future that call the redemption facet, unaware of this scenario, would enable the attack shown below.

Proof of Concept

The following scenario shows how it is possible to inflate the redemption payment deadline by creating multiple redemption requests providing a low `amountUBA`. The scenario considers a 40 second extension per redemption request.

```
0 - Request payment last timestamp: 1724159385
1 - Request payment last timestamp: 1724159425
2 - Request payment last timestamp: 1724159465
3 - Request payment last timestamp: 1724159505
4 - Request payment last timestamp: 1724159545
5 - Request payment last timestamp: 1724159585
6 - Request payment last timestamp: 1724159625
7 - Request payment last timestamp: 1724159665
8 - Request payment last timestamp: 1724159705
9 - Request payment last timestamp: 1724159745
10 - Request payment last timestamp: 1724159785
11 - Request payment last timestamp: 1724159825
12 - Request payment last timestamp: 1724159865
13 - Request payment last timestamp: 1724159905
14 - Request payment last timestamp: 1724159945
15 - Request payment last timestamp: 1724159985
16 - Request payment last timestamp: 1724160025
17 - Request payment last timestamp: 1724160065
18 - Request payment last timestamp: 1724160105
19 - Request payment last timestamp: 1724160145
```

```
    it("Coinspect - Inflates payment timestamp", async () => {
      // init
      const agentVault = await createAgent(agentOwner1,
underlyingAgent1);
      await depositAndMakeAgentAvailable(agentVault, agentOwner1);
      collateralPool = await CollateralPool.at(await
assetManager.getCollateralPool(agentVault.address));

      // Assume that the payment extension is 40s
      await assetManager.setRedemptionPaymentExtensionSeconds(40, { from:
governance });
      const resSettings = web3ResultStruct(await
assetManager.getSettings());
      (resSettings as
AssetManagerInitSettings).redemptionPaymentExtensionSeconds =
      await assetManager.redemptionPaymentExtensionSeconds();

assert.equal(resSettings.redemptionPaymentExtensionSeconds.toNumber(),
40);

    // Make 20 redemption requests to a specific agent of a small
amount
    const request = await mintAndRedeemFromAgentCustomAmt(
      agentVault,
      collateralPool.address,
      chain,
      underlyingMinter1,
      minterAddress1,
      underlyingRedeemer1,
      redeemerAddress1,
      true,
      1, // Custom Amount
      20 // Amount of subsequent redemptions
    );
  });
```

Where `mintAndRedeemFromAgentCustomAmt()` is a modified version of `mintAndRedeemFromAgent()` that allows specifying the redemption amount and times:

```
async function mintAndRedeemFromAgentCustomAmt(
    agentVault: AgentVaultInstance,
    collateralPool: string,
    chain: MockChain,
    underlyingMinterAddress: string,
    minterAddress: string,
    underlyingRedeemerAddress: string,
    redeemerAddress: string,
    updateBlock: boolean,
    redeemCustomAmt: Number,
    requestAmount: number
  ) {
    // minter
    chain.mint(underlyingMinterAddress, toBNExp(10000, 18));
    if (updateBlock) await updateUnderlyingBlock();
    // perform minting
    const lots = 3;
    const agentInfo = await
assetManager.getAgentInfo(agentVault.address);
    const crFee = await assetManager.collateralReservationFee(lots);
    const resAg = await assetManager.reserveCollateral(
      agentVault.address,
      lots,
      agentInfo.feeBIPS,
      constants.ZERO_ADDRESS,
      { from: minterAddress, value: crFee }
    );
    const crt = requiredEventArgs(resAg, "CollateralReserved");
    const paymentAmount = crt.valueUBA.add(crt.feeUBA);
    const txHash = await wallet.addTransaction(
      underlyingMinterAddress,
      crt.paymentAddress,
      paymentAmount,
      crt.paymentReference
    );
    const proof = await attestationProvider.provePayment(txHash,
underlyingMinterAddress, crt.paymentAddress);
    const res = await assetManager.executeMinting(proof,
crt.collateralReservationId, {
      from: minterAddress,
    });
    const minted = requiredEventArgs(res, "MintingExecuted");
    // redeemer "buys" f-assets
    await fAsset.transfer(redeemerAddress, minted.mintedAmountUBA, {
from: minterAddress });

    let lastRequest;
    let redemptionRequests;
    let request;
    for (let index = 0; index < requestAmount; index++) {
      // redemption request
      await impersonateContract(collateralPool,
toBN(512526332000000000), accounts[0]);
      lastRequest = await assetManager.redeemFromAgent (
        agentVault.address,
        redeemerAddress,
        redeemCustomAmt,
        underlyingRedeemerAddress,
        executorAddress1,
        { from: collateralPool, value: executorFee }
      );
      redemptionRequests = filterEvents(lastRequest,
"RedemptionRequested").map((e) => e.args);
      request = redemptionRequests[0];
      console.log(`${index} - Request payment last timestamp:
${request.lastUnderlyingTimestamp.toNumber()}`);
    }

    await stopImpersonatingContract(collateralPool);
    return request;
  }
```

FAS-39

Agents can bypass emergency pause by redeeming against themselves

Status
Solved

✓

Resolution
Fixed

Risk
Low

🟥⬜⬜

Impact
Medium

Likelihood
Low

Location

`./contracts/assetManager/facets/RedemptionRequestsFacet.sol:119`

Description

Agents could abuse from an emergency pause state to alter their collateral by self-closing a position they own.

The new `notEmergencyPaused` modifier is not applied to the following function, allowing agents to process self-redemptions even when the protocol is under an emergency state.

```
    function selfClose(
        address _agentVault,
        uint256 _amountUBA
    )
        external
        returns (uint256 _closedAmountUBA)
    {
        // in SelfClose.selfClose we check that only agent can do this
        return RedemptionRequests.selfClose(_agentVault, _amountUBA);
    }
```

Additionally, the minting process restricts an agent from self-minting when the protocol is emergency paused:

```
    function selfMint(
        Payment.Proof calldata _payment,
        address _agentVault,
        uint256 _lots
    )
        external
        onlyAttached
        notEmergencyPaused
        nonReentrant
    {
        Minting.selfMint(_payment, _agentVault, _lots.toUint64());
    }
```

In relationship to the previous item, users that managed to make a collateral reservation before an emergency pause will be able to execute their minting position:

```
    function executeMinting(
        Payment.Proof calldata _payment,
        uint256 _collateralReservationId
    )
        external
        nonReentrant
    {
        Minting.executeMinting(_payment,
_collateralReservationId.toUint64());
    }
```

This action generates an amount of **FAssets** even if the protocol is experiencing an emergency pause. However, restricting the minting execution could derive in unfair challenges or reservation fee collection by providing the non-payment proof.

Recommendation

Restrict agents from self-closing positions when there is an emergency pause. Additionally, evaluate if all means of generation/consumption of **FAssets** should be restricted during an emergency pause.

Status

Fixed on commit `2a2cad73ea9eca9f242359ff3dadd0b1437f15aa`.

The `notEmergencyPaused` modifier was added to `selfClose` restricting redemptions against themselves.

6. Disclaimer

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

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

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

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

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