cOinspect

You build, we defend.

flare

Smart Contract Audit

FAsset Core Vault

April, 2025

FAsset V1.1 Core Vault
Smart Contract Audit

Version: v250506 Prepared for: Flare April 2025

Security Assessment

1. Executive Summary
2. Summary of Findings
   - 2.2 Finding where caution is advised
   - 2.3 Solved issues & recommendations
3. Scope
4. Assessment
   - 4.1 Core Vault Analysis
   - 4.2 Security assumptions
   - 4.3 Decentralization Analysis
5. Detailed Findings

FAS-049 - Agents risk paying confirmation rewards due to CoreVault downtime

FAS-050 - Agents can mistakenly overpay fees

FAS-051 - Violation of checks-effects-interaction pattern

FAS-052 - Unbounded loop when processing requests could deny executing core vault operations

FAS-053 - Users are unprotected against missing or malformed vault redemption payments

FAS-054 - Core vault transfers force agents to get close to insolvency

FAS-055 - Escrows finalized close to expiry time decouple the Core Vault internal accountancy

FAS-056 - Large redemptions from core vault can be denied by smaller ones

FAS-057 - Array of escrows always grows

FASO-045 - Partially hardcoded API response can trigger wrong actions

FASO-046 - Replacement transactions might not go through due to insufficient funds

FASO-047 - Wrong notifier event

6. Disclaimer

1. Executive Summary

In **March, 2025**, Flare engaged Coinspect to perform a Smart Contract Audit of FAsset V1.1 Core Vault. The objective of the project was to evaluate the security of the smart contracts and off-chain services that support the Core Vault feature.

Core Vault provides a centralized liquidity reserve in the FAsset system, allowing agents to unlock collateral by depositing underlying Fassets. It improves scalability and stability by easing collateral constraints and supporting large-value user redemptions.

Solved	Caution Advised	Resolution Pending
High 0	High 0	High 0
Medium 0	Medium 3	Medium 0
Low 3	Low 0	Low 0
No Risk 6	No Risk 0	No Risk 0
Total **9**	Total **3**	Total **0**

In this security assessment Coinspect identified issues related to the following topics:

- Maximum transfer to vault limits and how it affects the overall protocol's health
- Escrow management and creation process
- Different paths and mechanisms that could temporarily or permanently halt the vault operations

2. Summary of Findings

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

2.2 Finding where caution is advised

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

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

Id	Title	Risk
FAS-053	Users are unprotected against missing or malformed vault redemption payments	Medium
FAS-055	Escrows finalized close to expiry time decouple the Core Vault internal accountancy	Medium
FAS-056	Large redemptions from core vault can be denied by smaller ones	Medium

2.3 Solved issues & recommendations

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

Id	Title	Risk
FAS-049	Agents risk paying confirmation rewards due to CoreVault downtime	Low

FAS-050	Agents can mistakenly overpay fees	Low
FASO-045	Partially hardcoded API response can trigger wrong actions	Low
FAS-051	Violation of checks-effects-interaction pattern	None
FAS-052	Unbounded loop when processing requests could deny executing core vault operations	None
FAS-054	Core vault transfers force agents to get close to insolvency	None
FAS-057	Array of escrows always grows	None
FASO-046	Replacement transactions might not go through due to insufficient funds	None
FASO-047	Wrong notifier event	None

3. Scope

The scope was set to be:

- The branch `fasset-core-vault-audit` at repository https://gitlab.com/flarenetwork/fasset at commit `2d3ac4e2a93033aa6585b8dc9965a01f1ac79f4c`.
- The repository https://gitlab.com/flarenetwork/fassets/fasset-bots at commit `1520e50a565f965d6dc810afb41359c40179110e`

4. Assessment

The Core Vault is a new liquidity management component in the FAssets v1.1 protocol, introduced to alleviate collateral bottlenecks and improve agent profitability. It is implemented as a multi-signature-controlled address on the XRP Ledger, designed to securely hold and manage underlying assets (e.g., XRP) that back fXRP minted on Flare. Hence, the implementation reviewed only supports the XRP chain.

The Core Vault limits risk with a strict release schedule: agent withdrawals, user redemptions, and escrow operations are authorized only once per day within a fixed signing window. Time-locked escrows minimize fund exposure and protect against signer compromise.

Funds held in the vault are used to facilitate two main services:

- Unlocking or re-balancing agent collateral on Flare when agents deposit or withdraw underlying assets.
- Servicing large user redemptions directly, bypassing the need to find an individual agent with sufficient liquidity.

Security is enforced through daily expiring and deterministic escrow flows, governance-set parameters, and an emergency fall-back to a secure custodian wallet in case of anomalies being detected. It is worth mentioning that Coinspect did not observe mechanisms to detect signs of compromise or anomalies.

4.1 Core Vault Analysis

4.1.1 Fund Inflows and Outflows

The following analysis outlines the defined mechanisms by which funds enter and exit the Core Vault, based on protocol specifications.

Inflows

1. **Agent Deposits to Unlock Collateral**: Agents may transfer underlying assets (e.g., XRP) to the Core Vault's predefined address on the underlying chain.

Once the system receives and validates the payment, the agent's collateral on Flare is unlocked. This flow is initiated through a manual agent request and supports further minting or reallocation of collateral.

2. **Daily Escrow Expiry**: Each day, one escrow expires and releases a predefined lot size of XRP (denoted as `L`) back into the Core Vault. This scheduled inflow is part of the vault's risk management model, ensuring controlled replenishment of liquidity.

Outflows

1. **Agent Withdrawals by Locking Collateral**: Agents may request a withdrawal of XRP from the Core Vault by locking additional collateral on Flare. These operations are authorized and executed by Core Vault multi-signers as explained below.

2. **Direct Redemption by Whitelisted Users**: Whitelisted users who meet KYC requirements may request to redeem their FAssets (fXRP) directly from the Core Vault. After burning the FAssets, the redemption is processed and paid out during the vault's daily signing window. These redemptions are subject to longer latency compared to agent redemptions.

3. **Escrow Creation from Excess Funds**: Each time the `triggerInstructions` function from the `CoreVaultManager` contract is called to process payments, the vault evaluates remaining liquidity. Any amount above the minimum reserve threshold (`M`) is split into new escrow accounts in fixed lot sizes (`L`). The escrow system locks funds with release dates structured sequentially one day apart, reducing exposure in the event of a compromise. The intended flow is for those escrows to expire, making the funds available again to the Core Vault for payments if needed. Any unused funds are then returned to escrow.

4. **Emergency Escrow Releases to Custodian**: In exceptional circumstances, such as a detected security incident, the Flare Foundation may authorize escrowed funds to be released to a predefined custodian address. This bypasses normal expiration logic and is executed under strict governance control.

4.1.2 Interactions with Bots

The Core Vault Manager smart contract emits important events that are consumed by the offchain bots. This system, triggers actions on each agent's bot allowing to process payments and submit the respective proofs to the smart contracts.

Coinspect identified that the `TransferToCoreVaultSuccessful` event is not handled by any bot, but the process relies on the legacy events of the FAsset system related to redemptions to update their bookkeeping.

4.1.3 Interactions with the Multisig's Backend

Core actions such as payment requests and escrow creations emit an event after the governance executes the `triggerInstructions` function at the `CoreVaultManager` contract. This function processes payments and handles balance surplus to manage the escrow creation process. Then, the multisig's backend fetches these events and triggers the signing process. After collecting enough signatures, the multisig creates the instructed escrows and makes the payments.

4.2 Security assumptions

For this security assessment, Coinspect made the following assumptions:

- Flare governance does not have malicious intent.
- Core Vault (CV) actors do not collude.
- The agents' underlying address is immutable. Changing this assumption will introduce Denial-of-Service risks in Core Vault.
- Agents will not be able to share the underlying address, as it enables Denial-of-Service scenarios in Core Vault.
- Agents are strictly limited to a single active transfer to or from the Core Vault at any given time.
- Core Vault multisig members are not compromised and the threshold chosen for this multisig is sufficient.

4.3 Decentralization Analysis

The FAsset Core Vault system enhances system liquidity using a federated control model, operated by a multi-signature group of trusted entities with Flare Foundation oversight. Its functionality relies on transaction signing based on smart contract events, introducing dependencies on these operators; direct user redemptions are additionally permissioned via KYC.

Associated counter-party and operational risks inherent to this semi-centralized design are mitigated through security measures like time-locked XRP escrows imposing daily capital limits.

5. Detailed Findings

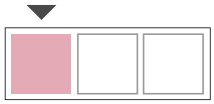
FAS-049

Agents risk paying confirmation rewards due to CoreVault downtime

Status
Solved

Resolution
Fixed

Risk
Low



Impact
Low

Likelihood
Low

Location

`fasset/contracts/assetManager/library/RedemptionConfirmations.sol`
`fasset/contracts/assetManager/library/CoreVault.sol`

Description

Agents may be forced to pay for third-party payment confirmations if the `CoreVault` contract is re-enabled and they have pending payment proofs.

Currently, once an agent calls `transferToCoreVault` to create a transfer request, they must submit payment proof via

`RedemptionConfirmations.confirmRedemptionPayment`, which internally invokes `CoreVault.confirmTransferToCoreVault()`. If `CoreVault` is disabled between the request and the proof submission, the confirmation will fail due to the `onlyEnabled` modifier:

```
function confirmTransferToCoreVault(
    IPayment.Proof calldata _payment,
    Agent.State storage _agent,
    uint64 _redemptionRequestId
) internal onlyEnabled
...
```

This not only leaves the agent's collateral locked until `CoreVault` is re-enabled, but also exposes the agent to the risk of paying a third-party confirmation reward if the re-enablement occurs after `confirmationByOthersAfterSeconds` of the transfer request creation:

```
if (!isAgent) {
    Agents.payForConfirmationByOthers(agent, msg.sender);
}
```

Coinspect considers the likelihood to be low since it would require specific timing with the `CoreVault` being disabled during the confirmation window.

Recommendation

Allow agents a grace period to submit payment confirmations after `CoreVault` is re-enabled, before accepting submissions from third parties. Also, since submitting payment proofs—whether valid or invalid—serves the agents' interests by unlocking their collateral, another approach would be to restrict third parties from submitting these proofs.

Status

Fixed in commit **fba4f5b6e2227eaa195d4272c9875783f8dd180c**. The Core Vault contract can no longer be disabled since, once the Core Vault manager address is set, it cannot be reset to `address(0)`.

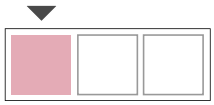
FAS-050

Agents can mistakenly overpay fees

Status
Solved

Resolution
Fixed

Risk
Low



Impact
Low

Likelihood
Low

Location

`fasset/contracts/assetManager/library/CoreVault.sol:80`

Description

When an agent creates a transfer to vault request, they could lose the fees provided in native assets if they mistakenly send more than required. As a consequence, the fee recipient gets more fees than required and the Agent has no means to recover them.

As shown in the snippet below, the function expects to receive *at least* the transfer fee.

```
require(msg.value >= transferFeeWei, "transfer fee payment too small");
```

Which is then entirely transferred to the fee recipient.

```
// pay the transfer fee
Transfers.transferNAT(state.nativeAddress, msg.value);  // guarded by
```

```
nonReentrant in the facet
```

Recommendation

Limit the maximum value of `msg.value`.

Status

Fixed in commit **dd5baf741540569968ce2b0fb6628a8eb0f818a0**. Funds exceeding the required fee are returned to the user. However, despite the reentrancy guard protection, Coinspect recommends moving the refund to the end of the function to follow the checks-effects-interactions pattern.

FAS-051

Violation of checks-effects-interaction pattern

Status
Solved

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
–

Location

`fasset/contracts/assetManager/library/Redemptions.sol (f75ddaa)`

Description

The `payOrBurnExecutorFee` function transfers NAT tokens to the executor address (`msg.sender`) before resetting the `executorFeeNatGWei` request state variable. This order of operations violates the checks-effects-interactions pattern, which dictates that state changes should occur before external calls to mitigate reentrancy risks.

```
function payOrBurnExecutorFee(
    Redemption.Request storage _request
)
    internal
{
    if (_request.executorFeeNatGWei == 0) return;
    if (msg.sender == _request.executor) {
        Transfers.transferNAT(_request.executor,
_request.executorFeeNatGWei * Conversion.GWEI);
```

```
    } else if (_request.executorFeeNatGWei > 0) {
        Agents.burnDirectNAT(_request.executorFeeNatGWei *
Conversion.GWEI);
    }
    _request.executorFeeNatGWei = 0;
}
```

Note that further analysis by Coinspect concluded that this violation does not pose an actual risk, since all calling functions implement appropriate reentrancy safeguards.

Recommendation

Consider following the checks-effects-interactions pattern by setting `executorFeeNatGWei` to zero prior to the token transfer.

Status

Fixed in commit **f75ddaa324690c2dd637d06c8b30a776bf0107c5**. The `executorFeeNatGWei` variable is now zeroed before paying this fee to the executor.

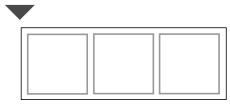
FAS-052

Unbounded loop when processing requests could deny executing core vault operations

Status
Solved

Resolution
Acknowledged

Risk
None



Impact
Recommendation

Likelihood
–

Location

`fasset/contracts/assetManager/implementation/CoreVaultManager.sol:176`

Description

The for loop below will run out of gas if too many `cancelableTransferRequests` are created.

```
for (uint256 i = 0; i < cancelableTransferRequests.length; i++) {
    TransferRequest storage req =
transferRequestById[cancelableTransferRequests[i]];
    require(keccak256(bytes(req.destinationAddress)) !=
destinationAddressHash, "request already exists");
}
```

However, Coinspect understands this is not possible due to:

1. The controlled number of agents.
2. The inability for agents to modify their underlying address.

Shall any of these conditions change, there is risk of Denial-of-Service preventing redemptions from the Core Vault.

Recommendation

Document the conditions and system assumptions that allow all the unbound loops at `CoreVaultManager` not running out of gas.

Status

Acknowledged. As documented in commit **30ac5b2f1abf7b80351a169dac01f950146edb3c**, the number of requests depend directly on the number of agents, which is limited by the governance.

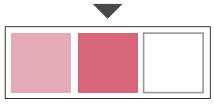
FAS-053

Users are unprotected against missing or malformed vault redemption payments

Status
Caution Advised

Resolution
Acknowledged

Risk
Medium



Impact
High

Likelihood
Low

Location

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

Description

Users requesting redemptions from the Core Vault have no way to challenge payments that are incorrect or not made—such as underpayments. This forces users to place a high degree of trust in the multisig operators.

The root cause is the `requestReturnFromCoreVault` function, which lacks logic to create a redemption request object. Without this, the system cannot validate the payment or trigger a default, as would normally be handled by `RedemptionConfirmations.confirmRedemptionPayment`.

```
Redemption.Request storage request =
Redemptions.getRedemptionRequest(_redemptionRequestId);
...
```

```
(bool paymentValid, string memory failureReason) =
_validatePayment(request, _payment);
```

Although the specification notes that these operations have low priority and may take longer than standard redemptions, the following scenario illustrates the problem:

- A user initiates a redemption, and their fAssets are immediately burned.
- Days later, due to a payment issue, no underlying is transferred from the Core Vault to the user.
- The user has no recourse to dispute or default the non-payment.

Recommendation

Enable users to challenge or default invalid payments from Core Vault redemptions in order to get the burned fAssets back on the Flare chain.

Status

Acknowledged. The Flare team stated that users are expected to fully trust the Core Vault by design.

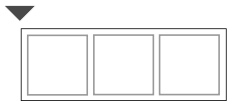
FAS-054

Core vault transfers force agents to get close to insolvency

Status
Solved

Resolution
Acknowledged

Risk
None



Impact
Recommendation

Likelihood
–

Location

`fasset/contracts/assetManager/library/CoreVault.sol:273`

Description

Agents attempting a transfer to the Core Vault must lower their collateral ratio to increase the maximum transferrable amount when `minimumAmountLeftBIPS` is set above zero. Consequently, if several agents perform this operation simultaneously, the system's overall health ratio may drop to a borderline-critical level, approaching liquidation.

When tranferring to the Core Vault, Agents can send up to a maximum value:

```
(uint256 maximumTransferAMG,) =
getMaximumTransferToCoreVaultAMG(_agent);
require(transferredAMG <= maximumTransferAMG, "too little minting left
after transfer");
```

This constraint depends on the collateral they have:

```
function getMaximumTransferToCoreVaultAMG(
    Agent.State storage _agent
)
    internal view
    returns (uint256 _maximumTransferAMG, uint256
_minimumLeftAmountAMG)
{
    _minimumLeftAmountAMG = _minimumRemainingAfterTransferAMG(_agent);
    _maximumTransferAMG = MathUtils.subOrZero(_agent.mintedAMG,
_minimumLeftAmountAMG);
}
```

This calculation is made estimating the maximum supported asset minting granularity for a given amount of collateral. And then, the minimum value between the pool collateral, agent collateral and agent pool tokens is used as the subtracting member to determine `_maximumTransferAMG`.

```
function _minimumRemainingAfterTransferAMG(
    Agent.State storage _agent
)
    private view
    returns (uint256)
{
    Collateral.CombinedData memory cd =
AgentCollateral.combinedData(_agent);
    uint256 resultWRTVault =
_minimumRemainingAfterTransferForCollateralAMG(_agent,
cd.agentCollateral);
    uint256 resultWRTPool =
_minimumRemainingAfterTransferForCollateralAMG(_agent,
cd.poolCollateral);
    uint256 resultWRTAgentPT =
_minimumRemainingAfterTransferForCollateralAMG(_agent,
cd.agentPoolTokens);
    return Math.min(resultWRTVault, Math.min(resultWRTPool,
resultWRTAgentPT));
}
```

```
function _minimumRemainingAfterTransferForCollateralAMG(
    Agent.State storage _agent,
    Collateral.Data memory _data
)
    private view
    returns (uint256)
{
    State storage state = getState();
    (, uint256 systemMinCrBIPS) =
AgentCollateral.mintingMinCollateralRatio(_agent, _data.kind);
    uint256 collateralEquivAMG =
Conversion.convertTokenWeiToAMG(_data.fullCollateral,
_data.amgToTokenWeiPrice);
    uint256 maxSupportedAMG =
collateralEquivAMG.mulDiv(SafePct.MAX_BIPS, systemMinCrBIPS);
```

```
        return maxSupportedAMG.mulBips(state.minimumAmountLeftBIPS);
    }
```

Consider the following scenario:

- Agent A, has the equivalent value of $1000 in each one of the three collateral types.
- Agent B, has $500.

Both agents are exposed to the same minting of $1200.

Then the calculation for Agent A returns a smaller value than Agent's B.

This issue assumes that `minimumAmountLeftBIPS` is non-zero, which yields a non-zero result when calling `_minimumRemainingAfterTransferForCollateralAMG`.

Recommendation

Improve the maximum boundary calculation to reduce the incentive of lowering the collateral ratio of an agent.

Status

Acknowledged. Although still possible, scenarios where users must decollateralize due to excess collateral are unlikely. The Flare team noted that this limitation could still be circumvented by transferring to the Core Vault multiple times.

Proof of Concept

The following test demonstrates that an agent with higher collateral is subject to a lower maximum transferable amount compared to one with less collateral. For this test, `minimumAmountLeftBIPS` was set to 1,

- Output for `fullAgentCollateral` set to `10e8`:

```
_agent.mintedAMG 104000000000
_maximumTransferUBA 52490546038
_minimumLeftAmountUBA 51509453962
```

- Output when `fullAgentCollateral` is increased significantly (e.g., `40e8`):

```
_agent.mintedAMG 104000000000
_maximumTransferUBA 0
_minimumLeftAmountUBA 170377378490
```

As a result, the maximum transferable amount drops to zero.

```
it.only("should transfer all backing to core vault", async () => {
    const agent = await Agent.createTest(context, agentOwner1,
underlyingAgent1);
    const minter = await Minter.createTest(context, minterAddress1,
underlyingMinter1, context.underlyingAmount(1000000));
    const redeemer = await Redeemer.create(context, minterAddress1,
underlyingMinter1);
    const cv = await Redeemer.create(context,
context.initSettings.coreVaultNativeAddress,
coreVaultUnderlyingAddress);

    await context.assetManager.setCoreVaultMinimumAmountLeftBIPS(1, {
from: context.governance });

    const fullAgentCollateral = toWei(10e8);
    await agent.depositCollateralsAndMakeAvailable(fullAgentCollateral,
fullAgentCollateral);

    await agent.collateralPool.enter(0, false, { from: minterAddress2,
value: toWei(3e8) });

    const [minted] = await minter.performMinting(agent.vaultAddress,
10);

    await context.updateUnderlyingBlock();
    const { 0: currentBlock, 1: currentTimestamp } = await
context.assetManager.currentUnderlyingBlock();

    const info = await agent.getAgentInfo();
    const transferAmount = info.mintedUBA;

    const cbTransferFee = await
context.assetManager.transferToCoreVaultFee(transferAmount);

    await expectRevert(
        context.assetManager.transferToCoreVault(agent.vaultAddress,
transferAmount, {
            from: agent.ownerWorkAddress,
            value: cbTransferFee.subn(1),
        }),
        "transfer fee payment too small"
    );

    const tx = await
context.assetManager.maximumTransferToCoreVault(agent.vaultAddress);
    const receipt = await tx.wait();

    for (const log of receipt.logs) {
        try {
            const parsed =
context.assetManager.interface.parseLog(log);
            console.log(`  Event: ${parsed.name}`);
```

```
            console.log("Args:");
            parsed.eventFragment.inputs.forEach((input, index) => {
                console.log(`  ${input.name}:
${parsed.args[index].toString()}`);
            });
        } catch (err) {
            // Ignore unrelated logs
        }
    }
});
```

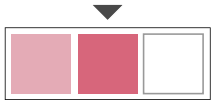
FAS-055

Escrows finalized close to expiry time decouple the Core Vault internal accountancy

Status
Caution Advised

Resolution
Acknowledged

Risk
Medium



Impact
High

Likelihood
Low

Location

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

Description

Escrows executed and finalized close to expiry times might be processed by the Core Vault Manager as expired, mistakenly assuming that the transferred funds are now available. As a consequence, any transfer request made between this stage and the bookkeeping call to `setEscrowsFinished` will have more funds available than the real balance of the core vault on XRP.

The state of escrows at `CoreVaultManager` contract on Flare is untied to what actually happens on XRP. In other words, escrow amounts and expiry times are not strictly validated in any way. Hence, the whole system trusts that the multisig's backend manages escrows under the specified parameters of the events emitted by the `CoreVaultManager` contract. Additionally, there is no

proof system to support or validate the state of escrows, being the only conditions to determine their extinction:

1. Release of preimage hashes
2. Reaching the expiry timestamp

However, because of this untied state, escrows could be created at the backend (e.g. due to a bug or backdoor) with different finalization or expiry conditions.

Besides that, even if we assume that escrows are created according to what each `EscrowInstructions` event specifies, because there is no proof system the following scenario could happen:

1. An escrow is close to expiry (e.g. some seconds or minutes away).
2. The escrow is finalized and funds are transferred from the Vault to the Custodian (underlying Core Vault balance is decreased).
3. The `processEscrows` function is called, since it is an external function allowing anyone to update the smart contract's state. This call assumes that the escrow expired, reducing the escrowed funds, adding them to the available funds variable.
4. A transfer request is made.
5. When the governance tries to call the `setEscrowsFinished` function, the execution underflows when trying to update the escrowed funds:

```
Escrow storage escrow = _getEscrow(escrowIndex);
require(!escrow.finished, "escrow already finished");
escrow.finished = true;
if (escrowIndex <= nextUnprocessedEscrowIndex) {
    availableFundsTmp -= escrow.amount;
} else {
    escrowedFundsTmp -= escrow.amount;
}
```

In summary:

1. When the system is bootstrapping and there is low liquidity at the Core Vault, requests for a high percentage of the liquidity are more likely. As a result, the scenario mentioned before also becomes more likely.

2. Even if the overflow does not happen, the Core Vault will be able to trigger a payment instruction for a request for an amount exceeding its actual balance due to its decoupling with the `availableFunds` variable.

This issue is considered to have low likelihood as it requires multiple conditions to happen such as finalizing an escrow right before the expiry and requesting a transfer right before setting the escrow as finished. The impact is considered high as transfers for more than the vault actually has can be requested. The impact is amplified for non-cancellable requests as user's

funds are burned and the only way for them to get the counterpart is by receiving the payment on the underlying chain.

Recommendation

Allow redeemers to provide payment default proofs after a sufficient period to recover burned assets.

Status

Acknowledged. Flare will control the timing so that `setEscrowsFinished` is called before escrows are released.

Proof of Concept

The following test shows how a transfer request between the escrow expiry and finalization call, triggers an underflow and breaks the tracking of funds.

The scenario is based on the `"should set already processed escrow as finished"` test case.

```
it("Coinspect - Finished escrow just before expiry breaks available
funds tracking", async () => {
// fund contract
const transactionId = web3.utils.keccak256("transactionId");
const proof = createPaymentProof(transactionId, 500);
await coreVaultManager.confirmPayment(proof);

// trigger instructions - not enough funds to create escrow
await coreVaultManager.triggerInstructions({ from: accounts[1] });
assertWeb3Equal(await coreVaultManager.availableFunds(), 500);
assertWeb3Equal(await coreVaultManager.escrowedFunds(), 0);

// add funds for fee
const transactionId1 = web3.utils.keccak256("transactionId1");
const proof1 = createPaymentProof(transactionId1, 15);
await coreVaultManager.confirmPayment(proof1);
// trigger instructions - create escrow
await coreVaultManager.triggerInstructions({ from: accounts[1] });
assertWeb3Equal(await coreVaultManager.availableFunds(), 300);
assertWeb3Equal(await coreVaultManager.escrowedFunds(), 200);

// move to the expiry of the escrow
await time.increase(2 * DAY);
await coreVaultManager.triggerInstructions({ from: accounts[1] });
assertWeb3Equal(await coreVaultManager.availableFunds(), 500); // We
assume that the escrow indeed was finalized
```

```
assertWeb3Equal(await coreVaultManager.escrowedFunds(), 0);

// request transfer
const amount = "400"; // The request believes that there are still 500
at the vault on XRP, but 200 were sent to the custodian
const paymentReference = web3.utils.keccak256("ref1");
await coreVaultManager.requestTransferFromCoreVault(
    destinationAddress1,
    paymentReference,
    amount,
    true,
    {
    from: assetManager,
    }
);
// trigger instructions
await coreVaultManager.triggerInstructions({ from: accounts[1] });
assertWeb3Equal(await coreVaultManager.availableFunds(), 500 - 400 -
15);

// // set escrow as finished; available funds will decrease
await coreVaultManager.setEscrowsFinished([preimageHash1], { from:
governance });
// assertWeb3Equal(await coreVaultManager.availableFunds(), 85 - 200);
// !!!! underflows
});
```

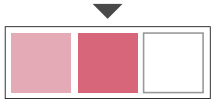
FAS-056

Large redemptions from core vault can be denied by smaller ones

Status
Caution Advised

Resolution
Acknowledged

Risk
Medium



Impact
High

Likelihood
Low

Location

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

Description

Redemption requests to the Core Vault involving large amounts—near or exceeding available funds—are never executed. This is because unfulfillable requests are skipped in favor of smaller ones that can be processed. Since these redemption requests are non-cancellable, this may result in a scenario where the required funds are never available, effectively leaving them locked.

This issue also relies on the fact that once all requests are processed, if there is a reminder of funds over the minimum Core Vault's balance, escrows are created.

Before making an action from the Core Vault, the multisig backend calls `triggerInstructions` at the `CoreVaultManager` contract. This function:

1. Checks for expired escrows and if so, adds their amount to the vault available funds
2. Processes cancellable requests, following the array ordering (FIFO)
3. Processes non-cancellable requests, following the array ordering (FIFO)
4. In case of surplus, creates escrows ensuring that at least a minimum balance is always held by the core vault.

The steps 2 and 3 validate that the transfer can be made by checking against the Core Vault's available funds:

```
if (availableFundsTmp >= transferRequestById[transferRequestId].amount
+ feeTmp) {
 // process request and emit event
} else {
    index++; // move on to the next request,
}
```

However, when creating a redemption request, users can request more than the available funds since it considers also the escrowed funds:

```
function getTotalCoreVaultAvailable()
    internal view
    returns (uint256)
{
    State storage state = getState();
    uint256 allFunds = uint256(state.coreVaultManager.availableFunds()
+ state.coreVaultManager.escrowedFunds());
    uint256 requestedAmount =
uint256(state.coreVaultManager.totalRequestAmountWithFee()) +
        getCoreVaultUnderlyingPaymentFee(); // extra fee for the
upcoming request
    return MathUtils.subOrZero(allFunds, requestedAmount);
}
```

As a result, there could be some cases when an overly large redemption request is never fulfilled.

Consider the following scenario (neglecting fees):

- `Available Funds = 1100`
- `Escrows = 400`, two escrows with `escrowAmount = 200`.

Then `allFunds = 1500`. Minimum balance is `1000`.

2. Request B (cancellable) = 200
3. Request A (non cancellable) = 1200

Then, `reserved funds = 1400`. When triggering instructions, since `availableFunds < Request A = 1200`, is skipped and only B is processed. Then:

- `Available Funds = 1100 - 200 = 900`. No escrows are created.

At some point, another transfer to Core Vault of `200` is made.

- `Available Funds = 1100`. No escrows are created.

And some returns are requested afterwards, when triggering instructions:

- Request B = 100 (cancellable). Fulfilled.
- Request C = 100 (cancellable). Fulfilled.
- Request A (non cancellable) = 1200. Not fulfilled.

No escrows are created.

The fact that these redemptions create non-cancellable requests, force users that burned their fAssets to wait until there are enough funds in the Core Vault for their transfer to be processed.

Recommendation

Do not prioritize smaller, cancellable requests over larger ones. If prioritization is necessary, consider enabling cancellation of currently non-cancellable requests.

Status

Acknowledged. If any large requests stall due to insufficient funds, Flare will release funds to the custodian, who is then expected to redeposit them into the multisig. Note that this implies using the custodian account for non-emergency tasks.

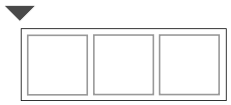
FAS-057

Array of escrows always grows

Status
Solved

Resolution
Acknowledged

Risk
None



Impact
Recommendation

Likelihood
–

Location

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

Description

The array of escrows at the `CoreVaultManager` contracts increases if triggering instructions creates new escrows and the contract has no cleanup. As a result, the cost of interacting with this always-increasing ramps up in time potentially affecting all interactions that rely on the escrows' state.

```
Escrow memory escrow = Escrow({
    preimageHash: preimageHash,
    amount: escrowAmountTmp,
    expiryTs: escrowEndTimestamp,
    finished: false
});
escrows.push(escrow);
```

Although escrows are only created under certain conditions after calling the `triggerInstructions` function, and the specification mentions that the

governance will call this function once per day, there are no restrictions to enforce this design choice.

Recommendation

Allow pruning old enough escrows.

Status

Acknowledged. Flare stated that the core vault will operate for under a year before transitioning to FAssetV2, and they expect only a small number of escrows per day. If more than one or two are created, they will raise the daily escrow limit.

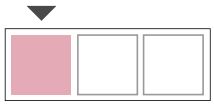
FASO-045

Partially hardcoded API response can trigger wrong actions

Status
Solved

Resolution
Fixed

Risk
Low



Impact
Low

Likelihood
Low

Location

`fasset-bots/packages/fasset-bots-api/src/bot-api/agent/services/agent.service.ts`

Description

The `getRedeemableCVData` API endpoint returns hardcoded values, potentially misleading users or systems into making decisions based on incorrect assumptions.

This behavior stems from the `getVaultRedeemableCVData` function, which hardcodes the `requestableLotsCV` and `minimumLotsToRedeem` fields:

```
return { redeemableLotsOwner: ownerLots.toNumber(), requestableLotsCV: 123, minimumLotsToRedeem: 10, lotSize: lotSizeAsset };
```

A similar issue is present in the `getAgentVaults` function behind the `vaults` endpoint:

```
transferableToCV: "10,200.24",
underlyingSymbol: cli.context.chainInfo.symbol,
transferableFromCV: "10,200.24",
redeemCapacity: (toBN(info.mintedUBA).div(lotSize)).toString()
```

Recommendation

Ensure these values are retrieved dynamically from the actual source. If not feasible, remove the fields to avoid misinforming consumers.

Status

Fixed in commits **0643ebfeb49204287e9462fcff0566baa1850529** and **3c3fb4bac5bf9957e0cc2c92e77798846482bfcf**. The `getVaultRedeemableCVData` function as well as the fixed return values from the `getAgentVaults` function were removed from the code..

FASO-046

Replacement transactions might not go through due to insufficient funds

Status
Solved

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
–

Location

`fasset-bots/packages/simple-wallet/src/chain-clients/utxo/TransactionService.ts`

Description

The `simple-wallet` does not ensure that Replace-By-Fee (RBF) transactions include enough inputs to cover the increased fee. As a result, RBF attempts for underlying withdrawal transactions may be invalid.

Currently, `simple-wallet` uses RBF in two cases:

- **Payment transactions**: Since these are time-sensitive, RBF is used to cancel them near the deadline by replacing them with a near-zero-value transaction. This prevents the bot from broadcasting the Payment and risking a double payout.
- **Underlying withdrawal transactions**: These are not time-bound, so RBF is used simply to speed up confirmation.

This issue specifically affects underlying withdrawals, and would only result in delays. The limitation is already acknowledged in the code:

```
// TODO what if not enough utxos to cover - add confirmed ones
return [tr, rbfUTXOs];
```

Recommendation

Ensure RBF transactions are constructed with sufficient inputs to cover the increased fee.

Also, consider documenting where each RBF policy rule is implemented directly in the codebase.

Status

Acknowledged. The Flare team indicated that RBF transactions for underlying withdrawals are not subject to time restrictions.

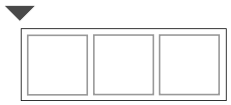
FASO-047

Wrong notifier event

Status
Solved

Resolution
Fixed

Risk
None



Impact
Recommendation

Likelihood
–

Location

`fasset-bots/packages/fasset-bots-core/src/actors/AgentBotMinting.ts`

Description

The `sendSelfMintPaymentTooSmall` notification is incorrectly used for errors related to insufficient free collateral, making it harder to analyze logs accurately.

```
if (errorIncluded(error, ["self-mint payment too small"])) {
    ...
    await this.notifier.sendSelfMintPaymentTooSmall(String(lots),
String(maxMintLots));
    ...
} else if (errorIncluded(error, ["not enough free collateral"])) {
    ...
    await this.notifier.sendSelfMintPaymentTooSmall(String(lots),
String(freeCollateralLots));
```

Recommendation

Trigger a dedicated notifier for the "not enough free collateral" case to ensure accurate event logging.

Status

Fixed in commit **cb3f3913f944b38d3c32a974d928b89ac810dad5**. The `sendSelfMintCollateralTooLow` function is now triggered on errors related to insufficient free collateral.

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.