FAsset v2 Bots

Source Code Review

coinspect

flare

FAssetV2 Bots

Source Code Review

Version: v240220 | Prepared for: Flare | December 2023

Source Code Review

Executive Summary

In **October 2023**, Flare engaged Coinspect to perform a source code review of **FAsset Bots**, a suite comprised of off-chain services that handle key actors of the **FAsset Protocol**.

The `FAsset Offchain Bots` are the `Challenger`, `Liquidator`, `Agent`, `TimeKeeper`, `SystemKeeper` and `User` bots. These rely on several peripheral components and interact with the `FAsset Protocol` smart contracts.

Solved	Caution Advised	Resolution Pending
High 5	High 0	High 0
Medium 7	Medium 0	Medium 0
Low 2	Low 0	Low 0
No Risk 2	No Risk 0	No Risk 0
Total **16**	Total **0**	Total **0**

Coinspect identified five high-risk, seven medium-risk and two low-risk issues. Overall, auditors identified:

- Cases where agents could bypass challenges, allowing them to drain their underlying balances.
- A lack of event handling leading to unfair or skipped liquidations, reward claiming omission, among others.

- Architecture and design issues related to how events are listened and processed; how reverts would affect the global execution and state.
- Problems with private keys and database encryption handling.
- Issues in the mempool dealing with transaction reordering and lack of retries.
- Risks in the proofs and attestations system regarding the use of insecure defaults, and faulty proof structure for payments.

During **November 2023**, Coinspect reviewed the fixes provided by the Flare Team for the issues contained in this report. The status of each issue was updated accordingly with the respective commits.

In **December 2023**, Coinspect reviewed the last set of fixes provided by the Flare Team.

Summary of Findings

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
FASO-003	Agents can be unfairly liquidated by deprecating tokens	High
FASO-006	Protocol might turn insolvent as agents in full liquidation status are not liquidated	High
FASO-007	Agents can drain the underlying's balance by spending multiple inputs bypassing any challenge	High
FASO-008	Malicious agents can bypass negative balance challenges	High
FASO-009	[Inherited] Payments with more than 255 inputs in UTXO chains are not supported	High
FASO-001	Insecure handling of bot operator's private keys	Medium
FASO-002	Reverts triggered by an event will skip remaining event processing	Medium
FASO-004	Liquidators fail to track vault collateral token updates and miss unhealthy agents	Medium
FASO-005	Insecure default blocks to wait for finalization	Medium
FASO-010	Airdrop distributions won't be claimed for collateral pools when the vault opts out	Medium
FASO-011	Critical bot TXs won't be performed if stuck in the mempool	Medium

FASO-012	Reverts during even processing result in corrupted TrackedState	Medium
FASO-013	Weak wallet encryption/decryption passwords are supported	Low
FASO-014	Multiple attempts may be required before successfully creating a new Agent	Low
FASO-015	Weak test coverage increases exposure to attacks and adversarial scenarios	None
FASO-016	Agents can steal underlying balance sending more than fifty transactions	None

Assessment and Scope

The audit started on **October 2nd, 2023** and was conducted on the following commits:

- `1c8378f26744aed5ed3b96bb58dab8aa585d090d` of the https://gitlab.com/flarenetwork/fasset-bots repository.
- `567482c2f5f6b20f060fff7bab6245278145c28e` of the https://gitlab.com/flarenetwork/simple-wallet repository.

Scope and Reviewed Threats

Coinspect auditors focused on scenarios that could impair the `FAsset-Bots`' performance and compromise the functioning of the `FAsset` protocol itself. The threats and adversarial scenarios reviewed were related to the following topics and fields:

- Processing target chain events and handling reorganizations when necessary.
- Managing transaction submission, expiration, return value assessment, retries, and resubmission.
- Maintaining awareness of wallet's account nonces.
- Estimating gas prices specific to the network.
- Identifying race conditions between observed events and transaction execution.
- Detecting gas waste attacks.
- Assessing blockchain congestion scenarios.
- Identifying vectors for bot denial of service attacks.

Overview

Overall, the code was easy to read, and was accompanied with documentation and specifications. In addition, the testing suite is well implemented, encompassing a comprehensive range of tests and scenarios. However, Coinspect identified room for improvement when it comes to tests coverage, as several branches are not tested which increases the likelihood of encountering bugs in production (`FASO-015`).

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` and the `Attestation Client` work correctly.

The project architecture allows operators to run separate bots by running each script. This is achieved by the creation of isolated entities (actors) that are executed with independent `runners`. All the bots but the `Agent`'s rely on the `TrackedState` implementation. This implementation is in charge of listening, processing and returning events to the consumer bots. Coinspect reported that the `TrackedState` could get its data corrupted in the event of an early revert when processing events, affecting all the internal states that depend on the remaining elements of the loop (`FASO-012`). The `Agent` bot has its own event listening mechanism, implemented directly into the `Agent`'s bot file.

Main Actors

Agent

The agent is in charge of performing all automated actions: processing mints and redemptions, and managing collateral. He is responsible for updating liquidation parameters, such as collateral prices and ratios, proving underlying addresses (if required by the underlying chain), among other key actions. On top of the automated bot actions, Agent Operators can interact with their agent instance by using CLI commands. Coinspect discovered that the agent's creation relies on the failure of the transaction simulation, as it is forced to loop over all token suffixes until an unused one is found because agent operators have no flexibility to specify the initial index, leading to `FASO-014`.

A single step of this bot is comprised by four key stages that handle:

- Unprocessed events
- Pending redemptions
- Waiting and cleanups (related to time-locked actions)
- Daily tasks (open redemptions and mints, reward claiming, among others)

The event listening mechanism relies on a three-step process to collect and process events:

1. Listens to events since the registered block, up to the current block
2. Each event is processed in the main loop
3. Depending on the event type, specific actions are performed

Coinspect identified two issues related to the event listening and processing mechanism:

1. The collateral swap event is not processed, which allows agents to get unfairly liquidated if the process is not fulfilled on time (`FASO-003`)
2. As the event processing architecture is inside a `try-catch` logic, in face of a revert, the remaining events will not be processed. This potentially corrupts the Agent's bot database, leading to inconsistent states (`FASO-002`).

Affecting the daily tasks handling, Coinspect detected that airdrops distributions for pools will not be claimed if the vault opted out. This is because the claiming process is done sequentially, inside the same `try-catch` block (`FASO-010`).

Lastly, Coinspect identified that the top-up mechanism is not properly tested, and strongly suggests adding tests for this critical functionality.

Challenger

This actor monitors all payments and redemptions where agents are involved. A challenge is triggered with the `FAsset` smart contracts, if an agent misbehaves regarding redemption payments, and minimum underlying balance. This bot should work as synced as possible, as it performs time-sensitive operations. Its main procedure consists in collecting events returned by `TrackedState`, filtering them by type, finally executing the action specified by each event.

Coinspect identified that this actor does not handle UTXO based transactions properly: it assumes that there is only one relevant input per transaction. This assumption led to two different issues:

1. An agent spending two inputs, where the first has a low value, the system fails to detect spendings from subsequent UTXOs. This allows a malicious agent to drain its underlying account without being challenged (`FASO-007`).
2. An underflow when checking the agent's underlying balance can be triggered with the same split UTXO mechanism as `FASO-007`. This leads to a negative balance, voiding the challenge mechanism (`FASO-008`).

TimeKeeper

The heartbeat of the `FAsset` Protocol is in charge of submitting the last proven underlying block number, timestamp and number of confirmations to the smart contracts. The values are later consumed by sensitive parts of the protocol, such as redemptions and minting deadlines.

SystemKeeper and Liquidator

These two actors work in a complimentary way. `SystemKeeper` is in charge of taking agents in or out of liquidation state, at the right time (e.g., when undercollateralized or challenged). This actor's main procedure consumes the events from `TrackedState`, and pursues actions when the collateral price changes, or when an agent executes a minting operation.

The `Liquidator` tracks down key agent operations that alter the collateral ratio, checking if they are liquidatable. If the liquidation threshold is met, the bot calls the `AssetManager` contract, triggering the agent's liquidation. In terms of event consuming, it works the same way as the `SystemKeeper`. Coinspect reported that the liquidator bot only targets agents that are liquidatable because of an unhealthy collateral ratio, skipping any liquidation to those agents that were challenged and are in `FULL LIQUIDATION` status (`FASO-006`). Also, the liquidator bot does not properly handle collateral swaps, skipping the liquidation. This means that those agents that did not swap their collateral will still be considered healthy by the calculations made with the internal state (`FASO-004`).

User

Provides the base actions and interactions that users can perform with Agents, such as reserving collateral, executing minting positions, and requesting for redemptions. Users interact with this implementation via CLI Commands.

Peripheral Entities and Services

The suite has multiple services that provide key functionalities to each core actor, such as: utilities, indexer helpers, attestation client helpers, a simple wallet, a tracked state, among others.

Within the peripheral services, Coinspect identified that the `BlockchainIndexerHelper` uses an insecure configuration for blocks finalization, as default - no `config` is provided (`FASO-005`). Each type of transaction has its own proof structure, which is consumed by the attestation services and proof verification systems. Coinspect detected that an issue related to payment proofs, reported in a previous assessment, is still present (`FASO-009`). With regards to awaiting for requests, Coinspect identified that parts of the codebase have no timeouts, and wait for responses indefinitely. For instance, in `StateConnectorClientHelper` in `waitForRoundFinalization()`, the iteration of the `while-loop` is constrained by a `sleep` call, exiting the loop only when certain condition is met, i.e., the round finalizes.

In terms on how the system interacts with the wallet to send transactions, Coinspect noted that there is no retry or mempool-handling logic. This means that stuck transactions cannot be bumped or dropped. Ultimately, the execution of those transactions will halt with higher nonces (`FASO-011`).

Lastly, users create new accounts that are stored into their local database. The private keys of those accounts are encrypted and then stored. However, the encryption password, used both for creation and recovery, can be weak, having no policy enforcement (`FASO-013`). In addition, the current project structure requires users to provide their agent private key along with other sensitive information into the same `.env` file (`FASO-001`).

Detailed Findings

FASO-003

Agents can be unfairly liquidated by deprecating tokens

Status
Solved

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

`src/state/TrackedState.ts`
`src/state/TrackedAgentState.ts`

Description

The Agent Bot omits the `CollateralTypeDeprecated` event. As a consequence the vault collateral token might not be switched and, once it is no longer valid, the Agent can be liquidated. Additionally, the Agent Bot will not be able to perform collateral top-ups, as it will never find the new collateral token in `context`, created when starting the bot.

Upon Bot startup, the context retrieves all the collateral types (active and deprecated), to later store the address of each stablecoin:

`src/config/create-asset-context.ts`:

```
    const collaterals = await assetManager.getCollateralTypes();
    const stableCoins = await createStableCoins(collaterals);
    return {
        nativeChainInfo: botConfig.nativeChainInfo,
        chainInfo: chainConfig.chainInfo,
        blockchainIndexer: chainConfig.blockchainIndexerClient,
        wallet: chainConfig.wallet,
        attestationProvider: new
AttestationHelper(chainConfig.stateConnector,
chainConfig.blockchainIndexerClient, chainConfig.chainInfo.chainId),
        assetManager: assetManager,
        priceChangeEmitter: priceChangeEmitter,
        wNat: wNat,
        fAsset: await FAsset.at(await assetManager.fAsset()),
        collaterals: collaterals,
        stablecoins: stableCoins,
        addressUpdater: addressUpdater,
    };
```

Then, if a top-up transaction should be made the bot attempts to match the current collateral token with the stablecoin, in the context created upon startup:

```
    async depositVaultCollateral(amountTokenWei: BNish) {
        const vaultCollateralTokenAddress = (await
this.getVaultCollateral()).token;
        const vaultCollateralToken =
requireNotNull(Object.values(this.context.stablecoins).find((token) =>
token.address === vaultCollateralTokenAddress));
        await vaultCollateralToken.approve(this.vaultAddress,
amountTokenWei, { from: this.ownerAddress });
        return await
this.agentVault.depositCollateral(vaultCollateralTokenAddress,
amountTokenWei, { from: this.ownerAddress });
    }
```

As there is no component that modifies the current collateral token when listening `CollateralTypeDeprecated`:

1. The bot will make top-ups of the recently deprecated token, not altering its collateral ratio in the contracts and draining the owner's wallet.
2. Any bot will be able to liquidate the unsuspecting agent if the collateral token is not manually switched, as expired collateral tokens won't count for the CR calculation. When the switching deadline is reached, the Agent faces

a discrete jump in their collateral ratio as all the deprecated token balance is dismissed:

```
// A simple way to force agents still holding expired collateral tokens
into liquidation is just to
// set fullCollateral for expired types to 0.
// This will also make all liquidation payments in the other collateral
type.
uint256 fullCollateral = CollateralTypes.isValid(collateral) ?
collateral.token.balanceOf(owner) : 0;
```

Recommendation

Add collateral deprecation handling.

Status

Fixed on commits `891b88965ea59cb24c02bfcf61442b8962800b20` and `3ead1e25171846605bd48c6ee9d1df8f069a02c1`.

The agent's tracked state now handles the `AgentCollateralTypeChanged` event triggered when the vault's collateral is changed.

FASO-006

Protocol might turn insolvent as agents in full liquidation status are not liquidated

Status

Solved

Resolution

Fixed

Risk

High

Impact

High

Likelihood

High

Location

`src/actors/Liquidator.ts`

Description

The `Liquidator` does not handle the `FULL_LIQUIDATION` status of an agent, meaning that no liquidation for those agents that misbehave will be triggered. As a consequence, no agent will be liquidated. In the event of having a collateral price drop, that position would start to build up debt harming the global protocol's health.

A liquidation is triggered when a minting position is executed or when a price epoch is finalized:

```
for (const event of events) {
    if (eventIs(event, this.state.context.priceChangeEmitter,
"PriceEpochFinalized")) {
```

```
        console.log(`Liquidator ${this.address} received event
'PriceEpochFinalized' with data ${formatArgs(event.args)}.`);
        logger.info(`Liquidator ${this.address} received event
'PriceEpochFinalized' with data ${formatArgs(event.args)}.`);
        await this.checkAllAgentsForLiquidation();
    } else if (eventIs(event, this.state.context.assetManager,
"MintingExecuted")) {
        console.log(`Liquidator ${this.address} received event
'PriceEpochFinalized' with data ${formatArgs(event.args)}.`);
        logger.info(`Liquidator ${this.address} received event
'MintingExecuted' with data ${formatArgs(event.args)}.`);
        await this.handleMintingExecuted(event.args);
    }
}
```

Then, the liquidation process checks that the current status is `LIQUIDATION` in order to liquidate the agent:

```
if (newStatus === AgentStatus.LIQUIDATION) {
    const fBalance = await
this.state.context.fAsset.balanceOf(this.address);
    console.log(`Trying to liquidate agent ${agent.vaultAddress}`);
    await this.state.context.assetManager.liquidate(agent.vaultAddress,
fBalance, { from: this.address });
    logger.info(`Liquidator ${this.address} liquidated agent
${agent.vaultAddress}.`);
}
```

Because of this, a challenged agent that now has the `FULL_LIQUIDATION` status will never be detected by the `Liquidator`. This leads to profits losses, potentially driving the protocol into insolvency (assuming that no other external bots with a correct implementation handle this condition).

Proof of Concept

The following test shows how an agent in `FULL_LIQUIDATION` status is not liquidated by the main steps of the `Liquidator`. It is checked that the liquidator does not receive any vault collateral token or spends FAssets as no liquidation was performed.

Run the script in the `test-hardhat/integration/challenger.ts` file.

Prerequisites:

- Some imports have to be added:

```
import { sleep, toBN, toBNExp } from "../../src/utils/helpers";
import { artifacts, web3 } from "../../src/utils/web3";
```

```
import {createTestLiquidator} from "../test-utils/helpers";
const IERC20 = artifacts.require("IERC20");
```

- Add the following addresses in the constructor (declaring them as global strings):

```
    liquidatorAddress = accounts[7];
    minter2Address = accounts[8];
```

Script

```
it("Coinspect - Will not liquidate agents in full liquidation", async
() => {
        const challenger = await
createTestChallenger(challengerAddress, state);
        const liquidator = await
createTestLiquidator(liquidatorAddress, state);

const spyChlg = spy.on(challenger, "doublePaymentChallenge");
        // create test actors
        const agentBot = await
createTestAgentBotAndMakeAvailable(context, orm, ownerAddress);
        const vaultCollateralToken = await IERC20.at((await
agentBot.agent.getVaultCollateral()).token);

const minter = await createTestMinter(context, minterAddress, chain);
        const minter2 = await createTestMinter(context, minter2Address,
chain);

const redeemer = await createTestRedeemer(context, redeemerAddress);
        await challenger.runStep();

// create collateral reservation and perform minting
        await createCRAndPerformMintingAndRunSteps(minter, agentBot, 3,
orm, chain);

// Generate balance in funder minter
        await createCRAndPerformMintingAndRunSteps(minter2, agentBot,
3, orm, chain);

// transfer FAssets
        const fBalance = await
context.fAsset.balanceOf(minter.address);
        await context.fAsset.transfer(redeemer.address, fBalance, {
from: minter.address });
        // create redemption requests and perform redemption
        const [reqs] = await redeemer.requestRedemption(3);
        const rdReq = reqs[0];
        // run agent's steps until redemption process is finished
        for (let i = 0; ; i++) {
            await time.advanceBlock();
            chain.mine();
            await agentBot.runStep(orm.em);
            // check if redemption is done
```

```
            orm.em.clear();
            const redemption = await agentBot.findRedemption(orm.em,
rdReq.requestId);
            console.log(`Agent step ${i}, state =
${redemption.state}`);
            if (redemption.state === AgentRedemptionState.DONE) break;
        }
        // repeat the same payment (already confirmed)
        await performRedemptionPayment(agentBot.agent, rdReq);
        // run challenger's and agent's steps until agent's status is
FULL_LIQUIDATION
        for (let i = 0; ; i++) {
            await time.advanceBlock();
            chain.mine();
            await sleep(3000);
            await challenger.runStep();
            await agentBot.runStep(orm.em);
            const agentStatus = await getAgentStatus(agentBot);
            console.log(`Challenger step ${i}, agent status =
${AgentStatus[agentStatus]}`);
            if (agentStatus === AgentStatus.FULL_LIQUIDATION) break;
        }
        const agentStatus = await getAgentStatus(agentBot);
        assert.equal(agentStatus, AgentStatus.FULL_LIQUIDATION);
        expect(spyChlg).to.have.been.called.once;

// Try to liquidate agent in full liquidation
        // liquidator "buys" f-assets
        console.log("Transferring Fassets to liquidator...");
        const funderBalance = await
context.fAsset.balanceOf(minter2.address);
        await context.fAsset.transfer(liquidator.address,
funderBalance, { from: minter2.address });

// FAsset and collateral balance
        const fBalanceBefore = await
state.context.fAsset.balanceOf(liquidatorAddress);
        const cBalanceBefore = await
vaultCollateralToken.balanceOf(liquidatorAddress);

// As the only trigger is price changes, check if the liquidator would
liquidate the agent
        // mock price changes and run liquidation trigger
        console.log("Finalizing Price Epoch...");
        await context.ftsoManager.mockFinalizePriceEpoch();

console.log("Liquidating...");
        await liquidator.runStep();

const fBalanceAfter = await
state.context.fAsset.balanceOf(liquidatorAddress);
        const cBalanceAfter = await
vaultCollateralToken.balanceOf(liquidatorAddress);

// The balance is unchanged, meaning that the liquidator bot omitted
the liquidation of the agent
        expect(cBalanceAfter.eq(cBalanceBefore)).to.be.true;
        expect(fBalanceAfter.eq(fBalanceBefore)).to.be.true;
    });
```

Recommendation

Handle the full liquidation status in the `Liquidator`.

Status

Fixed on commit `460278a546f800ed622b2f4b94d41a1e31b29c03`.

Liquidator bots now also trigger liquidations when hearing the `FullLiquidationStarted` event.

FASO-007

Agents can drain the underlying's balance by spending multiple inputs bypassing any challenge

Status
Solved

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

`src/actors/Challenger.ts`

Description

Agents can empty the underlying account by announcing a withdrawal transaction that spends a small amount in the first UTXO, draining the balance in a second input.

The Challenger Bot only considers the first matching input UTXO when processing transactions:

`checkForNegativeFreeBalance()`:

```
const spentAmount = transaction.inputs.find((input) => input[0] ===
agent.underlyingAddress)?.[1];
```

Then, the `transactions` array is built using the first matched UTXO's value and hash:

```
transactions.push({ txHash: transaction.hash, spent: spentAmount });
```

This mechanism to build the `transactions` array allows Agents to craft transactions spending multiple inputs, that will bypass the following check:

```
const totalSpent = sumBN(transactions, (tx) => tx.spent);
if (totalSpent.gt(agent.freeUnderlyingBalanceUBA)) {
    const transactionHashes = transactions.map((tx) => tx.txHash);
    this.runner.startThread((scope) =>
this.freeBalanceNegativeChallenge(scope, transactionHashes, agent));
}
```

A transaction with the following structure will bypass the negative balance check and could effectively drain the accounts balance:

```
    const fistUTXOAmt = toBN(underlyingBalanceUBA).div(toBN(1000));
    const spentUTXOs: UTXO[] = [
        { value: fistUTXOAmt }, // UTXO 1
        { value: toBN(underlyingBalanceUBA).sub(fistUTXOAmt) }, // UTXO
2
    ];
```

Proof of Concept

The following test scenario shows how an agent is able to drain the underlying account, without being challenged, because of `freeBalanceNegativeChallenge`. It can be seen how after performing the balance decreasing transactions, the Challenger Bot keeps detecting that the Agent is in `NORMAL` state which does not trigger the challenge.

Test Setup

To make this proof-of-concept, Coinspect adapted the `MockChainWallet`'s implementation to allow transactions that spend multiple UTXOs belonging to the same address.

Follow the instructions below to use this implementation:

1. Add the code in the `Appendix A` to `src/underlying-chain/interfaces/IBlockChainWallet.ts`.
2. In `src/mock/MockChain.ts` comment the current `MockChainWallet`'s implementation and add the code in the `Appendix B`
3. In `src/fasset-bots/IAssetBotContext.ts` add the `IBlockChainWalletMultipleUTXOs` into the `wallet` global:

```
wallet: IBlockChainWallet --------> wallet: IBlockChainWallet |
IBlockChainWalletMultipleUTXOs
```

4. Do the same as the previous step, but in `src/fasset/Agent.ts`:

```
    get wallet(): IBlockChainWallet {
        return this.context.wallet;
    }

// Replace for:
    get wallet(): IBlockChainWallet | IBlockChainWalletMultipleUTXOs {
        return this.context.wallet;
    }
```

Output

```
Tx Status: 0
Hash:
0x7cbd615090cee3e23141d704630ae4bd3e09bc9155bd21d281e020ee9cec1ef6
Reference:

INPUTS:
spender:  - value: 10000000
Total Spent: 10000000

OUTPUTS:
recipient: UNDERLYING_ACCOUNT_96954 - value: 10000000
Total Received: 10000000

MINTING STARTED: Minting 69 started for
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC.
Tx Status: 0
Hash:
0x8263644e7e11703c0ba80d5c3010a73571b68fae662215aa92db3907514036cb
Reference:
0x4642505266410001000000000000000000000000000000000000000000000045

INPUTS:
spender: MINTER_UNDERLYING_ADDRESS - value: 22000000000
Total Spent: 22000000000

OUTPUTS:
recipient: UNDERLYING_ACCOUNT_96954 - value: 22000000000
Total Received: 22000000000
```

```
MINTING EXECUTED: Minting 69 executed for
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC.
Error handling FTSO rewards for agent
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC: Error: Cannot create
instance of IFtsoRewardManager; no code at address
0x0000000000000000000000000000000000000000
Error handling airdrop distribution for agent
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC: Error: Cannot create
instance of IDistributionToDelegators; no code at address
0x0000000000000000000000000000000000000000

PAYING....
Tx Status: 0
Hash:
0x794253a89a4255d04dc500a4e91eceec45bb0827675fc406edf470ce513742d8
Reference:
0x4642505266410003000000000000000000000000000000000000000000000049

INPUTS:
spender: UNDERLYING_ACCOUNT_96954 - value: 22000000
spender: UNDERLYING_ACCOUNT_96954 - value: 21978000000
Total Spent: 22000000000

OUTPUTS:
recipient: UNDERLYING_ADDRESS - value: 22000000000
Total Received: 22000000000

Detected Tx Inputs:
UNDERLYING_ACCOUNT_96954,22000000,UNDERLYING_ACCOUNT_96954,21978000000
Found Spent Amount: 22000000

Total Spent Calculated By Challenger: 22000000
Free Underlying Balance UBA: 1200000000

Detected Tx Inputs:
UNDERLYING_ACCOUNT_96954,22000000,UNDERLYING_ACCOUNT_96954,21978000000
Found Spent Amount: 22000000

Total Spent Calculated By Challenger: 22000000
Free Underlying Balance UBA: 1200000000

Challenger step 0, agent status = NORMAL
Challenger step 1, agent status = NORMAL
```

Script

```
   it("Coinspect - Will not challenge negative balance with multiple
UTXOs", async () => {
        const challenger = await
createTestChallenger(challengerAddress, state);
        // create test actors
        const agentBot = await
createTestAgentBotAndMakeAvailable(context, orm, ownerAddress);
        const minter = await createTestMinter(context, minterAddress,
chain);
        await createCRAndPerformMintingAndRunSteps(minter, agentBot, 2,
orm, chain);
```

```
        await challenger.runStep();
        const underlyingBalanceUBA = (await
agentBot.agent.getAgentInfo()).underlyingBalanceUBA;
        // announce and perform underlying withdrawal
        const underlyingWithdrawal = await
agentBot.agent.announceUnderlyingWithdrawal();

let spenderAddr = agentBot.agent.underlyingAddress;
        let agentUnderlyingAddr = underlyingAddress;

const fistUTXOAmt = toBN(underlyingBalanceUBA).div(toBN(1000));
        const spentUTXOs: UTXO[] = [
            { value: fistUTXOAmt }, // UTXO 1
            { value: toBN(underlyingBalanceUBA).sub(fistUTXOAmt) }, //
UTXO 2
        ];

// Using This UTXO would trigger the negative underlying free balance
challenge
        // const spentUTXOs: UTXO[] = [{ value:
toBN(underlyingBalanceUBA) }];

const spent: SpentReceivedObject = {
            [spenderAddr]: spentUTXOs,
        };

const received1: SpentReceivedObject = {
            [agentUnderlyingAddr]: [{ value: underlyingBalanceUBA }],
        };

// Perform payment with multiple UTXOs
        console.log("\nPAYING....");
        await (agentBot.agent.wallet as
IBlockChainWalletMultipleUTXOs).addMultiTransaction(spent, received1,
underlyingWithdrawal.paymentReference);

chain.mine(chain.finalizationBlocks + 1);
        // run challenger's steps until agent's status is
FULL_LIQUIDATION
        for (let i = 0; ; i++) {
            await time.advanceBlock();
            chain.mine();
            await sleep(3000);
            await challenger.runStep();
            const agentStatus = await getAgentStatus(agentBot);
            console.log(`Challenger step ${i}, agent status =
${AgentStatus[agentStatus]}`);
            if (agentStatus === AgentStatus.FULL_LIQUIDATION) break;
        }
        // send notification
        await agentBot.runStep(orm.em);
        // check status
        const agentStatus2 = await getAgentStatus(agentBot);
        assert.equal(agentStatus2, AgentStatus.FULL_LIQUIDATION);
    });
```

Recommendation

Handle transactions that spend multiple inputs.

Status

Fixed on commit `5128e6ca3a5e781c8a210d7e704bdd9fab803fb4`.

The Challenger bot now performs negative balance calculations considering transactions with multiple inputs and outputs.

FASO-008

Malicious agents can bypass negative balance challenges

Status
Solved

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

`src/actors/Challenger.ts`

Description

Malicious agents can bypass negative balance challenges by triggering an underflow inside `checkForNegativeFreeBalance`. This is achieved by spending multiple UTXOs, where the value of the first input spent is lower than the redemption's.

After each redemption payment transaction, the free underlying balance is checked to prevent it from being negative. In the event of being negative (`current balance < required underlying balance`), the `freeBalanceNegativeChallenge` should be made. However, because the `checkForNegativeFreeBalance` function only uses the value for the first spent UTXO and then subtracts the redemption's value, when the first UTXO's value is lower than the redemption's, an underflow is triggered.

`checkForNegativeFreeBalance()`:

```
const spentAmount = transaction.inputs.find((input) => input[0] ===
agent.underlyingAddress)?.[1];
/* istanbul ignore next */
if (spentAmount == null) continue;
if (this.isValidRedemptionReference(agent, transaction.reference)) {
    // eslint-disable-next-line @typescript-eslint/no-non-null-
assertion
    const { amount } =
this.activeRedemptions.get(transaction.reference)!;
    transactions.push({ txHash: transaction.hash, spent:
spentAmount.sub(amount) });
```

This scenario makes the `spent` item of the transactions array negative, which will always be lower than the agent's `freeUnderlyingBalanceUBA` effectively bypassing a potential challenge.

```
// initiate challenge if total spent is big enough
const totalSpent = sumBN(transactions, (tx) => tx.spent);
if (totalSpent.gt(agent.freeUnderlyingBalanceUBA)) {
    const transactionHashes = transactions.map((tx) => tx.txHash);
    this.runner.startThread((scope) =>
this.freeBalanceNegativeChallenge(scope, transactionHashes, agent));
}
```

Proof of Concept

The following test scenario shows how an agent avoids being challenged by `freeBalanceNegativeChallenge`, by paying a redemption using two UTXOs, and the value of the first one is smaller than the redemption's value. It can be seen that the `totalSpent` amount calculated by the challenger bot is negative. The Challenger misses the logic inaccuracy, not challenging the malicious agent.

Test Setup

In order to make this proof of concept, Coinspect adapted the `MockChainWallet`'s implementation to allow transactions that spend multiple UTXOs belonging to the same address.

Follow the same instructions from `FASO-007` to setup the environment.

Output

```
Tx Status: 0
Hash:
0x4d3361c9d2344917f8fc867b0791b84d5a5b415937c7645797f1dc1e5687b258
Reference:

INPUTS:
spender:  - value: 10000000
Total Spent: 10000000

OUTPUTS:
recipient: UNDERLYING_ACCOUNT_50208 - value: 10000000
Total Received: 10000000

MINTING STARTED: Minting 69 started for
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC.
Tx Status: 0
Hash:
0xf2ae0496de58049bbe0dd71aca0c7dd7cb9100dee2bf58fb0f4d43848109480b
Reference:
0x4642505266410001000000000000000000000000000000000000000000000045

INPUTS:
spender: MINTER_UNDERLYING_ADDRESS - value: 33000000000
Total Spent: 33000000000

OUTPUTS:
recipient: UNDERLYING_ACCOUNT_50208 - value: 33000000000
Total Received: 33000000000

MINTING EXECUTED: Minting 69 executed for
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC.
Error handling FTSO rewards for agent
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC: Error: Cannot create
instance of IFtsoRewardManager; no code at address
0x0000000000000000000000000000000000000000
Error handling airdrop distribution for agent
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC: Error: Cannot create
instance of IDistributionToDelegators; no code at address
0x0000000000000000000000000000000000000000

PAYING....
Tx Status: 0
Hash:
0x73cebe3e6105493c75ed9e6cbaa01d2192a76172df478b9e7c802d7dd647bf1e
Reference:
0x464250526641000200000000000000000000000000000000000000000000004a

INPUTS:
spender: UNDERLYING_ACCOUNT_50208 - value: 1
spender: UNDERLYING_ACCOUNT_50208 - value: 39600000000
Total Spent: 39600000001

OUTPUTS:
recipient: REDEEMER_UNDERLYING_ADDRESS - value: 39600000001
Total Received: 39600000001

Detected Tx Inputs:
UNDERLYING_ACCOUNT_50208,1,UNDERLYING_ACCOUNT_50208,39600000000
Found Spent Amount: 1
```

```
Total Spent Calculated By Challenger: -19999999999
Free Underlying Balance UBA: 1800000000

Detected Tx Inputs:
UNDERLYING_ACCOUNT_50208,1,UNDERLYING_ACCOUNT_50208,39600000000
Found Spent Amount: 1

Total Spent Calculated By Challenger: -19999999999
Free Underlying Balance UBA: 1800000000

Challenger step 0, agent status = NORMAL
Challenger step 1, agent status = NORMAL
```

Script

```
   it("Coinspect - Underflow upon redemption payment", async () => {
        const challenger = await
createTestChallenger(challengerAddress, state);
        const spyChlg = spy.on(challenger, "doublePaymentChallenge");
        // create test actors
        const agentBot = await
createTestAgentBotAndMakeAvailable(context, orm, ownerAddress);
        const minter = await createTestMinter(context, minterAddress,
chain);
        const redeemer = await createTestRedeemer(context,
redeemerAddress);
        await challenger.runStep();
        // create collateral reservation and perform minting
        await createCRAndPerformMintingAndRunSteps(minter, agentBot, 3,
orm, chain);
        // transfer FAssets
        const fBalance = await
context.fAsset.balanceOf(minter.address);
        await context.fAsset.transfer(redeemer.address, fBalance, {
from: minter.address });

// perform redemption
        const [reqs] = await redeemer.requestRedemption(2);
        const rdReq = reqs[0];

// make the redemption payment
        const paymentAmount = reqs[0].valueUBA.sub(reqs[0].feeUBA);

const spentUTXOs: UTXO[] = [
            { value: toBN(1) }, // UTXO 1
            { value: toBN(paymentAmount).mul(toBN(2)) }, // UTXO 2
        ];

let spenderAddr = agentBot.agent.underlyingAddress;
        const spent: SpentReceivedObject = {
            [spenderAddr]: spentUTXOs,
        };

const received1: SpentReceivedObject = {
            [reqs[0].paymentAddress]: [
                {
                    value:
```

```
toBN(paymentAmount).mul(toBN(2)).add(toBN(1)),
                },
            ],
        };

// Perform payment with multiple UTXOs
        console.log("\nPAYING....");
        await (agentBot.agent.wallet as
IBlockChainWalletMultipleUTXOs).addMultiTransaction(spent, received1,
reqs[0].paymentReference);
        chain.mine(chain.finalizationBlocks + 1);

const agentStatus1 = await getAgentStatus(agentBot);
        assert.equal(agentStatus1, AgentStatus.NORMAL);

// run challenger's steps until agent's status is FULL_LIQUIDATION
        for (let i = 0; ; i++) {
            await time.advanceBlock();
            chain.mine();
            await sleep(3000);
            await challenger.runStep();
            const agentStatus = await getAgentStatus(agentBot);
            console.log(`Challenger step ${i}, agent status =
${AgentStatus[agentStatus]}`);
            if (agentStatus === AgentStatus.FULL_LIQUIDATION) break;
        }
        // send notification
        await agentBot.runStep(orm.em);
        const agentStatus2 = await getAgentStatus(agentBot);
        assert.equal(agentStatus2, AgentStatus.FULL_LIQUIDATION);
        expect(spyChlg).to.have.been.called.once;
    });
```

Recommendation

Prevent underflows when calculating the `spent` item and handle transactions that spend multiple input UTXOs.

Status

Fixed on commit `5128e6ca3a5e781c8a210d7e704bdd9fab803fb4`.

The Challenger bot now performs negative balance calculations considering transactions with multiple inputs and outputs.

FASO-009

[Inherited] Payments with more than 255 inputs in UTXO chains are not supported

Status
Solved

Resolution
Fixed

Risk
High

Impact
High

Likelihood
High

Location

```
src/verification/attestation-types/t-00001-payment.ts
```

Description

Payments made in UTXO chains spending more than 255 inputs are not supported. As a result, any attempt to encode or decode transactions with more than 255 inputs or outputs will fail.

The payment type has the following request when it comes to UTXO indexes:

```
    {
      key: "inUtxo",
      size: UTXO_BYTES,
      type: "NumberLike",
      description: `
Index of the source address on UTXO chains. Always 0 on non-UTXO
chains.
```

```
`,
    }
```

Where `UTXO_BYTES == 1`.

Then, this size is used (as `def.size`) across the verification `dir` to encode and decode data, for example:

```
  encodeRequest(request: ARBase): string {
    let definition =
this.getDefinitionForAttestationType(request.attestationType);
    if (!definition) {
      throw new AttestationRequestEncodeError(`Unsupported attestation
type id: ${request.attestationType}`);
    }
    let bytes = "0x";
    for (let def of [...REQUEST_BASE_DEFINITIONS,
...definition.request]) {
      const value = request[def.key as keyof ARBase];
      if (value === undefined) {
        throw new AttestationRequestEncodeError(`Missing key ${def.key}
in request`);
      }
      bytes += toUnprefixedBytes(value, def.type, def.size, def.key);
    }
    return bytes;
  }
```

This issue has been reported in previous reports of audits to other parts of the FAsset ecosystem:

1. FAsset Smart Contracts (`FAS-010: Payments on UTXO chains cannot be verified under certain conditions`) and
2. Attestation Client (`ATC-09: Attacker can prevent Payment and Balance Decreasing Attestations`)

Additionally, if the Attestation Client addresses the ATC-09 by supporting more than 255 inputs while maintaining `UTXO_BYTES` at 1 in the FAsset Bots project, this could lead to discrepancies and potential encoding/decoding issues.

Recommendation

Increase the amount of UTXOs supported. Additionally coordinate with the other Flare Teams to ensure that the Attestation Client and FAsset Smart Contracts follow the same criteria.

Status

Fixed on commit `5867ee4b17452c17972608f946ca6b6836f262fe`.

The `state-connector` was fixed an now considers UTXOs with more than 255 inputs.

FASO-001

Insecure handling of bot operator's private keys

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Description

The current project's structure does not fully protect bot runners credentials, increasing the likelihood of disclosing sensitive data into the repository.

Users and bot operators are forced by the project's architecture to place all their keys into the same `.env` file:

```
# DB ENCRYPTION

WALLET_ENCRYPTION_PASSWORD=

# NATIVE CHAIN

OWNER_ADDRESS=
OWNER_PRIVATE_KEY=

# UNDERLYING CHAIN
```

```
OWNER_UNDERLYING_ADDRESS=
OWNER_UNDERLYING_PRIVATE_KEY=

# RUN CONFIG PATH

RUN_CONFIG_PATH=

# FLARE_API_PORTAL_KEY

FLARE_API_PORTAL_KEY=

# INDEXER

INDEXER_API_KEY=

# RPC API KEYS (optional)

#NATIVE_RPC_API_KEY=
#XRP_RPC_API_KEY=
#BTC_RPC_API_KEY=
#DOGE_RPC_API_KEY=
```

This pattern could result in compromised sensitive credentials if they are accidentally pushed into a public repository.

Recommendation

Retrieve sensitive credentials from an external separate file. Additionally, enforce file system permissions for the file/s containing the keys, adding checks that prevent reading private keys from files that are too permissive.

Status

Fixed on commits `64c3032db6927602b489e4ee5ae661eaeaeb2408`, `ed6fa67beac8ff783ab88b5ce1ed0de2192cec91`, `62fc3116a3deecadadf0da39cb2a3df8abfaac38`, and `28887b1022565565c487c0db1ca6a48635c0eb61`.

A new implementation to handle secrets was added. Secrets are retrieved from a `secrets.json` file and permissions are checked before reading its content.

FASO-002

Reverts triggered by an event will skip remaining event processing

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Location

`src/actors/AgentBot.ts`

Description

Reverts at some point of the main `handleEvents()` loop will prevent the Agent bot from processing subsequent events, ignoring key actions and database updates they might trigger.

The main loop listens to unhandled events, and then performs actions triggered by each one:

```
async handleEvents(rootEm: EM): Promise<void> {
    await rootEm
        .transactional(async (em) => {
            const events = await this.readUnhandledEvents(em);
            // Note: only update db here, so that retrying on error
won't retry on-chain operations.
```

```
            for (const event of events) {

// ================================== EVENT HANDLING
==================================

}
        })
        .catch((error) => {
            console.error(`Error handling events for agent
${this.agent.vaultAddress}: ${error}`);
            logger.error(`Agent ${this.agent.vaultAddress} run into
error while handling events: ${error}`);
        });
}
```

Due to this design, if some error is encountered at any step of the event loop, it will be caught by the `catch` branch, skipping any action triggered by the remaining events. When a new bot step is run, the remaining unprocessed events of the previous step won't be recovered.

Coinspect has not identified any way to crash the code handling events yet. However, it is worth considering this could happen and improving the code to handle this scenario would result in a more resilient code.

Proof of Concept

The following test shows how an Agent Bot does not process a set of events that were enqueued after an event triggers an error. When a new step is run, those unprocessed events are not taken into account anymore.

To run this test:

- Place the script in `test-hardhat/integration/agentBot.ts`
- For event name and hash logging, add the following loop inside `handleEvents()`:

```
for (const event of events) {
    console.log(event.event, event.transactionHash)
}
```

- Add the following `throw` in the first line of the `checkAgentForCollateralRatiosAndTopUp()` function:

```
    throw "Some Throw inside checkAgentForCollateralRatiosAndTopUp"
```

Output

```
Run Step 1
AgentAvailable
0x47636c9bdf21ab496d588032f417662cc9fdcf59170bd10d0fb836da99d7b95a
CollateralReserved
0x057addd2031056d731639d6246d115e7643bceb443ac45b5a8cd61446573481a
PriceEpochFinalized
0xe71dc696fa436a8414091060c1815b576ab738dbe3739febf7214133b8771349
CollateralReserved
0x79aff57a54194a2b9dc0790aef4b78bce1f7b27fcbcf2609d0abc7a4f8a30923
CollateralReserved
0xd5eef0ae4e35dc5c0d673dc48ac4757a69f3f21dc1f978bb2a2e172bef5e7767
CollateralReserved
0xf319d8cce2eb69661e5d1f396a1d6edbd44bf08758b2d2c8f430417b4f354dd9
CollateralReserved
0xbe110da05eb9be61dfa26e4caed8c767eb7169fbbd80be3ef1b75b0b720ae5ef
CollateralReserved
0x41aadea901e431a9240059869c1956e590602b6402968d1420b7f0ed2d549454
CollateralReserved
0xd328cbd311a9b78abb59a67b8e0e7321f72c537af57a4a516b81ff21000a0877
CollateralReserved
0xd00794a9c7c2c9b6dad90424d0d97a1b783da474bd3d95c29dc7c3ee6e90a8a1
CollateralReserved
0x6becb99d5519ff7adf2d93ffdc7f006f8d1a1852e4fefa985a0e648ab3ffac6b
MINTING STARTED: Minting 71 started for
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC.
Error handling events for agent
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC: Some Throw inside
checkAgentForCollateralRatiosAndTopUp

Run Step 2
CollateralReserved
0xe8358cb6ef10b72d6b705c3647ff5464222aaeadccb831ec6e5b67d9d74dfea5
MINTING STARTED: Minting 766 started for
0xEA6aBEf9ea06253364Bb6cf53065dAFD2ca122FC.
```

Script

```
it("COINSPECT - Throw at some point of handleEvents", async () => {
    console.log("\nRun Step 1");
    const crt2 = await
minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await context.ftsoManager.mockFinalizePriceEpoch();
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await agentBot.runStep(orm.em);

// Advance one block as events are ordered increasingly
```

```
    await time.advanceBlock();
    chain.mine();

console.log("\nRun Step 2");
    const crt4 = await
minter.reserveCollateral(agentBot.agent.vaultAddress, 2);
    await agentBot.runStep(orm.em);
});
```

Recommendation

Create a retry mechanism that continues the event processing, after the one that triggered the revert.

If recovery is not an option, stop the Agent's Bot execution when there's a revert as next events could depend on corrupted state changes.

Status

On commits `c1257574f2dbd5909eb9986697c39c150eeb9b03` and `4152ff248abdacbc5e94602c3d2d07b62a916af4`:

A new event handling process including a retry logic for failed events was added. This new architecture creates an internal queue of handled and unhandled events, which is checked on the beginning of each bot's step, retrying the event handling of the unhandled events. However, the unhandled event list always grows as it is only cleaned up when a failed event gets successfully handled. This list will grow indefinitely if filled with ever-reverting events potentially incurring in unexpected and outstanding gas spendings (in case the retried event wastes gas each time).

Coinspect suggests including a maximum amount of retries for each unhandled event, removing this event after reaching this threshold.

Fixed on commit `fe74da8f1aefdb9250d623617e9f44d4448643aa`.

A maximum amount of retries was added. When this threshold is reached, the event is removed from the retry list.

FASO-004

Liquidators fail to track vault collateral token updates and miss unhealthy agents

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Medium

Location

`src/state/TrackedState.ts`
`src/state/TrackedAgentState.ts`

Description

The agent's tracked state does not detect collateral switches. As a consequence, it could consider that an agent is still healthy as collateral ratio calculations are made with the deprecated information.

The `TrackedAgentState` uses the following expression to calculate liquidation status transitions:

```
private possibleLiquidationTransitionForCollateral(collateral:
CollateralType, timestamp: BN): AgentStatus {
    const cr = this.collateralRatioBIPS(collateral);
    const settings = this.parent.settings;
    if (this.status === AgentStatus.NORMAL) {
```

```
        if (cr.lt(toBN(collateral.ccbMinCollateralRatioBIPS))) {
            return AgentStatus.LIQUIDATION;
        } else if (cr.lt(toBN(collateral.minCollateralRatioBIPS))) {
            return AgentStatus.CCB;
        }
    } else if (this.status === AgentStatus.CCB) {
        if (cr.gte(toBN(collateral.minCollateralRatioBIPS))) {
            return AgentStatus.NORMAL;
        } else if (cr.lt(toBN(collateral.ccbMinCollateralRatioBIPS)) ||
timestamp.gte(this.ccbStartTimestamp.add(toBN(settings.ccbTimeSeconds))
)) {
            return AgentStatus.LIQUIDATION;
        }
    } else if (this.status === AgentStatus.LIQUIDATION) {
        if (cr.gte(toBN(collateral.safetyMinCollateralRatioBIPS))) {
            return AgentStatus.NORMAL;
        }
    }
    return this.status;
}
```

Where the collateral ratio is calculated as it follows:

```
    collateralRatioBIPS(collateral: CollateralType): BN {
        const ratio =
this.collateralRatioForPriceBIPS(this.parent.prices, collateral);
        const ratioFromTrusted =
this.collateralRatioForPriceBIPS(this.parent.trustedPrices,
collateral);
        return maxBN(ratio, ratioFromTrusted);
    }

collateralBalance(collateral: CollateralType): BN {
        return Number(collateral.collateralClass) ===
CollateralClass.VAULT
            ?
this.totalVaultCollateralWei[this.agentSettings.vaultCollateralToken]
            : this.totalPoolCollateralNATWei;
    }
```

The `vaultCollateralToken` is fixed (specified when the bot starts) and does not change in the `TrackedState`, if there is a collateral deprecation, with a subsequent switch made by the owner through `AssetManager`. The FAsset protocol considers that expired collaterals will not count for the calculation of the CR:

```
// A simple way to force agents still holding expired collateral tokens
into liquidation is just to
// set fullCollateral for expired types to 0.
// This will also make all liquidation payments in the other collateral
type.
uint256 fullCollateral = CollateralTypes.isValid(collateral) ?
collateral.token.balanceOf(owner) : 0;
```

As there is no mechanism to track the current `vaultCollateralToken`, a liquidator bot will always consider that there is available collateral and won't trigger the liquidation when the deprecated token is no longer valid.

Recommendation

Handle collateral switch event.

Status

Fixed on commit `891b88965ea59cb24c02bfcf61442b8962800b20`.

The agent's tracked state now handles the `AgentCollateralTypeChanged` event triggered when the vault's collateral is changed.

FASO-005

Insecure default blocks to wait for finalization

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Location

```
src/underlying-chain/BlockchainIndexerHelper.ts
```

Description

When there is no finalization block `config` value, the code opts to use an insecure default for some chains (e.g., Bitcoin):

```
const waitBlocks = maxBlocksToWaitForTx ??
Math.max(this.finalizationBlocks, 1);
```

The default amount of blocks this function will wait until an underlying transaction is considered finalized is `1`. For Bitcoin, the suggested amount of wait blocks to consider finalization is `6`. In other words, a bot consuming from Bitcoin as an underlying chain using this default might conduct actions over a transaction in a block that has reorganization risk.

Recommendation

Use a safe default finalization value.

Status

On commit `1a8032c241d163f5ade7034b19f7f3a3a69ec666` the default was removed:

```
const waitBlocks = maxBlocksToWaitForTx ?? this.finalizationBlocks;
```

Coinspect identified that the contextual `finalizationBlocks` variable can be initialized with unsafe values (set directly in the agent's `config` file). Because the function reaching this line might have `maxBlocksToWaitForTx` as `undefined` (it is an optional parameter of the `waitForUnderlyingTransactionFinalization()` function), unsafe finalization values can still be used. If this is expected behavior, add a warning when unsafe configuration parameters are being used.

Fixed on commit `a0ace4efa472be35c53a67e3e26504783f0bd93b`.

The `finalizationBlocks` variable uses a constant value in the Attestation Client, that varies depending on each chain.

FASO-010

Airdrop distributions won't be claimed for collateral pools when the vault opts out

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Medium

Location

`src/actors/AgentBot.ts`

Description

Airdrop rewards for collateral pools can be potentially lost or unclaimed for long periods of time in the event of opting out of the airdrop. An unsuspecting owner might then schedule the agent's destruction, leaving the airdrop unclaimed.

The first call to claim the airdrop distribution makes it through the `AgentVault`. If the owner decided to `optOut` of the distribution, only in his vault, the execution will catch the revert, skipping the subsequent claim to the collateral pool:

```
try {
    // airdrop distribution rewards
    logger.info(`Agent ${this.agent.vaultAddress} started checking for
airdrop distribution.`);
    const IDistributionToDelegators =
```

```
artifacts.require("IDistributionToDelegators");
    const distributionToDelegators = await
IDistributionToDelegators.at(await
addressUpdater.getContractAddress("DistributionToDelegators"));

const { 1: endMonthVault } = await
distributionToDelegators.getClaimableMonths({ from:
this.agent.vaultAddress });
    const { 1: endMonthPool } = await
distributionToDelegators.getClaimableMonths({ from:
this.agent.collateralPool.address });
    logger.info(`Agent ${this.agent.vaultAddress} is claiming airdrop
distribution for vault ${this.agent.vaultAddress} for month
${endMonthVault}.`);
    await
this.agent.agentVault.claimAirdropDistribution(distributionToDelegators
.address, endMonthVault, this.agent.vaultAddress, {
        from: this.agent.ownerAddress,
    });
    logger.info(`Agent ${this.agent.vaultAddress} is claiming airdrop
distribution for pool ${this.agent.collateralPool.address} for
${endMonthPool}.`);
    await
this.agent.collateralPool.claimAirdropDistribution(distributionToDelega
tors.address, endMonthPool, { from: this.agent.ownerAddress });
} catch (error) {
    console.error(`Error handling airdrop distribution for agent
${this.agent.vaultAddress}: ${error}`);
    logger.error(`Agent ${this.agent.vaultAddress} run into error while
handling airdrop distribution: ${error}`);
}
```

The Agent owner can opt out of the airdrop distribution independently on each entity (vault and pool). Therefore, the owner can keep the airdrop distributions active only for the pool. The `DistributionToDelegators` contract checks upon claiming that the receiver has not opted out:

```
function _checkOptOut(address _account) internal view {
    require(!optOut[_account], ERR_OPT_OUT);
}
```

This means that the simulation of first call attempting to claim the airdrop for the Agent Vault (if opted out) will trigger an error, skipping the next claim.

Recommendation

Check that either the vault and pool have not opted out of the airdrop before claiming.

Status

On commit 1c77aa839e5c251c5d907452018530da0cf58a61 the following logic was implemented:

```
const claimableVault = await
distributionToDelegators.getClaimableAmountOf(this.agent.vaultAddress,
endMonthVault);
if (toBN(claimableVault).gtn(0)) {
    logger.info(`Agent ${this.agent.vaultAddress} is claiming airdrop
distribution for vault ${this.agent.vaultAddress} for month
${endMonthVault}.`);
    await
this.agent.agentVault.claimAirdropDistribution(distributionToDelegators
.address, endMonthVault, this.agent.vaultAddress, { from:
this.agent.ownerAddress });
}
const claimablePool = await
distributionToDelegators.getClaimableAmountOf(this.agent.collateralPool
.address, endMonthPool);
if (toBN(claimablePool).gtn(0)) {
    logger.info(`Agent ${this.agent.vaultAddress} is claiming airdrop
distribution for pool ${this.agent.collateralPool.address} for
${endMonthPool}.`);
    await
this.agent.collateralPool.claimAirdropDistribution(distributionToDelega
tors.address, endMonthPool, { from: this.agent.ownerAddress });
}
```

This structure only performs the claiming call if there are enough tokens to be claimed. However, `distributionToDelegators.getClaimableAmountOf()` fails if the claiming user decided to opt out:

```
function getClaimableAmountOf(address _account, uint256 _month)
external view override entitlementStarted
    returns(uint256 _amountWei)
{
    _checkOptOut(_account);
    _checkIsMonthClaimable(getMonthToExpireNext(), _month);
    (, _amountWei) = _getClaimableWei(_account, _month);
}
```

After an Agent decides to opt out from the airdrop distributions granted to the Vault, a revert is triggered when trying to retrieve the claimable amounts. This skips the claiming process for the Collateral Pool.

Fixed on commit 94c85a26a9018440ca08e469e1844dfd9fa76799.

Rewards and airdrop distributions are claimed independently, meaning that each claim call has its own `try-catch` logic without interfering with other claiming process in the event of failure.

FASO-011

Critical bot TXs won't be performed if stuck in the mempool

Status

Solved

Resolution

Fixed

Risk

Medium

Impact

High

Likelihood

Low

Description

The wallet implementation does not handle the possibility that transactions get stuck in the mempool. Further, in several parts of the codebase a transaction hash is awaited to continue the execution. In the event of a gas price surge, a transaction might get stuck in the mempool, disrupting the functioning of the actors.

This lack of retry logic can be identified in the following cases, for example:

`src/actors/Liquidator.ts`:

```
const fBalance = await
this.state.context.fAsset.balanceOf(this.address);
await this.state.context.assetManager.liquidate(agent.vaultAddress,
fBalance, { from: this.address });
```

```
logger.info(`Liquidator ${this.address} liquidated agent
${agent.vaultAddress}.`);
```

`src/actors/SystemKeeper.ts`:

```
await
this.state.context.assetManager.startLiquidation(agent.vaultAddress, {
from: this.address });
logger.info(
    `SystemKeeper ${this.address} started liquidation for agent
${agent.vaultAddress}. Agent's status changed from ${
        AgentStatus[agent.status]
    } to ${AgentStatus[newStatus]}.`
);
```

`src/actors/UserBot.ts`:

```
const txHash = await minter.performMintingPayment(crt);
logger.info(
    `User ${requireEnv("USER_ADDRESS")} paid on underlying chain for
reservation ${
        crt.collateralReservationId
    } to agent's ${agentVault} with transaction ${txHash}.`
);
```

As the gas price is always estimated, bot operators have no control or flexibility over this parameter. A sudden gas price surge will make any transaction stuck in the mempool until the price lowers. Any other transaction sent after the stuck one will not be executed, even if the gas price is met. This would violate the correlative nonce rule (considering that the transaction with a lower nonce is still in the mempool).

All bots perform time sensitive actions requiring celerity and proper control of their execution queue.

Recommendation

Design a retry logic. It could be implemented via transactions replacement by nonce when possible. Make sure that duplicates or other types of transactions - susceptible to a challenge through this retry/replacement process - are note created.

Status

Fixed on commits:

- Simple Wallet: `b309bf3bc0cb7f101137afdbd669408cdc95ead5`, `335ddd7c954e4e9859ea88ebb287b6d2129e4b1b`.
- FAsset Bots Native: `6de1c3c312552fb638251ae0ae7007f0ad75df01`, `e8e5c77095b112a8218e7b976428013741747be7`.

The Flare Team added a retry logic for each chain's wallet using specific triggers for each.

FASO-012

Reverts during even processing result in corrupted TrackedState

Status
Solved

Resolution
Fixed

Risk
Medium

Impact
High

Likelihood
Low

Location

`src/state/TrackedState.ts`

Description

Several actors get the unhandled events from the `TrackedStates` main loop, if this execution has a revert at some point, the execution will continue and will skip the unprocessed events. This scenario might leave several internal variables outdated that don't reflect the current state of the `FAsset Smart Contract Protocol`.

Skipping or altering the event processing could lead to corrupted states. As different events might inter-depend on states (e.g., two different events that update key variables used for collateral rate calculation), the consequences can be critical.

All actors but the `AgentBot` get the events from the `TrackedStates`. This process first listens events from several sources (via `readUnhandledEvents`) and, before returning the collected events, the internal tracked states are updated through `registerStateEvents`.

```
async readUnhandledEvents(): Promise<EvmEvent[]> {
    logger.info(`Tracked State started reading unhandled native events
FROM block ${this.lastEventBlockHandled}.`);
    // get all needed logs for state
    const nci = this.context.nativeChainInfo;
    const lastBlock = (await web3.eth.getBlockNumber()) -
nci.finalizationBlocks;
    const events: EvmEvent[] = [];
    for (let lastHandled = this.lastEventBlockHandled; lastHandled <
lastBlock; lastHandled += nci.readLogsChunkSize) {

/// Event listening logic

}
    // mark as handled
    this.lastEventBlockHandled = lastBlock;
    // run state events
    events.sort((a, b) => a.blockNumber - b.blockNumber);
    logger.info(`Tracked State finished reading unhandled native events
TO block ${this.lastEventBlockHandled}.`);
    await this.registerStateEvents(events);
    return events;
}
```

```
    async registerStateEvents(events: EvmEvent[]): Promise<void> {
        try {
            for (const event of events) {
                /// State updates according to each event

for (const collateral of this.collaterals.list) {
                    const contract = await
tokenContract(collateral.token);
                    if (eventIs(event, contract, "Transfer")) {
                        logger.info(`Tracked State received event
'Transfer' with data ${formatArgs(event.args)}.`);

this.agents.get(event.args.from)?.withdrawVaultCollateral(contract.addr
ess, toBN(event.args.value));

this.agents.get(event.args.to)?.depositVaultCollateral(contract.address
, toBN(event.args.value));

this.agentsByPool.get(event.args.from)?.withdrawPoolCollateral(toBN(eve
nt.args.value));

this.agentsByPool.get(event.args.to)?.depositPoolCollateral(toBN(event.
args.value));
                    }
                }
            }
        } catch (error) {
            console.error(`Error handling events for Tracked State:
${error}`);
            logger.error(`Tracked State run into error while handling
```

```
events: ${error}`);
        }
    }
```

If at some point of the main event loop in `registerStateEvents` a revert is triggered, the remaining events will be skipped and the execution will finish in the `catch` branch. Afterwards, the collected events are returned from `readUnhandledEvents`.

It is worth noting that, at the moment, Coinspect did not find any way to trigger a revert when processing events. However, this could happen if new code is added, or a dependency/library is updated and an issue is introduced.

Coinspect advises bulletproofing this loop in order to obtain a more resilient procedure.

Recommendation

Do not skip over remaining events when there is an exception in one event. If many order-sensitive events must be processed atomically, revert the execution and prevent consumers from using potentially corrupted states.

Status

Fixed on commit `e4bad21cbbe330df0fcf99f29be9e9c38f427196`.

Coinspect asked to the Flare Team about the re-initialization process as it discards all previously stored states. The Flare Team responded that in case of data corruption, users can effortlessly restore the data of previously created agents using the `getAgentInfo` method. This process also allows users to gradually rebuild the `TrackedState` from the latest block, for newly created agents.

FASO-013

Weak wallet encryption/decryption passwords are supported

Status
Solved

Resolution
Fixed

Risk
Low

Impact
Medium

Likelihood
Low

Location

`src/underlying-chain/WalletKeys.ts`
`src/utils/encryption.ts`

Description

When adding a new account to the database, an encryption password is required. However, there are no complexity requirements for this password, allowing for weak encryption passwords.

Furthermore, the code uses `SHA256` to hash the password. `SHA256` is not meant for password protection and a PKDF algorithm such as `argon2` or `scrypt` should be used instead.

`src/underlying-chain/WalletKeys.ts`:

```
async addKey(address: string, privateKey: string): Promise<void> {
    if (await this.getKey(address)) return;
    // set cache
    this.privateKeyCache.set(address, privateKey);
    // persist
    const wa = new WalletAddress();
    wa.address = address;
    wa.encryptedPrivateKey = encryptText(this.password, privateKey);
    await this.em.persist(wa).flush();
}
```

`src/utils/encryption.ts`:

```
export function encryptText(password: string, text: string): string {
    const passwordHash = crypto.createHash("sha256").update(password,
"ascii").digest();
    const initVector = crypto.randomBytes(16);
    const cipher = crypto.createCipheriv("aes-256-gcm", passwordHash,
initVector);
    const encBuf = cipher.update(text, "utf-8");
    return Buffer.concat([initVector, encBuf]).toString("base64");
}
```

As there are no complexity checks, weak or reused passwords are allowed. In the event of a database leak, adversaries could potentially brute-force the encryption trying to get the account's private key.

Recommendation

Enforce a minimum password length. Also, Coinspect suggests that the password be randomly generated.

Change the algorithm to `scrypt` or `argon2`.

Status

On commit `12905c51e761a6be59885ff4545dc4d8f21bd449`:

The password complexity is now checked. However, the hashing algorithm on `src/utils/encryption.ts`, is still `sha256`.

Fixed on commits `468baef6a3ea777cde8e17001ef0ef0a17a29f18` and `2b615cffb958a7a9a742291fc9472a8767af73e7`. Secrets now can be generated automatically and the hashing algorithm was changed for `scrypt`. /--

FASO-014

Multiple attempts may be required before successfully creating a new Agent

Status
Solved

Resolution
Fixed

Risk
Low

Impact
Low

Likelihood
Medium

Location

`src/fasset/Agent.ts`

Description

When creating new agents, the default index value used to build the token name is zero. This means that the creation process is forced to loop over every single index, relying on the transaction's simulation revert to increase the index:

```
static async create(ctx: IAssetAgentBotContext, ownerAddress: string,
agentSettings: AgentSettings, index: number = 0): Promise<Agent> {
    const desiredErrorIncludes = "suffix already reserved";
    try {
        const response = await
ctx.assetManager.createAgentVault(web3DeepNormalize(agentSettings), {
from: ownerAddress });
        // more create logic
    } catch (error: any) {
        if (error instanceof Error &&
```

```
error.message.includes(desiredErrorIncludes)) {
            index++;
            agentSettings.poolTokenSuffix =
this.incrementPoolTokenSuffix(agentSettings.poolTokenSuffix, index);
            return Agent.create(ctx, ownerAddress, agentSettings,
index);
        } else {
            throw new Error(error);
        }
    }
}
```

This means that when creating new agents, those willing to run the Agent Bot cannot opt to start looping over greater indexes to speed up this process. Also, in the event of a simulation failure, the transaction would be sent, wasting gas.

Recommendation

Allow Agent operators to specify a suffix index to start its creation.

Status

Fixed on commit `d3c6a717d4fe55725e5d8b80be167536ff9089b4`.

The recursive structure triggered when a token index (name) was taken when creating an agent was removed. Now, users are required to set the token index as an input. In other words, if for some reason the creation fails (e.g. the token name was already taken), users will need to set a new token index in their agent creation settings `config` file.

FASO-015

Weak test coverage increases exposure to attacks and adversarial scenarios

Status
Solved

Resolution
Fixed

Risk
None

Impact
Recommendation

Likelihood
–

Location

```
test/
test-hardhat/
```

Description

Several functions and branches are not tested. Keeping an exhaustive and complete test suite reduces the likelihood of encountering bugs in production.

The Hardhat's coverage report shown that the coverage could be improved:

```
=============================== Coverage summary
===============================
Statements    : 84.14% ( 2945/3500 )
Branches      : 73.12% ( 860/1176 )
Functions     : 84.03% ( 521/620 )
Lines         : 84.24% ( 2855/3389 )
```

```
================================================================================
=========
```

Coinspect identified that some functionalities, for example Agent's top-ups are not properly tested.

Recommendation

Increase the overall coverage to 95% or more.

Status

Fixed.

The Flare Team stated that an overall coverage of 97% is reached when running both Hardhat and E2E tests.

FASO-016

Agents can steal underlying balance sending more than fifty transactions

Status
Solved

Resolution
Acknowledged

Risk
None

Impact
Recommendation

Likelihood
–

Description

Only a set of 50 unprocessed transactions are taken into account to calculate the negative underlying balance challenge. A malicious agent with high traffic of operations can accumulate 50 redemptions, and pay them in a single block.

To exploit this, an agent must have an active withdrawal announcement, putting the withdrawal transaction in the 51th place. The global value of this transaction can be disguised by leveraging `FASO-007`:

```
    transactions.sort((a, b) => (a.spent.gt(b.spent) ? -1 :
a.spent.lt(b.spent) ? 1 : 0));
    // extract highest MAX_REPORT transactions
    transactions = transactions.slice(0, MAX_NEGATIVE_BALANCE_REPORT);
    // initiate challenge if total spent is big enough
    const totalSpent = sumBN(transactions, (tx) => tx.spent);
```

Where the constant is `MAX_NEGATIVE_BALANCE_REPORT == 50`.

Coinspect was not able to exploit this vector, and concluded that this is because the fees generated by the 50 transactions exceed the lowest redemption's value (a draining `tx` would be caught by ordering unprocessed transactions by value). However, this path could become exploitable if a parameter that impacts on the underlying balance generation is changed in the future.

Recommendation

Be aware of this scenario when tuning or changing a parameter that affects the underlying balance generation/accumulation.

Status

Acknowledged.

The Flare Team stated:

```
The limit of 50 is due to gas limit in the fasset contract
freeBalanceNegativeChallenge, so we cannot fix it. Anyway, all
redemption transactions must eventually be reported (because otherwise
the agent loses redemption collateral after 1 day), so the draining can
be only temporary.
```

Disclaimer

The information presented in this document is provided “as is” and without warranty. Security Audits are a “point in time” analysis, and as such, it's possible that something in scope may have changed since the tasks reflected in this report were executed. This report shouldn't be considered a perfect representation of the risks threatening the analyzed systems and/or applications in scope.

Appendix

Appendix A: IBlockChainWalletMultipleUTXOs, UTXO and SpentReceivedObject

```
export type UTXO = {
    value: NumberLike;
    // ... Add any other properties you want, like txid, vout, etc.
};

export type SpentReceivedObject = {
    [address: string]: UTXO[];
};

export interface IBlockChainWalletMultipleUTXOs {
    // Create a transaction with a single source and target address.
    // Amount is the amount received by target and extra fee / gas can be
added to it to obtain the value spent from sourceAddress
    // (the added amount can be limited by maxFee).
    // Returns new transaction hash.
    addTransaction(
        sourceAddress: string,
        targetAddress: string,
        amount: NumberLike,
        reference: string | null,
        options?: TransactionOptionsWithFee,
        awaitForTransaction?: boolean
    ): Promise<string>;

// Add a generic transaction from a set of source addresses to a set of
target addresses.
    // Total source amount may be bigger (but not smaller!) than total
target amount, the rest (or part of it) can be used as gas/fee (not all
need to be used).
    // This variant is typically used on utxo chains.
    // Returns new transaction hash.
    addMultiTransaction(spent: SpentReceivedObject, received:
SpentReceivedObject, reference: string | null): Promise<string>;

// Creates a new account and returns the address.
    // Private key is kept in the wallet.
    createAccount(): Promise<string>;

// Add existing account.
    // Private key is kept in the wallet.
    addExistingAccount(address: string, privateKey: string):
Promise<string>;
```

```
// Return the balance of an address on the chain. If the address does not
exist, returns 0.
    getBalance(address: string): Promise<BN>;

// Return the current or estimated transaction fee on the chain.
    getTransactionFee(): Promise<BN>;
}
```

Appendix B: MockChainWallet

```
// UTXO implementation
export class MockChainWallet implements IBlockChainWalletMultipleUTXOs {
    constructor(
        public chain: MockChain,
    ) { }

async getBalance(address: string): Promise<BN> {
        return this.chain.balances[address] ?? BN_ZERO;
    }

async getTransactionFee(): Promise<BN> {
        return this.chain.requiredFee;
    }

addExistingAccount(): Promise<string> {
        throw new Error("Method not implemented.");
    }

async addTransaction(from: string, to: string, value: BNish, reference:
string | null, options?: MockTransactionOptionsWithFee): Promise<string> {
        const transaction = this.createTransaction(from, to, value,
reference, options);
        this.chain.addTransaction(transaction);
        return transaction.hash;
    }

async addMultiTransaction(spent: SpentReceivedObject, received:
SpentReceivedObject, reference: string | null, options?:
MockTransactionOptions): Promise<string> {
        const transaction = this.createMultiTransaction(spent, received,
reference, options);
        this.chain.addTransaction(transaction);
        return transaction.hash;
    }

createTransaction(from: string, to: string, value: BNish, reference: string
| null, options?: MockTransactionOptionsWithFee): MockChainTransaction {
        options ??= {};
        value = toBN(value);
        const maxFee = this.calculateMaxFee(options);
        if (maxFee.lt(this.chain.requiredFee)) {
            // mark transaction failed if too little gas/fee is added (like
```

```
EVM blockchains)
            options = { ...options, status: TX_FAILED };
        }
        const success = options.status == null || options.status ===
TX_SUCCESS;
        const spent = success ? value.add(maxFee) : maxFee;
        const received = success ? value : BN_ZERO;

const spentObj: SpentReceivedObject = { [from]: [{ value: spent }] };
        const receivedObj: SpentReceivedObject = { [to]: [{ value: received
}] };

return this.createMultiTransaction(spentObj, receivedObj, reference,
options);
    }

createMultiTransaction(spent_: SpentReceivedObject, received_:
SpentReceivedObject, reference: string | null, options?:
MockTransactionOptions): MockChainTransaction {

const inputs: TxInputOutput[] = Object.entries(spent_).flatMap(([address,
utxos]): TxInputOutput[] => {
            return utxos.map(utxo => [address, toBN(utxo.value)]);
        });

const outputs: TxInputOutput[] =
Object.entries(received_).flatMap(([address, utxos]): TxInputOutput[] => {
            return utxos.map(utxo => [address, toBN(utxo.value)]);
        });

const totalSpent = inputs.reduce((a, [_, x]) => a.add(x), BN_ZERO);
        const totalReceived = outputs.reduce((a, [_, x]) => a.add(x),
BN_ZERO);

const status = options?.status ?? TX_SUCCESS;

if (!totalSpent.gte(totalReceived)) fail("mockTransaction: received more
than spent");
        if (!totalSpent.gte(totalReceived.add(this.chain.requiredFee)))
fail("mockTransaction: not enough fee");

const hash = this.chain.createTransactionHash(inputs, outputs, reference);

console.log(`Tx Status: ${status}`)
        console.log(`Hash: ${hash}`)
        console.log(`Reference: ${reference}`)

console.log(`\nINPUTS:`)
        for (let input of inputs) {
            console.log(`spender: ${input[0]} - value: ${input[1]}`)
        }
        console.log(`Total Spent: ${totalSpent}`)

console.log(`\nOUTPUTS:`)
        for (let output of outputs) {
            console.log(`recipient: ${output[0]} - value: ${output[1]}`)
        }
        console.log(`Total Received: ${totalReceived}`)

console.log("\n");
```

```
        return { hash, inputs, outputs, reference, status };
    }

async createAccount(): Promise<string> {
        const accountId = Math.floor(Math.random() * 100000) + 1;
        return `UNDERLYING_ACCOUNT_${accountId}`;
    }

private calculateMaxFee(options: TransactionOptionsWithFee) {
        if (options.maxFee != null) {
            return toBN(options.maxFee);
        } else if (options.gasLimit != null) {
            return toBN(options.gasLimit).mul(toBN(options.gasPrice ??
this.chain.estimatedGasPrice));
        } else {
            return toBN(this.chain.requiredFee);
        }
    }
}
```

File hashes

FAsset Bot Directory

```
a2ce8069ffcc4191040b978e58dc4a544930c7e74e584b69d480ebe23cf61562  ./src/bot-api/agent/agentServer.ts
8de6562e26e7a50ba978f1eecfc2e90efa26687ca31ff9ca75ab416da90aa78d  ./src/bot-api/agent/main.ts
10f6caa45450ea641b846969b46a2c07f7a22e526dfd7fe3b6cec67f5051900c  ./src/bot-api/agent/auth/auth-
header-api-key.strategy.ts
f40d61972228c4542b00d8b27450a32ca698c0d6ac8998567a2b121f07787403  ./src/bot-
api/agent/auth/auth.module.ts
85677dbe82510d71b66963a59408545ac7aee6c76f49b860cc8e858aa41862de  ./src/bot-
api/agent/controllers/agent.controller.ts
ecc5d029add27cfcb3aaef6ca6fe8fc1d2dfd570347c94b87fadfc70b1b45096  ./src/bot-
api/agent/controllers/pool.controller.ts
131e52784dbba1d6c543bae4584e90292c360d294709e762b5d17994d5507053  ./src/bot-
api/agent/controllers/underlying.controller.ts
7280b73de5489c3192d2e2a46f7df376e64740924c18b9ebe1dedfeb5588a8ee  ./src/bot-
api/agent/controllers/vault.controller.ts
d3718977db4aa99eff2a56665bf559e39c5f4c95a2c06fa66a719c8a36d52be3  ./src/bot-api/agent/agent.module.ts
aea93084be440a53d5409df8ac1d1f7578f3f1c08579e6e1135f31916007ce96  ./src/bot-
api/agent/services/agent.service.ts
588b6434abbb2cc70cbd54ff9c05a903d2f6979c5a819d92cca4cba3ecbdc181  ./src/bot-api/common/ApiResponse.ts
4b96c008142d69f39f3f41deb3479770e9d0b6eee9959c6c8e4de590507f2abd  ./src/bot-
api/common/AgentResponse.ts
942a5214e3b6ce38d16cc8cc8c722e86f40c7ef2e04c5b3227bc982e0f6a71c8  ./src/config/create-asset-
context.ts
136052b5d1ba7123fd48c09586d4d47007689b53a114633681c072b5cc4a68cf  ./src/config/json-loader.ts
85162db04dff20615d4c031937477f71f39efd2c46ea71541a964d4c9abd72ae  ./src/config/config-files.ts
1f68b6984c80a87738a8dd59ce14ca23079266e00fed4fd9c90e36e9e00707d4  ./src/config/contracts.ts
33e7dbe4d333d0f93fe49469bf5686349ad389480122bd4106b409fd23f03f4e  ./src/config/orm.ts
7d4c1198f89592d05a0c9965072d5b20418e2a347e6b3d86a825534b1a352f8f  ./src/config/orm-types.ts
b15a30b6209179a94d6975bfb03034defa8402559a9e1cb1a736cec16f8b157d  ./src/config/BotConfig.ts
0400e8bbef59b3ea146ac71f5d8fdb84135b10423e44518fa5efa0fac43ab8d0  ./src/utils/helpers.ts
1dbbd7ea4aa379d09d0f580c317ce21f5c2cdddc6edb2380487bacb000298a5c  ./src/utils/web3helpers.ts
8f77a986d0683ebb5684e6c8f0ca64b148eda063f63c0e3a9721694467d6ce35  ./src/utils/web3normalize.ts
```

```
d68fe29bb1780d42fb7ed325bc8497d59dc39fb0837f74ced94d2292da6a064e  ./src/utils/web3.ts
2470b7dd8679291afafea3ee9ec3704544e0a9c82b985d42444324feec83f57a  ./src/utils/logger.ts
744ce88b115d54699c60c4531b161383ca4ccb39493468f5634c9baa560b91ce  ./src/utils/formatting.ts
683f2a83ab1daae5a57e668cb4714b39e034abc5897892f33d1570feedf2c14b  ./src/utils/Notifier.ts
3a76aa1cf1e0cbda04930a484f0850084e872b79c9b8599206d7e9beaa3c0876  ./src/utils/printlog.ts
0486c0e6a6e8de1726121a0e8419d4dc27dbe74a85f74afc80d559e0d7a90bd4  ./src/utils/MerkleTree.ts
1be1ea1da8e92720fa90fcf97258883493566e430d6e191c0c203179fc2beb6c  ./src/utils/events/ScopedRunner.ts
2c16bf611417e4383bf139c190ce48a5134d579368e6292663ae89878450656b
./src/utils/events/Web3EventDecoder.ts
fff95bf0c355bc2feef31c6ab97d89e80b75ecaeadad3ca3a85a04b09ec6103f  ./src/utils/events/common.ts
9fc1f962d8931cfa73685f2481aa0e9603614ef0abe2d99ad8304c0c96fd55d3  ./src/utils/events/ScopedEvents.ts
a41988a70431ef775fb3cb5136c6c2bde80829a3f0356811c0ec498c6f9640f7
./src/utils/events/Web3ContractEventDecoder.ts
5b2ff9b68ab3e91b813b4940c568e5535cb2ffdc1662cf40a872fba5518cdfc9  ./src/utils/events/truffle.ts
2785b7feb016df55c1b9d5432d41641a877290c66ab85a54db0e15a0b37bbac8  ./src/utils/encryption.ts
1488116b8e872bd482f25e1521efa6e954b36e384130711d40c64eaee34506bf  ./src/utils/mini-truffle-
contracts/cancelable-promises.ts
7bf6c94ab82d2d6f741fdab374ee4aa2aff526d723fb56eaca1fdd9441f90ff5  ./src/utils/mini-truffle-
contracts/artifacts.ts
7bb5e5d46371be84155d4b4d7d3da079d7534a6c12bd6b76fb3d87df727ec24b  ./src/utils/mini-truffle-
contracts/contracts.ts
878b4eb1b9268eb77e2d8e2aa9451841fa9140aab2a529302baca48e9d568dd5  ./src/utils/mini-truffle-
contracts/types.ts
7474ba6f497979b994cc29b016b845a77209816d385d97957d5c79d68dba4dfb  ./src/utils/mini-truffle-
contracts/finalization.ts
6b671abdae04c8114e4866e648203524f5ceb2ca58d13758b705d403bbdc3e54  ./src/utils/mini-truffle-
contracts/methods.ts
ec268e6ba8a2cedc70f1c16c245568305a9b9e79264f57ac2858ed841517c827  ./src/utils/fasset-helpers.ts
1a7f8b30b86ebba2708b2ce6c41d8e9417556c9e77bf6f5e1ddaf3ebdd521524
./src/utils/StaticAttestationDefinitionStore.ts
f1d841ebf4b4c5b8b01fa949f5bfb8e5ee11a5a738da08e67d85ce23ff6d74dc  ./src/cli/apiKey.ts
200b249d05173ceaceb235e20d802529f9cacf0b8b0c104be3287ee069b9973f  ./src/cli/fakePriceReader.ts
5a36565ea0020be28ad9820ffeaf86b436dfa3d9d2705597204b3c9070bd2c29  ./src/cli/agent.ts
a4ce28c947e699cec5e05d06491ad1a95795205578736bfbf73b63195f3c03b0  ./src/cli/testGovernance.ts
0096fd7e5c8641ffcea2ce62661f62a8912e31689dae162f1bce175d9d7e5ca1  ./src/cli/user.ts
1a48cacf431a4e07856e2946e3877a83cc7934e179e80f24f4aa1107916bf3cf  ./src/state/Prices.ts
d130458ac826aeb56ab6daafddd5a600a4cc7913b14244e8c4ab34dd00a967c5  ./src/state/CollateralPrice.ts
154ea56cf54d5e6468a63f7d9aeb407cd4df2e3a71b5100bd3dd076124052882  ./src/state/TokenPrice.ts
d8b08273aa08f20d8344a4a049adc0624a065d863b883ff5442fd494d6a36574  ./src/state/TrackedState.ts
b3ac6d2193d19b3582aa0e037b7ba2798a44647606a9fdcbcf1504b15c5d718f  ./src/state/TrackedAgentState.ts
881b9f62a06bd098a6579de3c60ae9c752357f137f0e6bf61f711812d5bc17e9
./src/state/CollateralIndexedList.ts
8e0fca82650f853b9381888c0838d6c657931d8ba272cbd1d16f0e3f40963d83  ./src/mikro-orm.config.ts
3a5a924d95e1a8c50576130c351be2a94d673acef36d65b88be416b6c7504074
./src/verification/generated/attestation-types-enum.ts
3195b89ac0f38583fd09113034bcdcd462d5a8486b74d1d44e8ca7207498f91e
./src/verification/generated/attestation-request-types.ts
5f9cc7d8d4e66eb2b1308d790d623e66608dcdd1df37b349166aa7d156ee0e52
./src/verification/generated/attestation-random-utils.ts
164c015865c7828f61da1200b633076cb54d9219d218b0f757ff5c30f11c6213
./src/verification/generated/attestation-hash-types.ts
769ece6cb9187be7b3fea437ac1fac6b7ec8c2e7af119bd25402e3047dcfef42
./src/verification/sources/sources.ts
c9c69dbeae8fd9dc8a87574903c440eb80db9704c7db465f34fb5c3605371611  ./src/verification/attestation-
types/t-00004-referenced-payment-nonexistence.ts
b5d99e2b7818385ec2b43ae31bbbf19dba9fce65776843d4a814e51d08ba0aa0  ./src/verification/attestation-
types/attestation-types.ts
b6fb74dd19630bbc8d7457034dd88e8e98991bc50908fb56010f9a3523e483e0  ./src/verification/attestation-
types/attestation-types-helpers.ts
078c772c21f3e0ba1e9f79f8738ce5ca55d581a61d866dc8b59efb298eb18817  ./src/verification/attestation-
types/t-00002-balance-decreasing-transaction.ts
cb2665f21baf160ff6e98623448ca073687eb852755ec0d11ce17a554845d3d0  ./src/verification/attestation-
types/attestation-types-utils.ts
7fba82cd1c2b02a099fb77abd9f13d52c209fd96f11dc72d1971e4639fd214cd  ./src/verification/attestation-
types/t-00003-confirmed-block-height-exists.ts
7211c1edc051f0c023c8eca866372b5c6337d6dd630423561b335b4c2e2ce4be  ./src/verification/attestation-
types/AttestationDefinitionStore.ts
a7a33eb11430c482901b67eb461cdaf0ca59b5271daa7011f49c9a78593f7fc3  ./src/verification/attestation-
types/t-00001-payment.ts
```

```
38f18be41552c2019940e70b47c0949d2183af84905f144a43ca1a93a26826b1  ./src/verification/attestation-types/verifier-configs.ts
f895a891da8a1c5603e8fce7ce6d6eb359e514390bd4db683325d906f34dec39  ./src/underlying-chain/AttestationHelper.ts
e241423dfa07197298f6fa665e0aaa486a5b875655db57b3571386fc9144f2f0  ./src/underlying-chain/StateConnectorClientHelper.ts
99d85eda14370656acd6ddba65b0f27795eeafba64e844b9aca223d66d3cacd0  ./src/underlying-chain/BlockchainWalletHelper.ts
cbf44276079dd9f5f0d2c579250886bd3e03975b79db035fe7ea9e34b7277819  ./src/underlying-chain/BlockchainIndexerHelper.ts
f1e1ffbb196a6270db7d6e16a81a08995947fcc93e806ba0e4173d8bd16946de  ./src/underlying-chain/WalletKeys.ts
13d71c683328b8087c9ddb1a54e6431a1f54ccd7c923fec15003ccc9c5b9c881  ./src/underlying-chain/interfaces/IBlockChain.ts
f774b66c872846dbbd783310e318253754304e1b0f1d5bbc31eec833d55f5b8d  ./src/underlying-chain/interfaces/IStateConnectorClient.ts
79b735a082ea732e26277232abd033b17b845df701d79747113aef0e70342277  ./src/underlying-chain/interfaces/IBlockChainWallet.ts
dbfd4094d921650db4965ae79145be3fe145b6f19d117e9c3ff2645f2c49d3b2  ./src/fasset/PaymentReference.ts
adc0611588b93ef4f544c6f974d438a2a8f7e4b9334086d4c919f07867e335dc  ./src/fasset/Agent.ts
71ffd7c9ac7f26d2e02e925988c0cd8f18bcc2af5c8d9c5c4a17670ea7a927e8  ./src/fasset/AssetManagerTypes.ts
848d31006d3c21901a8554a071081ee04b763ca274bc0781135991b8a6d80ba5  ./src/fasset/ChainInfo.ts
9a2e8636acecff77af2c6fa3a695f89cdb02eb9a9c37b54e2131e50c1c88606d  ./src/fasset/LiquidationStrategyImpl.ts
5749414f11a2b8dc84369375c6949a01d14f85f5707007c3711bf656befbab13  ./src/fasset/Conversions.ts
ab9adabf7c0c4f28e5e1db85b2cde0a2e8987fda0944c3d50167ee41798dd568  ./src/fasset/CollateralData.ts
723d4fcf2a8b91f9517f9881ce82c6ae4e8054dca433474468fcfa3cc62fe9e8  ./src/actors/TimeKeeper.ts
12c170debe05f620820e0cac5e5f9038aec786f6ed18646f9f9fd7eb57fe9182  ./src/actors/ActorBaseRunner.ts
9edfb5a94072e9975c3050e912b3f27457a46e88855da2235768ea9fce0e9844  ./src/actors/Challenger.ts
7e8a60bfc307c4d7927a1a9dded65ae643fa5a1d9e6228314a260286a31c1fae  ./src/actors/Liquidator.ts
c2a5b99b0c0b99309f0b1fab2f76652a2d1745fa5ea4933686404ce6475a8c6b  ./src/actors/UserBot.ts
33453461e1808f7b447ae7b829897abaedb95de539fe61c53ae197505585630d  ./src/actors/AgentBot.ts
6a5ba019e5dcda6aa2b41e8dc500f0950579623f785b7a665ddac6a5d4da4536  ./src/actors/AgentBotRunner.ts
4be30bcc9f528c15cd61a01eba594c51353816f83d22a21dab05ced1a22ef003  ./src/actors/AgentBotCliCommands.ts
2e047c43b7c2aca9ea6218fbf92b2601f6364169c14983022c0d6a9c98a0e553  ./src/actors/SystemKeeper.ts
a85ad26f8d2330fe3838d65443bcecfa75987a79ea31b92e7f1a93e094aa8fdf  ./src/run/run-agent.ts
0ccda74534ba3b41b19f29bd0903b901fcdd120c746c29d25598f300b89780c4  ./src/run/run-systemKeeper.ts
1ef916f32bbf43a0c3dd90d94d6c87b145fac3564b04ccf1df59e62ea4ede65f  ./src/run/run-challenger.ts
a6a70cb80b225853a7238047be1fca10a476050930575e421ca7f833128b7af5  ./src/run/run-liquidator.ts
894d320b26d1092c78d15f924fd14e2f6d5db6869e9316dbee7881293335ff7f  ./src/run/run-timekeeper.ts
63901e319fe2ec848d9ca327c62255ee5052c77382cde638cb9af6bf79508f05  ./src/entities/wallet.ts
8635e4c1fc4d84c7a860ee7c0db3d47849b8ca7914b47f0be8a6a2aefa3a9a78  ./src/entities/agent.ts
e5f9f2468211795a7fb288abaf38576355ed4dcc8129a1041248b77196259c17  ./src/entities/common.ts
9358838f1669908a0062ca202dc09890c0e68cf2c598a3a71c83941923a1b3ba  ./src/fasset-bots/ActorBase.ts
d297391c68319e5eb367777c2f861cf56b7366f0402c206678d28db3f9ff9b95  ./src/fasset-bots/IAssetBotContext.ts
```

Simple-Wallet

```
6fed8cc2a47d2ecbd000a27052a1ea09297b16f4b875d191c63cab4180eb8c96  ./src/@types/bitcore-lib-ltc/index.d.ts
af69589d856b0f8bff3ae3ad9f49a16b8886c55fddad3cf17d59ca5837377f6d  ./src/@types/wallet-address-validator/index.d.ts
e0bee33166a46941c2ddd9173471e5f3da6a3ec0d6774f26de614b771c44c789  ./src/@types/bitcore-lib-doge/index.d.ts
1748226e6745f01deab8ffd3e6f610e52300f3592e707a18820af04df7f9680e  ./src/chain-clients/AlgoWalletImplementation.ts
b754aa168b015d26a78c9aeddb639bbb33a51597b6e33c60bda1cd2c05370c31  ./src/chain-clients/BtcWalletImplementation.ts
1262097499a0d9828c9ef96048b31c727369fc1e7c00704f69a291397f465fa7  ./src/chain-clients/UtxoCore.ts
f75acb0b8e14b713f2cc08d2d6e13cdcd2172956c22506f814cf066c26d91968  ./src/chain-clients/LtcWalletImplementation.ts
48d81e9f3ff21f1d4629f8b5344b3a14bc964eb6c9ab14ce041b64e196c4b79d  ./src/chain-
```

```
clients/DogeWalletImplementation.ts
9f818dea0da211b6b29d5563e5d68e28cbc4abc7c8462921f5e7b3111a684fdd  ./src/chain-
clients/XrpWalletImplementation.ts
45869c74842aa9582c4f901e36cdf12241fef85973416529e65487834e209beb  ./src/utils/utils.ts
ea5c43eec6d9553465c4a961943f244ee15f53f78db6b68aa280e4a000fd0fc6  ./src/utils/constants.ts
57d7723b863cc58201803a34a73f08cba3dccd8f46570d4f55455f591c25115b  ./src/types.ts
941df5c2735e9eac216660a2d95d3ae3b31833371dd75294b746dd8e8c564a57  ./src/index.ts
ca18af22cae610b8e16b7d7f296a1f8095236156a85744c02e2a86e7a56fc285
./src/interfaces/WriteWalletRpcInterface.ts
```