Flare fAsset

Smart Contract Audit

coinspect

Flare

fAsset

Smart Contract Audit

V220829 Prepared for Flare • June 2022

1. Executive Summary

In **June 2022, Flare** engaged Coinspect to perform a source code review of **fAsset**. The objective of the project was to evaluate the security of the smart contracts.

The audit focused on the implementation correctness of the fAsset system smart contracts.

The design of the system and other components of the Flare network on which fAsset relies were assumed trusted and its incentives aligned with the fAsset system. These dependencies include the FTSO oracles used to price assets against the collateral, and the attestation providers that prove the presence or absence of transactions and their expected properties in the underlying chains.

The fAsset system security also depends on the correct functioning of the off-chain code responsible for critical tasks, such as underlying chain monitoring, reporting of illegal payments and tracking pending payments. However, the off-chain code was not in scope for this engagement.

As with other similar systems, the security of the funds is highly dependent on how the configurable parameters are selected.

The following issues were identified during the initial assessment:

High Risk	Medium Risk	Low Risk
3	4	0
Fixed 3	Fixed 2	Fixed 0

FAS-1, **FAS-2** and **FAS-5** allow attackers to move the underlying funds without being penalized, by exploiting errors in the implementation to bypass the protection offered by challengers. **FAS-7** describes a scenario where redeemers could not receive the expected amount of collateral after burning their fAssets. **FAS-4** shows how agents could grief challengers in order to discourage them from reporting illegal payments. **FAS-6** is related to an assumption regarding the agents' underlying EOA address that could be abused to unfairly liquidate them.

2. Assessment and Scope

The audit started on **June 13 2022** and was conducted on the **fasset-audit** branch of the git repository at https://gitlab.com/flarenetwork/fasset as of commit **3e1acb12978d5913102746d91b7892bbeb99baae** of **June 21 2022**.
The audited files have the following sha256sum hash:

```
5c466adb97e4839475718c235e82420c26993b77e8bf284d66bc760358a12838  ./generated/contracts/AttestationClientMock.sol
7d0e0aa5402a0825e9dd8c8ef997bfe94f6d5187603d320a465268a932cc2081  ./generated/contracts/AttestationClientBase.sol
12e153d52f292348464cfabb29e8cab5903d997305caf6d740b9dbd4475c240c  ./generated/contracts/StateConnectorMock.sol
bea344e023852265e0b3e4e216f5819b90e42444a9c0443c8cd015354f0ddb40  ./generated/contracts/AttestationClientSC.sol
ad138cdbc714438fbadb30fb2642dd179b4658de7f55cc1611508a8680129671  ./generated/interface/IAttestationClient.sol
9e3a2676b35ce795387e7d86b8428cf0cc5d70ee1c570781ebfd9318fb5f0ee8  ./generated/interface/IStateConnector.sol
0393db48b434ef7e973781224acf2c89c4a41889cd11e0faede6502ab159b021  ./fasset/mock/FtsoRegistryMock.sol
816a7880e5da49f457bef40c9ae3180f6f3034d35ab2a805e93e08ea3bac8ad6  ./fasset/mock/ERC20Mock.sol
9993a7ea512cadd3a18b06dd4bd23e87d53095817738410c47fc81d5382bd841  ./fasset/mock/ImportContractsMock.sol
1e7e548332c1527712b8f5641089366d9911e2c7331ae46f9ccd39b42ec2c87c  ./fasset/mock/FtsoManagerMock.sol
e080b19f8be1fd1deff7e604ce4ff8088edc77fba3f401f97c4e384d62dd569e  ./fasset/mock/FtsoMock.sol
fe543a033e812cc33fe2ccffbed04f8f4098a89c467ba7667fc6fe4796a5bbdf  ./fasset/mock/AssetManagerMock.sol
3d87f707d0833dd82d28d92eb3351b88d798ab5efdb8dae2ea95ba7579b13bfe  ./fasset/implementation/AgentVaultFactory.sol
c8110e237541466c07b0a5e15a4cd957bbb83a2405c0982797f3608a5b791045  ./fasset/implementation/Whitelist.sol
2e50313a609bbeda772d616f05cec1f9127f6e5f01257b450884829cdd65c1aa  ./fasset/implementation/AssetManager.sol
db486a3766ee20a0646b054cb712ea301ab14ac582099d1243b3706824f19cd3  ./fasset/implementation/AssetManagerController.sol
ab3fe346977d551561b30e99619110076d611e394a631df1ff0092de4d66c4ae  ./fasset/implementation/AgentVault.sol
f730bce6ad765c2ae438c0f5e0c4be43e59f81af2dfa8b507bad87f0d58a6db3  ./fasset/implementation/FAsset.sol
5406c9024caa300de5e54ccb254e722bd156eb92928cc3dba504bedfa6d129fb  ./fasset/library/UnderlyingAddressOwnership.sol
845a05996b4723431931cd25a50e4b05f790966bb68d9df2e34c83551fdc1508  ./fasset/library/Conversion.sol
d548f6136639d3a17e379ca656996ded05b031e073d0a983f6ce249fe9fd5a59  ./fasset/library/Minting.sol
f9a17926700fca91979de14e73c9b97e92c9795b7f217ade41c27e5dd8c4cd41
./fasset/library/UnderlyingWithdrawalAnnouncements.sol
f012a0348b1e71b7e416eae95f522fdc7439be902dab9f9b521e05cdadd7a567  ./fasset/library/Redemption.sol
64178a0f7762c6e97f8319609681259d42e14309e9eee144aa039357f5a70f98  ./fasset/library/AssetManagerSettings.sol
8de4865abb5de4a6401ea554396fed11988a197b6ff03db7af5e5321bda2aa7f  ./fasset/library/PaymentConfirmations.sol
2ca16d9526009f0d1c109308be04e920061f94cb870a71546f061676802eb768  ./fasset/library/SettingsUpdater.sol
40995fec603d1b6d0eb0ebf18a1fcb2d983eaf1ae4282ee9f5cb2fa7b895185a  ./fasset/library/Liquidation.sol
e52fa88e3196709138db115222f03c82eb74c00649e2b5f0cf4a9192209c700a  ./fasset/library/AMEvents.sol
b92d049ccadf22d487044822877de39fc117f9e2472635d79170e3bd4bf0502f  ./fasset/library/AvailableAgents.sol
24a1906336b258f87792e92ba980a6a048cdf8839a63e42d13904796f403fcc9  ./fasset/library/PaymentReference.sol
76c830f49df16603b4a816dd6a06de4e79dd9df5366dd21d24bea512781570ae  ./fasset/library/mock/RedemptionQueueMock.sol
e82514a3336a25d0b8b1eaccfba1150e44d482f02da43bbac005e1bdeaedc87d  ./fasset/library/mock/ConversionMock.sol
9078acc6cf5c27e6803312bec7d21966dd86793382710297e6bd344ac6dca936  ./fasset/library/StateUpdater.sol
c80a095884c13cd450ae3bc4c351d2bf9288b055ad67af4e0182d30672e1d890  ./fasset/library/FullAgentInfo.sol
cff93fbae0bbec095c2ede59216d9b4bd978e918f5bc321b2292f2f31a1a3757  ./fasset/library/Agents.sol
fa3d8661288363210bcba0391d7a05a006d4e44460a2b05fd71ab02d249f00a7  ./fasset/library/RedemptionQueue.sol
78307db70462af163ae2cc487669da6bc239824572525020f4596cec10a2aacd  ./fasset/library/AssetManagerState.sol
dcdec53addb979c82d23666d781f2facb1450eb7353f88a6717e21a3374fd6e3  ./fasset/library/CollateralReservations.sol
03dfb636a4a84e9d588b47825072c499735a76c07deaf5e68492178b4b9eaa3b  ./fasset/library/AgentCollateral.sol
7979a016a02167539dcf6acb912dd25eb87c09aae782f2e446465bbeb9db7958  ./fasset/library/UnderlyingFreeBalance.sol
a56515b7552d5cc85c270eacfca226971541e7b3673b367781c1471c5adb3b09  ./fasset/library/TransactionAttestation.sol
cbef92e379f9205601d92c2c97e38a281752157ce5afc5a3753eaffd962da766  ./fasset/library/Challenges.sol
abe9853adb2fc0640e84365b89348576f1ad84bb77424261f12c25681f2dbc49  ./fasset/interface/IAssetManager.sol
c08c40f452b6888e79dcb99384caad1b10a390815601bca4b1419abb4d7eb4ff  ./fasset/interface/IWNat.sol
338955619d75b1de8121608adab404f38ad55873ebab43772fcdf7ad2404d999  ./fasset/interface/IAgentVaultFactory.sol
df82a6c93de5e38596b6d38da3fd99369d45625a6f3858b29103e879e78546ad  ./fasset/interface/IWhitelist.sol
95e28b33ef15e2d98e1034899dcfcbe53f2e9f9d63ebfc87c110985bc456efc7  ./fasset/interface/IFAsset.sol
e226f09b2b7e244506b20735ccfabbe358fcea04ac87c2267ab9d9b2d07ebc4c  ./fasset/interface/IAssetManagerEvents.sol
4822c8abce65acaebbe43df0282af18cc33d59cce58eb5e1079c3aaba5721dd6  ./fasset/interface/IAgentVault.sol
a4b3014e37916881aa9aa2e7a7d80b5fb931f166df67539490b22984abfa9b50  ./governance/mock/AddressUpdatableMock.sol
3555f618268fabdfc2023b634afbbf9d743ff911f3c96d100610478c43c70faa  ./governance/mock/GovernedMock.sol
7a35ab0fb87846c3e055c3fcf4542d3df4c00375076d5e59882d706ce3e501c1  ./governance/implementation/GovernedBase.sol
3f160dde7943001b1eedc94876e14bd52785a410560ea4b298060070347825f9  ./governance/implementation/Governed.sol
71c1435f13dd898993b8c861b181857c1ebf6876c3b91adeb1183a9024709619  ./governance/implementation/AddressUpdatable.sol
26599de22dce1bb771dec0d22e2b3c8bf99b0f5230a702c91c2daff3935d29e6  ./utils/mock/SafePctMock.sol
01c86baab70a125e3e2fea8b136b8418207a3ab953678804b9f9399a8e07d18e  ./utils/mock/SafeMath64Mock.sol
6d1f3b24b29eb3feb9e45d3e3109d068e0b23ae6f4a72086d859339e7441e584  ./utils/lib/SafePct.sol
4122c1d3c3e748172f897a95765b913de2aef6c0ece943287087f5177c826a02  ./utils/lib/SafeBips.sol
39dff1b27e26a139810d011ab0aeb66cd9857ea339612d859314655656f86801  ./utils/lib/SafeMath64.sol
0ccebdfb356b3533eb9acb45090c62fab097026e573228310a3e17647c704112  ./utils/lib/MathUtils.sol
3617f640c89459bb1f020fbc0f6f4adc0e2c574735970af0586b651618661ac4  ./utils/Imports.sol
```

The fAsset system allows minting fAssets in the Flare network that represent assets from other blockchains such as BTC, XRP and Ethereum. This platform is implemented as an overcollateralized lending system, where fAsset agents lock

collateral in the form of FLR, the Flare network native asset. The collateral backing the minted assets must be always above a configurable per fAsset collateral ratio or the agent can be liquidated.

By design, the underlying chain funds are not actually locked, but they are expected to be held in the declared EOA account. These funds, which depositors transfer to the agent's underlying chain EOA address, are intended to be transferred back to the redeemers when fAssets are redeemed. If an agent fails to comply with the expected underlying payment, its collateral can be used to pay back the redeemer.

The fAsset system relies on off-chain "challengers" that are incentivized to monitor transactions in the underlying blockchain and report any illegal payment that moves funds out of the agent's EOA address without proper justification. In that scenario, the agent collateral is fully liquidated at a discounted price. However, Coinspect identified different issues that allowed agents to bypass this protection mechanism and break the agent's underlying expectation (see **FLR-1**, **FLR-2** and **FLR-5**).

Because the underlying funds are not locked, agents are always in control of them. During a quick price swift (e.g., the FLR token price decreases fast in comparison to the underlying asset price), if an agent becomes undercollateralized and is not liquidated in time, it can choose to escape with all the underlying funds. As a result, redeemers will not be able to obtain the total amount of underlying assets that correspond to the fAssets being redeemed (see **FLR-7**).

The code reviewed limits certain operations to a configurable allowlist. Flare team clarified that the whitelist is only intended for the initial testing period and will be removed afterward.

The following components were assumed trusted and properly incentivized and implemented during this audit:

1. FTSO oracles
2. Attestation provider
3. Off-chain
    a. fAsset agents
    b. Attestation clients
    c. Challengers

The system relies on the FTSO oracle system to price each underlying asset against the collateral locked in the agent's vaults. It is worth noting that by design, as explained in the systems specification, two pairs of prices are compared: one

obtained from every price provider and other from only the trusted providers. The value resulting in a higher collateral ratio is utilized in order to protect the fAsset agents from being liquidated by the colluding FTSO price providers. **If this kind of collusion is considered possible, Coinspect recommends evaluating the possibility of FTSO price providers manipulating the asset prices in order to inflate the fAssets agent's collateral ratio instead.**

3. Summary of Findings

Id	Title	Total Risk	Fixed
FAS-1	Agents can withdraw underlying funds unpenalized: ref type check	High	✔
FAS-2	Agents can withdraw underlying funds unpenalized: redemptionId check	High	✔
FAS-3	Attackers can lock agents' selfMint payment	Medium	✔
FAS-4	Predictable payment references facilitate griefing challengers	Medium	✔
FAS-5	Agents can mint fAssets without locking underlying assets	High	✔
FAS-6	Agents can be liquidated with old transactions	Medium	✓
FAS-7	Redeemers can receive less funds than corresponds to their burned fAssets	Medium	✘
FAS-8	Missing _validateSettings	Info	✔
FAS-9	burnAddress can be updated	Info	✔

4. Detailed Findings

FAS-1 Agents can withdraw underlying funds unpenalized: ref type check

Total Risk	Impact	Location
High	High	Challenges.sol Redemption.sol PaymentReference.sol
Fixed ✔	Likelihood: High	

Description

Agents can modify payment references and reuse any `redemptionId` to withdraw money from the underlying chain without being challenged and penalized.

There are two challenges that should catch this misbehavior: `illegalPaymentChallenge` and `doublePaymentChallenge`. The first challenge is used for penalizing payments that do not match a redemption ticket. The second challenge is used for flagging the reuse of a ticket.

Because the `redemptionId` is a subarray of the `paymentReference`, an agent can reuse a `redemptionId` with a different `paymentReference` to avoid being punished by either methods.

The `isValid` function implementation fails to properly validate the `_refType`:

```
    function isValid(bytes32 _reference, uint256 _type) internal view returns (bool) {
        uint256 refType = uint256(_reference) & _type;
        uint256 refLowBits = uint256(_reference) & ((1 << TYPE_SHIFT) - 1);
        // for valid reference, type must match and low bits may never be 0 (are either id or address)
        return refType == _type && refLowBits != 0;
    }
```

But in the `doublePaymentChallenge` the whole `referencePayment` must match to be penalized:

```
// payment references must be equal
require(_payment1.paymentReference == _payment2.paymentReference, "challenge: not duplicate");
```

And as a consequence it is possible to craft many payment references with different reference tags that are considered a valid REDEMPTION.

By exploiting this issue an agent can withdraw the underlying assets held in their address without penalization.

A test case exploiting this scenario is shared below. This proof of concept modifies an existing payment reference by replacing the first 8 bytes with 0xffffffff:

```
it("must fail", async () => {
    const agent = await Agent.createTest(context, agentOwner1, underlyingAgent1);
            const minter = await Minter.createTest(context, minterAddress1, underlyingMinter1,
context.underlyingAmount(10000));
    const redeemer = await Redeemer.create(context, redeemerAddress1, underlyingRedeemer1);
    const challenger = await Challenger.create(context, challengerAddress1);
    // make agent available
    const fullAgentCollateral = toWei(3e8);
    await agent.depositCollateral(fullAgentCollateral);
    await agent.makeAvailable(500, 2_2000);
    // update block
    await context.updateUnderlyingBlock();
    // perform minting
    const lots = 3;
    const crt = await minter.reserveCollateral(agent.vaultAddress, lots);
    const txHash = await minter.performMintingPayment(crt);
    const minted = await minter.executeMinting(crt, txHash);
    assertWeb3Equal(minted.mintedAmountUBA, await context.convertLotsToUBA(lots));
    // redeemer "buys" f-assets
    await context.fAsset.transfer(redeemer.address, minted.mintedAmountUBA, { from: minter.address });
    // perform redemption
    const [redemptionRequests, remainingLots, dustChanges] = await redeemer.requestRedemption(lots);
    assertWeb3Equal(remainingLots, 0);
    assert.equal(dustChanges.length, 0);
    assert.equal(redemptionRequests.length, 1);
    const request = redemptionRequests[0];
    assert.equal(request.agentVault, agent.vaultAddress);
    const tx1Hash = await agent.performRedemptionPayment(request);
    const fakeTxHash = await agent.performFakeRedemptionPayment(request);
    // others cannot confirm redemption payment immediately or challenge it as illegal payment
      await expectRevert(challenger.confirmActiveRedemptionPayment(request, tx1Hash, agent), "only agent
vault owner");
    await expectRevert(challenger.illegalPaymentChallenge(agent, tx1Hash), "matching redemption active");
        await expectRevert(challenger.illegalPaymentChallenge(agent, fakeTxHash), "matching redemption
active");
       await expectRevert(challenger.doublePaymentChallenge(agent, tx1Hash, fakeTxHash), "challenge: not
duplicate")
    console.log("Illegal payment or double payment should catch this and not revert");
});

async        performFakeRedemptionPayment(request:        EventArgs<RedemptionRequested>,        options?:
MockTransactionOptionsWithFee) {
    const paymentAmount = request.valueUBA.sub(request.feeUBA);
    let ref = request.paymentReference;
    let newRef = "0xffffffffffffffff" + ref.substring(18, ref.length);
    return await this.performPayment(request.paymentAddress, paymentAmount, newRef, options);
}
```

This same issue affects the `paymentsMakeFreeBalanceNegative` challenge and the `ANNOUNCED_WITHDRAWAL` logic which will not catch the misbehavior.

Attack scenario

An attacker could perform the next steps to exploit this issue

1. Create an agent so it can hold at least 2 lots.
2. Self mint two lots or wait until another user mints.
3. Create a redemption ticket for 1 lot.
4. Use the redemption ticket twice (or as many times as desired) to execute payments using the exploit above.
5. Call `confirmRedemtpionPayment` to account for the legal payment so that the free underlying balance is updated.
6. Recover the collateral.
7. Attacker ends up with 1 extra lot of underlying.

Recommendation

Correct the `isValid` function.

Make sure the checks in both challenges are consistent and that two different payment references can not exist for the same redemption id.

Status

This issue was addressed by commit **`a040c3d47ad95fff23f4fb281c42d425c09484c4`**

FAS-2 Agents can withdraw underlying funds unpenalized: redemptionId check

Total Risk	Impact	Location

High	High	`Challenges.sol`
		`Redemption.sol`
		`PaymentReference.sol`

Fixed ✔	Likelihood High	

Description

Agents can modify payment references and reuse any `redemptionId` to withdraw money from the underlying chain without being challenged and penalized.

There are two challenges that should catch this misbehavior: `illegalPaymentChallenge` and `doublePaymentChallenge`. The first challenge is used for penalizing payments that do not match a redemption ticket. The second challenge is used for flagging the reuse of a ticket.

Because the `redemptionId` is a subarray of the `paymentReference`, an agent can reuse a `redemptionId` with a different `paymentReference` to avoid being punished by either methods.

The `decodeId` function is used to obtain the `redemptionId` in the `illegalPaymentChallenge`:

```
uint64 redemptionId = PaymentReference.decodeId(_payment.paymentReference);
```

Which is only the last 64 bits of the 256 bits `paymentReference`:

```
function decodeId(bytes32 _reference) internal pure returns (uint64) {
    return uint64(uint256(_reference) & ((1 << 64) - 1));
}
```

But in the `doublePaymentChallenge` the whole `referencePayment` must match to be penalized:

```
 // payment references must be equal
require(_payment1.paymentReference == _payment2.paymentReference, "challenge: not duplicate");
```

The redemption id is generated in the `_createRemptionRequest` function when a user calls the `redeem` function:

```
        // emit event to remind agent to pay
        emit AMEvents.RedemptionRequested(_data.agentVault,
            requestId,
            _redeemerUnderlyingAddressString,
            redeemedValueUBA,
            redemptionFeeUBA,
            lastUnderlyingBlock,
            lastUnderlyingTimestamp,
            PaymentReference.redemption(requestId));
    }
```

The code in PaymentReference.sol contract casts the 64 bits unsigned int to 256 bits before bitwise oring with the payment type identifier:

```
    function redemption(uint64 _id) internal pure returns (bytes32) {
        return bytes32(uint256(_id) | REDEMPTION);
    }
```

Because of that, an agent can modify a `paymentReference` by changing the bits between the `REDEMPTION` tag and the 64 bits of the `_id` is so that the `redemptionId` value is not affected when retrieved by the `decodeId` function. This prevents the agent from being challenged from both an illegal or a double illegal payment.

Exploiting this issue an agent can withdraw the underlying assets held in their address without penalization.

A test case exploiting this scenario is shared below. This proof of concept modifies an existing payment reference by replacing the first 16 bytes after the reference type tag with 0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa. After decoding the payment reference, the redemption Id will be 1.

```
it("must fail", async () => {
    const agent = await Agent.createTest(context, agentOwner1, underlyingAgent1);
            const minter = await Minter.createTest(context, minterAddress1, underlyingMinter1,
context.underlyingAmount(10000));
    const redeemer = await Redeemer.create(context, redeemerAddress1, underlyingRedeemer1);
    const challenger = await Challenger.create(context, challengerAddress1);
    // make agent available
    const fullAgentCollateral = toWei(3e8);
    await agent.depositCollateral(fullAgentCollateral);
    await agent.makeAvailable(500, 2_2000);
    // update block
    await context.updateUnderlyingBlock();
    // perform minting
    const lots = 3;
```

```
    const crt = await minter.reserveCollateral(agent.vaultAddress, lots);
    const txHash = await minter.performMintingPayment(crt);
    const minted = await minter.executeMinting(crt, txHash);
    assertWeb3Equal(minted.mintedAmountUBA, await context.convertLotsToUBA(lots));
    // redeemer "buys" f-assets
    await context.fAsset.transfer(redeemer.address, minted.mintedAmountUBA, { from: minter.address });
    // perform redemption
    const [redemptionRequests, remainingLots, dustChanges] = await redeemer.requestRedemption(lots);
    assertWeb3Equal(remainingLots, 0);
    assert.equal(dustChanges.length, 0);
    assert.equal(redemptionRequests.length, 1);
    const request = redemptionRequests[0];
    assert.equal(request.agentVault, agent.vaultAddress);
    const tx1Hash = await agent.performRedemptionPayment(request);
    const fakeTxHash = await agent.performFakeRedemptionPaymentID(request);
    // others cannot confirm redemption payment immediately or challenge it as illegal payment
      await expectRevert(challenger.confirmActiveRedemptionPayment(request, tx1Hash, agent), "only agent
vault owner");
    await expectRevert(challenger.illegalPaymentChallenge(agent, tx1Hash), "matching redemption active");
        await expectRevert(challenger.illegalPaymentChallenge(agent, fakeTxHash), "matching redemption
active");
       await expectRevert(challenger.doublePaymentChallenge(agent, tx1Hash, fakeTxHash), "challenge: not
duplicate")
    console.log("Illegal payment or double payment should catch this and not revert");
});

async       performFakeRedemptionPaymentID(request:       EventArgs<RedemptionRequested>,       options?:
MockTransactionOptionsWithFee) {
    const paymentAmount = request.valueUBA.sub(request.feeUBA);
    let ref = request.paymentReference;
    let newRef = "0x4642505266410002aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" + ref.substring(50, ref.length);
    return await this.performPayment(request.paymentAddress, paymentAmount, newRef, options);
}
```

This same issue affects the `paymentsMakeFreeBalanceNegative` challenge and the `ANNOUNCED_WITHDRAWAL` logic which will not catch the misbehavior.

The attack scenario described in **FAS-1** applies to this issue.

Recommendation

Fix the `decodeId` function to take into account all bytes provided.

Make sure the checks in both challenges are consistent and that two different payment references can not exist for the same redemption id.

Status

This issue was addressed by commit **`f6aaf1a85b9ac46318c8326706440a0f2bb04f24`**.

FAS-3	Attackers can lock agents' selfMint payment

Total Risk	Impact	Location
Medium	Low	`Minting.sol`
Fixed ✔	Likelihood: High	

Description

Agents' self-mint payment can get locked until more collateral is added or available.

When an agent uses the `selfMint` function for minting fAssets, if there is not enough collateral to cover the minting value, the `selfMint` function fails and the funds get locked.

This is especially problematic as any user can monitor the underlying chain and reserve collateral for minting between the agent's payment and the self-minting transaction.

The agent then can attempt to add more collateral in order to be able to execute the self-mint. But the attacker can also preempt this new self-mint request and reserve the new collateral the agent just deposited.

A mitigation factor for this attack is the cost of reserving collateral or the minting fee in case the attacker decides to actually mint the fAssets.

Recommendation

The self minting process should reserve collateral the same way as any minting process.

Status

Flare does not consider this issue needs to be fixed, as they explained that as a mitigating factor the agent can selfMint 0 lots in order to update the free underlying balance to include all funds that were deposited.

This mitigation procedure will be added to the documentation to help agents operators know how to free their funds if locked in this scenario.

FAS-4	Predictable payment references facilitate griefing challengers	

Total Risk
Medium
	Impact
	Medium
		Location
		`UnderlyingWithdrawalRequests.sol`
		`PaymentReferences.sol`
		`Challenges.sol`

Fixed ✔	Likelihood Medium	

Description

Attackers can trick challengers into attempting failed challenge attempts in order to harm them. As a result, challengers will spend gas and get no commission for their efforts. This can discourage them from reporting similar transactions.

Payment references are predictable. These are formed by joining a payment type tag plus an incremental id counter. This allows the attacker to send a payment, with the know-to-be-next payment reference.

Then, for example, the attackers do not need to announce an underlying withdrawal and wait for the transaction to return the intended reference payment. They can send a payment with the predicted payment reference, and wait for any challengers being submitted to the network. Then they can front-run this challenge with an announcement in order to prevent the challenge from succeeding.

This could be repeated until an automated challenger bot runs out of gas or until challengers are discouraged from continuing to report illegal payments.

Recommendation

Make reference payments not predictable in order to force the announcement to be performed before the payment.

Status

This issue was addressed by commit **`6c4d93113a9eeedb30625c9835f5d2a1fe80f434.`**

The `ID` can still be correctly predicted by generating it in the right block, but challengers can defend themselves against griefing attempts using better strategies (e.g. starting the challenge one block after it).

FAS-5 Agents can mint fAssets without locking underlying assets

Total Risk	Impact	Location

High	High	`Minting.sol`
		`CollateralReservation.sol`
		`PaymentReference.sol`
		`Agents.sol`
		`UnderlyingAddressOwnership.sol`

Fixed ✔	Likelihood: High	

Description

Agents can mint fAssets at creation time, without locking the corresponding underlying asset at creation time.

When an agent is created the `AssetManager` requires an `EOA` proof. This proof can send any amount of underlying assets without being penalized.

Attackers can perform the next steps:

1. Send a transaction to their underlying address with the value to be minted using a precalculated Collateral Reservation ID.
2. Perform the `EOA` proof sending that value out of the address.
3. Create the agent with the proof and add the required collateral.
4. Wait until the right moment, allow third party minting and create a Collateral Request that uses the previously calculated ID.
5. Use the transaction at step 1 for minting fAssets, these assets will not be locked in the underlying address and cannot be penalized.

This can be done due to a set of properties in the contracts.

The minting process accepts old transactions. While the `selfMint` and the `confirmTopupPayment` functions require the transaction to be new enough, this is not true for the `mintingExecuted` function. This check only exist on the former functions

```
require(_payment.blockNumber >= agent.underlyingBlockAtCreation,
    "self-mint payment too old");
```

The `EOA` proof can send arbitrary money to any address. This is an unnecessary property of the proof.

Collateral Reservation IDs are manipulable. During the audit the Flare team added a protection to prevent guessing the ID (per Coinspect's recommendation in finding **FAS-4**). But this fix is not enough to prevent these kinds of attacks. Basically, the

randomized ID is the last used ID (starting at 1) plus the block number modulus 1000. Because the agents can choose when collateral can be reserved, they can select the block where the reservation happens and force it to match the precalculated ID.

Even though, ultimately, by design agents can always remove all funds deposited in their underlying address, this issue allows them to do this without being liquidated and bypassing the challengers controls set in place with this purpose. This breaks the expectations for Agent_100 type agents as described in the protocol's specification.

As a mitigating factor, the corresponding redemption ticket is created. When this ticket is redeemed the agent will be forced to return the underlying funds or collateral in case of default.

Recommendation

Do not accept transactions that happened before the agent creation time.

Improve the ID randomization.

Evaluate adding a challenge that enables penalizing an agent if the EOA underlying balance is below the expected amount. This challenge would be useful to report an ill behaved agent in case any other unknown scenario is exploited. Implementing this solution would require the ability to prove the balance in a certain block and keeping history of the expected free balance.

Status

Fixed in commit **651a0c53bacec8519dd494facc2bbe298dafd9d0**. The code now checks that the minting payment is from a later block than the EOA proof payment

FAS-6	Agents can be liquidated with old transactions	
Total Risk Medium	Impact High	Location `Challenges.sol`
Fixed ✓	Likelihood Low	

Description

Challengers can use transactions performed by the agent underlying address before the agent creation to report an illegal transaction and fully liquidate them.

The challenges do not verify the transaction being denounced occurred after the agent creation timestamp.

The current code assumes no transaction was performed by the underlying EOA before the agent creation, but this is not enforced nor documented.

Recommendation

Do not accept challenges of transactions that happened before the agent creation time. If this is the intended behavior, clearly document it to avoid agents being deployed with underlying addresses with previous transactions. Alternatively, enforce this requirement during the EOA verification step.

Status

This issue was acknowledged by Flare. Momentarily, a warning was added in the specification alerting about this issue (section Agent's underlying address).

FAS-7	Redeemers can receive less funds than corresponds to their burned fAssets

Total Risk	Impact	Location
Medium	High	`Agents.sol`

Fixed	Likelihood
✘	Low

Description

Users could receive less collateral than expected when redeeming their fAssets.

The redeemer's fAssets are burned when the redemption process starts. Then, a redemption ticket is picked to fulfill the redemption request, and the agent that created it is expected to transfer the amount of underlying assets corresponding to this ticket. If the agent fails to comply in time, the redeemer can report the lack of payment and the collateral in the agent's vault is used to compensate the redeemer.

The `payout` function does not revert if the collateral in the vault is not enough to pay the redeemer. Instead, the minimum between what is available and what is expected is transferred:

```
function payout(
    AssetManagerState.State storage _state,
    address _agentVault,
    address _receiver,
    uint256 _amountNATWei
)
    internal
    returns (uint256 _amountPaid)
{
    IAgentVault vault = IAgentVault(_agentVault);
    // don't want the calling method to fail due to too small balance for payout
    _amountPaid = Math.min(_amountNATWei, fullCollateral(_state, _agentVault));
    vault.payout(_state.settings.wNat, _receiver, _amountPaid);
}
```

As a consequence, if there is an uncollateralized agent in the system, the redeemer will receive less than expected for the amount of fAssets it burned.

This will depend on the redemption ticket and agent that were chosen from the FIFO redemption queue by the system. It could happen that there are enough funds in other agents to comply with the redemption but because of the redemption ticket chosen the redeemer results being harmed.

Recommendation

Allow redeemers to specify if they accept burning their fAssets for less than the expected amount. Consider re-minting the burned assets for the redeemer.

Status

This issue was acknowledged. Flare does not expect this situation to occur “... because the required collateral ratio and the liquidation system mean that the price asset/flare should fall by a factor of 2 in relatively short time for such a situation to arise”.

FAS-8	Missing _validateSettings		
Total Risk **Info**	Impact -	Location `SettingsUpdater.sol`	
Fixed ✔	Likelihood -		

Description

The `_validateSettings` method is missing in the `SettingsUpdater` contract.

Recommendation

Implement the missing method.

Status

This issue was acknowledged and fixed during the audit in commit `c10e43f7f358511ba6d6a321d15e0ab27852ffff`.

FAS-9	burnAddress can be updated		

Total Risk
Info
	Impact
	-
		Location
		`AssetManagerSettings.sol`
		`Minting.sol`
		`CollateralReservations.sol`
		`Agents.sol`	

Fixed ✔	Likelihood -		

Description

The address used to burn funds is a setting and can be updated.

Recommendation

Consider making the burn address immutable.

Status

Flare team stated this will be set to a constant for deployment.

5. Disclaimer

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