Flare

Smart Contract Audit

coinspect

Flare

coinspect

Flare

Smart Contract Audit

Prepared for Flare • August 2021

v210809

1. Executive Summary

2. Scope

3. Assessment

4. Summary of Findings

5. Detailed Findings

   FLR-001 - WLFR token unlimited free minting

   FLR-002 - Inflation contract locked after failing balance assertion

   FLR-003 - Double voting power

   FLR-004 - FTSO oracle can manipulate the election mechanism to perpetually receive all rewards

   FLR-005 - OOG results in errors not logged

6. Disclaimer

1. Executive Summary

In **July 2021**, Flare Network engaged Coinspect to perform a source code review of the Flare Network smart contracts.

The objective of the project was to evaluate the security of the smart contracts that implement critical functionality in the network, including but not limited to:

- Flare Keeper,
- Inflation and Supply
- Flare Time Series Oracles and Price Providers whitelisting
- Vote Power Token and delegation mechanism

Neither Flare Network's consensus protocol nor the incentives model and their security assumptions were in scope during this audit.

All the reviewed components had clear specifications and the code was extensively documented with commentaries. Several unit and integration tests are included in the repository.

Overall, Coinspect found the project's code quality to be above average.

The following issues were identified during the assessment:

High Risk	Medium Risk	Low Risk	Zero Risk
3	1	0	1

Coinspect reported three high risk and one medium risk findings. The most critical issue, FLR-001, allows anybody to mint unlimited WFLR tokens for free. FLR-002

results in the Inflation contract being unable to execute any action. FLR-003 permits a delegator to double its voting power by abusing the revocacion procedure. FLR-004 allows colliding oracle price providers to bypass the randomized rewards distribution mechanism and could hurt FRL holders as a consequence in certain scenarios.

During August 2021, **Coinspect verified all security relevant findings had been correctly addressed** by the Flare team.

The present report details the tasks performed and the vulnerabilities found during this audit as well as several suggestions aimed at improving the overall code quality, and warnings regarding potential issues.

2. Scope

The audit started on **July 6, 2020** and was conducted on the **audit_2_coinspect** branch of the git repository at

https://gitlab.com/flarenetwork/flare-smart-contracts/

The last commit reviewed during the engagement was commit 9deb04874295a1b7b2401c74c13a93ea71910779 of **July 15, 2020**:

```
commit 9deb04874295a1b7b2401c74c13a93ea71910779
Merge: a29d9ba 364c586
Author: Ilan Doron <ilan@flare.network>
Date:   Thu Jul 15 21:02:47 2021 +0000

   Merge branch 'master' into 'audit_2_coinspect'

   Update audit branch with last changes

   See merge request flarenetwork/flare-smart-contracts!238
```

The scope of the audit was limited to the following Solidity source files, shown here with their sha256sum hash:

```
a4817fc02eddf7272315fe7c7ddc2ea736bfd67fb2df146fa62a5e1413631c48
./validator/implementation/ValidatorRewardManager.sol
33928d5cfdcd9c8b6e79310e56c4869bd4bb6f5aec9be9ba901b5efa2f4f3924
./validator/interface/IIValidatorRewardManager.sol
53bbbbb6b2a8b89ced059ad535e41cec101c52de8cff127a7c580eba3f5ca1c6  ./airdrop/implementation/Distribution.sol
e7f85a1660bb12038a0841a0eb18cf4c081e2fe23a39626b23e6085a20d234fa  ./genesis/mock/FlareKeeperMock.sol
9a49c1490576538d082c42b967cfe7e59379bf182af0ac562b401ae29567a8d4  ./genesis/mock/EndlessLoopMock.sol
ade3f36b4ac0d83bf60bceed0212dbd4771eed94803bc558694b6759a3f6ff33  ./genesis/implementation/StateConnector.sol
88936810564e9238eaba9bfff9507f1a5baca72fe20f35dc949a2e57b4c244b3  ./genesis/implementation/FlareKeeper.sol
0a1f2fbc510b0fb8ea0de123e1ca9beedf1c743c20cc7e66ddb9049446cfd4d0  ./genesis/implementation/PriceSubmitter.sol
84454d480c304ef34943c2bf0f99b51c9593202907d5ceb34e6f2f1d31e51aec  ./genesis/interface/IIVoterWhitelister.sol
9d1f256cdb13e19f6b3edfa825240d112f503e075cfb843c9ac3fc3221a75a89  ./genesis/interface/IIPriceSubmitter.sol
6fb637e0e86a6b57ad5381c478842cf80690380693a799edbfa1a51c432cdf25  ./genesis/interface/IFlareKeep.sol
97fd12845c9f8f90c1963e57f5ad99641cfc9b5a44f68f276ee33ceafbb3c1ab  ./genesis/interface/IIStateConnector.sol
bcf997b976a37f855ba07283e20230292252c8281eab7993a9bf3c64e38914f2  ./genesis/interface/IIFtsoRegistry.sol
9c074aec811921000f3f00c59008af29850843e16efde5de6f40a6f2cff7e7f7  ./fFasset/mock/DummyFAssetMinter.sol
```

854e411f4be944a8f36e48e4ad33e1b1c5c0048e9293e34f55c7004541cf539a ./fFasset/mock/FAssetToken.sol
a7d2ed5f0610ab17604a1f911540591454e97c1638deda4e984c888513792348 ./fFasset/interface/ICollateralizable.sol
68f4f7eb28f2ba281d7459334fe0679ec990ddfa9f3568b148450860d91afb1c ./governance/mock/HistoryCleanerMock.sol
f9224f524e699c637ca654ebb2481112de003b43572d6e9dded8dd87679cb12b ./governance/implementation/GovernedBase.sol
93cf605f3233de4520b4e27474677166106690aa8e2563e135d5d98720356700 ./governance/implementation/Governed.sol
931230c9315e1ae36510019d77c391dcad4f17a5594c2cc52c80f2006e5c9d1a
./governance/implementation/GovernedAtGenesis.sol
a94bf6057ee10aacfa349b7048e0e114dda8c09bc924d5eafcf63f4d81bd7188
./governance/implementation/InflationAllocation.sol
f0ce169637a7e27f390e379892268ea9565a5c487cc61e4a0f6864a04a3b7ba1 ./userInterfaces/IFtso.sol
2e4d403cc106278ec92ae1f66cb9b047d9162c140316851c6bc4708bb5703d71 ./userInterfaces/IVPContractEvents.sol
2c52ee7ac5f40613d6743ca5098cfae80356bc5331a15f668960e946ac64a9f3 ./userInterfaces/IPriceSubmitter.sol
30f9fe088dc46141ddc1624dc514b0addbf6046fe47ce90dc9416c14b1f3d0a6 ./userInterfaces/IWFlr.sol
3597ce40bde72ca6d8210afe964afe3c2aafcab90b1d54996055c4b7f5cb21db ./userInterfaces/IValidatorRewardManager.sol
56198f51925b5675ecf7694eb4e8b433fc86854901a5ca4d173652a979095ae5 ./userInterfaces/IVPToken.sol
173373b5264d17185a1a724e25ee2cc32aa9f313ee9a2807f99468dc26e8ca86 ./userInterfaces/IDistribution.sol
1a46640b1312ceb56e05802a4f54e720c9edfb93455b7f6ac6c5160f0f16d557 ./userInterfaces/IFtsoRegistry.sol
acdf56e26e80cd09f33080fe7671fbca86de8b7fe8df0c2f64fb501f5e0b497d ./userInterfaces/IGovernanceVotePower.sol
843460f7c515da5255f649f6fc61e7ca8ff63b0db1b924264ffc049aad10df0c ./userInterfaces/IFtsoRewardManager.sol
8d5fc620599e917d9b9731f1cb3f26f2af2ed348d0ec451c8a0b6fa73fd4d7ed ./userInterfaces/IFtsoManager.sol
f94136177fa518493960a641439664d0f62311ed94e8758c5674ffc65e5870e0 ./userInterfaces/IVoterWhitelister.sol
7746cc2f97c87312227a00fee7b0a4bf0f11198ed332201817029e93dc93972e ./inflation/mock/InflationReceiverMock.sol
04a6a98d7f024bc957b6a824c85779641d28291bb5978e114cde7856e15e1ff6 ./inflation/mock/InflationMock.sol
48fe3d6d9c1b47e45e03752ce35d6eb11dec25796cc8200c6bb697d6261a6354 ./inflation/mock/SuicidalMock.sol
f60b1fc3f4b7fe240e1386c89d2d73a290b5ff0d08749df7340a42693c56365e
./inflation/mock/SharingPercentageProviderMock.sol
816bb2f01c999cc07b1d50f1c46c24903447fc45cbac6c5865e3dfa2269238b9 ./inflation/implementation/Inflation.sol
258e5a35f095a9258574101dcc41d4dce316a3149adc06d5b0cf5602b5138f80 ./inflation/lib/InflationAnnum.sol
41fd9d888058a75f8a6900eb4969e325ab20d86213bb68ce59e1c09388e0585a ./inflation/lib/InflationAnnums.sol
48cefde730ec7e1c7414343fb5700f000e5d5631bf60d2b512ff1a14b349dfb2 ./inflation/lib/RewardServices.sol
e4e371511562f44ba2921365cbb47e5045ba5d10d0b22b5cdd47a0cc2c2ed11e ./inflation/lib/RewardService.sol
d224d8085e583087f425f226eb1cffff4b69f379cf97fc133ffd8b583fb5bb23 ./inflation/interface/IIInflationReceiver.sol
7c89c7b1f2d57344ddc1363fecc85d40244a2893050f8511a0efd80334939b37
./inflation/interface/IIInflationPercentageProvider.sol
d85bf841fdc1937dbc907adc7126f6d6a7fc54ec552c3a43d2f238f52b7050b7
./inflation/interface/IIInflationSharingPercentageProvider.sol
8987b038958572bbd87aae6862ac2ec3770aa158037a1ff4523767b3bf1691af ./utils/mock/SafePctMock.sol
c755beca5aa8d3c2fa8d950d1bfe16ab9d5d1adabe33f02107d43a29cff60ae0 ./utils/mock/VoterWhitelisterMock.sol
0ea9b26b1502d7507ed8a9ddea625718d24b3830749d80c71a077393541e01fc ./utils/mock/DateTimeLibraryContractMock.sol
34c3f2393555b6d0a7d4f9d187dc53f19a9e49f4f748cb83b26ed7dccbf94f9f ./utils/implementation/GovernedAndFlareKept.sol
64f5c2dbfc7bf6cdd5ce7efc6f02e24c72fdc93bcd244f7c368ee4822152e081 ./utils/implementation/VoterWhitelister.sol
4f6864f6815039f156c252b56bcb8615f1e5ca65f7dca474ef3af0a5c615ac73 ./utils/implementation/RevertErrorTracking.sol
da7d13381925ca65b5b184d651fb0a5af86a7146b4bf5b8735c64a51bc51bb68 ./utils/implementation/DateTimeLibrary.sol
48be41613733916fc28fdfa65421a6d984d6274089e4712eff6e546acfbb61ea ./utils/implementation/SafePct.sol
f13e59455e32bd2279b2abfb06076bf0a12f5f33cedf4c5a5be274cc2a033aa3 ./utils/implementation/FtsoRegistry.sol
3617f640c89459bb1f020fbc0f6f4adc0e2c574735970af0586b651618661ac4 ./utils/Imports.sol
f170036b3bacfbd728886b1bef3ee8b445d0a2cf75f83f84823709c7b2576d1a ./ftso/mock/FtsoMedianMock.sol
1c54eb90ff0fa3f2172860b190a4b9d0368abcf8b0796355063a7361239da7c9 ./ftso/mock/MockFtso.sol
95d7935dbb52febce58a278f5d8b148d7355a18ac76b22eeab46b6fcc1c7a7ab ./ftso/mock/FtsoManagerMock.sol
d693101cf908fb102b895da65ac1f94baa561a94071ca5fb533522c98fcacfde ./ftso/mock/MockVPToken.sol

a2480d06f1302daa8cbf7a5b4fc03a49d7a4b636e769242bc750c9b1e77e8361 ./ftso/mock/FtsoEpochMock.sol
8e00a0b073b77d9494fa4b25e433a3232dc0e838330a80b0e04b5e58d51949c9 ./ftso/mock/FtsoVoteMock.sol
ac48182ddeb8d1960a768beb6755f1bc8723a307269505e9634692ca94740a29 ./ftso/implementation/FtsoRewardManager.sol
0fbf1fb4ef301cf4a012719182198ea3059f51b726ae590242969674d48766c8 ./ftso/implementation/Ftso.sol
6444484b78516a5bb8e8c4ef91b8d02e8f095c15ef6cfab10b994da0d1ad4081 ./ftso/implementation/FtsoManager.sol
e94c6e6be7139dee60b72178b7fec0b54c132edea476bde1ea216eefb6b16389 ./ftso/lib/FtsoVote.sol
99b989ff1306f21ea51666ed5bc9d25fabde3238fbd46f2429cbb47930875b93 ./ftso/lib/FtsoMedian.sol
8bb054df7cb9b511687372668c7dd90c3c1f5e63586fc253ca9925027a4ea8d9 ./ftso/lib/FtsoEpoch.sol
c18bf269b10a77d67977a5e831e684ba098663643d71eec1a24783979b712c45 ./ftso/lib/FtsoManagerSettings.sol
9a5d66f08aa1ae761d14d6e12b0b411029872a30e60a56a9269d377012bf02b4 ./ftso/interface/IIFtsoManager.sol
490a1f669e7f9a4db6f6595ed94b043062b9291ffc3b6d59fbeb5d733f4590f4 ./ftso/interface/IIFtso.sol
a702d2f6284d441b996c439739692eafe87de1441d3476e7da5914eef75ecfd8 ./ftso/interface/IIFtsoRewardManager.sol
6f79c8bb0d97fa1676dabb5341c488a65542515512d324c79d349e362fa45993 ./token/mock/VPTokenMock.sol
9329f8aaa0f19d294f2b4163b582de766d18b65fd27baba7ad0655aed1964624 ./token/mock/DummyVPToken.sol
8d1f9982daff1372f94fd6542da0c874aba4d4cf57a64e0a86878b7515e3017d ./token/mock/FlashLoanMock.sol
83fd0b4c5d3b4a31b59be1d949517700ed5d8973608f2bbc9cf1610ffdba0eb0 ./token/mock/CheckPointsByAddressMock.sol
ae1613c09279cad51e3bac43e4e501afe39b2f30f415f5821e4a5afdd9ed198d ./token/mock/PercentageDelegationMock.sol
eefc49cdda92a74e1157efc0cddc63c8d528b9657feeb913e6fcba27745f04c1 ./token/mock/CheckPointHistoryMock.sol
efbc2243520e596bbd7c7a7d1f2478b31a109cba0665ff99148bc6a218e6fa4f ./token/mock/TransferToWflrMock.sol
474bb95a1a5a63736ba31c8b8aed2b30d416197c23973e94d3ffbc8060d73ebe ./token/mock/VotePowerMock.sol
8a06b88a399e75a55a404fb8fbcc970db8643f6b47a5ce4ab24a306dc1eb4d3c ./token/mock/ExplicitDelegationMock.sol
81d0d2ed4f8baf2a0f10f1e42c32211e0e053b08fcc3ee1a190e988f2b04eab3 ./token/mock/DelegatableMock.sol
1500e26abd3a2294262e58f75b8f748703ca075da5216518833ae5d0d9b5a118 ./token/mock/CheckPointableMock.sol
665b02aa8b2de933877ea357380fda8d456c05c2ca2bafb701a93853de1cb031 ./token/implementation/WFlr.sol
95331f4166aa0416959c264a63bbe69b1bb00b9d3831513d44ef9cd18521657f ./token/implementation/CheckPointable.sol
5abf84373aea3196b1ab4f74b945937773530c37abe70c15d154ed651cbabf75 ./token/implementation/VPToken.sol
28f898059f24b6fa90bfc943787ab705eecbfb60f6e2abb8099b03f065e74d31 ./token/implementation/Delegatable.sol
ad54442c6eab1e13cbd71d9afcf1f1e6bb1b09f32a88b579ad489fcb595ac600
./token/implementation/CleanupBlockNumberManager.sol
e33ec7e494fce4f533163331a336d59079e4f8a34fad061025c8e86b85b54996 ./token/implementation/VPContract.sol
9e20303a56a6f568536d8ee5297888df46055be0a60ae0233358d97f0dabeaec ./token/lib/DelegationHistory.sol
214342adb4251914a86776fdabc3f5666846d673ca14026424e1033edb0ac9dd ./token/lib/CheckPointHistoryCache.sol
7ade30958ce8f8c063284a805e9d0cc81472dfcfb5913a9ac87e4a3e40eefbdb ./token/lib/ExplicitDelegation.sol
975449c3564c2777c2e5345ebcf264dec6251bceb9401e93e330cf6628e65ec0 ./token/lib/PercentageDelegation.sol
1f245cacad26d8f1de6d6bad9d98d1d36ea238f862096d49c5ca44228f47869b ./token/lib/VotePowerCache.sol
47b9d40515de40aa3955b175d05a052ace9a49bd286f8e34e2d119fbe21c662a ./token/lib/VotePower.sol
2b80bce2e24d9bc1fdcd7c40ab75912b3137400b3307bcd643b0be7ec34a8794 ./token/lib/CheckPointsByAddress.sol
699a525427b0320b46b0d2ee15c05b30d94c768609c3f70ffa0fd9f73d66e973 ./token/lib/CheckPointHistory.sol
8e6c13265d5663ddc6d7ec875aafdf2b5651b6d9617a3716eb14555c255cb5aa ./token/interface/IICleanable.sol
8fa276e095a28d48a5d852365d377f0659b136f48e6799903b05950169172ace ./token/interface/IIGovernanceVotePower.sol
9ed8ef3eea363a870a12986cc58246f7753d7812f0b8b7564486daf465ac0d6c ./token/interface/IIVPContract.sol
522831bde385e5157220e44d3f77d7aa77f28efa8c9c1d0ff92cc06c995b7966 ./token/interface/IIVPToken.sol
15bbe0293021fd0c1010bcaf2dfe2f663983e1e5a9b72819e1d8200e5e114b8b ./token/interface/IIHistoryCleanup.sol
2edb2932ad6d8ef5865ff06895e6c24c26e4102e9c46bf22d550e0e4da0018a8 ./supply/implementation/Supply.sol
fd48455ffc44b6df0c335df9c261c2ed050dee4908a3c90dc178fe7554538e89 ./supply/interface/IIRewardPool.sol
85c837056aa319146e08dc16954d13ccf4cb89bd06217785b3d0bd5c78f468db ./supply/interface/IISupply.sol

The following documentation was consulted during this engagement:

1. Flare design review presentation prepared by Flare's team
2. The Flare Network and Spark Token whitepaper https://flare.xyz/app/uploads/2020/08/flare_v1.1.pdf
3. System specification documentation https://gitlab.com/flarenetwork/flare-smart-contracts/-/blob/audit_2_coinspect/docs/specs/
4. Flare Consensus Protocol FCP.pdf

The consensus protocol used in Flare network, the attacks that could arise because of its particularities and the design of the economic incentives or lack thereof were not in scope for this audit.

3. Assessment

This audit focused on several smart contracts that implement system critical functionality for the Flare Network such as:

- Token contracts
- Flare Keeper, a special system trigger contract.
- Flare Inflation tracking and allocation.
- Reward manager and distribution to rewarded services
- Supply accounting system of FLR tokens
- Vote Power (delegation mechanisms and revocation)
- FTSO (Flare Time Series Oracles)
- FTSO Manager
- Price providers and the whitelisting process

A detailed explanation of how these components work and interact can be found in Flare Network's website and project repository and the material listed in the previous section of this report.

The specifications include attack scenarios and how they were contemplated and mitigated and the rationale behind several choices made for the implementation.

The smart contracts are specified to be compiled with Solidity compiler version 0.7.6. The repository includes a comprehensive set of unit and integration tests.

These contracts are designed to be upgradable, and most of their parameters can be reconfigured after deployment by a special Governance address controlled by the Flare team.

During this engagement, several potential attack scenarios were considered and evaluated by reading the code and writing tests, including:

- Pseudo random number generation and utilization
- Potential issues with vote power delegation
- Revocation of vote power delegated in the past and its related cached view
- Price providers collusion to manipulate prices
- Price providers collusion to unfairly accumulate rewards
- Transactions replay and the commit/reveal scheme used to submit prices to oracles
- Accounting of balances maintained by the system contracts
- Data structures and gas utilization abuse

For this audit, Coinspect auditors assumed the security properties granted by the Flare Network consensus layer behaved as described in the whitepaper. More specifically: reorganizations and the manipulation of transactions ordering were considered impossible.

Even though the cryptoeconomics and incentives mechanisms were not fully evaluated during this engagement, which focused on the implementation, **some high-level concerns were identified during the code review in relation to oracle price manipulation scenarios and are described in the following paragraphs**.

Incentive not to harm FLR value

The FLR holders have a strong incentive to keep the FTSO oracle system correctly working.

However, Coinspect observed that when voting, the voting power does not represent the current token balances/delegation status. Voting power is calculated using a checkpoint in the past.

This is documented in the system specs:

Voting campaigns that don't involve token locks should be able to use a vote power snapshot. For this, all vote power data is checkpointed when updated. Any voting campaign in flare will use a randomly chosen block number from the past. This means when any address casts its vote for a specific campaign, its vote power would be taken from a specific past block that was chosen for this campaign. The address vote power for this campaign will not reflect its present balance and delegation. This design allows for a usable (non-locked) token and a single voter power snapshot of token holdings. Voting campaigns are a generic concept; in flare oracle, (FTSO) price feeds will use the vote power data to choose the "correct price". Meaning each price submission will be weighted according to the vote power scheme described here.

Due to reward distribution constraints that will be described in the reward manager specification, the same vote power block will be used for a rather long period of time. This time frame will be named a "reward epoch" which will include many short price epochs. Meaning, FTSO price feeds commencing over a period of a few days will continuously derive vote power from the same vote power block in the past.

As a consequence, malicious actors could:

1. Obtain FLR tokens
2. Wait for the voting power block to be elected
3. Sell their FLR tokens
4. Keep voting for the duration of the current reward epoch (which lasts 5-7 days) with the voting power calculated to their balance before selling the tokens.

Collusion attacks

According to the system's documentation:

Flare does not rely on money or economic incentives to secure the network. The incentive for F-asset holders to provide data to the system is the safety of their application. FLR token holders are incentivized by the potential to earn something called the oracle reward.

The FTSO or 'Flare Time Series Oracle' is a service used to gather price (and data) signals of off-chain assets for consumption by other smart contracts on the Flare blockchain. The system is designed to encourage FLR holders to submit accurate prices to the FTSO contracts.

Users will submit prices in epochs of a few minutes. And rewards are distributed in reward epochs of several days. Vote power can be delegated and revoked.

The oracles implement a protection mechanism to prevent the following scenario where everybody submits same price:

In a collusion attack, where a majority (based on weight) of submitters agree on a price and submit that same price, the weighted median is determined by these submitters. In addition, if we reward all the submissions that have the same price as the already rewarded submissions that are strictly within the 25% of the median, all such submitters are automatically rewarded.
This would create the incentive to just submit the same price as everyone else and participating in the collusion brings guaranteed reward. In addition, any deviation from the agreed upon price practically guarantees that such a submission would not be rewarded.

Coinspect observed one drawback for this protection mechanism: it could be counterproductive for non-volatile assets such as stable coins, whose price might stay constant for several epochs, discouraging price providers from submitting prices to this FTSO as they might not get rewarded.

It is not clear if this protection and the rewards are enough to prevent collusion attacks if the majority agrees to manipulate the price and where the resulting profit is enough to compensate for the rewards lost to the random selection in the edge cases. Also, note that F-asset holders are not rewarded, and their only incentive to vote the right price is to maintain their asset value. F-asset holders and FLR holders can be viewed as an opposing force to the FLR holders.

The revocation of ill-behaved actors' voting power only applies to delegated voting power; the actual voting power of token holders can not be revoked.

As detailed above, the colluding actors could sell their tokens before the collusion takes place.

It is worth adding that there are other protections in place to control arbitrary price manipulation, for example, a configurable maximum deviation from the previously agreed price.

4. Summary of Findings

ID	Description	Risk	Fixed
FLR-001	WLFR token unlimited free minting	High	✔
FLR-002	Inflation contract locked after failing balance assertion	High	✔
FLR-003	Double voting power	Medium	✔
FLR-004	FTSO oracle can manipulate the election mechanism to perpetually receive all rewards	High	✔
FLR-005	OOG results in errors not logged	Info	✘

During August 2021, Coinspect verified all security relevant findings had been correctly addressed by the Flare team.

5. Detailed Findings

FLR-001	WLFR token unlimited free minting		
Total Risk **High**	Impact High	Location token/implementation/WFlr.sol	
Fixed ✔	Likelihood High		

Description

Attackers can mint unlimited amounts of the WFLR (wrapped FLR) for free.

The WLFR tokens can then be used to inflate the attackers voting power. Alternatively, the attackers can empty the FLR stored in the WLFR contract by withdrawing the freshly minted WFLR and prevent users from recovering their FLR tokens.

The vulnerable code is located in the `depositTo` function in the WLFR token contract. The payable function receives the FLR, mints new WFLR tokens to the intended recipient, and then forwards the received FLR value to the recipient as well, instead of storing it in the contract:

```
/**
 * @notice Deposit Flare from msg.sender to recipient and and mint WFLR ERC20.
 * @param recipient A payable address to receive Flare and minted WFLR.
 */
function depositTo(address payable recipient) external payable override {
    require(recipient != address(0), "Cannot deposit to zero address");
    // Mint WFLR
    _mint(recipient, msg.value);
    // Emit deposit event
    emit Deposit(recipient, msg.value);
    // Transfer Flare to recipient (last statement, to prevent reentrancy)
    recipient.transfer(msg.value);
}
```

It is worth noting the function documentation matches the current implementation. Moreover, there is a unit test in WFLR.ts (`"Should accept FLR deposits from another account.”`) which passes because it also verifies the current and documented behavior.

Recommendation

Remove the `transfer` call from the function. For example, see the WETH10 implementation: https://github.com/WETH10/WETH10/blob/e952d1ec4c149e85d93ed2ce4040ac571ed6bc19/contracts/WETH10.sol#L94.

Status

Fixed at commit `e8464d575ccbffb5a11144887e5b99c604c90b62` by removing the `transfer` call.

FLR-002 Inflation contract locked after failing balance assertion

Total Risk	Impact	Location
High	High	contracts/inflation/implementation/Inflation.sol
Fixed	Likelihood	
✔	High	

Description

Attackers can prevent the Inflation process from executing, blocking the distribution of newly minted FLR tokens and the payment of reward claims as a result.

The situation seems unrecoverable until a new `Inflation` contract is deployed and registered in the `Keeper` contract. This action must be performed by Governance and could require more or less time depending on how Governance decisions are made and executed at the moment in time where the bug is triggered.

The `Inflation` contract fails to handle `FLR` sent via self destruct contracts. This causes the `mustBalance` modifier to fail and revert the calls made to the `receiveMinting` method by the Flare `Keeper` contract.

The `Inflation` contract code attempts to consider `selfDestruct` originated balances, but the check is incorrect as it compares the `msg.value` of the call with the expected `topup amount`.

```
[contracts/inflation/implementation/Inflation.sol]
function receiveMinting() external payable onlyFlareKeeper mustBalance {
    uint256 amountPostedWei = inflationAnnums.receiveTopupRequest();
    // Assume that if we got more than we posted, we must have been a self-destruct
    // recipient in this block.
    uint256 selfDestructProceeds = msg.value.sub(amountPostedWei);
    if (selfDestructProceeds > 0) {
```

```
        totalSelfDestructReceivedWei =
              totalSelfDestructReceivedWei.add(selfDestructProceeds);
    }
    emit MintingReceived(amountPostedWei, selfDestructProceeds);
}
```

```
[contracts/inflation/implementation/Inflation.sol]
function getExpectedBalance() private view returns(uint256 _balanceExpectedWei) {
    return inflationAnnums.totalInflationTopupReceivedWei
        .sub(inflationAnnums.totalInflationTopupWithdrawnWei)
        .add(totalSelfDestructReceivedWei)
        .sub(totalSelfDestructWithdrawnWei);
}
```

The `receiveMinting` function in the `Inflation` contract is called from the `FlareKeeper trigger` function, which does not take into account sudden changes in the `Inflation` contract's balance:

```
[contracts/genesis/implementation/FlareKeeper.sol]
function trigger() external inflationSet mustBalance returns (uint256 _toMintWei) {
    require(block.number > systemLastTriggeredAt, ERR_BLOCK_NUMBER_SMALL);
    systemLastTriggeredAt = block.number;
    uint256 currentBalance = address(this).balance;

    if (currentBalance > lastBalance) {
        uint256 balanceExpected = lastBalance.add(expectedMintRequest);
        if (currentBalance == balanceExpected) {
            uint256 minted = currentBalance.sub(lastBalance);
            totalMintingReceivedWei = totalMintingReceivedWei.add(minted);
            emit MintingReceived(minted);
            try inflation.receiveMinting{ value: minted }() {
                totalMintingWithdrawnWei = totalMintingWithdrawnWei.add(minted);
                emit MintingWithdrawn(minted);
            } catch Error(string memory message) {
                addKeepError(address(this), message);
            }
```

An attacker can abuse this fact to prevent the `Inflation` contract from functioning normally.

Recommendation

The `receiveMinting` function should compare the last known balance against the current contract balance and assume that the difference came from self-destructed contracts.

Status

Fixed at commit `b18566d19d2fb7a67492dfd8b0db59397ac73025` following the same approach used in similar files when considering the self-destruct originated funds.

FLR-003 Double voting power

Total Risk	Impact	Location
Medium	Medium	contracts/token/lib/VotePowerCache.sol
Fixed	Likelihood	
✔	High	

Description

An attacker can abuse the voting power cache system to double its voting power.

The vote power cache enables a delegator to revoke the voting power it delegated in a block in the past.

The main issue arises when the revocation of the delegated power is recovered by the revocator.

```
[contracts/token/lib/VotePowerCache.sol]
function revokeAt(
        CacheState storage _self,
        VotePower.VotePowerState storage _votePower,
        address _from,
        address _to,
        uint256 _revokedValue,
        uint256 _blockNumber
    ) internal {
    if (_revokedValue == 0) return;
    bytes32 keyFrom = keccak256(abi.encode(_from, _blockNumber));
    if (_self.revocationCache[keyFrom].revocations[_to] != 0) {
        revert("Already revoked");
    }
    (uint256 valueFrom,) = valueOfAt(_self, _votePower, _from, _blockNumber);
    (uint256 valueTo,) = valueOfAt(_self, _votePower, _to, _blockNumber);
    bytes32 keyTo = keccak256(abi.encode(_to, _blockNumber));
    _self.revocationCache[keyFrom].revokedTotal =
                _self.revocationCache[keyFrom].revokedTotal.add(_revokedValue);
     _writeCacheValue(_self, keyFrom, valueFrom.add(_revokedValue));
```

```
    _writeCacheValue(_self, keyTo, valueTo.sub(_revokedValue,
       "Revoked value too large"));
    _self.revocationCache[keyFrom].revocations[_to] = _revokedValue;
}
```

The attackers can exploit this issue to delegate power to a secondary account, vote with the secondary account, revoke the power given to that account and then vote with the primary account again.

Voting power is accounted for in the `revealPrice` function in the `FTSO` contract. The value in the cache is used for this purpose:

```
[contracts/ftso/implementation/Ftso.sol]
function _revealPrice(address _voter, uint256 _epochId, uint256 _price, uint256
_random) internal {
    require(_price < 2**128, ERR_PRICE_TOO_HIGH);
    require(epochs._epochRevealInProcess(_epochId), ERR_PRICE_REVEAL_FAILURE);
    require(epochVoterHash[_epochId][_voter] == keccak256(abi.encode(_price, _random,
_voter)), ERR_PRICE_INVALID);
    // get epoch
    FtsoEpoch.Instance storage epoch = epochs.instance[_epochId];
    require(epoch.initializedForReveal || (epoch.fallbackMode &&
epochs.trustedAddressesMapping[_voter]), ERR_EPOCH_NOT_INITIALIZED_FOR_REVEAL);

    (uint256 votePowerFlr, uint256 votePowerAsset) = _getVotePowerOf(epoch, _voter);
    uint256 voteId = votes._createInstance(
            _voter,
            votePowerFlr,
            votePowerAsset,
            epoch.votePowerFlr,
            epoch.votePowerAsset,
            _price
        );
...
```

This function calls `_getVotePowerOfAt` which in turns calls `votePowerOfAtCached`.

This attack can be performed one price epoch per reward epoch. Depending on how the attacked oracle price is being used, its manipulation could result in more or less severe consequences. For example: one price epoch could trigger liquidations of many collateralized positions.

Recommendation

Do not count revoked vote power when calculating an account's vote power.

Status

Fixed at commit `9b60294a8e3e8721d21a91db56c72e62324bfe0c` by not adding the voting power to the delegating account.

FLR-004	FTSO oracle can manipulate the election mechanism to perpetually receive all rewards	
Total Risk **High**	Impact High	Location contracts/ftso/implementation/FtsoManager.sol
Fixed ✔	Likelihood Medium	

Description

An FTSO oracle can abuse the pseudo-random mechanism utilized to pick which FTSO oracle (from all the FTSO oracles available) will be rewarded for the current price epoch in order to continue being elected to get rewarded for all the following price epochs.

FTSO rewards are the main incentive for price providers to supply up-to-date data to the FTSO subsystem. For every price epoch, a randomly chosen FTSO oracle obtains all the rewards being distributed for that epoch.

Because the random value generated by the previous epoch's elected FTSO is utilized, once a FTSO is elected for rewards, it can manipulate the process to continue being elected indefinitely. In order to do so, the malicious FTSO's price providers would need to collide, and will benefit with all the inflation generated rewards allocated to the Flare's FTSO subsystem. As a result, the remaining FTSOs in the system will never be rewarded for their services.

The rewards distribution and FTSO election mechanism are implemented in the `_finalizePriceEpoch` function:

```
    /**
```

```
     * @notice Finalizes price epoch
     */
    function _finalizePriceEpoch(IIFtso[] memory _ftsos) internal {
        uint256 numFtsos = _ftsos.length;

        // Are there any FTSOs to process?
        if(numFtsos > 0) {
            // choose winning ftso
            uint256 chosenFtsoId;
            if (lastUnprocessedPriceEpoch == 0 ||
                        priceEpochs[lastUnprocessedPriceEpoch-1].chosenFtso == address(0)) {
                // pump not yet primed
                //slither-disable-next-line weak-prng           // only used for first epoch
                chosenFtsoId = uint256(keccak256(abi.encode(
                        block.difficulty, block.timestamp
                    ))) % numFtsos;

            } else {
                // at least one finalize with real FTSO

                  // ftso random calculated safely from inputs
                  chosenFtsoId =
                        uint256(keccak256(abi.encode(
                               IIFtso(priceEpochs[lastUnprocessedPriceEpoch-1].chosenFtso
                        ).
                                       getCurrentRandom()
                    ))) % numFtsos;
            }
```

When contrasted with the documentation in /docs/specs/FTSO.md, Coinspect noticed the implementation differs from the expected as described in the following quote:

> *When finalizing a reward epoch, FtsoManager sums up the random values across all the FTSOs that had submissions in the given period, adds the epoch timestamp, hashes the result and reduces it modulo the number of submitting FTSOs to determine which FTSO is considered for the reward.*

Coinspect observed the `_finalizeRewardEpoch` function does also generate a pseudo random value, in this case, in order to select the power block intended to be

used for the next reward epoch. However, in this case, the values provided from all the FTSOs in the system are used in addition to the current block timestamp.

The following output log excerpt from a Coinspect's proof of concept test shows how once being selected, a FTSO can perpetuate itself as the election winner for all the following rounds:

```
. epoch #3: submitting prices
. epoch #3: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
80245733601281189815040401495987632527441427983030161085959315928336261143214
_finalizePriceEpoch() chosenFtsoId = 3
_finalizePriceEpoch() epoch #3 -> chosenFtso = 0x18b9306737eaf6e8fc8e737f488a1ae077b18053


. epoch #4: submitting prices
. epoch #4: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
98851696006435613807158402720063884020960140988567086050201156353501125977248
_finalizePriceEpoch() chosenFtsoId = 2
_finalizePriceEpoch() epoch #4 -> chosenFtso = 0x22474d350ec2da53d717e30b96e9a2b7628ede5b


. epoch #5: submitting prices
. epoch #5: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
98851696006435613807158402720063884020960140988567086050201156353501125977248
_finalizePriceEpoch() chosenFtsoId = 2
_finalizePriceEpoch() epoch #5 -> chosenFtso = 0x22474d350ec2da53d717e30b96e9a2b7628ede5b


. epoch #6: submitting prices
. epoch #6: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
17897577689199082549040705789028457944578058190922433190972842936040427815 14
_finalizePriceEpoch() chosenFtsoId = 2
_finalizePriceEpoch() epoch #6 -> chosenFtso = 0x22474d350ec2da53d717e30b96e9a2b7628ede5b


. epoch #7: submitting prices
```

```
. epoch #7: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
53043096227359053818872138093809337670750929190802556077421333924524463103737
_finalizePriceEpoch() chosenFtsoId = 2
_finalizePriceEpoch() epoch #7 -> chosenFtso = 0x22474d350ec2da53d717e30b96e9a2b7628ede5b


. epoch #8: submitting prices
. epoch #8: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
60748059476122350776289021709948099136229761952063928055508919721071551745963
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #8 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e

!!! ftso0 was chosen!

!!! ftso0 will get the rewards forever from now on


. epoch #9: submitting prices
. epoch #9: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
69905498232133061315283679783375232193375639567589363798167845350809654601651
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #9 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e
!!! ftso0 was chosen!


. epoch #10: submitting prices
. epoch #10: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
69905498232133061315283679783375232193375639567589363798167845350809654601651
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #10 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e
!!! ftso0 was chosen!


. epoch #11: submitting prices
. epoch #11: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
69905498232133061315283679783375232193375639567589363798167845350809654601651
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #11 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e
!!! ftso0 was chosen!
```

```
. epoch #12: submitting prices
. epoch #12: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
69905498232133061315283679783375232193375639567589363798167845350809654601651
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #12 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e
!!! ftso0 was chosen!


. epoch #13: submitting prices
. epoch #13: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
69905498232133061315283679783375232193375639567589363798167845350809654601651
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #13 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e
!!! ftso0 was chosen!


. epoch #14: submitting prices
. epoch #14: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
69905498232133061315283679783375232193375639567589363798167845350809654601651
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #14 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e
!!! ftso0 was chosen!


[...]


. epoch #19: submitting prices
. epoch #19: revealing prices
_finalizePriceEpoch() prev chosenFtso getCurrentRandom() =
69905498232133061315283679783375232193375639567589363798167845350809654601651
_finalizePriceEpoch() chosenFtsoId = 0
_finalizePriceEpoch() epoch #19 -> chosenFtso = 0x53369fd4680ffe3dff39fc6dda9cfbfd43daea2e
!!! ftso0 was chosen!

    ✓ Price Providers in a FTSO collide to reap all rewards perpetually
```

By colliding in this manner, price providers in a single FTSO could harm all FLR holders.

Recommendation

Improve the FTSO selection process used for reward distribution purposes in order to prevent a single FTSO oracle from collecting all rewards being distributed.

Status

Fixed at commit `b2dac9d7a41030719108ef065159924e410df987` by using all the FTSOs random values.

FLR-005 OOG results in errors not logged

Total Risk	Impact	Location
Info	Low	contracts/genesis/implementation/FlareKeeper.sol
Fixed	Likelihood	
✘	Low	

Description

The Flare Keeper contract will not be able to emit error events and handle the out of gas scenario if one of the registered contracts runs out of gas when its `keep` function is invoked and it has not been configured with a gas limit.

The following code in the `trigger` function iterates over all the registered contracts:

```
// Perform trigger operations here
for (uint256 i = 0; i < len; i++) {
    uint256 blockHoldoffRemainingForContract = blockHoldoffsRemaining[keepContracts[i]];
    if (blockHoldoffRemainingForContract > 0) {
        blockHoldoffsRemaining[keepContracts[i]] = blockHoldoffRemainingForContract - 1;
       emit ContractHeldOff(address(keepContracts[i]), blockHoldoffRemainingForContract);
    } else {
        // Figure out what gas to limit call by
        uint256 startGas = gasleft();
        uint256 gasLimit = gasLimits[keepContracts[i]];

        uint256 useGas = gasLimit > 0 ? gasLimit : startGas;
        // Run keep for the contract, consume errors, and record
        try keepContracts[i].keep{gas: useGas} ()
        {
            emit ContractKept(address(keepContracts[i]));
        // Catch all requires with messages
        } catch Error(string memory message) {
            addKeepError(address(keepContracts[i]), message);
        // Catch everything else...out of gas, div by zero, asserts, etc.
        } catch {
            uint256 endGas = gasleft();
            // Interpret out of gas errors
            if (startGas.sub(endGas) >= gasLimits[keepContracts[i]]) {
```

```
                    blockHoldoffsRemaining[keepContracts[i]] = blockHoldoff;
                    addKeepError(address(keepContracts[i]), ERR_OUT_OF_GAS);
                } else {
                    // Don't know error cause...just log it as unknown
                    addKeepError(address(keepContracts[i]), "unknown");
                }
            }
        }
    }
```

This only applies to the case where a gas limit has not been configured for a contract.

Recommendation

Do not pass all gas left in the Keeper to the contract, but keep some gas in order to enable the handling of the scenario where all the gas passed got consumed.

6. 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.