hexens × Secured Finance

DEC.24

SECURITY REVIEW REPORT FOR SECURED FINANCE

CONTENTS

- Initialize can be frontrun
- Post-commit comment issue
- Public constants should be marked as private
- Unused FILBalanceUpdated event
- Using experimental ABIEncoderV2 in production
- Development imports in production code
- Redundant calculation in price feed
- owner Arguments are Shadowed from OwnableUpgradeable::owner in DebtToken.sol
- Consider renaming inaccurate Chainlink references

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the smart contracts of Secured Finance, a stable coin protocol on the Filecoin EVM blockchain. The protocol is a fork of Liquity V1, with some changes to configuration variables and the price feed oracles.

Our security assessment was a full review of the code differences, spanning a total of 3 weeks.

During our audit, we have identified 1 high severity vulnerability and 4 medium severity vulnerabilities. These issues were all found in the code changes compared to Liquity.

We have also identified several minor severity vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/Secured-Finance/stablecoin-contracts/tree/2d585be1d6d03f1f6e8cb6acb0e46837315928a9

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

AUDITING DETAILS

STARTED
30.12.2024

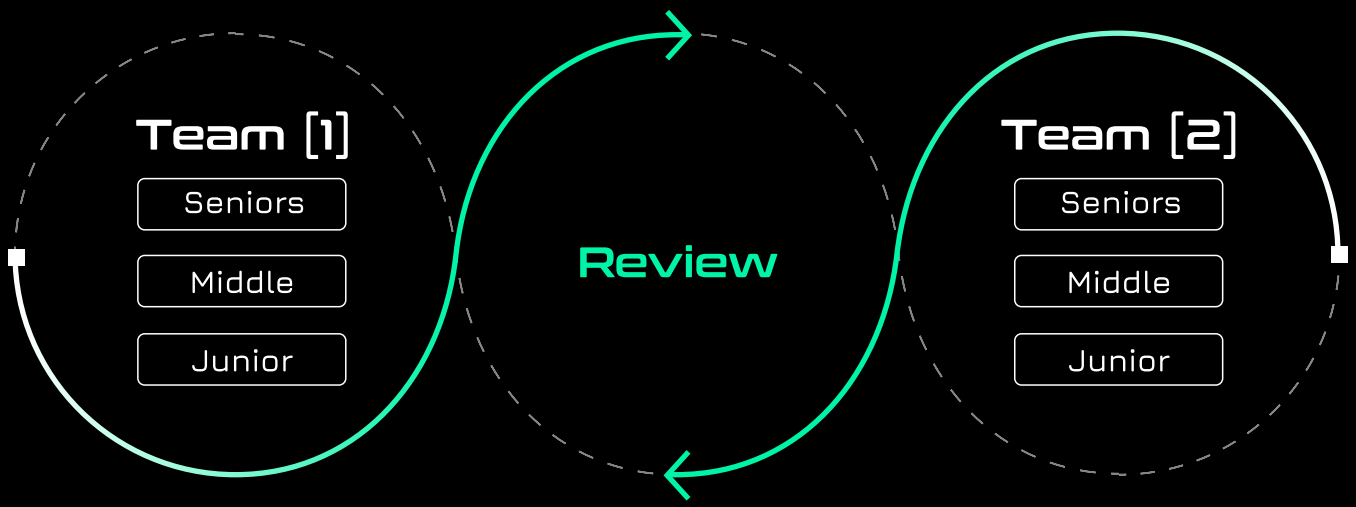
DELIVERED
20.01.2025

Review Led by

HANNAY AL MOHANNA
Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

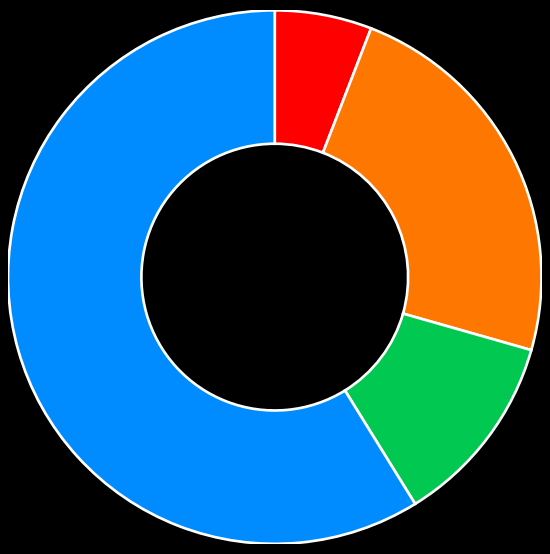
ISSUE SYMBOLIC CODES

Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

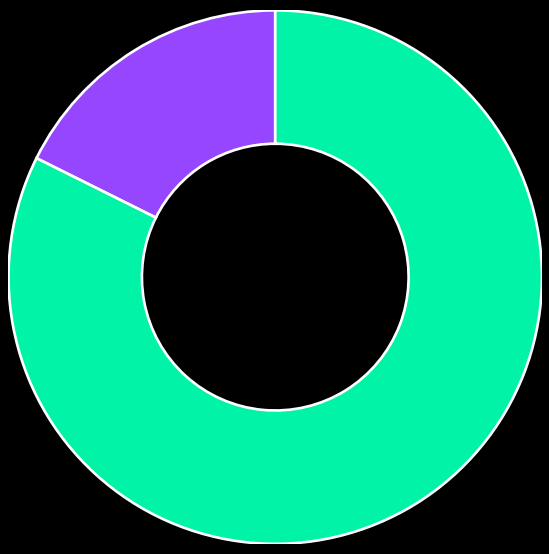
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	4
Low	2
Informational	10

Total: 17



- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

SECFIN1-15

DOS IN OFF-CHAIN TOKEN ALLOCATION PROCESS BY BACKRUNNING LOCKUPCONTRACTFACTORY::DEPLOY LOCKUPCONTRACT

SEVERITY: High

PATH:

deployments/allocate.js#L70-L86

REMEDIATION:

Consider deploying the beneficiary lockup contracts first and validating correct on/off-chain states before allocating large token amounts to the admin, Uniswap pool, and community issuance addresses. This allows for the lockup contracts to be redeployed before significant sums of tokens are moved, should any MEV or front/backrunning issues occur which result in unintended states.

The issues reported in finding SECFIN1-14 and particularly finding SECFIN1-16 should also be remediated before token allocation/significant on-chain state changes are attempted to further limit the risk of race condition and state desync issues.

STATUS: Fixed

DESCRIPTION:

Beneficiary protocol token allocation logic is outlined in the **deployments/allocate.js** utility. The logic for mainnet allocation can be summarized as follows:

1. Protocol tokens are initially allocated among the admin, Uniswap pool, and community issuance vault addresses.
2. **LockupContract** contracts are then deployed for new protocol token investor addresses stored in the **deployments/inputs/mainnet.js** config file.
3. The **deployer** address calls **LockupContractFactory::deployLockupContract(investorAddr, oneYearFromDeployment)**, deploying lockup contracts with a fixed one year duration, via the **sendAndWaitForTransaction** helper utility, which uses the ethers.js **wait()** method to wait for 1 block confirmation.
4. After the transaction has been confirmed, the beneficiary for the newly deployed lockup contract address is queried via the **LockupContractFactory::beneficiaryToLockupContract** getter.
5. The local deployment state is saved to a JSON mapping for the current environment (e.g., **deployments/outputs/mainnet.json**), using the transaction hash from step 3 and the beneficiary address from step 4.
6. A final set of checks are performed to ensure that valid lockup contracts with the correct one year timelock have been deployed for each expected beneficiary.

An issue is present where any existing beneficiary - or via exploitation of finding **SECFIN1-14** , an arbitrary non-beneficiary address - can perform a back-running attack between steps 3 and 4 above, by calling **LockupContractFactory::deployLockupContract(investorAddr, _unlockTime)** with a valid beneficiary/investor address and/or any **_unlockTime** further than **block.timestamp + 1 years**.

By setting a lockup period beyond one year during the back-ran call, an attacker can deliberately cause the final **unlockTime** check to fail in step 6, resulting in a nodeJS **AssertionError**. Subsequent token allocation steps may then halt, requiring the allocation process to be restarted.

Alternatively, if subsequent token allocation steps do not halt or do not account for this error before sending funds to the lockup addresses saved in the deployment state, then this could result in investors funds being permanently locked should an attacker choose to set an excessive unlock time during the back-ran calls. This could result in a substantial loss of funds.

If values that do not result in an **AssertionError** for **_beneficiary** and **_unlockTime** are used during the back-ran call, all beneficiaries "valid" lockup contracts saved to both the off-chain environment deployment state and **LockupContractFactory** will have a deployer set to the attacker's address through the **lockupContractToDeployer** mapping.

```
//deployments/allocate.js
...
//@audit Step 1
// Allocate tokens to the admin, community issuance, and unipool
  const accounts = [configParams.walletAddrs.ADMIN, unipool.address,
communityIssuance.address];
  const amounts = [
    configParams.allocationAmounts.ADMIN,
    configParams.allocationAmounts.UNIPOOL,
    configParams.allocationAmounts.COMMUNITY_ISSUANCE,
  ];
...
const oneYearFromDeployment = (Number(supplyStartTime) +
timeVals.SECONDS_IN_ONE_YEAR).toString();
...
  // Deploy LockupContracts - one for each beneficiary
  const lockupContracts = {};

  for (const [investor, investorAddr] of
Object.entries(configParams.beneficiaries)) {
    ...
    // @audit Step 2
    if (deploymentState[investor] && deploymentState[investor].address) {
        ...
    } else {
      // @audit Step 3
      const txReceipt = await mdh.sendAndWaitForTransaction(
        lockupContractFactory.deployLockupContract(investorAddr,
oneYearFromDeployment),
      );
      // @audit Step 4
      const address = await
lockupContractFactory.beneficiaryToLockupContract(investorAddr);
      lockupContracts[investor] = new ethers.Contract(
        address,
        lockupContractEthersFactory.interface,
        deployerWallet,
      );
```

```
      // @audit Step 5
      deploymentState[investor] = {
        address: address,
        txHash: txReceipt.transactionHash,
      };

      mdh.saveDeployment(deploymentState);
    }
    ...
  }
  console.log("LOCKUP CONTRACT CHECKS");
  // @audit Step 6
  // Check lockup contracts exist for each beneficiary with correct unlock
time
  for (investor of Object.keys(lockupContracts)) {
    const lockupContract = lockupContracts[investor];
    // check LC references correct ProtocolToken
    const storedProtocolTokenAddr = await lockupContract.protocolToken();
    assert.equal(protocolToken.address, storedProtocolTokenAddr);
    // Check contract has stored correct beneficary
    const onChainBeneficiary = await lockupContract.beneficiary();
    assert.equal(
      configParams.beneficiaries[investor].toLowerCase(),
      onChainBeneficiary.toLowerCase(),
    );
    // Check correct unlock time (1 yr from deployment)
    const unlockTime = await lockupContract.unlockTime();
    assert.equal(oneYearFromDeployment, unlockTime);
    ...
  }
...
```

Proof of Concept:

Add the following code block to **deployments/allocate.js**, including the attacker's simulated back-ran call to **lockupContractFactory.deployLockupContract**, before executing the allocation logic against a local testnet:

```
//deployments/allocate.js
...
++    const attackerWallet = ethers.Wallet.createRandom()
++    const attackerSigner = attackerWallet.connect(ethers.provider)
++    const attackerFactory = lockupContractFactory.connect(attackerSigner)
++    const hundredYearFromDeployment = (Number(supplyStartTime) +
timeVals.SECONDS_IN_ONE_YEAR * 100).toString();

      await deployerWallet.sendTransaction({to:
attackerWallet.address,value: ethers.utils.parseEther("1.0")})

      const txReceipt = await mdh.sendAndWaitForTransaction(
        lockupContractFactory.deployLockupContract(investorAddr,
oneYearFromDeployment),
++      attackerFactory.deployLockupContract(investorAddr,
hundredYearFromDeployment)
      );

      const address = await
lockupContractFactory.beneficiaryToLockupContract(investorAddr);

...
```

Observe that the final lockup contract checks will raise an **AssertionError** due to the excessive lockup. Note that this will have occurred after the on-chain admin/unipool/community vault tokens have been allocated, and crucially after the local deployment state has been saved.

```
...
LOCKUP CONTRACT CHECKS
AssertionError: expected '1768009691' to equal BigNumber{ _hex:
'0x0123789a5b', …(1) }
    at main (.../deployments/allocate.js:124:12)
    at processTicksAndRejections (node:internal/process/task_queues:105:5) {
  showDiff: true,
  actual: '1768009691',
  expected: BigNumber { _hex: '0x0123789a5b', _isBigNumber: true },
  operator: 'deepStrictEqual'
}
...
```

SECFIN1-14

MISSING BENEFICIARY SIGNATURE VALIDATION IN LOCKUPCONTRACTFACTORY::DEPLOY LOCKUPCONTRACT

SEVERITY: Medium

PATH:

contracts/ProtocolToken/LockupContractFactory.sol#L53-L70

REMEDIATION:

Consider adding signature validation to LockupContractFactory::deployLockupContract, ensuring that investors/beneficiaries permissions are validated before their beneficiaryToLockupContract can be set to a new lockup contract. Alternatively, the LockupContractFactory owner may supply the signatures.

STATUS: Fixed

DESCRIPTION:

LockupContractFactory::deployLockupContract allows any account to create a **LockupContract** and set a **beneficiary** and **unlockTime**. The factory maintains two mappings, **beneficiaryToLockupContract** and **lockupContractToDeployer** , which store beneficiary to **LockupContract** contract mappings, and lockup contract addresses to their deployers respectively.

By calling **LockupContractFactory::deployLockupContract** with **_beneficiary** and **_unlockTime** arguments, is possible for an arbitrary account to overwrite the **beneficiaryToLockupContract** mapping for any existing beneficiary, essentially “resetting” the beneficiary’s lockup contract to a newly deployed empty contract without a means to set the

original contract address back. This also sets the **lockupContractToDeployer** mapping to the attacker's address for the newly deployed contract. The new lockup contract may also be set with an arbitrarily large unlock time.

As **LockupContractFactory** is considered the authority for lockup-to-beneficiary records, this issue affects any contract or off-chain asset relying on the factory for accurate records.

One critical component that relies on this mapping is the off-chain token allocation utility found in **deployments/allocate.js**, which relies on the factory to verify lockup addresses against expected **beneficiary** addresses during investor token allocation. Refer to finding SECFIN1-15 for further implications of this issue.

```
//contracts/ProtocolToken/LockupContractFactory.sol
...
    function deployLockupContract(address _beneficiary, uint _unlockTime)
external override {
        address protocolTokenAddressCached = protocolTokenAddress;
        _requireProtocolTokenAddressIsSet(protocolTokenAddressCached);
        LockupContract lockupContract = new LockupContract(
            protocolTokenAddressCached,
            _beneficiary,
            _unlockTime
        );

        lockupContractToDeployer[address(lockupContract)] = msg.sender;
        beneficiaryToLockupContract[_beneficiary] = address(lockupContract);
        emit LockupContractDeployedThroughFactory(
            address(lockupContract),
            _beneficiary,
            _unlockTime,
            msg.sender
        );
    }
...
```

ORACLE NOT VALIDATED DURING INITIALIZATION, POTENTIALLY RESULTING IN FETCHPRICE OF 0

SEVERITY: Medium

PATH:

contracts/PriceFeed.sol#L79-L93

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

_setAddresses doesn't verify if the oracle is actually working while setting the status to working, which can result in a **lastGoodPrice** being set to 0, making the fetch price 0.

When the price feed contract is initialized, it calls the **_setAddresses** function. In this function, it assumes the oracle is working and sets the initial status to **Status.chainlinkWorking**. If the oracle is broken, instead of reverting, it just returns an unsuccessful response with both the price and decimals set to 0.

Even so, it still calls **_storeChainlinkPrice(chainlinkResponse)**, and that response sets the **lastGoodPrice** to 0.

This causes the **fetchPrice** function to return 0 when the oracle isn't working, because it relies on the **lastGoodPricebeing** returned when the oracle fails.

```
//contracts/PriceFeed.sol
...
    function _setAddresses(address _priceAggregatorAddress, address
_tellorCallerAddress) private {
      checkContract(_priceAggregatorAddress);
      checkContract(_tellorCallerAddress);

      priceAggregator = AggregatorV3Interface(_priceAggregatorAddress);
      tellorCaller = ITellorCaller(_tellorCallerAddress);

      // Explicitly set initial system status
      status = Status.chainlinkWorking;

      // Get an initial price from Chainlink to serve as first reference for
lastGoodPrice
      ChainlinkResponse memory chainlinkResponse =
_getCurrentChainlinkResponse();

      _storeChainlinkPrice(chainlinkResponse);
}
...
    function _storeChainlinkPrice(
        ChainlinkResponse memory _chainlinkResponse
    ) internal returns (uint) {
        uint scaledChainlinkPrice = _scaleChainlinkPriceByDigits(
            uint256(_chainlinkResponse.answer),
            _chainlinkResponse.decimals
        );
        _storePrice(scaledChainlinkPrice);

        return scaledChainlinkPrice;
    }
...
    function _storePrice(uint _currentPrice) internal {
        lastGoodPrice = _currentPrice;
        emit LastGoodPriceUpdated(_currentPrice);
    }
...
```

Validate the oracle response in the **_setAddresses** function:

```
//contracts/PriceFeed.sol
...
function _setAddresses(address _priceAggregatorAddress, address
_tellorCallerAddress) private {
    checkContract(_priceAggregatorAddress);
    checkContract(_tellorCallerAddress);

    priceAggregator = AggregatorV3Interface(_priceAggregatorAddress);
    tellorCaller = ITellorCaller(_tellorCallerAddress);

    // Explicitly set initial system status
    status = Status.chainlinkWorking;

    // Get an initial price from Chainlink to serve as first reference for
lastGoodPrice
    ChainlinkResponse memory chainlinkResponse =
_getCurrentChainlinkResponse();
++  require(!_chainlinkIsBroken(chainlinkResponse) && !
_chainlinkIsFrozen(chainlinkResponse), "PriceFeed: Chainlink must be working
and current");
    _storeChainlinkPrice(chainlinkResponse);
}
...
```

SECFIN1-13

MISSING LOGIC TO COMPARE PYTH PRICE WITH TELLOR WHEN PYTH IS WORKING BUT PRICE IS INCORRECT

SEVERITY: Medium

PATH:

contracts/PriceFeed.sol#L117-L165

REMEDIATION:

Consider adding logic to switch to Tellor pricing if the price returned by Pyth deviates from a last known good Pyth price by 50%.

STATUS: Fixed

DESCRIPTION:

If the status is **chainlinkWorking** and the Pyth oracle is not frozen and **not broken**, the price is considered **valid**. However, the current logic doesn't handle switching to the Tellor oracle if the price returned by Pyth is too different from a previous price (50% lower or 50% higher).

It should compare the price to Tellor if the difference is too large. Since Pyth does not support retrieving the past rounds, the system should always compare the latest price to another reliable source that can be represented as the previous round such as last good price.

```
//contracts/PriceFeed.sol
...
    function fetchPrice() external override returns (uint price) {
        // Get current and previous price data from Chainlink, and current
price data from Tellor
        ChainlinkResponse memory chainlinkResponse =
_getCurrentChainlinkResponse();
        TellorResponse memory tellorResponse = _getCurrentTellorResponse();

        // --- CASE 1: System fetched last price from Chainlink  ---
        if (status == Status.chainlinkWorking) {
            // If Chainlink is broken, try Tellor
            if (_chainlinkIsBroken(chainlinkResponse)) {
                // If Tellor is broken then both oracles are untrusted, so
return the last good price
                if (_tellorIsBroken(tellorResponse)) {
                    _changeStatus(Status.bothOraclesUntrusted);
                    return lastGoodPrice;
                }
                /*
                 * If Tellor is only frozen but otherwise returning valid
data, return the last good price.
                 * Tellor may need to be tipped to return current data.
                 */
                if (_tellorIsFrozen(tellorResponse)) {
                    _changeStatus(Status.usingTellorChainlinkUntrusted);
                    return lastGoodPrice;
                }

                // If Chainlink is broken and Tellor is working, switch to
Tellor and return current Tellor price
                _changeStatus(Status.usingTellorChainlinkUntrusted);
                return _storeTellorPrice(tellorResponse);
            }

            // If Chainlink is frozen, try Tellor
            if (_chainlinkIsFrozen(chainlinkResponse)) {
                // If Tellor is broken too, remember Tellor broke, and
return last good price
                if (_tellorIsBroken(tellorResponse)) {
                    _changeStatus(Status.usingChainlinkTellorUntrusted);
                    return lastGoodPrice;
```

```
                }

                // If Tellor is frozen or working, remember Chainlink froze,
and switch to Tellor
                _changeStatus(Status.usingTellorChainlinkFrozen);

                if (_tellorIsFrozen(tellorResponse)) {
                    return lastGoodPrice;
                }

                // If Tellor is working, use it
                return _storeTellorPrice(tellorResponse);
            }

            // If Chainlink is working and Tellor is broken, remember Tellor
is broken
            if (_tellorIsBroken(tellorResponse)) {
                _changeStatus(Status.usingChainlinkTellorUntrusted);
            }

            // If Chainlink is working, return Chainlink current price (no
status change)
            return _storeChainlinkPrice(chainlinkResponse);
        }

        // --- CASE 2: The system fetched last price from Tellor ---
        if (status == Status.usingTellorChainlinkUntrusted) {
            // If both Tellor and Chainlink are live, unbroken, and
reporting similar prices, switch back to Chainlink
            if
(_bothOraclesLiveAndUnbrokenAndSimilarPrice(chainlinkResponse,
tellorResponse)) {
                _changeStatus(Status.chainlinkWorking);
                return _storeChainlinkPrice(chainlinkResponse);
            }

            if (_tellorIsBroken(tellorResponse)) {
                _changeStatus(Status.bothOraclesUntrusted);
                return lastGoodPrice;
            }
```

```
            /*
             * If Tellor is only frozen but otherwise returning valid data,
just return the last good price.
             * Tellor may need to be tipped to return current data.
             */
            if (_tellorIsFrozen(tellorResponse)) {
                return lastGoodPrice;
            }

            // Otherwise, use Tellor price
            return _storeTellorPrice(tellorResponse);
        }

        // --- CASE 3: Both oracles were untrusted at the last price fetch
---
        if (status == Status.bothOraclesUntrusted) {
            /*
             * If both oracles are now live, unbroken and similar price, we
assume that they are reporting
             * accurately, and so we switch back to Chainlink.
             */
            if
(_bothOraclesLiveAndUnbrokenAndSimilarPrice(chainlinkResponse,
tellorResponse)) {
                _changeStatus(Status.chainlinkWorking);
                return _storeChainlinkPrice(chainlinkResponse);
            }

            // Otherwise, return the last good price - both oracles are
still untrusted (no status change)
            return lastGoodPrice;
        }

        // --- CASE 4: Using Tellor, and Chainlink is frozen ---
        if (status == Status.usingTellorChainlinkFrozen) {
            if (_chainlinkIsBroken(chainlinkResponse)) {
                // If both Oracles are broken, return last good price
                if (_tellorIsBroken(tellorResponse)) {
                    _changeStatus(Status.bothOraclesUntrusted);
                    return lastGoodPrice;
                }
```

```
                // If Chainlink is broken, remember it and switch to using
Tellor
                _changeStatus(Status.usingTellorChainlinkUntrusted);

                if (_tellorIsFrozen(tellorResponse)) {
                    return lastGoodPrice;
                }

                // If Tellor is working, return Tellor current price
                return _storeTellorPrice(tellorResponse);
            }

            if (_chainlinkIsFrozen(chainlinkResponse)) {
                // if Chainlink is frozen and Tellor is broken, remember
Tellor broke, and return last good price
                if (_tellorIsBroken(tellorResponse)) {
                    _changeStatus(Status.usingChainlinkTellorUntrusted);
                    return lastGoodPrice;
                }

                // If both are frozen, just use lastGoodPrice
                if (_tellorIsFrozen(tellorResponse)) {
                    return lastGoodPrice;
                }

                // if Chainlink is frozen and Tellor is working, keep using
Tellor (no status change)
                return _storeTellorPrice(tellorResponse);
            }

            // if Chainlink is live and Tellor is broken, remember Tellor
broke, and return Chainlink price
            if (_tellorIsBroken(tellorResponse)) {
                _changeStatus(Status.usingChainlinkTellorUntrusted);
                return _storeChainlinkPrice(chainlinkResponse);
            }

            // If Chainlink is live and Tellor is frozen, just use last good
price (no status change) since we have no basis for comparison
```

```
            if (_tellorIsFrozen(tellorResponse)) {
                return lastGoodPrice;
            }

            // If Chainlink is live and Tellor is working, compare prices.
Switch to Chainlink
            // if prices are within 5%, and return Chainlink price.
            if (_bothOraclesSimilarPrice(chainlinkResponse, tellorResponse))
{
                _changeStatus(Status.chainlinkWorking);
                return _storeChainlinkPrice(chainlinkResponse);
            }

            // Otherwise if Chainlink is live but price not within 5% of
Tellor, distrust Chainlink, and return Tellor price
            _changeStatus(Status.usingTellorChainlinkUntrusted);
            return _storeTellorPrice(tellorResponse);
        }

        // --- CASE 5: Using Chainlink, Tellor is untrusted ---
        if (status == Status.usingChainlinkTellorUntrusted) {
            // If Chainlink breaks, now both oracles are untrusted
            if (_chainlinkIsBroken(chainlinkResponse)) {
                _changeStatus(Status.bothOraclesUntrusted);
                return lastGoodPrice;
            }

            // If Chainlink is frozen, return last good price (no status
change)
            if (_chainlinkIsFrozen(chainlinkResponse)) {
                return lastGoodPrice;
            }

            // If Chainlink and Tellor are both live, unbroken and similar
price, switch back to chainlinkWorking and return Chainlink price
            if
(_bothOraclesLiveAndUnbrokenAndSimilarPrice(chainlinkResponse,
tellorResponse)) {
                _changeStatus(Status.chainlinkWorking);
                return _storeChainlinkPrice(chainlinkResponse);
            }
```

```
            // Otherwise if Chainlink is live and deviated <50% from it's
previous price and Tellor is still untrusted,
            // return Chainlink price (no status change)
            return _storeChainlinkPrice(chainlinkResponse);
        }
    }
...
```

POTENTIAL OFF-CHAIN STATE DESYNC IN DEPLOYMENTS/ALLOCATE.JS

SEVERITY: Medium

PATH:

deployments/allocate.js#L81-L84

REMEDIATION:

Consider unpacking the topic for the LockupContractDeployedThroughFactory event emitted in LockupContractFactory::beneficiaryToLockupContract, and using the beneficiary address from the actual transaction instead of relying on a getter result, which may reflect a different state by the start of the next block.

STATUS: Fixed

DESCRIPTION:

The deployment state saved in lines 81-84 of **deployments/allocate.js** uses the transaction hash of the call to **LockupContractFactory:deployLockupContract** and the beneficiary address of the subsequent getter call to **LockupContractFactory::beneficiaryToLockupContract** without accounting for transaction ordering.

This results in a state desync, as the beneficiary set in the transaction denoted by **txHash** may not necessarily be the beneficiary reported by **beneficiaryToLockupContract** at the start of the next block. This may occur due to back-running or MEV, as outlined in finding **SECFIN1-15**, and will result in an incorrect deployment state being stored relative to on-chain events. This may have significant implications for additional off-chain infrastructure which read or write to the deployment state.

```
//deployments/allocate.js
...
      const txReceipt = await mdh.sendAndWaitForTransaction(
        lockupContractFactory.deployLockupContract(investorAddr,
oneYearFromDeployment),
      );

      const address = await
lockupContractFactory.beneficiaryToLockupContract(investorAddr);
      lockupContracts[investor] = new ethers.Contract(
        address,
        lockupContractEthersFactory.interface,
        deployerWallet,
      );

      deploymentState[investor] = {
        address: address,
        txHash: txReceipt.transactionHash,
      };
...
```

SECFIN1-8

MISSING ALLOCATIONS WHEN TRIGGERANNUALALLOCATION IS SKIPPED

SEVERITY: Low

PATH:

contracts/ProtocolToken/ProtocolToken.sol#L227-L234

REMEDIATION:

See description.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The **triggerAnnualAllocation()** function in the **ProtocolToken.sol** contract is responsible for allocating a certain amount of tokens or rewards annually. It checks whether the allocation for the current year has been triggered and, if not, allocates the tokens for that year. However, if this function is not called in a given year, only the current year's allocation will be applied when the function is eventually called, resulting in missed allocations for any skipped year(s).

```
//contracts/ProtocolToken/ProtocolToken.sol
...
    function triggerAnnualAllocation() external {
        require(_totalSupply != 0, "ProtocolToken: initial allocation has
not been done yet");

        uint passedYears = (block.timestamp - allocationStartTime) / 365
days;
        require(
            !allocationTriggered[passedYears],
            "ProtocolToken: annual allocation is not yet available"
        );

        allocationTriggered[passedYears] = true;
        _mint(annualAllocationRecipient,
_totalSupply.mul(annualAllocationRate).div(_100pct));
    }
...
```

Consider introducing a new function that allows manual allocation for any missed year:

```
//contracts/ProtocolToken/ProtocolToken.sol
...
++    function triggerSpecificYearAllocation(uint256 year) external {
++        require(_totalSupply != 0, "ProtocolToken: initial allocation has
not been done yet");
++
++        uint passedYears = (block.timestamp - allocationStartTime) / 365
days;
++        require(passedYears >= year, "The specified year is in the
future")
++        require(
++            !allocationTriggered[year],
++            "ProtocolToken: annual allocation already done"
++        );
++
++        allocationTriggered[year] = true;
++        _mint(annualAllocationRecipient,
_totalSupply.mul(annualAllocationRate).div(_100pct));
++    }
...
```

Commentary from the client:

“ - As **annualAllocationRate** doesn't have historical data, **triggerAnnualAllocation()** can mint tokens of past allocation with the current **annualAllocationRate** value.
It has a possibility of an unexpected allocation amount in cases where **annualAllocationRate** is changed.
To avoid those cases, we keep the current logic.”

TELLOR MAY RETURN A FAILED STATE WITH A NON STALE DURATION

SEVERITY: Low

PATH:

contracts/Dependencies/TellorCaller.sol#L61

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In **TellorCaller::getTellorCurrentValue**, the oracle call will return "fail" if the data is older than 12 hours. However, if the the **pricefeed** (caller of this function) allows data that is up to 24 hours old, it won't align with this validation. This extra check isn't needed because the **fetchPrice** logic already handles this.

```
//contracts/Dependencies/TellorCaller.sol
...
uint256 public constant STALENESS_AGE = 12 hours;
...
  function getTellorCurrentValue()
      public
      view
      override
      returns (bool ifRetrieve, uint256 _value, uint256 timestamp)
  {
    ...
    require(block.timestamp.sub(_timestamp) <= STALENESS_AGE, "TellorCaller:
StalePrice");
    ...
  }
```

Consider removing the staleness check in **getTellorCurrentValue**, as its handled in the **PriceFeed::fetchPrice**.

```
//contracts/Dependencies/TellorCaller.sol
...
    function getTellorCurrentValue()
        public
        view
        override
        returns (bool ifRetrieve, uint256 _value, uint256 timestamp)
    {
        // retrieve the most recent 20+ minute old btc price.
        // the buffer allows time for a bad value to be disputed
        (bool _ifRetrieve, bytes memory _data, uint256 _timestamp) =
tellor.getDataBefore(
            btcQueryId,
            block.timestamp.sub(DISPUTE_BUFFER)
        );

        if (!_ifRetrieve || _timestamp == 0 || _data.length == 0) {
            return (false, 0, _timestamp);
        }

        // decode the value from bytes to uint256
        _value = abi.decode(_data, (uint256));

        // check whether value is too old
--      require(block.timestamp.sub(_timestamp) <= STALENESS_AGE,
"TellorCaller: StalePrice");

        // return the value and timestamp
        return (true, _value, _timestamp);
    }
...
```

PRICE DIFFERENCE THRESHOLD IN _BOTHORACLESSIMILARPRICE (5% INSTEAD OF 3%)

SEVERITY: Informational

PATH:

contracts/PriceFeed.sol#L368-L373

REMEDIATION:

- Option 1: Keep the threshold at 5% and update the comment to reflect this.
- Option 2: If the intention is to enforce a 3% price difference, change MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES to 3e16.

STATUS: Fixed

DESCRIPTION:

In the function **_bothOraclesSimilarPrice**, the price from one oracle is compared to the price from the other oracle. The function checks if the price difference is greater than 3%. However, the current threshold is mistakenly set to 5% (5e16), not 3% as intended.

```
//contracts/PriceFeed.sol
...
uint public constant MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES = 5e16; // 5%
...
   function _bothOraclesSimilarPrice(
        ChainlinkResponse memory _chainlinkResponse,
        TellorResponse memory _tellorResponse
    ) internal pure returns (bool) {
        ...
        /*
         * Return true if the relative price difference is <= 3%: if so, we
assume both oracles are probably reporting
         * the honest market price, as it is unlikely that both have been
broken/hacked and are still in-sync.
         */
        return percentPriceDifference <=
MAX_PRICE_DIFFERENCE_BETWEEN_ORACLES;
    }
...
```

INITIALIZE CAN BE FRONTRUN

SEVERITY: Informational

PATH:

All initialize functions for upgradeable contracts.

REMEDIATION:

Consider initializing contracts automatically upon deployment.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The contracts that are designed to be upgradeable, utilize an **initialize** function to facilitate future upgrades. This approach aligns with common upgradeability patterns, but it introduces a significant risk: the **initialize** function is marked as public/external, allowing any actor to call it with arbitrary or malicious values.

If the contract is not initialized in the same transaction as its deployment, a legitimate actor's call to **initialize** can be frontrun by a malicious actor. This would enable the malicious actor to initialize the contract with incorrect or harmful parameters, potentially compromising the integrity and functionality of the system.

As per OpenZeppelin recommendation:

>The guidelines are now to make it impossible for anyone to run **initialize** on an implementation contract, by adding an empty constructor with the **initializer** modifier. So the implementation contract gets initialized automatically upon deployment.

For example contracts/ActivePool.sol#L32-L45:

```
...
    function initialize(
        address _borrowerOperationsAddress,
        address _troveManagerAddress,
        address _stabilityPoolAddress,
        address _defaultPoolAddress
    ) external initializer {
        __Ownable_init();
        _setAddresses(
            _borrowerOperationsAddress,
            _troveManagerAddress,
            _stabilityPoolAddress,
            _defaultPoolAddress
        );
    }
...
```

Commentary from the client:

“ - Not needed. Contract deployment and initialization are triggered in one transaction by the ProxyAdmin contract using the Openzeppelin library following the general upgradeable contract deployment way.”

POST-COMMIT COMMENT ISSUE

SEVERITY: Informational

PATH:

contracts/ProtocolToken/CommunityIssuance.sol#L39-L44

REMEDIATION:

Update the comment to reflect the now dynamic protocolTokenSupplyCap.

STATUS: Fixed

DESCRIPTION:

The **CommunityIssuance** contract's code was updated to allow the minting of governance tokens after deployment. However, the comment in the contract was not updated to reflect this change.

```
//contracts/ProtocolToken/CommunityIssuance.sol
...
    /*
     * The community ProtocolToken supply cap is the starting balance of the
Community Issuance contract.
     * It should be minted to this contract by ProtocolToken, when the token
is deployed.
     *
     * Set to 32M (slightly less than 1/3) of total ProtocolToken supply.
     */
-     uint public constant override protocolTokenSupplyCap = 32e24; // 32
million
+    uint public override protocolTokenSupplyCap;
...
```

PUBLIC CONSTANTS SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH:

contracts/ProtocolToken/CommunityIssuance.sol#L20

contracts/ProtocolToken/CommunityIssuance.sol#L36

REMEDIATION:

The affected variables should be marked as private instead of public.

STATUS: Fixed

DESCRIPTION:

In **CommunityIssuance.sol** there are constant variables that are declared **public**. However, setting these constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code.

```
//contracts/ProtocolToken/CommunityIssuance.sol
...
uint public constant SECONDS_IN_ONE_MINUTE = 60;
...
uint public constant ISSUANCE_FACTOR = 999998681227695000;
...
```

UNUSED FILBALANCEUPDATED EVENT

SEVERITY: Informational

PATH:

contracts/Interfaces/IPool.sol#L9-10

REMEDIATION:

Remove event declarations which serve no purpose in current or future contract versions.

STATUS: Fixed

DESCRIPTION:

The **IPool** interface declares the unused **FILBalanceUpdated** event, which is not emitted in any other contract.

```
//contracts/Interfaces/IPool.sol
...
event FILBalanceUpdated(uint _newBalance);
...
```

USING EXPERIMENTAL ABIENCODERV2 IN PRODUCTION

SEVERITY: Informational

PATH:

contracts/MultiTroveGetter.sol#L4

contracts/IPyth.sol#L4

contracts/Dependencies/PythCaller.sol#L4

REMEDIATION:

Use pragma abicoder v2 instead of the experimental version.

STATUS: Fixed

DESCRIPTION:

MultiTroveGetter , IPyth and PythCaller contracts use experimental ABIEncoderV2. The **ABIEncoderV2** became stable in Solidity 0.7.5 and can be enabled using **pragma abicoder v2** instead of the experimental pragma.

```
//contracts/MultiTroveGetter.sol, contracts/IPyth.sol, contracts/
Dependencies/PythCaller.sol
...
pragma experimental ABIEncoderV2;
...
```

SECFIN1-4

DEVELOPMENT IMPORTS IN PRODUCTION CODE

SEVERITY: Informational

PATH:

contracts/SortedTroves.sol#L7

contracts/Dependencies/TroveBase.sol#L6-7

All other contracts importing Dependencies/console.sol

REMEDIATION:

Remove redundant imports and development artefacts.

STATUS: Fixed

DESCRIPTION:

Several contracts import **Dependencies/console.sol**. **SortedTroves.sol** imports the unused interface **IBorrowerOperations.sol**, and **TroveBase.sol** unnecessarily imports **BaseMath.sol** and **ProtocolMath.sol** when they are already inherited via **ProtocolBase.sol**.

```
//contracts/ActivePool.sol and others
...
import "./Dependencies/console.sol";
...
```

```
//contracts/SortedTroves.sol
...
import "./Interfaces/IBorrowerOperations.sol
...
```

```
//contracts/Dependencies/TroveBase.sol
...
import "./ProtocolBase.sol";
import "./ProtocolMath.sol";
import "./BaseMath.sol"
...
```

```
//contracts/Dependencies/ProtocolBase.sol
...
import "./BaseMath.sol";
import "./ProtocolMath.sol";
...
```

REDUNDANT CALCULATION IN PRICE FEED

SEVERITY: Informational

PATH:

contracts/PriceFeed.sol#L396-L398

REMEDIATION:

Refactor PriceFeed to remove _scaleTellorPriceByDigits , as it is redundant in its current implementation.

STATUS: Fixed

DESCRIPTION:

Since the **TARGET_DIGITS** and **TELLOR_DIGITS** have the same value (18), the following calculation will always return the price given by the input parameter because the multiplication by 1 (**10 ** (18 - 18)**).

```
//contracts/PriceFeed.sol
...
uint public constant TARGET_DIGITS = 18;
uint public constant TELLOR_DIGITS = 18;
...
function _scaleTellorPriceByDigits(uint _price) internal pure returns (uint)
{
  return _price.mul(10 ** (TARGET_DIGITS - TELLOR_DIGITS));
}
...
```

SECFIN1-18

OWNER ARGUMENTS ARE SHADOWED FROM OWNABLEUPGRADEABLE::OWNER IN DEBTTOKEN.SOL

SEVERITY: Informational

PATH:

contracts/DebtToken.sol (multiple)

contracts/Dependencies/OpenZeppelin/access/OwnableUpgradeable.sol#L44

REMEDIATION:

Consider renaming the affected owner function arguments to avoid shadowing.

STATUS: Fixed

DESCRIPTION:

Multiple functions in the **DebtToken** contract use **owner** as an argument name, which shadows the **OwnableUpgradeable::owner** function. Although this does not currently introduce any security issues, this is against best practices and may result in readability and maintainability issues over subsequent upgrades.

```
//contracts/DebtToken.sol
...
    function _approve(address owner, address spender, uint256 amount)
internal {
...
    function allowance(address owner, address spender) external view
override returns (uint256) {
...
     function permit(
        address owner,
...
    function nonces(address owner) external view override returns (uint256)
{
...
```

```
//contracts/Dependencies/OpenZeppelin/access/OwnableUpgradeable.sol
...
    function owner() public view virtual returns (address) {
...
```

SECFIN1-19

CONSIDER RENAMING INACCURATE CHAINLINK REFERENCES

SEVERITY: Informational

PATH:

Throughout codebase, particularly contracts/PriceFeed.sol#L97-L117

REMEDIATION:

Consider updating Chainlink references to Pyth where appropriate.

STATUS: Acknowledged, see commentary

DESCRIPTION:

Pyth has replaced the Chainlink oracle's role in this codebase, however there are still references to Chainlink in place of Pyth throughout the codebase. This may result in reduced readability over time.

For example, PriceFeed::fetchPrice:

```
...
    /*
     * fetchPrice():
     * Returns the latest price obtained from the Oracle. Called by protocol
functions that require a current price.
     *
     * Also callable by anyone externally.
     *
     * Non-view function - it stores the last good price seen by the
PriceFeed contract.
     *
     * Uses a main oracle (Chainlink) and a fallback oracle (Tellor) in case
Chainlink fails. If both fail,
     * it uses the last good price seen by the PriceFeed contract.
     *
     */
    function fetchPrice() external override returns (uint price) {
        // Get current and previous price data from Chainlink, and current
price data from Tellor
        ChainlinkResponse memory chainlinkResponse =
_getCurrentChainlinkResponse();
        TellorResponse memory tellorResponse = _getCurrentTellorResponse();


        // --- CASE 1: System fetched last price from Chainlink  ---
        if (status == Status.chainlinkWorking) {
            // If Chainlink is broken, try Tellor
            if (_chainlinkIsBroken(chainlinkResponse)) {
...
```

Commentary from the client:

“ - Not needed to fix it because:

- To use Pyth, we chose to create a wrapper contract, **PythCaller**, for Pyth that has the same interface as Chainlink instead of fixing the PriceFeed itself. It means we can say this protocol expects to use Chainlink.
- There is a possibility to use Chainlink in future updates if they start to support Filecoin.”


hexens
×
Secured Finance