// HALBORN

Liquid Loans - Protocol

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **June 20th, 2023 - July 12th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	07/09/2023	Manuel Garcia
0.2	Document Updates	07/11/2023	Manuel Garcia
0.3	Final Draft	07/12/2023	Manuel Garcia
0.4	Draft Review	07/12/2023	Piotr Cielas
0.5	Draft Review	07/12/2023	Gabi Urrutia
1.0	Remediation Plan	07/28/2023	Manuel Garcia
1.1	Remediation Plan Review	07/28/2023	Piotr Cielas
1.2	Remediation Plan Review	07/31/2023	Gabi Urrutia
1.3	Remediation Plan Updates	08/21/2023	Manuel Garcia
1.4	Remediation Plan Updates Review	08/21/2023	Piotr Cielas
1.5	Remediation Plan Updates Review	08/21/2023	Gabi Urrutia
1.6	Remediation Plan Updates	08/29/2023	Manuel Garcia
1.7	Remediation Plan Updates Review	08/29/2023	Piotr Cielas
1.8	Remediation Plan Updates Review	08/29/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Manuel Garcia	Halborn	Manuel.Diaz@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Liquid Loans engaged Halborn to conduct a security assessment on their smart contracts beginning on June 20th, 2023 and ending on July 12th, 2023.

The Liquid Loans protocol is a decentralized borrowing protocol that allows users to draw 0% interest loans against native currency used as collateral. It is based on a fork of the Liquity protocol that is meant to run on PulseChain.

This security assessment was scoped to some smart contracts in the Liquid-Loans-Official/monorepo GitHub repository. The code in this repository is a fork of the Liquity protocol, per client request, only a pre-defined set of contracts involving changes in the original protocol were verified. Any code that is out of these contracts is left out of scope. More information can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided 3 weeks for the engagement and assigned a team of one full-time security engineer to verify the security of the smart contracts in scope. The security team consists of a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessments is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security risks that were mostly addressed by Liquid Loans. The main one was the following:

- Fetchcaller now returns the last price returned by the oracle with

at least a 15-minute delay.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs (MythX).
- Static Analysis of security for scoped contract, and imported functions (Slither).
- Testnet deployment (Foundry, Brownie).

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Liquid Loans Monorepo

- Repository: Liquid-Loans-Official/monorepo
- Commit ID: 7c3c0d5aa4ec0b78863882443c998dfa47388772
- Smart contracts in scope:
    1. CommunityPoints.sol
    2. LockupContract.sol
    3. LockupContractCreator.sol
    4. LockupContractFactory.sol
    5. LockupSacrifice.sol
    6. PriceFeed.sol
    7. FetchCaller.sol
    8. UsingFetch.sol
    9. StabilityPool.sol (_computeRewardsPerUnitStaked() function)

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	3	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) FETCHCALLER IS PRONE TO PRICE MANIPULATION ATTACKS	Low (2.2)	RISK ACCEPTED
(HAL-02) PRICEFEED CAN RETURN STALE PRICES	Low (4.1)	RISK ACCEPTED
(HAL-03) PRICEFEED ADDRESSES CANNOT BE CHANGED	Low (2.0)	SOLVED - 07/28/2023
(HAL-04) HARDCODED ARRAY LENGTH	Informational (1.4)	SOLVED - 07/28/2023
(HAL-05) APPROVE RESTRICTION CAN BE BYPASSED	Informational (1.4)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) FETCHCALLER IS PRONE TO PRICE MANIPULATION ATTACKS - LOW (2.2)

Description:

The FetchCaller contract is used by the PriceFeed contract to retrieve prices from the fetch oracle in the Pulse chain.

The fetch oracle is a decentralized oracle that allows anyone to introduce new data into the oracle by providing funds as collateral. If the data is later on disputed and determined to be false, the data is removed from the oracle and the user loses the funds used as collateral for the data.

For this reason, when retrieving prices with the getFetchCurrentValue() function, a 15-minute delay is added, so only prices that have been in the oracle for at least 15 minutes are allowed.

However, once a price is retrieved, the FetchCaller contract saves it as lastStoredPrice and lastStoredTimestamp. If some price is retrieved later with a timestamp earlier than the stored one, the previous price is returned as it is considered the most recent price.

However, this behavior is prone to price manipulation attacks, as a user can introduce a malicious price, and if not disputed for at least 15 minutes they can call the getFetchCurrentValue which saves this price as the last stored price.

If the malicious price is later on disputed and removed from the oracle and a replacement for the previous price is provided, the getFetchCurrentValue still returns this malicious price as its timestamp is more recent than the previous one reported by the Fetch oracle.

This is partially mitigated by the fact that the previous price is also checked for a price variation of 50%. Limiting the impact of the price manipulation to a manipulation of a 50% in value.

Code Location:

Listing 1: src/Dependencies/FetchCaller.sol (Line 59)

```
41 function getFetchCurrentValue(
42     bytes32 _queryId
43 )
44     external
45     override
46     returns (bool ifRetrieve, uint256 value, uint256
↳ _timestampRetrieved)
47 {
48     (bytes memory data, uint256 timestamp) = getDataBefore(
49         _queryId,
50         block.timestamp - 15 minutes
51     );
52     uint256 _value = abi.decode(data, (uint256));
53     if (timestamp == 0 || _value == 0) return (false, _value,
↳ timestamp);
54     if (timestamp > lastStoredTimestamp) {
55         lastStoredTimestamp = timestamp;
56         lastStoredPrice = _value;
57         return (true, _value, timestamp);
58     } else {
59         return (true, lastStoredPrice, lastStoredTimestamp);
60     }
61 }
```

Proof Of Concept:

1. The attacker introduces a malicious price into the oracle.
2. After 15 minutes, calls `getFetchCurrentValue` and this price is stored.
3. The price is disputed and removed from the oracle.
4. The malicious price is still returned by the `getFetchCurrentValue()` function.

```
[FAIL. Reason: Assertion failed.] testFail_PriceFeed_PriceCache() (gas: 12572)
Logs:
  Calling getFetchCurrentValue():
  Value:  374712420912
  Timestamp:  201
  Removing price from the oracle.
  Calling getFetchCurrentValue():
  Value:  374712420912
  Timestamp:  201
  Same value was returned.

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 527.46µs

Failing tests:
Encountered 1 failing test in test/PriceFeed.t.sol:PriceFeedTest
[FAIL. Reason: Assertion failed.] testFail_PriceFeed_PriceCache() (gas: 12572)
```

BVSS:

AO:A/AC:M/AX:H/C:N/I:C/A:N/D:N/Y:N/R:N/S:U (2.2)

Recommendation:

If the `lastStoredTimestamp` is greater than the last retrieved timestamp, consider using the price from the secondary oracle.

Reference:

Fetch Oracle Whitepaper

Remediation Plan:

RISK ACCEPTED: Addressing this concern would allow users to challenge the most recent price and revert to a previously more favorable price. Considering this, the Liquid Loans team decided to take the associated risk by implementing off-chain security mechanisms, effectively setting up automated systems for resolving price disputes.

4.2 (HAL-02) PRICEFEED CAN RETURN STALE PRICES - LOW (4.1)

Description:

The fetchPrice() function from the PriceFeed contract allows the Liquid Loan protocol to fetch the price from the fetch oracle on PulseChain. This function uses Fetch as a main oracle and a fallback oracle in case Fetch fails. If both fail, the last good price seen by LiquidLoans is used.

In the extreme case of both oracles failing, the last price seen by Liquid Loans is returned. This means that if both oracles fail, a stale price might be returned. This might not be ideal in the case of extreme price fluctuations, returning stale prices and lead could lead to arbitrage opportunities that may impact users' deposits. In such cases of price fluctuations, reverting the transaction might be a better option than returning stale prices, as an impact in availability is considered less severe than an impact on deposits.

Code Location:

Listing 2: src/PriceFeed.sol (Line 401)

```
218 function fetchPrice() external override returns (uint) {
219     //Get current and previous price data from Fetch and current
 ↳ price data from SecondaryOracle
220     FetchResponse memory fetchResponse = _getCurrentFetchResponse
 ↳ ();
221     FetchResponse memory prevFetchResponse =
 ↳ _getPreviousFetchResponse(
222         fetchResponse.timestamp
223     );
224     SecondaryOracleResponse
225         memory secondaryResponse = _getCurrentSecondaryResponse();
226
227     //--- CASE 1: System fetched last price from Fetch  ---
228     if (status == Status.fetchWorking) {
```

```
229         //If Fetch is broken, try SecondaryOracle
230         if (_fetchIsBroken(fetchResponse)) {
231             //If SecondaryOracle is broken then both oracles are
 ↳ untrusted, so return the last good price
232             if (_secondaryIsBroken(secondaryResponse)) {
233                 _changeStatus(Status.bothOraclesUntrusted);
234                 return lastGoodPrice;
235             }
236             /*
237              * If SecondaryOracle is only frozen but otherwise
 ↳ returning valid data, return the last good price.
238              */
239             if (_secondaryIsFrozen(secondaryResponse)) {
240                 _changeStatus(Status.usingSecondaryFetchUntrusted)
 ↳ ;
241                 return lastGoodPrice;
242             }
243 
244             //If Fetch is broken and SecondaryOracle is working,
 ↳ switch to SecondaryOracle and return current SecondaryOracle price
245             _changeStatus(Status.usingSecondaryFetchUntrusted);
246             return _storeSecondaryPrice(secondaryResponse);
247         }
248 
249         //If Fetch is frozen, try SecondaryOracle
250         if (_fetchIsFrozen(fetchResponse)) {
251             //If SecondaryOracle is broken too, remember
 ↳ SecondaryOracle broke, and return last good price
252             if (_secondaryIsBroken(secondaryResponse)) {
253                 _changeStatus(Status.usingFetchSecondaryUntrusted)
 ↳ ;
254                 return lastGoodPrice;
255             }
256 
257             //If SecondaryOracle is frozen or working, remember
 ↳ Fetch froze, and switch to SecondaryOracle
258             _changeStatus(Status.usingSecondaryFetchFrozen);
259 
260             if (_secondaryIsFrozen(secondaryResponse)) {
261                 return lastGoodPrice;
262             }
263 
264             //If SecondaryOracle is working, use it
265             return _storeSecondaryPrice(secondaryResponse);
```

```
266         }
267
268         //If Fetch price has changed by > 50% between two
 ↳ consecutive rounds, compare it to SecondaryOracle's price
269         if (_fetchPriceChangeAboveMax(fetchResponse,
 ↳ prevFetchResponse)) {
270             //If SecondaryOracle is broken, both oracles are
 ↳ untrusted, and return last good price
271             if (_secondaryIsBroken(secondaryResponse)) {
272                 _changeStatus(Status.bothOraclesUntrusted);
273                 return lastGoodPrice;
274             }
275
276             //If SecondaryOracle is frozen, switch to
 ↳ SecondaryOracle and return last good price
277             if (_secondaryIsFrozen(secondaryResponse)) {
278                 _changeStatus(Status.usingSecondaryFetchUntrusted)
 ↳ ;
279                 return lastGoodPrice;
280             }
281
282             /*
283              * If SecondaryOracle is live and both oracles have a
 ↳ similar price, conclude that Fetch's large price deviation between
284              * two consecutive rounds was likely a legitmate
 ↳ market price movement, and so continue using Fetch
285              */
286             if (
287                 _bothOraclesSimilarPrice(fetchResponse,
 ↳ secondaryResponse)
288             ) {
289                 return _storeFetchPrice(fetchResponse);
290             }
291
292             //If SecondaryOracle is live but the oracles differ
 ↳ too much in price, conclude that Fetch's initial price deviation
 ↳ was
293             //an oracle failure. Switch to SecondaryOracle, and
 ↳ use SecondaryOracle price
294
295             _changeStatus(Status.usingSecondaryFetchUntrusted);
296             return _storeSecondaryPrice(secondaryResponse);
297         }
298
```

```
299         //If Fetch is working and SecondaryOracle is broken,
 ↳ remember SecondaryOracle is broken
300         if (_secondaryIsBroken(secondaryResponse)) {
301             _changeStatus(Status.usingFetchSecondaryUntrusted);
302         }
303
304         //If Fetch is working, return Fetch current price (no
 ↳ status change)
305         return _storeFetchPrice(fetchResponse);
306     }
307
308     //--- CASE 2: The system fetched last price from
 ↳ SecondaryOracle ---
309     if (status == Status.usingSecondaryFetchUntrusted) {
310         //If both SecondaryOracle and Fetch are live, unbroken,
 ↳ and reporting similar prices, switch back to Fetch
311         if (
312             _bothOraclesLiveAndUnbrokenAndSimilarPrice(
313                 fetchResponse,
314                 secondaryResponse
315             )
316         ) {
317             _changeStatus(Status.fetchWorking);
318             return _storeFetchPrice(fetchResponse);
319         }
320
321         if (_secondaryIsBroken(secondaryResponse)) {
322             _changeStatus(Status.bothOraclesUntrusted);
323             return lastGoodPrice;
324         }
325
326         /*
327          * If SecondaryOracle is only frozen but otherwise
 ↳ returning valid data, just return the last good price.
328          */
329         if (_secondaryIsFrozen(secondaryResponse)) {
330             return lastGoodPrice;
331         }
332
333         //Otherwise, use SecondaryOracle price
334         return _storeSecondaryPrice(secondaryResponse);
335     }
336
337     //--- CASE 3: Both oracles were untrusted at the last price
```

```
 ↳ fetch ---
338     if (status == Status.bothOraclesUntrusted) {
339         /*
340          * If both oracles are now live, unbroken and similar
 ↳ price, we assume that they are reporting
341          * accurately, and so we switch back to Fetch.
342          */
343         if (
344             _bothOraclesLiveAndUnbrokenAndSimilarPrice(
345                 fetchResponse,
346                 secondaryResponse
347             )
348         ) {
349             _changeStatus(Status.fetchWorking);
350             return _storeFetchPrice(fetchResponse);
351         }
352
353         //Otherwise, return the last good price - both oracles are
 ↳  still untrusted (no status change)
354         return lastGoodPrice;
355     }
356
357     //--- CASE 4: Using SecondaryOracle, and Fetch is frozen ---
358     if (status == Status.usingSecondaryFetchFrozen) {
359         if (_fetchIsBroken(fetchResponse)) {
360             //If both Oracles are broken, return last good price
361             if (_secondaryIsBroken(secondaryResponse)) {
362                 _changeStatus(Status.bothOraclesUntrusted);
363                 return lastGoodPrice;
364             }
365
366             //If Fetch is broken, remember it and switch to using
 ↳ SecondaryOracle
367             _changeStatus(Status.usingSecondaryFetchUntrusted);
368
369             if (_secondaryIsFrozen(secondaryResponse)) {
370                 return lastGoodPrice;
371             }
372
373             //If SecondaryOracle is working, return
 ↳ SecondaryOracle current price
374             return _storeSecondaryPrice(secondaryResponse);
375         }
376
```

```
377         if (_fetchIsFrozen(fetchResponse)) {
378             //if Fetch is frozen and SecondaryOracle is broken,
 ↳ remember SecondaryOracle broke, and return last good price
379             if (_secondaryIsBroken(secondaryResponse)) {
380                 _changeStatus(Status.usingFetchSecondaryUntrusted)
 ↳ ;
381                 return lastGoodPrice;
382             }
383 
384             //If both are frozen, just use lastGoodPrice
385             if (_secondaryIsFrozen(secondaryResponse)) {
386                 return lastGoodPrice;
387             }
388 
389             //if Fetch is frozen and SecondaryOracle is working,
 ↳ keep using SecondaryOracle (no status change)
390             return _storeSecondaryPrice(secondaryResponse);
391         }
392 
393         //if Fetch is live and SecondaryOracle is broken, remember
 ↳  SecondaryOracle broke, and return Fetch price
394         if (_secondaryIsBroken(secondaryResponse)) {
395             _changeStatus(Status.usingFetchSecondaryUntrusted);
396             return _storeFetchPrice(fetchResponse);
397         }
398 
399         //If Fetch is live and SecondaryOracle is frozen, just use
 ↳  last good price (no status change) since we have no basis for
 ↳ comparison
400         if (_secondaryIsFrozen(secondaryResponse)) {
401             return lastGoodPrice;
402         }
403 
404         //If Fetch is live and SecondaryOracle is working, compare
 ↳  prices. Switch to Fetch
405         //if prices are within 5%, and return Fetch price.
406         if (_bothOraclesSimilarPrice(fetchResponse,
 ↳ secondaryResponse)) {
407             _changeStatus(Status.fetchWorking);
408             return _storeFetchPrice(fetchResponse);
409         }
410 
411         //Otherwise if Fetch is live but price not within 5% of
 ↳ SecondaryOracle, distrust Fetch, and return SecondaryOracle price
```

```
412         _changeStatus(Status.usingSecondaryFetchUntrusted);
413         return _storeSecondaryPrice(secondaryResponse);
414     }
415
416     //--- CASE 5: Using Fetch, SecondaryOracle is untrusted ---
417     if (status == Status.usingFetchSecondaryUntrusted) {
418         //If Fetch breaks, now both oracles are untrusted
419         if (_fetchIsBroken(fetchResponse)) {
420             _changeStatus(Status.bothOraclesUntrusted);
421             return lastGoodPrice;
422         }
423
424         //If Fetch is frozen, return last good price (no status
 ↳ change)
425         if (_fetchIsFrozen(fetchResponse)) {
426             return lastGoodPrice;
427         }
428
429         //If Fetch and SecondaryOracle are both live, unbroken and
 ↳  similar price, switch back to fetchWorking and return Fetch price
430         if (
431             _bothOraclesLiveAndUnbrokenAndSimilarPrice(
432                 fetchResponse,
433                 secondaryResponse
434             )
435         ) {
436             _changeStatus(Status.fetchWorking);
437             return _storeFetchPrice(fetchResponse);
438         }
439
440         //If Fetch is live but deviated >50% from it's previous
 ↳ price and SecondaryOracle is still untrusted, switch
441         //to bothOraclesUntrusted and return last good price
442         if (_fetchPriceChangeAboveMax(fetchResponse,
 ↳ prevFetchResponse)) {
443             _changeStatus(Status.bothOraclesUntrusted);
444             return lastGoodPrice;
445         }
446
447         //Otherwise if Fetch is live and deviated <50% from it's
 ↳ previous price and SecondaryOracle is still untrusted,
448         //return Fetch price (no status change)
449         return _storeFetchPrice(fetchResponse);
450     }
```

```
451 }
```

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:C/Y:C/R:N/S:U (4.1)

Recommendation:

Consider reverting instead of returning the last stored price in case both oracles fail.

Remediation Plan:

RISK ACCEPTED: The Liquid Loans team accepted the risk of this finding, as reverting the process would lead to the suspension of all user operations reliant on the current price, including Vault activities, liquidations, redemptions, and more. However, in the existing setup, users still retain the ability to at least close their positions or redeem their USDL in the event of a failure in either oracle.

Additionally, there is a valid concern that altering the logic to enable reversion and consequently locking down the entire protocol might introduce the possibility of a permanent disruption, rendering the protocol inoperable indefinitely.
Hence, considering the underlying design rationale and the associated risk inherent in altering this aspect of the protocol, the Liquid Loans team opted to maintain the current logic unchanged.

4.3 (HAL-03) PRICEFEED ADDRESSES CANNOT BE CHANGED - LOW (2.0)

In the PriceFeed contract, the owner has to set the oracle address in the setAddresses() function. This function can only be called once by the contract deployer, as at the end of the function the ownership is renounced and the contract is left without an owner.

In this function, the fetch oracle address is checked to ensure that the address is valid, and the oracle is working. However, the fallback oracle is not being checked, meaning if the owner of the contract mistakenly sets a wrong fallback oracle address the address is accepted as it is not checked, and the owner cannot change it after the address is set.

Code Location:

Listing 3: src/PriceFeed.sol (Line 112)

```
89 function setAddresses(
90     address _fetchCallerAddress,
91     address _secondaryOracleAddress
92 ) external onlyOwner {
93     checkContract(_fetchCallerAddress);
94     checkContract(_secondaryOracleAddress);
95
96     fetchCaller = IFetchCaller(_fetchCallerAddress);
97     secondaryOracle = ISecondaryOracle(_secondaryOracleAddress);
98
99     //Explicitly set initial system status
100    status = Status.fetchWorking;
101
102    //Get an initial price from Fetch to serve as first reference
 ↳ for lastGoodPrice
103    FetchResponse memory fetchResponse = _getCurrentFetchResponse
 ↳ ();
104
105    require(
106        !_fetchIsBroken(fetchResponse) && !_fetchIsFrozen(
 ↳ fetchResponse),
107        "PriceFeed: Fetch must be working and current"
```

```
108     );
109
110     _storeFetchPrice(fetchResponse);
111
112     _renounceOwnership();
113 }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:U (2.0)

Recommendation:

Do not renounce ownership after calling `setAddresses()` or check both oracles with `_bothOraclesLiveAndUnbrokenAndSimilarPrice()` instead.

Remediation Plan:

SOLVED: The Liquid Loans team fixed this issue by checking both oracles using the `_bothOraclesLiveAndUnbrokenAndSimilarPrice()` function in commit `ba8022f`.

4.4 (HAL-04) HARDCODED ARRAY LENGTH - INFORMATIONAL (1.4)

Description:

On the CommunityPoints contract, the number of release slots is set to 25 through a constant in the contract; therefore, it can be easily changed to any other value. This contract is consumed by the LockupSacrifice contract, and although it also contains the RELEASE_SLOTS constant, it is not used in some functions and events in the contracts. Therefore, if this constant is changed just before deployment to other value different from 25, the LockupSacrifice contract would not work properly.

Code Location:

Listing 4: src/LOAN/LockupSacrifice.sol (Line 49)

```
43 function _getNextWithdrawAvailable(
44     address _beneficiary
45 ) internal view returns (uint, uint) {
46     (
47         bool registered_,
48         uint256 total_,
49         uint256[25] memory entitlements_
50     ) = communityPoints.getEntitlements(_beneficiary); //@audit-
↳ issue Hardcoded array length
51
```

BVSS:

AO:A/AC:H/AX:H/C:N/I:C/A:C/D:N/Y:N/R:N/S:U (1.4)

Recommendation:

Replace the hardcoded number with the RELEASE_SLOTS constant.

Remediation Plan:

SOLVED: The Liquid Loans team fixed this issue by using the RELEASE_SLOTS constant in commit ba8022f.

4.5 (HAL-05) APPROVE RESTRICTION CAN BE BYPASSED - INFORMATIONAL (1.4)

Description:

The LOANToken contract prevents the LOANTokens minted to the team multi-signature wallet from being transferred to any address that is not a LockupContract for the first year. This restriction is enforced both for transferring and increasing the allowance every time the caller is the multisignature wallet.

However, it is possible for the multi-signature to increase the allowance through the permit() function, as it does not enforce any restriction for the multi-signature. Although this is partially mitigated by the fact that even if the allowance is increased, the transferFrom() function still enforces the restriction if the sender is the multi-signature address.

Code Location:

Listing 5: src/LOAN/LOANToken.sol

```
335 function permit(
336     address owner,
337     address spender,
338     uint amount,
339     uint deadline,
340     uint8 v,
341     bytes32 r,
342     bytes32 s
343 ) external override {
344     require(deadline >= now, "LOAN: expired deadline");
345     bytes32 digest = keccak256(
346         abi.encodePacked(
347             "\x19\x01",
348             domainSeparator(),
349             keccak256(
350                 abi.encode(
351                     _PERMIT_TYPEHASH,
352                     owner,
```

```
353                     spender,
354                     amount,
355                     _nonces[owner]++,
356                     deadline
357                 )
358             )
359         )
360     );
361     address recoveredAddress = ecrecover(digest, v, r, s);
362     require(recoveredAddress == owner, "LOAN: invalid signature");
363     _approve(owner, spender, amount);
364 }
```

BVSS:

AO:A/AC:H/AX:H/C:N/I:C/A:C/D:N/Y:N/R:N/S:U (1.4)

Recommendation:

Add the `_requireCallerIsNotMultisig()` restriction to the `permit()` function if called during the first year.

Remediation Plan:

ACKNOWLEDGED: Since the `transferFrom()` function already implements the necessary restrictions for the multisig address, and considering that the multisig is under the ownership of the Liquid Loans management team, no modifications to the code have been introduced and the Liquid Loans team acknowledged the issue.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

Slither results for LockupContract.sol	
Finding	**Impact**
LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) ignores return value by loanToken.transfer(beneficiary,balance) (src/LOAN/LockupContract.sol#86)	High
LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) uses a dangerous strict equality: - released == 0 && currentReleaseSlot == 0 (src/LOAN/LockupContract.sol#107)	Medium
LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) uses a dangerous strict equality: - unlockAmount == 0 (src/LOAN/LockupContract.sol#92)	Medium

Finding	Impact
Reentrancy in LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103): External calls: - loanToken.transfer(beneficiary,unlockAmount) (src/LOAN/LockupContract.sol#96) State variables written after the call(s): - currentReleaseSlot ++ (src/LOAN/LockupContract.sol#98) LockupContract.currentReleaseSlot (src/LOAN/LockupContract.sol#37) can be used in cross function reentrancies: - LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) - LockupContract.constructor(address,address,uint256,LockupContract -.LockupClass) (src/LOAN/LockupContract.sol#58-76) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) - nextUnlockTime = startTime + (currentReleaseSlot * UNLOCK_TIME_SLOT) (src/LOAN/LockupContract.sol#99-101) LockupContract.nextUnlockTime (src/LOAN/LockupContract.sol#33) can be used in cross function reentrancies: - LockupContract._requireLockupDurationHasPassed() (src/LOAN/LockupContract.sol#145-150) - LockupContract.constructor(address,address,uint256,LockupContract -.LockupClass) (src/LOAN/LockupContract.sol#58-76) - LockupContract.nextUnlockTime (src/LOAN/LockupContract.sol#33) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) - released += unlockAmount (src/LOAN/LockupContract.sol#97) LockupContract.released (src/LOAN/LockupContract.sol#35) can be used in cross function reentrancies: - LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) - LockupContract.released (src/LOAN/LockupContract.sol#35) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103)	Medium
LockupContract.constructor(address,address,uint256,LockupContract. -LockupClass)._beneficiary (src/LOAN/LockupContract.sol#60) lacks a zero-check on : - beneficiary = _beneficiary (src/LOAN/LockupContract.sol#71)	Low

Finding	Impact
LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) uses timestamp for comparisons Dangerous comparisons: - block.timestamp >= (startTime + (UNLOCK_TIME_SLOT * 24)) (src/LOAN/LockupContract.sol#120)	Low
LockupContract._requireLockupDurationHasPassed() (src/LOAN/LockupContract.sol#145-150) uses timestamp for comparisons Dangerous comparisons: - require(bool,string)(block.timestamp >= nextUnlockTime,LockupContract: The lockup duration must have passed) (src/LOAN/LockupContract.sol#146-149)	Low
End of table for LockupContract.sol	

Slither results for LockupContractCreator.sol	
Finding	**Impact**
LockupContractCreator.setParamsAndDeployLockupContract -(address,address,address,address) (src/LOAN/LockupContractCreator.sol#40-71) ignores return value by loanToken.transfer(teamLockA,entitlementA) (src/LOAN/LockupContractCreator.sol#65)	High
LockupContractCreator.setParamsAndDeployLockupContract -(address,address,address,address) (src/LOAN/LockupContractCreator.sol#40-71) ignores return value by loanToken.transfer(teamLockB,entitlementB) (src/LOAN/LockupContractCreator.sol#66)	High
LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) ignores return value by loanToken.transfer(beneficiary,balance) (src/LOAN/LockupContract.sol#86)	High
LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) uses a dangerous strict equality: - released == 0 && currentReleaseSlot == 0 (src/LOAN/LockupContract.sol#107)	Medium
LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) uses a dangerous strict equality: - unlockAmount == 0 (src/LOAN/LockupContract.sol#92)	Medium

Finding	Impact
Reentrancy in LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103): External calls: - loanToken.transfer(beneficiary,unlockAmount) (src/LOAN/LockupContract.sol#96) State variables written after the call(s): - currentReleaseSlot ++ (src/LOAN/LockupContract.sol#98) LockupContract.currentReleaseSlot (src/LOAN/LockupContract.sol#37) can be used in cross function reentrancies: - LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) - LockupContract.constructor(address,address,uint256, -LockupContract.LockupClass) (src/LOAN/LockupContract.sol#58-76) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) - nextUnlockTime = startTime + (currentReleaseSlot * UNLOCK_TIME_SLOT) (src/LOAN/LockupContract.sol#99-101) LockupContract.nextUnlockTime (src/LOAN/LockupContract.sol#33) can be used in cross function reentrancies: - LockupContract._requireLockupDurationHasPassed() (src/LOAN/LockupContract.sol#145-150) - LockupContract.constructor(address,address,uint256, -LockupContract.LockupClass) (src/LOAN/LockupContract.sol#58-76) - LockupContract.nextUnlockTime (src/LOAN/LockupContract.sol#33) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) - released += unlockAmount (src/LOAN/LockupContract.sol#97) LockupContract.released (src/LOAN/LockupContract.sol#35) can be used in cross function reentrancies: - LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) - LockupContract.released (src/LOAN/LockupContract.sol#35) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103)	Medium
LockupContractCreator.setParamsAndDeployLockupContract -(address,address,address,address)._beneficiaryB (src/LOAN/LockupContractCreator.sol#44) lacks a zero-check on : - teamLockB = lockupContractFactory.deployLockupContract(_- beneficiaryB,startTime,LockupContract -.LockupClass.B) (src/LOAN/LockupContractCreator.sol#59-63)	Low

Finding	Impact
LockupContractCreator.setParamsAndDeployLockupContract -(address,address,address,address)._beneficiaryA (src/LOAN/LockupContractCreator.sol#43) lacks a zero-check on : - teamLockA = lockupContractFactory.deployLockupContract(_- beneficiaryA,startTime,LockupContract.LockupClass.A) (src/LOAN/LockupContractCreator.sol#54-58)	Low
LockupContract.constructor(address,address,uint256, -LockupContract.LockupClass)._beneficiary (src/LOAN/LockupContract.sol#60) lacks a zero-check on : - beneficiary = _beneficiary (src/LOAN/LockupContract.sol#71)	Low
Reentrancy in LockupContractCreator.setParamsAndDeployLockupContract -(address,address,address,address) (src/LOAN/LockupContractCreator.sol#40-71): External calls: - teamLockA = lockupContractFactory.deployLockupContract(_- beneficiaryA,startTime,LockupContract.LockupClass.A) (src/LOAN/LockupContractCreator.sol#54-58) - teamLockB = lockupContractFactory.deployLockupContract(_- beneficiaryB,startTime,LockupContract.LockupClass.B) (src/LOAN/LockupContractCreator.sol#59-63) State variables written after the call(s): - teamLockB = lockupContractFactory.deployLockupContract(_- beneficiaryB,startTime,LockupContract.LockupClass.B) (src/LOAN/LockupContractCreator.sol#59-63)	Low
LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) uses timestamp for comparisons Dangerous comparisons: - block.timestamp >= (startTime + (UNLOCK_TIME_SLOT * 24)) (src/LOAN/LockupContract.sol#120)	Low
LockupContract._requireLockupDurationHasPassed() (src/LOAN/LockupContract.sol#145-150) uses timestamp for comparisons Dangerous comparisons: - require(bool,string)(block.timestamp >= nextUnlockTime,LockupContract: The lockup duration must have passed) (src/LOAN/LockupContract.sol#146-149)	Low
End of table for LockupContractCreator.sol	

Slither results for LockupContractFactory.sol	
Finding	**Impact**
LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) ignores return value by loanToken.transfer(beneficiary,balance) (src/LOAN/LockupContract.sol#86)	High
LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) uses a dangerous strict equality: - released == 0 && currentReleaseSlot == 0 (src/LOAN/LockupContract.sol#107)	Medium
LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) uses a dangerous strict equality: - unlockAmount == 0 (src/LOAN/LockupContract.sol#92)	Medium

Finding	Impact
Reentrancy in LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103): External calls: - loanToken.transfer(beneficiary,unlockAmount) (src/LOAN/LockupContract.sol#96) State variables written after the call(s): - currentReleaseSlot ++ (src/LOAN/LockupContract.sol#98) LockupContract.currentReleaseSlot (src/LOAN/LockupContract.sol#37) can be used in cross function reentrancies: - LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) - LockupContract.constructor(address,address,uint256,LockupContract.LockupClass) (src/LOAN/LockupContract.sol#58-76) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) - nextUnlockTime = startTime + (currentReleaseSlot * UNLOCK_TIME_SLOT) (src/LOAN/LockupContract.sol#99-101) LockupContract.nextUnlockTime (src/LOAN/LockupContract.sol#33) can be used in cross function reentrancies: - LockupContract._requireLockupDurationHasPassed() (src/LOAN/LockupContract.sol#145-150) - LockupContract.constructor(address,address,uint256,LockupContract.LockupClass) (src/LOAN/LockupContract.sol#58-76) - LockupContract.nextUnlockTime (src/LOAN/LockupContract.sol#33) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103) - released += unlockAmount (src/LOAN/LockupContract.sol#97) LockupContract.released (src/LOAN/LockupContract.sol#35) can be used in cross function reentrancies: - LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) - LockupContract.released (src/LOAN/LockupContract.sol#35) - LockupContract.withdrawLOAN() (src/LOAN/LockupContract.sol#78-103)	Medium

Finding	Impact
LockupContractFactory.setLOANTokenAddress(address)._-loanTokenAddress (src/LOAN/LockupContractFactory.sol#43) lacks a zero-check on : - loanTokenAddress = _loanTokenAddress (src/LOAN/LockupContractFactory.sol#46)	Low
LockupContract.constructor(address,address,uint256,LockupContract.LockupClass)._-beneficiary (src/LOAN/LockupContract.sol#60) lacks a zero-check on : - beneficiary = _beneficiary (src/LOAN/LockupContract.sol#71)	Low
LockupContract._getUnlockAmount(uint256) (src/LOAN/LockupContract.sol#105-125) uses timestamp for comparisons Dangerous comparisons: - block.timestamp >= (startTime + (UNLOCK_TIME_SLOT * 24)) (src/LOAN/LockupContract.sol#120)	Low
LockupContract._requireLockupDurationHasPassed() (src/LOAN/LockupContract.sol#145-150) uses timestamp for comparisons Dangerous comparisons: - require(bool,string)(block.timestamp >= nextUnlockTime,LockupContract: The lockup duration must have passed) (src/LOAN/LockupContract.sol#146-149)	Low
End of table for LockupContractFactory.sol	

Slither results for LockupSacrifice.sol	
Finding	**Impact**
Reentrancy in LockupSacrifice.withdrawLOAN() (src/LOAN/LockupSacrifice.sol#75-88): External calls: - loanToken.transfer(msg.sender,entitlement_) (src/LOAN/LockupSacrifice.sol#79) State variables written after the call(s): - _beneficiaries[msg.sender].withdrawn[i - 1] = true (src/LOAN/LockupSacrifice.sol#80) LockupSacrifice._beneficiaries (src/LOAN/LockupSacrifice.sol#30) can be used in cross function reentrancies: - LockupSacrifice._getNextWithdrawAvailable(address) (src/LOAN/LockupSacrifice.sol#43-68) - LockupSacrifice.getLOANtokenWithdrawnEntitlements(address) (src/LOAN/LockupSacrifice.sol#111-115) - LockupSacrifice.withdrawLOAN() (src/LOAN/LockupSacrifice.sol#75-88)	Medium

Finding	Impact
LockupSacrifice.getLOANtokenEntitlements(address) (src/LOAN/LockupSacrifice.sol#103-109) ignores return value by (entitlements_) = communityPoints.getEntitlements(_beneficiary) (src/LOAN/LockupSacrifice.sol#106-107)	Medium
Reentrancy in LockupSacrifice.withdrawLOAN() (src/LOAN/LockupSacrifice.sol#75-88): External calls: - loanToken.transfer(msg.sender,entitlement_) (src/LOAN/LockupSacrifice.sol#79) Event emitted after the call(s): - SacrificeEntitlementReleased(msg.sender,entitlement_,i - 1,block.timestamp) (src/LOAN/LockupSacrifice.sol#81-86)	Low
LockupSacrifice._getNextWithdrawAvailable(address) (src/LOAN/LockupSacrifice.sol#43-68) uses timestamp for comparisons Dangerous comparisons: - i < RELEASE_SLOTS && block.timestamp >= releaseSlots[i] (src/LOAN/LockupSacrifice.sol#57)	Low
End of table for LockupSacrifice.sol	

Slither results for PriceFeed.sol	
Finding	**Impact**
Reentrancy in PriceFeed.setAddresses(address,address) (src/PriceFeed.sol#89-113): External calls: - fetchResponse = _getCurrentFetchResponse() (src/PriceFeed.sol#103) - (ifRetrieve,value,_timestampRetrieved) = fetchCaller.getFetchCurrentValue(PLSUSD_FETCH_REQ_ID) (src/PriceFeed.sol#121-136) State variables written after the call(s): - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#110) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202)	Low

Finding	Impact
Reentrancy in PriceFeed.fetchPrice() (src/PriceFeed.sol#218-451): External calls: - fetchResponse = _getCurrentFetchResponse() (src/PriceFeed.sol#220) - (ifRetrieve,value,_timestampRetrieved) = fetchCaller.getFetchCurrentValue(PLSUSD_FETCH_REQ_ID) (src/PriceFeed.sol#121-136) - prevFetchResponse = _getPreviousFetchResponse(fetchResponse.timestamp) (src/PriceFeed.sol#221-223) - (ifRetrieve,value,_timestampRetrieved) = fetchCaller.getFetchPreviousValue(PLSUSD_FETCH_REQ_ID,timestamp) (src/PriceFeed.sol#143-156) - secondaryResponse = _getCurrentSecondaryResponse() (src/PriceFeed.sol#224-225) - (_response.ifRetrieve,_response.value,_response.timestamp,_response.success) = secondaryOracle.getPrice() (src/PriceFeed.sol#569-574) State variables written after the call(s): - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#246) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#265) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#289) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#296) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#305) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#318) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#334) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#350) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#374) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#390) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#396) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#408) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#413) - lastGoodPrice = _currentPrice (src/PriceFeed.sol#202)	Low

Finding	Impact
Reentrancy in PriceFeed.fetchPrice() (src/PriceFeed.sol#218-451): External calls: - fetchResponse = _getCurrentFetchResponse() (src/PriceFeed.sol#220) - (ifRetrieve,value,_timestampRetrieved) = fetchCaller.getFetchCurrentValue(PLSUSD_FETCH_REQ_ID) (src/PriceFeed.sol#121-136) - prevFetchResponse = _getPreviousFetchResponse(fetchResponse.timestamp) (src/PriceFeed.sol#221-223) - (ifRetrieve,value,_timestampRetrieved) = fetchCaller.getFetchPreviousValue(PLSUSD_FETCH_REQ_ID,timestamp) (src/PriceFeed.sol#143-156) - secondaryResponse = _getCurrentSecondaryResponse() (src/PriceFeed.sol#224-225) - (_response.ifRetrieve,_response.value,_response.timestamp,_response.success) = secondaryOracle.getPrice() (src/PriceFeed.sol#569-574) Event emitted after the call(s): - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#437) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#413) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#408) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#289) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#296) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#246) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#265) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#396) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#449) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#318) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#350) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeFetchPrice(fetchResponse) (src/PriceFeed.sol#305) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203) - _storeSecondaryPrice(secondaryResponse) (src/PriceFeed.sol#374) - LastGoodPriceUpdated(_currentPrice) (src/PriceFeed.sol#203)	Low

Finding	Impact
PriceFeed.setAddresses(address,address) (src/PriceFeed.sol#89-113) uses timestamp for comparisons Dangerous comparisons: - require(bool,string)(! _fetchIsBroken(fetchResponse) && ! _fetchIsFrozen(fetchResponse),PriceFeed: Fetch must be working and current) (src/PriceFeed.sol#105-108)	Low
PriceFeed._fetchIsBroken(PriceFeed.FetchResponse) (src/PriceFeed.sol#166-184) uses timestamp for comparisons Dangerous comparisons: - _response.timestamp == 0 || _response.timestamp > block.timestamp (src/PriceFeed.sol#175)	Low
PriceFeed._fetchIsFrozen(PriceFeed.FetchResponse) (src/PriceFeed.sol#159-164) uses timestamp for comparisons Dangerous comparisons: - block.timestamp.sub(_fetchResponse.timestamp) > TIMEOUT (src/PriceFeed.sol#163)	Low
PriceFeed._-bothOraclesLiveAndUnbrokenAndSimilarPrice(PriceFeed.FetchResponse, -zPriceFeed.SecondaryOracleResponse) (src/PriceFeed.sol#453-470) uses timestamp for comparisons Dangerous comparisons: - _secondaryIsBroken(_secondaryOracleResponse) || _secondaryIsFrozen(_secondaryOracleResponse) || _fetchIsBroken(_fetchResponse) || _fetchIsFrozen(_fetchResponse) (src/PriceFeed.sol#460-463)	Low
PriceFeed._secondaryIsFrozen(PriceFeed.SecondaryOracleResponse) (src/PriceFeed.sol#532-537) uses timestamp for comparisons Dangerous comparisons: - block.timestamp.sub(_response.timestamp) > TIMEOUT (src/PriceFeed.sol#536)	Low
PriceFeed._secondaryIsBroken(PriceFeed.SecondaryOracleResponse) (src/PriceFeed.sol#539-557) uses timestamp for comparisons Dangerous comparisons: - _response.timestamp == 0 || _response.timestamp > block.timestamp (src/PriceFeed.sol#548)	Low
End of table for PriceFeed.sol	

Slither results for FetchCaller.sol	
Finding	**Impact**

Finding	Impact
UsingFetch.getDataBefore(bytes32,uint256) (src/Dependencies/UsingFetch.sol#59-68) ignores return value by (None,_value,_timestampRetrieved) = fetch.getDataBefore(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#64-67)	Medium
UsingFetch.getIndexForDataBefore(bytes32,uint256) (src/Dependencies/UsingFetch.sol#169-175) ignores return value by fetch.getIndexForDataBefore(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#174)	Medium
UsingFetch.getTimestampbyQueryIdandIndex(bytes32,uint256) (src/Dependencies/UsingFetch.sol#271-277) has external calls inside a loop: fetch.getTimestampbyQueryIdandIndex(_queryId,_index) (src/Dependencies/UsingFetch.sol#276)	Low
UsingFetch.retrieveData(bytes32,uint256) (src/Dependencies/UsingFetch.sol#299-305) has external calls inside a loop: fetch.retrieveData(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#304)	Low
UsingFetch.isInDispute(bytes32,uint256) (src/Dependencies/UsingFetch.sol#285-291) has external calls inside a loop: fetch.isInDispute(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#290)	Low
End of table for FetchCaller.sol	

Slither results for UsingFetch.sol	
Finding	**Impact**
UsingFetch.getDataBefore(bytes32,uint256) (src/Dependencies/UsingFetch.sol#59-68) ignores return value by (None,_value,_timestampRetrieved) = fetch.getDataBefore(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#64-67)	Medium
UsingFetch.getIndexForDataBefore(bytes32,uint256) (src/Dependencies/UsingFetch.sol#169-175) ignores return value by fetch.getIndexForDataBefore(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#174)	Medium
UsingFetch.getTimestampbyQueryIdandIndex(bytes32,uint256) (src/Dependencies/UsingFetch.sol#271-277) has external calls inside a loop: fetch.getTimestampbyQueryIdandIndex(_queryId,_index) (src/Dependencies/UsingFetch.sol#276)	Low

Finding	Impact
UsingFetch.retrieveData(bytes32,uint256) (src/Dependencies/UsingFetch.sol#299-305) has external calls inside a loop: fetch.retrieveData(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#304)	Low
UsingFetch.isInDispute(bytes32,uint256) (src/Dependencies/UsingFetch.sol#285-291) has external calls inside a loop: fetch.isInDispute(_queryId,_timestamp) (src/Dependencies/UsingFetch.sol#290)	Low
End of table for UsingFetch.sol	

The findings obtained as a result of the Slither scan were reviewed, and they were not included in the report because they were determined false positives.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

Report for LOAN/LockupContract.sol
https://dashboard.mythx.io/#/console/analyses/e4477a3d-23d7-4bac-a8b1-7af476503c14
https://dashboard.mythx.io/#/console/analyses/c5916f09-e964-4ae1-9055-9863cc3934cc

Line	SWC Title	Severity	Short Description
97	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
98	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
100	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+" discovered
101	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
109	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
113	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
113	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
113	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
120	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+" discovered
120	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered

Report for LOAN/LockupContractFactory.sol
https://dashboard.mythx.io/#/console/analyses/52f723e3-3902-4419-a8e0-a8f108c0252f

Line	SWC Title	Severity	Short Description
26	(SWC-123) Requirement Violation	Low	Requirement violation.
55	(SWC-110) Assert Violation	Low	An assertion violation was triggered.
55	(SWC-123) Requirement Violation	Low	Requirement violation.

Report for LOAN/LockupSacrifice.sol
https://dashboard.mythx.io/#/console/analyses/b009e84f-f51b-4bc7-8460-aa5341f21572

Line	SWC Title	Severity	Short Description
55	(SWC-116) Timestamp Dependence	Low	A control flow decision is made based on The block.timestamp environment variable.

Report for UsingFetch.sol
https://dashboard.mythx.io/#/console/analyses/ec988988-e17a-405c-b885-e7db0cc5d097

Line	SWC Title	Severity	Short Description
199	(SWC-101) Integer Overflow and Underflow	High	The arithmetic operator can underflow.
276	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.

Report for PriceFeed.sol
https://dashboard.mythx.io/#/console/analyses/4f830188-918c-43ed-a907-ec48f48b4274

Line	SWC Title	Severity	Short Description
144	(SWC-107) Reentrancy	Low	Write to persistent state following external call
144	(SWC-107) Reentrancy	Low	Read of persistent state following external call
144	(SWC-113) DoS with Failed Call	Low	Multiple calls are executed in the same transaction.
196	(SWC-107) Reentrancy	Low	Read of persistent state following external call
228	(SWC-107) Reentrancy	Low	Read of persistent state following external call
234	(SWC-107) Reentrancy	Low	Read of persistent state following external call
574	(SWC-107) Reentrancy	Low	Write to persistent state following external call
574	(SWC-107) Reentrancy	Low	Read of persistent state following external call

The findings obtained as a result of the MythX scan were examined, and they were not included in the report because they were determined false positives.

THANK YOU FOR CHOOSING

// HALBORN