BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

defi.money

ChainlinkEMA

June 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Defi.money-ChainlinkEMA
Website	defi.money
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/defidotmoney/dfm-contracts/blob/67a8a66e0e194176cc478f3dd2fa3daf996e580b/contracts/cdp/oracles/ChainlinkEMA.sol
Resolution 1	https://github.com/defidotmoney/dfm-contracts/blob/7d348762d57985b745baeb743c99565b0ee900e0/contracts/cdp/oracles/ChainlinkEMA.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	2	2		
Medium	0			
Low	1			1
Informational	2	1		1
Governance	0			
Total	5	3		2

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

ChainlinkEMA

The ChainlinkEMA contract is an oracle handler contract that fetches a price from a Chainlink oracle. Contrary to standard oracle handler contracts, this contract does not only return the nominal price but incorporates an EMA calculation mechanism which returns the 10m EMA for the corresponding price.

There are two distinct scenarios:

a) Standard EMA calculation: The most recent EMA and the next matched price for an interval is taken and the next EMA is calculated. This is done using a loop until the most recent period is reached.

b) New EMA calculation: Since the previous approach would require at some point a potentially large gas-consumption due to increased looping-size, an additional mechanism was implemented which is triggered whenever more than 20 minutes since the past update have been passed. In that scenario it will simply fetch 20 (or potentially less) prices from the last 20 minutes and apply the EMA calculation upwards beginning from the most historical price

Privileged Functions

- none

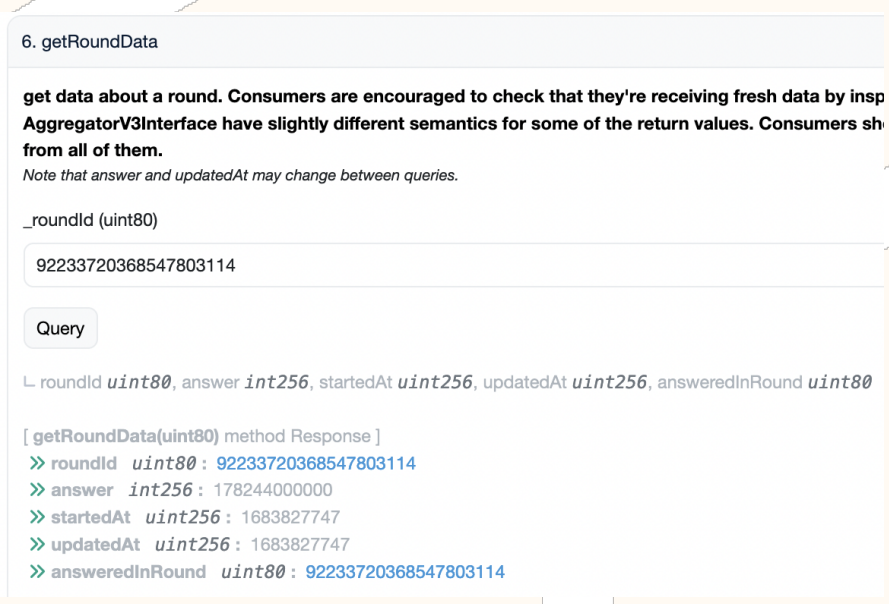
Issue_01	Incorrect assumption that querying non-existing data in historical roundId will break EMA calculation
Severity	**High**

Description	The _getNextRoundData function incorporates a try/catch pattern to fetch the next round data:
	
	try chainlinkFeed.getRoundData(roundId + 1) returns (uint80 round, int answer, uint, uint updatedAt, uint80) {
	
	return _validateAndFormatResponse(round, answer, updatedAt);
	}
	
	catch {
	// handle case where chainlink phase has increased
	uint80 nextRoundId = ((roundId >> 64) + 1) << 64;
	return _getRoundData(nextRoundId);
	}
	
	The rationale behind this try/catch pattern is the assumption that the getRoundData call will revert if we have reached the end of an aggregatorRound. This assumption seems to be however incorrect.
	
	Consider an example based on the current ETH/USD implementation:
	
	https://etherscan.io/address/0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419#readContract
	
	current aggregatorId = 6
	past aggregatorId = 5

In our example, we would take a look at the transition when the aggregatorId was changed from 5 to 6 and fetch the latest roundId for this state:

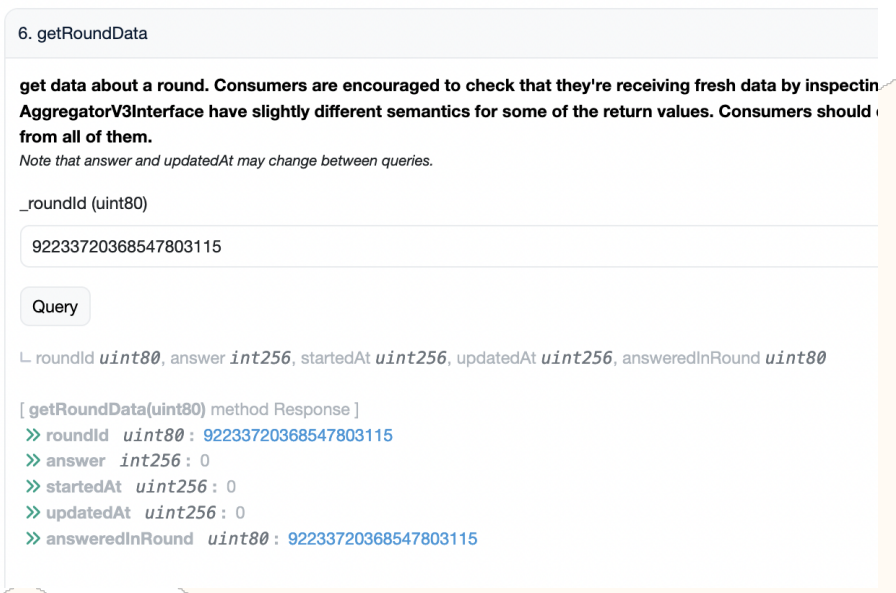
aggregatorId = 5
roundId = 45034
-> 92233720368547803114

This value returns a valid answer:



If we now increase the value by 1, we cross the end of this roundId. Based on the assumption in the code, this means the **call reverts.**

Surprise: It does not revert but return 0,0,0:



Therefore, the try/catch pattern will not fall into the catch condition and the _validateAndFormatResponse call reverts the whole transaction, effectively DoS'ing the price fetching mechanism up to 20 minutes.

Interestingly, the Chainlink docs state the following:

"*To get the historical data for previous aggregators, decrement the phaseId and start from round 1. Loop over the getRoundData function. Start at 73786976294838206465 and increment it until you get a revert. This means that you reached the last round for the underlying aggregator. The same process could be repeated for previous phaseIds (3,2,1).*"

As provided in our PoC, their documentation is simply **wrong**.

Recommendations

As we mentioned above,per their documentation, the call should revert. This means, we have two potential outcomes whenever the end of an aggregatorId is reached:

a) Revert (as stated in their docs)
b) Return 0,0,0 (as provided in our PoC)

	The problem is, there is no guarantee on which outcome can happen, as it might be different based on their past aggregator implementations and it might also be different in the future. This means, the function must be refactored to properly handle **both conditions**, and once one of these conditions is met, the function should invoke the getRoundData function with roundId = 1 for the next aggregator round. This change must be executed with utmost importance and must be checked properly. Bailsec therefore retains the right to charge a small nominal fee depending on the refactoring methodology of the function.
Comments / Resolution	Resolved.

Issue_02	Querying Chainlink price from the 0th round in a new phase returns a price equal to zero
Severity	**High**
Description	Chainlink feed is a proxy contract that receives the price from its underlying aggregator. Each time the aggregator is changed phaseId gets incremented and the new aggregator roundId starts at 1. The phaseId and aggregator roundId are packed inside a single uint80 value: uint80(uint256(phaseId) << 64 | aggregatorId); The ChainlinkEMA contract assumes in several places that the aggregator roundId starts at 0 and tries to query the price at that

	point. The issue is when you try to query the price at a non-existing roundId it will return zero values for price and updatedAt. This results in reverts as _validateAndFormatResponse function only allows price > 0. There are two instances of this issue. _getNextRoundData function where: - If the getRoundData *potentially* reverts due to hitting the last roundId in a phase, it tries to query the first data point in the next phase. - phaseId is increased by 1, but the roundId equals 0. _calculateNewEMA function where: - It tries to fetch the Chainlink response for the 0th roundId for the existing aggregator. - This occurs when the response.roundId is equal to the 1st roundId in a phase. The reason why this issue is raised as high severity is the fact that it will break the price fetching mechanism for a maximum of 20 minutes.

Recommendations	Implement the following changes:
	- if (response.roundId & type(uint64).max == 0) {
	+if (response.roundId & type(uint64).max == 1) {
	
	- uint80 nextRoundId = ((roundId >> 64) + 1) << 64;
	+uint80 nextRoundId = (((roundId >> 64) + 1) << 64) + 1;

Comments / Resolution	Resolved.

Issue_03	Inconsistency within _calculateNewEMA will slightly alter EMA
Severity	**Informational**
Description	In `_calculateNewEMA`, a chainlink response which was updated in the same timestamp as `observationTimestamp` would be incorporated in the EMA calculation. This is inconsistent with `_calculateLatestEMA` which will only incorporate data from chainlink which happened in a timestamp before the `observationTimestamp`. It is standard in EMA implementations to only incorporate data from the past, rather than the same block as the observation timestamp.
Recommendations	- if(response.updatedAt > observationTimestamp) + if(response.updatedAt => observationTimestamp)
Comments / Resolution	Resolved.

Issue_04	Frequent Chainlink updates can cause gas over-consumption
Severity	**Low**
Description	The initial EMA calculation is done by fetching the SMA over the period in question. In our example that would be the average of the past 10 intervals. However, the current system is using a different approach which favors the first data point much more than all recent data points. This will result in a difference compared to the SMA.
Recommendations	Consider using the SMA instead.
Comments / Resolution	Acknowledged, this risk is accepted by the dev and therefore the first 20 observations are used.

Issue_05	Frequent Chainlink updates can cause gas over-consumption
Severity	**Informational**
Description	Some chains like Arbitrum have an average block time of less than 1 second. Theoretically, Chainlink can push multiple updates within a second. In case a ChainlinkEMA contract is configured with: - Large number of observations - Long-duration interval Coupled with frequent Chainlink updates it can result in excessive loop counts as for each roundId the _calculateNewEma and _calculateLatestEMA tries to fetch the response.
Recommendations	No change is needed just take into account the heartbeat of the Chainlink feed while configuring each ChainlinkEMA contract.
Comments / Resolution	Acknowledged.