CD SECURITY

AUDIT REPORT

Athena

May 2024

Introduction

A time-boxed security review of the **Athena** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Athena**

Athena is an DeFi insurance protocol where insurance providers interact with insurance takes. Insurance providers can gain added capital efficiency by both gaining premium rewards from insurance takers and staking rewards from Aave. Insurance takers can also interact by buying covers and extending or shortening/closing them. At the moment the control of the contracts and insurance claims are centralised under a few protocol-controlled accounts but the protocol aims at decentralizing this in the future.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 8f752452ec95a9226594d81e399a4a726682012e**

Scope

The following smart contracts were in scope of the audit:

- `managers/LiquidityManager.sol`
- `managers/StrategyManager.sol`
- `libs/VirtualPool.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 1 issues
- Medium: 3 issues
- Low: 4 issues

Findings Summary

ID	Title	Severity
[H-01]	Cover premium rewards can be lost	High
[M-01]	First leveraged pool is extra penalized	Medium
[M-02]	User can spam covers to DoS cover usage	Medium
[M-03]	Users are missing out on Aave v3 rewards	Medium
[L-01]	Possible silent overflow	Low
[L-02]	Avoid using `transfer`	Low
[L-03]	Consider using OppenZeppelin's `safeTransfer`	Low
[L-04]	Avoid using `tx.origin`	Low

Detailed Findings

[H-01] Cover premium rewards can be lost

Severity

Impact: Medium

Likelihood: High

Description

Premiums paid by the cover holders are periodically distributed to the liquidity providers. This is done on all calls affecting the pool utilization.

It goes down to a function `VirtualPool::_refreshSlot0` which has all the logic for updating the current tick and new utilization and updating the cover premium rewards.

The way it works is that it calculates the current tick and then checks which covers are expiring between the last tick checked and the new current tick:

```
      if (isInitialized) {
        (slot0, utilization, premiumRate) = self
          ._crossingInitializedTick(slot0, nextTick);
      }

      // Remove parsed tick size from remaining time to current timestamp
      remaining -= secondsToNextTickEnd;
      secondsParsed = secondsToNextTickEnd;
      slot0.tick = nextTick + 1;
    } else {
      // ...
    }

    slot0.liquidityIndex += PoolMath.computeLiquidityIndex(
      utilization,
      premiumRate,
      secondsParsed
    );
```

The issue is that the liquidity index is computed with the new utilization and premium rate. Hence any last rewards from the closed cover will be lost. The most extreme case would be a cover that's opened and closed between when `_refreshSlot0` is called, then all the premiums from that position would be lost.

Recommendations

Consider calculating the liquidity index before crossing an initialized tick as well.

Client

Fixed

[M-01] First leveraged pool is extra penalized

Severity

Impact: Low

Likelihood: High

Description

To handle the increased risk from liquidity providers providing the same liquidity in multiple pools the protocol has a *leverage fee* that is taken on the profits made, this only kicks in when a liquidity provider provides liquidity in more than one pool:

`VirtualPool::_payRewardsAndFees`:

```
    uint256 leverageFee;
    if (1 < nbPools_) {
      // The risk fee is only applied when using leverage
      leverageFee =
        (rewards_ * (self.leverageFeePerPool * nbPools_)) /
        HUNDRED_PERCENT;
    } // ...
```

The issue here is that the first "additional" pool will get unfairly penalized. If you stake in one pool you get 0 `leverageFee` applied. If you stake in two pools you get `2*leverageFee` applied. While you stake in a third pool you only get one additional `leverageFee` for this pool (`3*leverageFee`). I.e. the second pool costs 2 leverage fees, compared to the rest.

Recommendations

Consider just taking the `leverageFee` for the additional pools:

```diff
-         (rewards_ * (self.leverageFeePerPool * nbPools_)) /
+         (rewards_ * (self.leverageFeePerPool * nbPools_ - 1)) /
```

Client

Acknowledged

"The leverage risk fee is meant to be applied on a per pool basis and not on a per extra pool basis. This is because when using leverage in two pools this creates liquidity dependency for both pools. The fee evolution is more abrupt but this is by design."

[M-02] User can spam covers to DoS cover usage

Severity

Impact: High

Likelihood: Low

Description

Cover expiry is tracked per tick so that when covers expire the rate, seconds per tick and utilization can be recalculated. Whenever an action happens that involves a change in any of these values these covers need to be tracked. This happens in `VirtualPool::_crossingInitializedTick`:

```
    uint256[] memory coverIds = self.ticks[tick_];

    uint256 coveredCapitalToRemove;

    uint256 nbCovers = coverIds.length;
    for (uint256 i; i < nbCovers; i++) {
      // Add all the size of all covers in the tick
      coveredCapitalToRemove += self.covers[coverIds[i]].coverAmount;
    }
```

Here each cover is looped over and its cover amount is added to the total removed which will later be used to update `premiumRate`, `slot0_.secondsPerTick`, and `utilization`.

The issue is that a user can open any number of covers for dust amounts and fill up a tick so that this loop can use more gas than is available in a block. Thus causing a denial of service to all interactions involving opening, updating or withdrawing a position or opening/updating or claiming a cover.

Hence even though this would be expensive and complicated for the attacker in terms of gas and complicated to perform (as they would need to spread the creation of covers out across multiple blocks), the effect would be a total DoS of the `LiquidityManager`.

Note, there is a way for the admin to `purgeExpiredCoversUpTo` but this only handles if there are too many covers spread across ticks. This would also be impossible to use if the total covers within *one tick* would be too many.

Recommendations

Consider just tracking the aggregate amounts that are expiring per tick instead of each cover. This would remove the need to loop over the covers and thus the possibility for DoS.

Client

Fixed

[M-03] Users are missing out on Aave v3 rewards

Severity

Impact: Low

Likelihood: High

Description

Aave can hand out extra rewards

See here

Currently in StrategyManager there is no way to claim these rewards. This is in itself not a problem as a protocol can through an Aave gov vote get rights to claim on behalf of their contracts. But once that is done, there is no way for these to be redistributed back to the users staking.

Hence a user is disincentivized from moving their staking position from Aave to Athena since they will be losing out on their rewards.

Recommendations

Consider implementing a way to re-add the rewards from Aave to the users staking in Aave through StrategyManager.

Client

Fixed

"Integrating individual computation of rewards per user would be too time consuming at the moment and rewards may be negligible. We have implemented two privileged functions in the StrategyManager to avoid the loss of extra rewards. If the amounts become worthwhile, we will review how they can be redistributed to users."

[L-01] Possible silent overflow

When calculating the last tick for a cover this is the code in VirtualPool::_registerCover is used:

```
uint32 lastTick = self.slot0.tick +
  uint32(durationInSeconds / newSecondsPerTick);
```

Here there is a possibility for overflow if durationInSeconds / newSecondsPerTick is larger than type(uint32).max.

This could happen if a user opens a cover with a very high premium, meaning they want a cover for a very long time. Then instead of getting a very long cover they would get a much lower tick. Possibly even already expired.

Now, this would require someone to open a *very* long cover which is very unlikely.

Consider validating that durationInSeconds / newSecondsPerTick is not bigger than type(uint32).max and then reverting.

Client

Fixed

[L-02] Avoid using transfer

In StrategyManager.sol there's a way for the owner to rescue funds in the form of ether:

```
function rescueTokens(
  address token,
  address to,
  uint256 amount
) external onlyOwner {
  if (token == address(0)) {
    payable(to).transfer(amount);
  } else {
```

`payable.transfer` only forwards a fixed 2300 gas for the call. This could cause smart contract wallets or DAO treasury contracts to run out of gas since they might do logic in their `receive` functions.

Since it's recommended to use multisigs for handling any protocol funds this could force the `owner` to rescue funds to a less secure account.

Consider using low-level `call.value(amount)` with the corresponding result check or using the OpenZeppelin `Address.sendValue`. Also, it is recommended to add a `nonReentrant` modifier as it protects from reentrancy attacks.

Client

Fixed

[L-03] Consider using OpenZeppelin's `safeTransfer`

In `StrategyManager.sol` there's also a way for the owner to rescue funds in the form of IERC20 tokens:

```
function rescueTokens(
  address token,
  address to,
  uint256 amount
) external onlyOwner {
  // ...
  } else {
    IERC20(token).transfer(to, amount);
  }
}
```

There are a lot of weird tokens out there. Some tokens don't revert on failure and thus require the sender to check the returned bool value. While some tokens also don't return a bool value.

This could cause tokens to be rescued to not be rescued or the transfers to not work.

Further reading: https://github.com/d-xo/weird-erc20

Consider using OpenZeppelins `SafeTransfer` library.

Client

Fixed

[L-04] Avoid using `tx.origin`

In `StrategyManager.sol` `tx.origin` is used to validate if an account is whitelisted:

```
modifier onlyWhiteListedLiquidityProviders() {
  if (
    // @dev using tx origin since the contract is called by the liquidity manager
    isWhitelistEnabled && !whiteListedLiquidityProviders[tx.origin]
  ) revert OnlyWhitelistCanDepositLiquidity();
  _;
}
```

Using `tx.origin` should be avoided for a number of reasons:

https://swcregistry.io/docs/SWC-115/ https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/tx-origin/ https://docs.soliditylang.org/en/develop/security-considerations.html#tx-origin

It's also warned that it might be deprecated in the future.

Consider adding the sender as a parameter to these calls so `msg.sender` can be verified instead.

Client

Acknowledged

"We recognize that `tx.origin` is meant to be used with care and only when needed. The whitelist feature is meant as a temporary measure that may even be removed before the first production deployment. Additionally, the `StrategyManager` is going to be heavily reformed soon in order to include new strategies in a modular manner. The new version will not have any whitelist feature. Adding the original caller as a parameter would then cause the need for a new `LiquidityManager` deployment or induce unnecessary gas costs. For these reasons we believe it's best to leave the current usage of `tx.origin` that we deem safe to use in the current (temporary) setting."