Overlay Security Review

Reviewers

Mudit Gupta, Lead Security Researcher

Gerard Persoon, Lead Security Researcher

Harikrishnan Mulackal, Lead Security Researcher

Report prepared by: Pablo Misirov and Harikrishnan Mulackal

May 5, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at https://spearbit.com.

2 Introduction

Overlay is a novel protocol designed around trading assets without requiring a counterparty. This is done by staking OVL (the underlying protocol token) as a collateral, where the protocol mints / burns OVL tokens depending on whether the trade resulted in a profit or loss. To prevent inflation, markets have to be deployed by Overlay governance which tunes the parameters using off-chain risk models. There is also an on-chain component to the risk model which introduces extra checks to trades, oracle updates, etc.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Overlay according to the specific commit by a three person team. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 23 days in total, Overlay engaged with Spearbit to review overlay-market contracts. In this period of time a total of 31 issues were found.

Summary

Project Name	Overlay
Repository	V1-core
Commit	5208a322c976dabe5e9f1d7ce...
Type of Project	Markets on streams of data, DeFi
Audit Timeline	Feb 24 - March 18, 2022
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	2
Medium Risk	2
Low Risk	5
Gas Optimizations	13
Informational	9
Total Issues	31

5 Findings

5.1 High Risk

5.1.1 Use `unchecked` in `TickMath.sol` and `FullMath.sol`

Severity: *High Risk*

Context: Overlay TickMath, Euler TickMath, Overlay FullMath, Euler FullMath

Description: Uniswap math libraries rely on wrapping behaviour for conducting arithmetic operations. Solidity version `0.8.0` introduced checked arithmetic by default where operations that cause an overflow would revert. Since the code was adapted from Uniswap and written in Solidity version `0.7.6`, these arithmetic operations should be wrapped in an `unchecked` block.

Recommendation: Add an `unchecked` block to the following functions in `TickMath.sol` and `FullMath.sol`:

- `getSqrtRatioAtTick()`
- `getTickAtSqrtRatio()`
- `mulDiv()`
- `mulDivRoundingUp()`

The Uniswap protocol has a reference implementation for these changes in a branch named 0.8.

Overlay: Fixed in commit 1f6a974.

Spearbit: Acknowledged.

5.1.2 Liquidation might fail

Severity: *High Risk*

Context: OverlayV1Market.sol#L345-L376, Position.sol#L221-L247

Description: The `liquidate()` function checks if a position can be liquidated and via `liquidatable()`, uses `maintenanceMarginFraction` as a factor to determine if enough value is left. However, in the rest of the `liquidate()` function `liquidationFeeRate` is used to determine the fee paid to the liquidator.

It is not necessarily true that enough value is left for the fee, as two different ways are used to calculate this which means that positions might be liquidated.

This is classified as high risk because liquidation is an essential functionality of Overlay.

```
contract OverlayV1Market is IOverlayV1Market {
    function liquidate(address owner, uint256 positionId) external {
        ...
        require(pos.liquidatable(..., maintenanceMarginFraction),"OVLV1:!liquidatable");
        ...
        uint256 liquidationFee = value.mulDown(liquidationFeeRate);
        ...
        ovl.transfer(msg.sender, value - liquidationFee);
        ovl.transfer(IOverlayV1Factory(factory).feeRecipient(), liquidationFee);
    }
}
```

```
library Position {
    function liquidatable(..., uint256 maintenanceMarginFraction) ... {
        ...
        uint256 maintenanceMargin = posNotionalInitial.mulUp(maintenanceMarginFraction);
        can_ = val < maintenanceMargin;
    }
}
```

Recommendation: Also take into account `liquidationFee` to determine if a position can/should be liquidated.

Note: function `build()` also calls `liquidatable()`.

Overlay: Agreed. The way the liquidation fee amount is calculated, it's taken from the remaining maintenance margin once the `liquidate()` function is called (less any burn of margin as insurance).

So the liquidation fee in its current form is not as a percentage of the current `notionalWithPnl()` like trading fees are, which means that it won't affect the ability to liquidate the position. We should potentially change this.

Fixed in commit 082c6c7.

Spearbit: Acknowledged.

5.2 Medium Risk

5.2.1 Rounding down of `snapAccumulator` might influence calculations

Severity: *Medium Risk*

Context: `Roller.sol#L23-L78`

Description: The function `transform()` lowers `snapAccumulator` with the following equation: `(snapAccumulator * int256(dt)) / int256(snapWindow)`. During the time that `snapAccumulator * dt` is smaller than `snapWindow` this will be rounded down to `0`, which means `snapAccumulator` will stay at the same value. Luckily, `dt` will eventually reach the value of `snapWindow` and by then the value won't be rounded down to `0` any more. Risk lies in calculations diverging from formulas written in the whitepaper.

Note: Given medium risk severity because the probability of this happening is high, while impact is likely low.

```
function transform(...) ... {
    ...
    snapAccumulator -= (snapAccumulator * int256(dt)) / int256(snapWindow);
    ...
}
```

Recommendation: Confirm that rounding down does not influence calculations too much.

Overlay: Confirmed and agreed. Fixed in commit 9b1865e.

Spearbit: Acknowledged.

5.2.2 Verify pool legitimacy

Severity: *Medium Risk*

Context: OverlayV1UniswapV3Factory.sol#L19-L40, OverlayV1UniswapV3Feed.sol#L30-L78

Description: The `constructor` in `OverlayV1UniswapV3Factory.sol` and `OverlayV1UniswapV3Feed.sol` only does a partial check to see if the pool corresponds to the supplied tokens. This is accomplished by calling the pool's functions but if the pool were to be malicious, it could return any token. Additionally, checks can be bypassed by supplying the same tokens twice.

Because the `deployFeed()` function is permissionless, it is possible to deploy malicious feeds. Luckily, the `deployMarket()` function is permissioned and prevents malicious markets from being deployed.

```
contract OverlayV1UniswapV3Factory is IOverlayV1UniswapV3FeedFactory, OverlayV1FeedFactory {
    constructor(address _ovlWethPool, address _ovl, ...) {
        ovlWethPool = _ovlWethPool; // no check on validity of _ovlWethPool here
        ovl = _ovl;
    }
    function deployFeed(address marketPool, address marketBaseToken, address marketQuoteToken, ...)
         external returns (address feed_) { // Permissionless
        ... // no check on validity of marketPool here
    }
```

```
}

contract OverlayV1UniswapV3Feed is IOverlayV1UniswapV3Feed, OverlayV1Feed {
    constructor(
        address _marketPool,
        address _ovlWethPool,
        address _ovl,
        address _marketBaseToken,
        address _marketQuoteToken,
        ... ) ... {
        ...
        address _marketToken0 = IUniswapV3Pool(_marketPool).token0(); // relies on a valid _marketPool
        address _marketToken1 = IUniswapV3Pool(_marketPool).token1();

        require(_marketToken0 == WETH || _marketToken1 == WETH, "OVLV1Feed: marketToken != WETH");
        marketToken0 = _marketToken0;
        marketToken1 = _marketToken1;

        require(
            _marketToken0 == _marketBaseToken || _marketToken1 == _marketBaseToken,
            "OVLV1Feed: marketToken != marketBaseToken"
        );
        require(
            _marketToken0 == _marketQuoteToken || _marketToken1 == _marketQuoteToken,
            "OVLV1Feed: marketToken != marketQuoteToken"
        );
        marketBaseToken = _marketBaseToken; // what if _marketBaseToken == _marketQuoteToken == WETH ?
        marketQuoteToken = _marketQuoteToken;
        marketBaseAmount = _marketBaseAmount;

        // need OVL/WETH pool for ovl vs ETH price to make reserve conversion from ETH => OVL
        address _ovlWethToken0 = IUniswapV3Pool(_ovlWethPool).token0(); // relies on a valid
↪   _ovlWethPool
        address _ovlWethToken1 = IUniswapV3Pool(_ovlWethPool).token1();

        require(
            _ovlWethToken0 == WETH || _ovlWethToken1 == WETH,
            "OVLV1Feed: ovlWethToken != WETH"
        );
        require(
            _ovlWethToken0 == _ovl || _ovlWethToken1 == _ovl, // What if _ovl == WETH ?
            "OVLV1Feed: ovlWethToken != OVL"
        );
        ovlWethToken0 = _ovlWethToken0;
        ovlWethToken1 = _ovlWethToken1;

        marketPool = _marketPool;
        ovlWethPool = _ovlWethPool;
        ovl = _ovl;
    }
```

Recommendation: Verify that pools are indeed Uniswap pools and the supplied tokens do generate the supplied pool.

Note: Verifying that a legitimate Uniswap pool is used still allows for the possibility of malicious tokens making it into the pool.

Consider changing the `deployFeed()` function to be permissioned the same way `deployMarket()` also is.

When deploying a market via `deployMarket()` make sure only valid feeds and tokens are used.

Consider checking the pools using the example code below, note that:

- This can be done for both the `marketPool` and the `_ovlWethPool`
- Determine where to do this, in `OverlayV1UniswapV3Factory` and/or `OverlayV1UniswapV3Feed`
- This way the pool address doesn't even have to be supplied.
- You have to supply a fee to `getPool()`. It seems only 3 fees used.

```
IUniswapV3Factory constant UniswapV3Factory =
↪   IUniswapV3Factory(address(0x1F98431c8aD98523631AE4a59f267346ea31F984));
uint24[3] memory FeeAmount=[uint24(500),uint24(3000),uint24(10000)];
for (uint i=0;i<FeeAmount.length;i++) {
    pool = UniswapV3Factory.getPool(token1,token2,FeeAmount[i]);
    if (pool != address(0)) break;
}
```

Overlay: Fixed in commit b889200.

Spearbit: Acknowledged.

5.3 Low Risk

5.3.1 Liquidatable positions can be unwound by the owner of the position

Severity: *Low Risk*

Context: `OverlayV1Market.sol#L240`, `OverlayV1Market.sol#L345`

Description: The liquidation function can be front-runned since it does not require any deposits. In particular, the liquidation function can be front-runned by the owner of the position by calling `unwind`. This effectively means that users can prevent themselves from getting liquidated by watching the mempool and frontrunning calls to their liquidation position by calling `unwind`.

Although this behaviour is similar to liquidations in lending protocols where a borrower can front-run a liquidation by repaying the borrow, the lack of collateral requirements for both `unwind` and `liquidation` makes this case special.

Note: In practice, transactions for liquidations do not end up in the public mempool and are often sent via private relays such as flashbots. Therefore, a scenario where the user finds out about a liquidatable position by the public mempool is likely not common. However, a similar argument still applies.

Note: Overlay also allows the owner of the position to be the liquidator, unlike other protocols like compound. The difference in price computation for the liquidation and unwind mechanism may make it better for users to liquidate themselves rather than unwinding their position. However, a check similar to compound is not effective at preventing this issue since users can always liquidate themselves from another address.

Recommendation: Consider disallowing the unwinding of liquidatable positions as well as making the underlying oracle price for both `liquidation` and `unwind` to be the same.

Overlay: Disallowed unwinding of liquidatable position in commit 0d6f1c4.

Undecided on having the same oracle price for both `unwind` and `liquidation` due to the following edge case:

Assume a liquidator is targeting a position that is not yet liquidatable, but close:

1. liquidator places a trade on the market to increase the ask or decrease the bid through market impact, `lmbda.mulUp(volume)`
2. ask or bid price then moves enough from the trade in 1 to make the target position liquidatable
3. the liquidator liquidates the target position for the reward
4. liquidator unwinds their first position from 1.

If the reward is large enough, this would seem to be a good strategy

Spearbit: Acknowledged the fix for unwinding of liquidatable position. This could potentially be fixed by tuning the `maintenenceMargin`.

5.3.2 Adding constructor params causes creation code to change

Severity: *Low Risk*

Context: OverlayV1Deployer.sol#L30-L35

Description: Using constructor parameters in `create2` makes the construction code different for every case. This makes address calculation more complex as you first have to calculate the construction code, hash it and then do address calculation. What's worse is that Etherscan does not properly support auto-verification of contracts deployed via `create2` with different creation code. You'll have to manually verify all markets individually.

Additionally, needless salt in `OverlayV1Factory.sol#L129`.

Recommendation: Do a callback from the constructor to the deployer to fetch the parameters so the deployer caches the parameters before deploying the contract. This way, the construction code remains the same and generating the address becomes easier.

Overlay: Fixed in commit 1ce980a.

Spearbit: Acknowledged.

5.3.3 Potential wrap of timestamp

Severity: *Low Risk*

Context: Roller.sol#L23-L34

Description: In the `transform()` function, a revert could occur right after `timestamp32` has wrapped (e.g. when `timestamp > 2**32`).

```
function transform(... , uint256 timestamp, ...) ... {
    uint32 timestamp32 = uint32(timestamp % 2**32); // mod to fit in uint32
    ...
    uint256 dt = uint256(timestamp32 - self.timestamp); // could revert if timestamp32 has just wrapped
    ...
}
```

Recommendation: Note that this truncation would only occur in year 2107, and most protocols ignore this issue. However, a potential fix would look like as follows:

```
uint256 dt;
if (timestamp32 < self.timestamp) {
    dt = uint256(2**32) + uint256(timestamp32) - uint256(self.timestamp);
else
    dt = uint256(timestamp32 - self.timestamp);
}
```

Overlay: Fixed in 7fe8ff3.

Spearbit: Acknowledged.

5.3.4 Verify the validity of `_microWindow` and `_macroWindow`

Severity: *Low Risk*

Context: OverlayV1Feed.sol#L13-L16

Description: The `constructor` of `OverlayV1Feed` doesn't verify the validity of `_microWindow` and `_macroWindow`, potentially causing the price oracle to produce bad results if misconfigured.

```
constructor(uint256 _microWindow, uint256 _macroWindow) {
        microWindow = _microWindow;
        macroWindow = _macroWindow;
}
```

Recommendation: Consider adding sanity checks to be safe. Note: Doing checks in constructors are a one time thing, thereofre gas overhead is often acceptable.

Consider introducing the following checks:

1. `microWindow > 0`
2. `macroWindow > microWindow`
3. `macroWindow / microWindow > constant` some bound. From test cases, this is `60`.
4. `macroWindow < 1 day` or a similar bound.

Overlay: Fixed in commit 44b419c. Except for the 3rd recommendation.

Spearbit: Acknowledged.

5.3.5 Simplify `_midFromFeed()`

Severity: *Low Risk*

Context: OverlayV1Market.sol#L653-L657

Description: The calculation in `_midFromFeed()` is more complicated than necessary because: `min(x,y) + max(x,y) == x + y`. More importantly, the average operation `(bid + ask) / 2` can overflow and revert if `bid + ask >= 2**256`.

```
function _midFromFeed(Oracle.Data memory data) private view returns (uint256 mid_) {
    uint256 bid = Math.min(data.priceOverMicroWindow, data.priceOverMacroWindow);
    uint256 ask = Math.max(data.priceOverMicroWindow, data.priceOverMacroWindow);
    mid_ = (bid + ask) / 2;
}
```

Recommendation: Change the code as follows:

```
function _midFromFeed(Oracle.Data memory data) private view returns (uint256 mid_) {
-   uint256 bid = Math.min(data.priceOverMicroWindow, data.priceOverMacroWindow);
-   uint256 ask = Math.max(data.priceOverMicroWindow, data.priceOverMacroWindow);
-   mid_ = (bid + ask) / 2;
+   mid_ = Math.average(data.priceOverMicroWindow, data.priceOverMacroWindow);
}
```

Here, the `average` function is from Openzepplin.

Overlay: Fixed in commit 2bb5654.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 Use implicit truncation of timestamp

Severity: *Gas Optimization*

Context: Roller.sol#L29

Description: Solidity will truncate data when it is typecast to a smaller data type, see solidity explicit-conversions. This can be used to simplify the following statement:

```
uint32 timestamp32 = uint32(timestamp % 2**32); // mod to fit in uint32
```

Recommendation: Change the code as follows:

```
- uint32 timestamp32 = uint32(timestamp % 2**32); // mod to fit in uint32
+ uint32 timestamp32 = uint32(timestamp); // truncated by compiler
```

Overlay: Fixed in commit 08ef243.

Spearbit: Acknowledged.

5.4.2 Set `pos.entryPrice` to 0 after liquidation

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L345-L427

Description: The `liquidate()` function sets most of the values of `pos` to 0, with the exception of `pos.entryPrice`.

```
function liquidate(address owner, uint256 positionId) external {
    ...
    // store the updated position info data. mark as liquidated
    pos.notional = 0;
    pos.debt = 0;
    pos.oiShares = 0;
    pos.liquidated = true;
    positions.set(owner, positionId, pos);
    ...
}
```

Recommendation: Consider setting `pos.entryPrice` to 0. This is more in line with the rest of the code and can give a small gas refund.

Overlay: Fixed in commit 55318f5.

Spearbit: Acknowledged.

5.4.3 Store result of expression in temporary variable

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L145-L221, OverlayV1Market.sol#L240-L342, OverlayV1Market.sol#L488-L534

Description: Several gas optimizations are possible by storing the result of an expression in a temporary variable, such as the value of `oiFromNotional(data, capNotionalAdjusted)`.

```
function build( ... ) {
    ...
    uint256 price = isLong
        ? ask(data, _registerVolumeAsk(data, oi, oiFromNotional(data, capNotionalAdjusted)))
        : bid(data, _registerVolumeBid(data, oi, oiFromNotional(data, capNotionalAdjusted)));
        ...
        require(oiTotalOnSide <= oiFromNotional(data, capNotionalAdjusted), "OVLV1:oi>cap");
}
```

- A: The value of `pos.oiCurrent(fraction, oiTotalOnSide, oiTotalSharesOnSide)` could be stored in a temporary variable to save gas.
- B: The value of `oiFromNotional(data, capNotionalAdjustedForBounds(data, capNotional))` could also be stored in a temporary variable to save gas and make the code more readable.
- C: The value of `pos.oiSharesCurrent(fraction)` could be stored in a temporary variable to save gas.

```
function unwind(...) ... {
       ...
        uint256 price = pos.isLong
            ? bid(
                data,
                _registerVolumeBid(
                    data,
                    pos.oiCurrent(fraction, oiTotalOnSide, oiTotalSharesOnSide), // A1
                    oiFromNotional(data, capNotionalAdjustedForBounds(data, capNotional)) // B1
                )
            )
            : ask(
                data,
                _registerVolumeAsk(
                    data,
                    pos.oiCurrent(fraction, oiTotalOnSide, oiTotalSharesOnSide), // A2
                    oiFromNotional(data, capNotionalAdjustedForBounds(data, capNotional)) // B2
                )
            );
        ...
        if (pos.isLong) {
            oiLong -= Math.min(
                oiLong,
                pos.oiCurrent(fraction, oiTotalOnSide, oiTotalSharesOnSide) // A3
            );
            oiLongShares -= Math.min(oiLongShares, pos.oiSharesCurrent(fraction)); // C1
        } else {
            oiShort -= Math.min(
                oiShort,
                pos.oiCurrent(fraction, oiTotalOnSide, oiTotalSharesOnSide) // A4
            );
             oiShortShares -= Math.min(oiShortShares, pos.oiSharesCurrent(fraction)); // C2
        }
        ...
        pos.oiShares -= Math.min(pos.oiShares, pos.oiSharesCurrent(fraction)); // C3
}
```

The value of `2 * k * timeElapsed` could also be stored in a temporary variable:

```
function oiAfterFunding( ...) ... {
    ...
    if (2 * k * timeElapsed < MAX_NATURAL_EXPONENT) {
        fundingFactor = INVERSE_EULER.powDown(2 * k * timeElapsed);
    }
```

Recommendation: Consider using temporary variables to save gas and improve readability.

Overlay: Fixed in commit 0af31ff for `oiFromNotional()` i.e., B. The recommendations A and C are causing some stack too deep issues so haven't implemented yet.

Spearbit: Acknowledged.

5.4.4 Flatten code of OverlayV1UniswapV3Feed

Severity: *Gas Optimization*

Context: OverlayV1UniswapV3Feed.sol#L84-L282

Description: Functions `_fetch()`, `_inputsToConsultMarketPool()`, `_inputsToConsultOvlWethPool()` and `consult()` do a lot of interactions with small arrays and loops over them, increasing overhead and reading difficulty.

Recommendation: Consider making the code less generic and unroll it.

Note: If you don't unroll, for loops of arrays can be made more efficient by caching the array length. However as the loops are very small maybe its not worth the trouble.

Overlay: The overlay team is also working on potentially flattening this in the Balancer feed implementation.

The function `_fetch` was flattened in the commit bce944f.

Spearbit: Acknowledged.

5.4.5 Replace memory with calldata

Severity: *Gas Optimization*

Context: OverlayV1Deployer.sol#L21-L25, OverlayV1Market.sol#L102-L107

Description: External calls to functions with `memory` parameters can be made more gas efficient by replacing `memory` with `calldata`, as long as the memory parameters are not modified.

Recommendation: Consider replacing `memory` with `calldata` and check the gas costs are indeed lower.

Note: Also check with finding "Put risk parameters in an array".

```
contract OverlayV1Deployer is IOverlayV1Deployer {
-    function deploy(..., Risk.Params memory params) .. {
+    function deploy(..., Risk.Params calldata params) .. {
```

Overlay: Fixed in commit 6e96fa5.

Spearbit: Acknowledged.

5.4.6 No need to cache `immutable` values

Severity: *Gas Optimization*

Context: OverlayV1UniswapV3Feed.sol#L84-L87, OverlayV1Feed.sol#L13-L16

Description: Variables `microWindow` and `macroWindow` are `immutable`, so it is not necessary to cache them because the compiler inlines their value.

```
contract OverlayV1UniswapV3Feed is IOverlayV1UniswapV3Feed, OverlayV1Feed {
    function _fetch() internal view virtual override returns (Oracle.Data memory) {
        // cache micro and macro windows for gas savings
        uint256 _microWindow = microWindow;
        uint256 _macroWindow = macroWindow;
        ...
    }
}
abstract contract OverlayV1Feed is IOverlayV1Feed {
    ...
    uint256 public immutable microWindow;
    uint256 public immutable macroWindow;
    ...
    constructor(uint256 _microWindow, uint256 _macroWindow) {
        microWindow = _microWindow;
        macroWindow = _macroWindow;
    }
}
```

Recommendation: Use `microWindow` and `macroWindow` directly in function `_fetch()`.

Overlay: Fixed in commit ef5d0d3.

Spearbit: Acknowledged.

5.4.7 Simplify `circuitBreaker`

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L558-L574

Description: The function `circuitBreaker()` does a `divDown()` which can be circumvented to save gas and improving readability.

```
function circuitBreaker(Roller.Snapshot memory snapshot, uint256 cap) ... {
    ...
    if (minted <= int256(_circuitBreakerMintTarget)) {
        return cap;
    } else if (uint256(minted).divDown(_circuitBreakerMintTarget) >= 2 * ONE) {
        return 0;
    }
    ...
}
```

Recommendation: Consider changing the `circuitBreaker()` function as follows:

```
function circuitBreaker(Roller.Snapshot memory snapshot, uint256 cap) ... {
    ...
    if (minted <= int256(_circuitBreakerMintTarget)) {
        return cap;
-   } else if (uint256(minted).divDown(_circuitBreakerMintTarget) >= 2 * ONE) {
+   } else if (minted >= 2 * int256(_circuitBreakerMintTarget)) { // more like the 'if' above
        return 0;
    }
    ...
}
```

Overlay: Had added the divDown in the else clause to match the following, in the event of a rounding issue that caused adjustment to be negative.

```
uint256 adjustment = 2 * ONE - uint256(minted).divDown(_circuitBreakerMintTarget);
```

But seeing here now that `divDown` (vs `divUp`) would always have adjustment `>= 0`. So confirmed and agree. Fixed in commit 3ce32d9.

Spearbit: Acknowledged.

5.4.8 Optimizations if `data.macroWindow` is constant

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L578-L606, OverlayV1Market.sol#L465-L484

Description: Several checks are done in contract `OverlayV1Market` which involve `data.macroWindow` in combination with a linear calculation. If `data.macroWindow` does not change (as is the case with the UniswapV3 feed), it is possible to optimize the calculations by precalculating several values.

Recommendation: In the constructor of contract `OverlayV1Market.sol` calculate the following (please double check the calculations):

```
frontbackrunbound = Math.min( params.lmbda, params.delta * data.macroWindow * 2 / AVERAGE_BLOCK_TIME);
```

Also update `frontbackrunbound` when `lmbda` or `delta` are changed.

And then update `capNotionalAdjustedForBounds()` to:

```
function capNotionalAdjustedForBounds(Oracle.Data memory data, uint256 cap) public view returns
↪   (uint256) {
    ...
-   cap = Math.min(cap, frontRunBound(data));
    ...
-   cap = Math.min(cap, backRunBound(data));
+   cap = Math.min(cap, frontbackrunbound * data.reserveOverMicroWindow );
    ...
}
-function frontRunBound(Oracle.Data memory data) public view returns (uint256) {
-    return lmbda.mulDown(data.reserveOverMicroWindow);
-}
-function backRunBound(Oracle.Data memory data) public view returns (uint256) {
-    uint256 window = (data.macroWindow * ONE) / AVERAGE_BLOCK_TIME;
-    return delta.mulDown(data.reserveOverMicroWindow).mulDown(window).mulDown(2 * ONE);
-}
```

In the constructor of contract `OverlayV1Market.sol` calculate the following:

```
uint256 pow = params.priceDriftUpperLimit * data.macroWindow;
dpLowerLimit = INVERSE_EULER.powUp(pow);
dpUpperLimit = EULER.powUp(pow);
```

Also update `dpLowerLimit` and `dpUpperLimit` when `priceDriftUpperLimit` is changed.

```
function dataIsValid(Oracle.Data memory data) public view returns (bool) {
    ...
-   uint256 pow = priceDriftUpperLimit * data.macroWindow;
-   uint256 dpLowerLimit = INVERSE_EULER.powUp(pow);
-   uint256 dpUpperLimit = EULER.powUp(pow);
    ...
    return (dp >= dpLowerLimit && dp <= dpUpperLimit);
}
```

Note: Also see finding "Optimize power functions" for additional optimizations.

Overlay: Implemented the caching optimization for dpUpperLimit in the commit c505175, but decided against caching `frontbackrunbound` since I think it's a bit easier to read the code with the whitepaper when `frontRunBound()` and `backRunBound()` remain separate functions.

Spearbit: Acknowledged.

5.4.9 Remove unused / redundant functions and variables

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L536-L539, OverlayV1Market.sol#L642-L649, Position.sol#L79-L90, Position.sol#L251-L292, OverlayV1UniswapV3Feed.sol#L30-L78

Description: Functions `nextPositionId()` and `mid()` in `OverlayV1Market.sol` are not used internally and don't appear to be useful.

```
contract OverlayV1Market is IOverlayV1Market {
    function nextPositionId() external view returns (uint256) {
        return _totalPositions;
    }

    function mid(Oracle.Data memory data,uint256 volumeBid,uint256 volumeAsk) ... {
        ...
    }
}
```

The functions `oiInitial()` and `oiSharesCurrent()` in library `Position.sol` have the same implementation. The `oiInitial()` function does not seem useful as it retrieves current positions and not initial ones.

```
library Position {
    /// @notice Computes the initial open interest of position when built
    ...
    function oiInitial(Info memory self, uint256 fraction) internal pure returns (uint256) {
        return _oiShares(self).mulUp(fraction);
    }

    /// @notice Computes the current shares of open interest position holds
    ...
    function oiSharesCurrent(Info memory self, uint256 fraction) internal pure returns (uint256) {
        return _oiShares(self).mulUp(fraction);
    }
}
```

The function `liquidationPrice()` in library `Position.sol` is not used from the contracts. Because it type is internal it cannot be called from the outside either.

```
library Position {
    function liquidationPrice(... ) internal pure returns (uint256 liqPrice_) {
        ...
    }
}
```

The variables `ovlWethToken0` and `ovlWethToken1` are stored but not used anymore.

```
constructor(..., address _ovlWethPool,...) .. {
    ...
    // need OVL/WETH pool for ovl vs ETH price to make reserve conversion from ETH => OVL
    address _ovlWethToken0 = IUniswapV3Pool(_ovlWethPool).token0();
    address _ovlWethToken1 = IUniswapV3Pool(_ovlWethPool).token1();
    ...
    ovlWethToken0 = _ovlWethToken0;
    ovlWethToken1 = _ovlWethToken1;
    ...
}
```

Recommendation: Doublecheck the usefulness of the abovementioned functions and variables. Remove them if not useful or change them to become useful.

Overlay:

- `nextPositionId()` was for testing.
- `mid()` has been replaced with `_midFromFeed()`.
- `oiInitial()` this is likely confusing.
- `ovlWethToken0` and `ovlWethToken1`: unnecessary since the feed contract stores and uses `ovl` and `WETH`.

Removed `mid()`, `nextPositionId()` and `liquidationPrice()` in this commit. We will update `ovlWethToken0` and `ovlWethToken1` when addressing *Check pools* and *Flatten code of OverlayV1UniswapV3Feed*.

Spearbit: Acknowledged.

5.4.10 Optimize power functions

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L465-L469, OverlayV1Market.sol#L504-L506, OverlayV1Market.sol#L621-L640

Description: In contract `OverlayV1Market.sol`, several power calculations are done with `EULER` / `INVERSE_EULER` as a base which can be optimized to save gas.

```
function dataIsValid(Oracle.Data memory data) public view returns (bool) {
    ...
    uint256 dpLowerLimit = INVERSE_EULER.powUp(pow);
    uint256 dpUpperLimit = EULER.powUp(pow);
    ...
}
```

Note: As the Overlay team confirmed, less precision might be sufficient for this calculation.

```
OverlayV1Market.sol: fundingFactor = INVERSE_EULER.powDown(2 * k * timeElapsed);
OverlayV1Market.sol: bid_ = bid_.mulDown(INVERSE_EULER.powUp(pow));
OverlayV1Market.sol: ask_ = ask_.mulUp(EULER.powUp(pow));
```

Recommendation: Replace `EULER.powUp(x)` with `x.expUp()` and replace `INVERSE_EULER.powDown(x)` with `ONE.divDown(x.expUp());`. In function `dataIsValid()` an even further optimization is possible, as `(1/e)^x == exp(-x) == 1/exp(x)`. Consider using alternative `exp()` functions if less precision is required.

Note: Although this might not be worth the trouble, take into account the suggestions of finding: *Optimizations if `data.macroWindow` is constant*.

```
function dataIsValid(Oracle.Data memory data) public view returns (bool) {
    ...
-   uint256 dpLowerLimit = INVERSE_EULER.powUp(pow);
-   uint256 dpUpperLimit = EULER.powUp(pow);
+   uint256 dpUpperLimit = pow.expUp();
+   uint256 dpLowerLimit = ONE.divDown(dpUpperLimit);
    ...
}
```

This requires access to the `LogExpMath.exp()` function. As this function is private, something like the following needs to be added to library `FixedPoint.sol` (please double check the code):

```
library FixedPoint {
    function expUp(uint256 x) internal pure returns (uint256) {
        if (x == 0) return ONE;
        _require(x < 2**255, Errors.X_OUT_OF_BOUNDS);
        int256 x_int256 = int256(x);
        uint256 raw = uint256(LogExpMath.exp(x_int256));
        uint256 maxError = add(mulUp(raw, MAX_POW_RELATIVE_ERROR), 1);
        return add(raw, maxError);
    }
```

Note: The constants `EULER` and `INVERSE_EULER` could be rewritten in a more readable format, but they are no longer necessary with the above suggested changes.

`OverlayV1Market.sol`:

```
-uint256 internal constant EULER = 2718281828459045091;
+uint256 internal constant EULER = 2.718_281_828_459_045_091e18;

-uint256 internal constant INVERSE_EULER = 367879441171442334;
+uint256 internal constant INVERSE_EULER = 0.367_879_441_171_442_334e18;
```

Overlay: Fixed in commit 4b20c00f.

Spearbit: Acknowledged.

5.4.11 Redundant `Math.min()`

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L543-L574

Description: The function `capNotionalAdjustedForCircuitBreaker()` calculates `circuitBreaker()` and then does a `Math.min(cap,...)` with the result. However `circuitBreaker()` already returns a value that is `<= cap`. So the `Math.min(...)` function is unnecesary.

```
function capNotionalAdjustedForCircuitBreaker(uint256 cap) public view returns (uint256) {
        ...
        cap = Math.min(cap, circuitBreaker(snapshot, cap));
        return cap;
    }

function circuitBreaker(Roller.Snapshot memory snapshot, uint256 cap) public view returns (uint256) {
        ...
        if (minted <= int256(_circuitBreakerMintTarget)) {
             return cap;
        } else if (...) {
            return 0;
        }
        // so  minted > _circuitBreakerMintTarget, thus minted / _circuitBreakerMintTarget > ONE
        ...
        uint256 adjustment = 2 * ONE - uint256(minted).divDown(_circuitBreakerMintTarget);
        // so adjustment <= ONE
        return cap.mulDown(adjustment); // so this is <= cap
    }
```

Recommendation: Change `capNotionalAdjustedForCircuitBreaker()` as follows:

```
function capNotionalAdjustedForCircuitBreaker(uint256 cap) public view returns (uint256) {
        ...
-     cap = Math.min(cap, circuitBreaker(snapshot, cap));
+     cap = circuitBreaker(snapshot, cap);
}
```

Overlay: Fixed in commit 3fe9520.

Spearbit: Acknowledged.

5.4.12 Replace square with multiplication

Severity: *Gas Optimization*

Context: OverlayV1Market.sol#L515-L518

Description: The contract `OverlayV1Market.sol` contains the following expression several times: `x.powDown(2 * ONE)`. This computes the square of `x`. However, it can also be calculated in a more gas efficient way:

```
function oiAfterFunding(...) {
    ...
    uint256 underRoot = ONE -
             oiImbalanceBefore.divDown(oiTotalBefore).powDown(2 * ONE).mulDown(
                 ONE - fundingFactor.powDown(2 * ONE)
             );
    ...
}
```

Recommendation: Replace `x.powDown(2 * ONE)` with `mulDown(x,x)`. Alternatively add this functionality into the `FixedPoint.sol` library like balancer has done: Balancer FixedPoint.sol#L107-L117

Overlay: Fixed in commit ad4395d.

Spearbit: Acknowledged.

5.4.13 Retrieve roles via constants in import

Severity: *Gas Optimization*

Context: OverlayV1Factory.sol#L117, OverlayV1Factory.sol#L156-L157, IOverlayV1Token.sol#L9-L15, OverlayV1Token.sol#L10-L21, AccessControl.sol#L57

Description: Within contract `OverlayV1Factory.sol`, the roles `GOVERNOR_ROLE`, `MINTER_ROLE`, `BURNER_ROLE` are retrieved via an external function call. To save gas they could also be retrieved as constants via `import`.

Additionally, a role `ADMIN_ROLE` is defined in contract `OverlayV1Token.sol`, which is the same as `DEFAULT_ADMIN_ROLE` of `AccessControl.sol`. This `ADMIN_ROLE` could be replaced with `DEFAULT_ADMIN_ROLE`.

```
modifier onlyGovernor() {
-    require(ovl.hasRole(ovl.GOVERNOR_ROLE(), msg.sender), "OVLV1: !governor");
+    require(ovl.hasRole(GOVERNOR_ROLE, msg.sender), "OVLV1: !governor");
    _;
}
...
function deployMarket(...) {
    ...
-    ovl.grantRole(ovl.MINTER_ROLE(), market_);
+    ovl.grantRole(MINTER_ROLE, market_);
-    ovl.grantRole(ovl.BURNER_ROLE(), market_);
+    ovl.grantRole(BURNER_ROLE, market_);
    ...
}
```

Recommendation: Consider doing the following changes:

`IOverlayV1Token.sol`

```
+//Note: has to be outside the interface definition
+//Can use DEFAULT_ADMIN_ROLE from AccessControl.sol
+bytes32 constant MINTER_ROLE = keccak256("MINTER");
+bytes32 constant BURNER_ROLE = keccak256("BURNER");
+bytes32 constant GOVERNOR_ROLE = keccak256("GOVERNOR");

interface IOverlayV1Token is IAccessControlEnumerable, IERC20 {
-function ADMIN_ROLE() external view returns (bytes32);
-function MINTER_ROLE() external view returns (bytes32);
-function BURNER_ROLE() external view returns (bytes32);
-function GOVERNOR_ROLE() external view returns (bytes32);
}
```

`OverlayV1Token.sol`

```
-    bytes32 public constant ADMIN_ROLE = 0x00;
-    bytes32 public constant MINTER_ROLE = keccak256("MINTER");
-    bytes32 public constant BURNER_ROLE = keccak256("BURNER");
-    bytes32 public constant GOVERNOR_ROLE = keccak256("GOVERNOR");

    constructor() {
-        _setupRole(ADMIN_ROLE, msg.sender);
+        _setupRole(DEFAULT_ADMIN_ROLE , msg.sender);
        ...
-        _setRoleAdmin(MINTER_ROLE, ADMIN_ROLE);
+        _setRoleAdmin(MINTER_ROLE, DEFAULT_ADMIN_ROLE);

-        _setRoleAdmin(BURNER_ROLE, ADMIN_ROLE);
+        _setRoleAdmin(BURNER_ROLE, DEFAULT_ADMIN_ROLE);

-        _setRoleAdmin(GOVERNOR_ROLE, ADMIN_ROLE);
+        _setRoleAdmin(GOVERNOR_ROLE, DEFAULT_ADMIN_ROLE);
    }
```

Overlay: Fixed in commit fa0f15c.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 Double check action when `snapAccumulator == 0` in `transform()`

Severity: *Informational*

Context: Roller.sol#L23-L78

Description: The function `transform()` does a check for `snapAccumulator + value == 0` (where all variables are of type `int256`). This could be `true` if `value == -snapAccumulator` (or `snapAccumulator == value == 0`)

A comment shows this is to prevent division by `0` later on. The division is based on `abs(snapAccumulator) + abs(value)`. So this will only fail when `snapAccumulator == value == 0`.

```
function transform(...) ... {
    ...
    int256 accumulatorNow = snapAccumulator + value;
    if (accumulatorNow == 0) {
        // if accumulator now is zero, windowNow is simply window
        // to avoid 0/0 case below    ---> this comment might not be accurate
        return ...
    }
    ...
    uint256 w1 = uint256(snapAccumulator >= 0 ? snapAccumulator : -snapAccumulator); // w1 =
↪   abs(snapAccumulator)
    uint256 w2 = uint256(value >= 0 ? value : -value);    // w2 = abs(value)
    uint256 windowNow = (w1 * (snapWindow - dt) + w2 * window) / (w1 + w2); // only fails if w1 == w2
↪   == 0
    ...
}
```

Recommendation: Double check `windowNow` should indeed be reset to `window` when `accumulatorNow == 0`. Note: this seems logical, but isn't explicitly stated in the whitepaper. We recommend updating the comment.

Overlay: Fixed in 1ea0df.

Spearbit: Acknowledged.

5.5.2 Add `unchecked` in natural log (`ln`) function or remove the functions

Severity: *Informational*

Context: LogExpMath.sol#L297-L334

Description: The function `ln()` in contract `LogExpMath.sol` does not use `unchecked`, while the function `log()` does.

Note: Neither `ln()` nor `log()` are used, so they could also be deleted.

```
function log(int256 arg, int256 base) internal pure returns (int256) {
    unchecked {
        ...
    }
}

function ln(int256 a) internal pure returns (int256) {
    // no unchecked
}
```

Recommendation: Consider removing unused functions. Otherwise, consider adding `unchecked` to function `ln()` to make it equivalent to all the other functions.

Overlay: Fixed in commit 57b9bd6.

Spearbit: Acknowledged.

5.5.3 Specialized functions for the long and short side

Severity: *Informational*

Context: OverlayV1Market.sol#L145-L427

Description: The functions `build()`, `unwind()` and `liquidate()` contain a large percentage of code that is different for the `long` and `short` side.

Recommendation: Consider creating specialized functions for the `long` and `short` side, which might make the code easier to read.

Overlay: Did not implement this, due to contract size issues.

Spearbit: Acknowledged.

5.5.4 Beware of chain dependencies

Severity: *Informational*

Context: OverlayV1Market.sol#L25, OverlayV1UniswapV3Feed.sol#L14

Description: The contracts have a few dependencies/assumptions which aren't future proof and/or limit on which chain the code can be deployed.

The `AVERAGE_BLOCK_TIME` is different on several EVM based chains. As the the Ethereum mainchain, the `AVERAGE_BLOCK_TIME` will change to 12 seconds after the merge.

```
contract OverlayV1Market is IOverlayV1Market {
    ...
    uint256 internal constant AVERAGE_BLOCK_TIME = 14; // (BAD) TODO: remove since not futureproof
    ...
}
```

WETH addresses are not the same on different chains. See Uniswap Wrapped Native Token Addresses.

Note: Several chains have a different native token instead of ETH.

```
contract OverlayV1UniswapV3Feed is IOverlayV1UniswapV3Feed, OverlayV1Feed {
    address public constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    ...
}
```

Recommendation: Consider making the block time a configurable parameter, which is initially set via the constructor of `OverlayV1Market`. Additionally, consider supplying the WETH address as a parameter to the constructor of `OverlayV1UniswapV3Feed.sol`

Overlay: Fixed `AVERAGE_BLOCK_TIME` to be a `params[]` element in 2bb8552. Will tackle `WETH` hard code in feed flattening rewrite.

Spearbit: Acknowledged.

5.5.5 Move `_registerMint()` closer to `mint()` and `burn()`

Severity: *Informational*

Context: OverlayV1Market.sol#L240-L427

Description: Within functions `unwind()` and `liquidate()` there is a call to `_registerMint()` as well as calls to `ovl.mint()` and `ovl.burn()`. However these two are quite a few lines apart so it is not immediately obvious they are related and operate on the same values. Additionally `_registerMint()` also registers burns.

```
function unwind(...) ... {
    ...
    _registerMint(int256(value) - int256(cost));
    ... // 40 lines of code
    if (value >= cost) {
        ovl.mint(address(this), value - cost);
    } else {
        ovl.burn(cost - value);
    }
    ...
}
function liquidate(address owner, uint256 positionId) external {
    ...
    _registerMint(int256(value) - int256(cost));
    ... // 33 lines of code
    ovl.burn(cost - value);
    ...
}
```

Recommendation: Rename `_registerMint()` to `_registerMintAndBurn()`. Add a comment to `_registerMintAndBurn()` on indicate that a negative `value` means `burn`.

Move the call to `_registerMint()` close to the `ovl.mint()` and `ovl.burn()` calls in the source. Or possibly move the calls of `ovl.mint()` and `ovl.burn()` to `_registerMint()`, which lower the change on mistakes.

Nevertheless and as indicated by the Overlay team, this changes could make the code harder to understand.

Overlay: Fixed in commit 4894368.

Spearbit: Acknowledged.

5.5.6 Use of `Math.min()` is error-prone

Severity: *Informational*

Context: OverlayV1Market.sol, Position.sol

Description: Function `Math.min()` is used in two ways:

- To get the smallest of two values, e.g. `x = Math.min(x,y);`
- To make sure the resulting value is >=0, e.g. `x -= Math.min(x,y);` (note, there is an extra `-` in `-=`)

It is easy to make a mistake because both constructs are rather similar.

Note: No mistakes have been found in the code.

Examples to get the smallest of two values:

```
OverlayV1Market.sol: tradingFee     = Math.min(tradingFee, value);
OverlayV1Market.sol: cap        = Math.min(cap, circuitBreaker(snapshot, cap));
OverlayV1Market.sol: cap        = Math.min(cap, backRunBound(data));
```

Examples to make sure the resulting value is >=0:

```
OverlayV1Market.sol: oiLong         -= Math.min(oiLong,pos.oiCurrent(fraction, oiTotalOnSide,
↪  oiTotalSharesOnSide));
OverlayV1Market.sol: oiLongShares   -= Math.min(oiLongShares, pos.oiSharesCurrent(fraction));
OverlayV1Market.sol: oiShort        -= Math.min(oiShort,pos.oiCurrent(fraction, oiTotalOnSide,
↪  oiTotalSharesOnSide));
OverlayV1Market.sol: oiShortShares  -= Math.min(oiShortShares, pos.oiSharesCurrent(fraction));
OverlayV1Market.sol: pos.notional   -= uint120( Math.min(pos.notional, pos.notionalInitial(fraction)));
OverlayV1Market.sol: pos.debt       -= uint120( Math.min(pos.debt, pos.debtCurrent(fraction)));
OverlayV1Market.sol: pos.oiShares   -= Math.min(pos.oiShares, pos.oiSharesCurrent(fraction));
OverlayV1Market.sol: oiLong         -= Math.min(oiLong,pos.oiCurrent(fraction, oiTotalOnSide,
↪  oiTotalSharesOnSide));
OverlayV1Market.sol: oiLongShares   -= Math.min(oiLongShares, pos.oiSharesCurrent(fraction));
OverlayV1Market.sol: oiShort        -= Math.min(oiShort,pos.oiCurrent(fraction, oiTotalOnSide,
↪  oiTotalSharesOnSide));
OverlayV1Market.sol: oiShortShares  -= Math.min(oiShortShares, pos.oiSharesCurrent(fraction));
Position.sol:        posCost        -= Math.min(posCost, posDebt);
```

Recommendation: Consider using the following:

```
function minfloor(uint256 a, uint256 b) internal pure returns (uint256) {
    return a > b ? a - b : 0;
}
```

Then you can do something like the example below. This also makes the code easier to read.

```
-oiLong -= Math.min(oiLong, ... )
+oiLong = minfloor(oiLong, ... )
```

Overlay: Fixed in commit ad4d1ec.

Spearbit: Acknowledged.

5.5.7 Confusing use of term `burn`

Severity: *Informational*

Context: OverlayV1Market.sol#L510-L533

Description: The function `oiAfterFunding()` contains a comment that it burns a portion of the contracts. The term `burn` can be confused with burning of `OVL`.

The Overlay team clarified that:

> The total aggregate open interest outstanding (oiLong + oiShort) on the market decreases over time with funding. There's no actual burning of OVL.

```
function oiAfterFunding(...) ... {
    ...
    // Burn portion of all aggregate contracts (i.e. oiLong + oiShort)
    // to compensate protocol for pro-rata share of imbalance liability
    ...
    return (oiOverweightNow, oiUnderweightNow);
}
```

Recommendation: Update the comments.

Overlay: Fixed in commit dae3b82.

Spearbit: Acknowledged.

5.5.8 Document precondition for `oiAfterFunding()`

Severity: *Informational*

Context: OverlayV1Market.sol#L488-L494

Description: Function `oiAfterFunding` contains the following statement: `uint256 oiImbalanceBefore = oiOverweightBefore - oiUnderweightBefore;`

Nevertheless, if `oiOverweightBefore < oiUnderweightBefore` then statement will revert. Luckily, the `update()` function makes sure this isn't the case.

```
function oiAfterFunding(uint256 oiOverweightBefore, uint256 oiUnderweightBefore, ...) ... {
    ...
    uint256 oiImbalanceBefore = oiOverweightBefore - oiUnderweightBefore;
    // Could if oiOverweightBefore < oiUnderweightBefore
    ...
}

function update() public returns (Oracle.Data memory) {
    ...
    bool isLongOverweight = oiLong > oiShort;
    uint256 oiOverweight  = isLongOverweight ? oiLong  : oiShort; // oiOverweight is the largest of the
↪   two
    uint256 oiUnderweight = isLongOverweight ? oiShort :  oiLong; // oiUnderweight is the smallest of
↪   the two
    (oiOverweight, oiUnderweight) = oiAfterFunding(oiOverweight, oiUnderweight, ...);
    ...
}
```

Recommendation: Document the precondition for function `oiAfterFunding()`, e.g, the value of `oiOverweightBefore` must be >= than `oiUnderweightBefore`.

Overlay: Fixed in commit 95e92fe.

Spearbit: Acknowledged.

5.5.9 Format numbers intelligibly

Severity: *Informational*

Context: OverlayV1Factory.sol#L15-L42

Description: Solidity offers several possibilities to format numbers in a more readable way as noted below.

Recommendation: Consider formatting numbers as follows: `OverlayV1Factory.sol`:

```
+ // Note: 1 bps = 1e14
-uint256 public constant MIN_LMBDA = 1e16;    // 0.01
+uint256 public constant MIN_LMBDA = 0.01e18; // 0.01

-uint256 public constant MAX_LMBDA = 1e19;  // 10
+uint256 public constant MAX_LMBDA = 10e18; // 10

-uint256 public constant MAX_DELTA = 2e16;   // 2% (200 bps)
+uint256 public constant MAX_DELTA = 200e14; // 2% (200 bps)

-uint256 public constant MAX_CAP_PAYOFF = 1e19;  // 10x
+uint256 public constant MAX_CAP_PAYOFF = 10e18; // 10x

-uint256 public constant MAX_CAP_NOTIONAL = 8e24;         // 8,000,000 OVL (initial supply)
+uint256 public constant MAX_CAP_NOTIONAL = 8_000_000e18; // 8,000,000 OVL (initial supply)

-uint256 public constant MAX_CAP_LEVERAGE = 2e19;  // 20x
+uint256 public constant MAX_CAP_LEVERAGE = 20e18; // 20x

-uint256 public constant MAX_CIRCUIT_BREAKER_MINT_TARGET = 8e24;         // 8,000,000 OVL
+uint256 public constant MAX_CIRCUIT_BREAKER_MINT_TARGET = 8_000_000e28; // 8,000,000 OVL

-uint256 public constant MIN_MAINTENANCE_MARGIN_FRACTION = 1e16;    // 1%
+uint256 public constant MIN_MAINTENANCE_MARGIN_FRACTION = 0.01e18; // 1%

-uint256 public constant MAX_MAINTENANCE_MARGIN_FRACTION = 2e17;    // 20%
+uint256 public constant MAX_MAINTENANCE_MARGIN_FRACTION = 0.20e18; // 20%

-uint256 public constant MIN_MAINTENANCE_MARGIN_BURN_RATE = 1e16;    // 1%
+uint256 public constant MIN_MAINTENANCE_MARGIN_BURN_RATE = 0.01e18; // 1%

-uint256 public constant MAX_MAINTENANCE_MARGIN_BURN_RATE = 5e17;     // 50%
+uint256 public constant MAX_MAINTENANCE_MARGIN_BURN_RATE = 0.50e18; // 50%

-uint256 public constant MIN_LIQUIDATION_FEE_RATE = 1e15;  // 0.10% (10 bps)
+uint256 public constant MIN_LIQUIDATION_FEE_RATE = 10e14; // 0.10% (10 bps)

-uint256 public constant MAX_LIQUIDATION_FEE_RATE = 1e17;     // 10.00% (1000 bps)
+uint256 public constant MAX_LIQUIDATION_FEE_RATE = 1_000e14; // 10.00% (1000 bps)

-uint256 public constant MAX_TRADING_FEE_RATE = 3e15;  // 0.30% (30 bps)
+uint256 public constant MAX_TRADING_FEE_RATE = 30e14; // 0.30% (30 bps)

-uint256 public constant MIN_MINIMUM_COLLATERAL = 1e12;           // 1e-6 OVL
+uint256 public constant MIN_MINIMUM_COLLATERAL = 0.000_000_1e18; // 1e-6 OVL

-uint256 public constant MIN_PRICE_DRIFT_UPPER_LIMIT = 1e12;    // 0.01 bps/s
+uint256 public constant MIN_PRICE_DRIFT_UPPER_LIMIT = 0.01e14; // 0.01 bps/s
```

Overlay: Fixed in commit a82ddd7.

Spearbit: Acknowledged.