Sense Security Review

Reviewers
Gerard Persoon, Lead Security Researcher
Denis Milicevic, Lead Security Researcher
Max Goodman, Security Researcher
Kurt Barry, Security Review Consultant

February 4, 2022

1 Executive Summary

Over the course of 2 calendar weeks and 7 engineering weeks in total, Sense engaged with Spearbit to review Sense.

We found a total of 71 issues with Sense.

Repository	**Commit**
sense-finance/sense-v1	1b14c79bdf9925c143d311c0f948eedc9810d2ea

Summary

Type of Project	DeFi, Fixed Yield
Timeline	Jan 5th, 2022 - Jan 21st, 2022
Methods	Computer-Aided Verification, Manual Review
Documentation	Medium
Testing Coverage	Medium

Total Issues

High Risk	11
Medium Risk	13
Low Risk	19
Informational	17
Gas Optimizations	11

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

Contents

2 Spearbit

Spearbit is a decentralized network of expert Web3 security engineers. Together, we help secure the Web3 ecosystem. We offer security reviews and related services to Web3 projects. Our network has experience at every part of the stack, including protocol design, smart contracts, and the Solidity compiler itself. Spearbit brings in untapped security talent: expert freelance auditors want flexibility to work on interesting projects together. Learn more about us at https://spearbit.com.

3 Findings

3.1 High Risk Issues

3.1.1 The Variable `maxscale` Is Not Saved

Severity: *High Risk*

Context: `Divider.sol#L334-382`

Situation: In the function `_collect()` of `Divider.sol`, the value `maxscale` is updated in a temporary variable. However, this temporary variable is not written back to its origin. This means the value of `maxscale` is not kept over time.

```
function _collect(...) internal returns (uint256 collected) {
    ...
    Series memory _series = series[adapter][maturity];
    ...
    // If this is larger than the largest scale we've seen for this Series, use
↪   it
    if (cscale > _series.maxscale) {
        // _series is a local variable
        _series.maxscale = cscale;
        lscales[adapter][maturity][usr] = cscale;
        // If not, use the previously noted max scale value
    } else {
        lscales[adapter][maturity][usr] = _series.maxscale;
    }
} // _series is not saved to series[adapter][maturity]
```

Recommendation: Do one of the following:

- Replace memory with storage. This way any access to `_series` translates to `sload`/`sstore`.

```
Series storage _series = series[adapter][maturity];
```

- At the end of function `_collect()`, add the following to "save" the value of `_series.maxscale`. This is assuming `maxscale` is the only part that has to be saved.

```
series[adapter][maturity].maxscale = _series.maxscale;
```

Sense: Addressed in #163.

Spearbit: Acknowledged.

3.1.2 Reserves Not Always Updated In `onJoinPool()`

Severity: *High Risk*

Context: Space.sol#L148-223

Recommendation: In the `if` part, add something like the following before the call to `_cacheReserves()`:

```
reserves[_targeti] += reqAmountsIn[_targeti];
```

Sense: Addressed in #1.

Spearbit: Acknowledged.

3.1.3 Wrong Amount in `sponsorSeries`

Severity: *High Risk*

Context: Periphery.sol#L66-76

Situation: In function `sponsorSeries()`, a different amount is used with `safeTransferFrom()` than with `safeApprove()`, if the number of decimals of the stake token != 18.

Normally, `safeTransferFrom()` and `safeApprove()` should be the same amount.

```
function sponsorSeries(address adapter, uint48 maturity) external returns
↪   (address zero, address claim) {
    ...
    // Transfer stakeSize from sponsor into this contract
    uint256 stakeDecimals = ERC20(stake).decimals();
    ERC20(stake).safeTransferFrom(msg.sender, address(this),
↪   _convertToBase(stakeSize, stakeDecimals)); // amount 1

    // Approve divider to withdraw stake assets
    ERC20(stake).safeApprove(address(divider), stakeSize);  // amount 2
}
```

Recommendation: Spearbit recommends double checking which of these two amounts is the right amount and update the code. We also recommend considering adding unit tests with `Stake` tokens with less than 18 decimals.

Sense: Fixed here. We've gotten rid of `_convertToBase` so the amount should just be `stakeSize`.

Spearbit: Acknowledged.

3.1.4 Return value missing in `wrapUnderlying()` `WstETHAdapter`

Severity: *High Risk*

Context:

1. `WstETHAdapter.sol#L136-142`
2. `CAdapter.sol#L130-154`
3. `Periphery.sol#L253-272`

Situation: The function `wrapUnderlying()` of `WstETHAdapter` does not return any value, which means it returns 0. On the contrary, the function `wrapUnderlying()` of `CAdapter` returns the amount of tokens sent: `tBal`. The `CAdapter` version returns the amount of tokens sent (`tBal`), while the `WstETHAdapter` return `0`.

The contract `Periphery`, which calls `wrapUnderlying()`, expects a return value. It also bases it's following actions on this return value.

```
contract WstETHAdapter is BaseAdapter {
    ...
    function wrapUnderlying(uint256 amount) external override returns (uint256)
↪   {
        ...
        ERC20(WSTETH).safeTransfer(msg.sender, wstETH); // transfer wstETH to
↪   msg.sender // no return value
    }
}
contract CAdapter is CropAdapter {
    ...
    function wrapUnderlying(uint256 uBal) external override returns (uint256) {
        ...
        ERC20(target).safeTransfer(msg.sender, tBal);
        return tBal;
    }
}
contract Periphery is Trust {
    ...
    function addLiquidityFromUnderlying(...) ... {
        ...
        uint256 tBal = Adapter(adapter).wrapUnderlying(uBal);
        return _addLiquidity(adapter, maturity, tBal, mode); // tBal being used
↪   here
    }
}
```

Recommendation: In the function `wrapUnderlying()` of `WstETHAdapter()`, at the end, add the following:

```
return wstETH;
```

Add unit tests for `WstETHAdapter` to detect these types of errors.

Sense: Fixed here.

Spearbit: It was not fixed in the above commit. There is still a stack variable declared that will shadow the return variable. Seems it was fixed in a different commit here, but still no test coverage was added to this issue on the dev branch.

Spearbit: Acknowledged.

3.1.5 Send Reward And Stake Once

Severity: *High Risk*

Context: Divider.sol#L157-180, Divider.sol#L511-547

Situation: A reward and stake can be sent from `settleSeries()` or `backfillScale()`. However, this should only be done once.

Luckily `settleSeries()` can't be run twice as this is prevented by `_canBeSettled()`. However, `backfillScale()` might be called multiple times. This could result in the function trying to send the reward and the stake multiple times.

```
function settleSeries(address adapter, uint48 maturity) external nonReentrant
↪   whenNotPaused {
    ...
    // prevents calling this function twice
    require(_canBeSettled(adapter, maturity), Errors.OutOfWindowBoundaries);
    ...
    ERC20(target).safeTransferFrom(adapter, msg.sender,
↪   series[adapter][maturity].reward);
    ERC20(stake).safeTransferFrom(adapter, msg.sender, stakeSize);
    ...
}
function backfillScale(...) external requiresTrust {
    ...
    uint256 reward = series[adapter][maturity].reward;
    ERC20(target).safeTransferFrom(adapter, cup, reward);
    ERC20(stake).safeTransferFrom(adapter, stakeDst, stakeSize);
    ...
}
```

Recommendation: After sending the reward and the stake, set a flag to prevent sending it a second time.

Sense: Addressed in #155.

Spearbit: The reward is set to 0 now so won't be transferred twice. There may still, however, be a risk with `stakeSize`.

3.1.6 Untrusted `ERC-20 decimals()` Return Values Could Be Mutated, Uncached Access Shouldn't Be Considered Reliable

Severity: *High Risk*

Context:

- Space.sol#L133, GClaimManager.sol#L59, CAdapter.sol#L96,
- CAdapter.sol#L109, CAdapter.sol#L113-115, Divider.sol#L426-427,

- Periphery.sol#L70-71, Periphery.sol#L546-548, Divider.sol#L676.

Situation: Numerous parts of the logic above perform repeated calls to the `decimals()` function of `Target.sol` and `Underlying.sol`. At times, these are used in various calculations.

An attacker is able to mutate the variable returned by `decimals()` multiple times intra-transaction if they so wished, when it comes to a permissionless or untrusted `ERC-20`, `target` or `underlying`.

This could lead to exploits due to the return value for `decimals()` is used for calculating balance transfers and other important logic.

Here is an proof of concept example of an `EvilToken` contract cast with Rari's `ERC-20` abstract, returning different `decimals()` results based on timestamp:

```
import { ERC20 } from "contracts/ERC20.sol"; //Rari ERC20

contract EvilToken {
    function decimals() view public returns (uint8) {
        return uint8(block.timestamp % 18);
    }
}

contract Victim {
    function getTokensDecimals(address token) view public returns (uint8) {
        return ERC20(token).decimals();
    }
}
```

Recommendation: Any external calls should pass some logical pre-conditions. The result of those external calls should be stored in internal contracts and be re-used when the result is expected to be constant, as is the case with `decimals()`.

In this case, the external call to `decimals()` or other external "constants" should be consolidated in one call backed by preconditions. When these preconditions are met, the call should accept it, cache the value in its respective adapter, and have future dependent reads of this constant from that location, rather than from an external call to `decimals()`.

After they pass logical preconditions, the adapter should save any constants pertaining to `Target` and `Underlying` tokens, rather than doing an external call each time. The danger with external calls is that they could return a different result, bypass initial logical preconditions, and mutate the result to achieve an

exploit.

Sense: We've decided to cache some values from adapters here & here, but for others (like target decimals), we've decided to leave them out under the assumption that a malicious actor can cause problems in many ways, and we can't make strong guarantees about them without reviewing the code. The onus will be on us to clearly communicate which adapters have been audited and are seen as safe

Spearbit: It is indeed important to show which adapters are audited.

3.1.7 LP Oracle Should Enforce 18 Decimals Or Use Decimal Flexible Fixed Point Math

Severity: *High Risk*

Context: LP.sol#L87-L88

```
uint256 value = PriceOracle(msg.sender).price(tokenA).fmul(balanceA,
↪  FixedPointMathLib.WAD) +
   PriceOracle(msg.sender).price(tokenB).fmul(balanceB,
↪  FixedPointMathLib.WAD);
```

Situation: This calculation assumes all priced tokens have 18 decimals. This can be implicitly enforced with `Zero` tokens which can be deployed at 18 decimals every time. However, target tokens exist that are not 18 decimals, such as `USDC` (6 decimals) or any user-created `ERC-20` could set decimals to `uint8` bounds.

If a target token lower than 18 decimals is passed, this would yield the calculation to return an undervaluing of the `LP` tokens. If the target token was greater than 18 decimals, it could yield an overvalue.

Both of these could lead to attacks. According to the documentation, the creation of `Zeroes` and `Fuse` pools is intended to become permissionless. This means that users could build malicious pools that purposely undervalue or overvalue to steal the tokens of other users that join it.

Recommendation: Enforce and support only tokens with 18 decimals, which is likely the safest option, considering the complexity. This contract should check the decimals variable of both tokens passed.

Alternatively, consider making the calculation more flexible, by utilizing the token's decimals as the base units for the fixed point math, rather than the constant `WAD`.

```
uint256 value = PriceOracle(msg.sender).price(tokenA).fmul(balanceA,
↪  10**tokens[0].decimals()) +
   PriceOracle(msg.sender).price(tokenB).fmul(balanceB,
↪  10**tokens[1].decimals());
```

With permissionless tokens, there is still a risk of abuse and exploitation with this flexibility; a malicious party could create an `ERC-20` contract that reports a certain `decimals()` value under normal circumstances and a different one when they wish to conduct the attack. By checking `tx.origin`, `msg.sender`, or another global variable they may have control over, they may trigger an attack.

With the current set of contracts, there isn't a safe, flexible method, but another issues ("untrusted `ERC-20` decimals") is available for a fix for this. This issue suggests that the `decimals()` pass some preconditions and then be stored in the adapter. Pulling `decimals()` from the adapter instead should be safer.

```
uint256 value = PriceOracle(msg.sender).price(tokenA).fmul(balanceA,
↪  adapter.cachedDecimals(tokenA) +
   PriceOracle(msg.sender).price(tokenB).fmul(balanceB,
↪  adapter.cachedDecimals(tokenB));
```

This assumes an implementation where the adapter pulls then checks and caches the `decimals()` for tokens pertaining to it.

Sense: Addressed in #165.

Spearbit: I would consider it provisionally fixed, because that specific fix potentially expands the "untrusted `ERC-20` decimals" vulnerability, at least in the case of untrusted `ERC-20`, potentially added when permissionless.

Sense: Noted! However, there are no plans at this time to allow permissionless adapters to use the fuse pool (the consumer of the oracle).

Spearbit: Sounds good in this regard then, the tokens should not be untrusted and are expected to be audited + verified on Etherscan or reproducible bytecode confirmed.

3.1.8 Intra-Transaction Oracle Tampering Possible With LP Pricing Using Flashloans

Severity: *High Risk*

Context: `LP.sol#L87-L88`

```
uint256 value = PriceOracle(msg.sender).price(tokenA).fmul(balanceA,
↪  FixedPointMathLib.WAD) +
   PriceOracle(msg.sender).price(tokenB).fmul(balanceB,
↪  FixedPointMathLib.WAD);
```

Situation: This utilizes the current balances of the respective tokens within the `LP`. As these are current balances, they are trivial to manipulate via a flash loan. By setting arbitrary pool values only true for a certain point of the transaction, an attacker could potentially tamper with the price.

An attacker could also exploit this to create a skewed price of one of the tokens in the `fuse` pool and then drain the contract of the rest. Recent similar attacks have been done in the wild on `Fuse`. This resulted from weak Oracles, even in the case of ones with `TWAP`.

The associated `Zero` price should have a `TWAP` backing it according to the current commit hash, when it is completed and working. However, if the oracle is enabled on a pool without sufficiently safe liquidity, it may too be vulnerable to flash loan attacks, even with the `TWAP`.

The target price, which will be the other token used in the calculation, may also be manipulable. This token also does not appear to be a `TWAP` based on the 2 current adapter implementations. Therefore, it is even more trivial to manipulate, even with sufficient liquidity. This is true except in the case that a constant pricing is used, i.e. for `CETH`.

Compound-based adapters will likely use their built-in Oracle pricing. Prior implementations have been exploited in the past, even with stablecoins such as DAI leading to *millions* in losses.

Recommendation: Avoid using variables that are current and that the transaction originator (`tx.origin`) could temporarily change during the course of the transaction. This is most commonly done now by using time-weighted calculations of the variables at hand.

Utilizing these makes it much harder to pull off a profitable intra-transaction attack and will likely require to be spread over multiple blocks. The more blocks, the likelier any skewing by the prospective attacker is arbitraged/MEV'd out. This increases the risk of loss to a potential attacker and discouraging such an attack. As a result, intra-transaction attacks are practically risk-free.

One part of the Balancer pools is the ability to enable it as an oracle. These should *only* be enabled once they attain a minimal liquidity threshold that is

sustained over a period of time. Even if using `TWAP`, a low-liquidity pool could still yield profitable attacks within a relatively short number of blocks of the tampering transaction.

It is also important to consider multiple sources of truth; if available (and ideally on-chain), require them to be within some acceptable margin of error to each other. If they are outside of this margin, they should fail. The sources should always come from a source above a liquidity threshold, as one low liquidity source could be used to grief others.

Sense: We are exploring viable solutions to this with the Balancer team. Our work is in the direction of this suggestion:

> Additionally consider multiple sources of truth if available, ideally on-chain, and require them to be within some acceptable margin of error to each other, but fail if they are outside of this margin.

Namely, there are certain properties of Zeros that will let us bound a reasonable price (a Zero cannot go above 1 underlying, and the price will not be too dislocated from other Zeros of similar Series).

3.1.9 Avoid External Calls To Transient Or Unverified External Contracts

Severity: *High Risk*

Context: `PoolManager.sol#L190`

```
uint256 err = ComptrollerLike(comptroller)._deployMarket(false,
↪  constructorData, targetParams.collateralFactor);
```

`PoolManager.sol#L254`

```
uint256 errZero = ComptrollerLike(comptroller)._deployMarket(
```

`PoolManager.sol#L274`

```
uint256 errLpToken = ComptrollerLike(comptroller)._deployMarket(
```

Situation: External calls take place via the `_deployMarket` function through the set `Comptroller`. Rari Capital's on-chain Ethereum mainnet `Comptroller` calls out to a contract called `FuseAdminProxy`. This contract's implementation can be transient and it's underlying implementation source code is missing as well as unverified on Etherscan. It handles the deployment and potential control of `fuse` pool derived tokens.

Interacting with transient, unverified contracts on-chain can be quite dangerous, as its underlying logic could turn malicious at some point. Due to the difficulty in verifying states or effects, interacting with unverified/missing contracts could lead to some malicious contract interactions. Below are the listed on-chain contracts as reference points with links to their respective Etherscan page.

- `FuseAdmin` proxy contracts (`AdminUpgradabilityProxy.sol`) on Etherscan
- Unverified Implementation used by proxy on Etherscan
- `Unitroller` is directly called here, which is an upgradeable proxy contract on Etherscan

Currently, the above point to the following `Comptroller` implementation on Etherscan.

Recommendation: Make sure any dependencies of the code logic lead to *verified* contracts only.

With regards to transient or proxy and/or upgradeable contracts, keep in mind that there is a high degree of trust put into whatever personnel are managing said proxy. Be absolutely sure to trust the proxy personnel to not eventually deploy a breaking and/or malicious contract. If Sense Team must, then ideally, the team will consider only interacting with proxy patterns that may allow the dependent users to opt-in to logic or implementation changes.

Sense: Noted that the Rari Capital team has been informed of the unverified contract and will see to it to be verified.

3.1.10 Check `zeroParams` and `lpTokenParams` Is Not Empty

Severity: *High Risk*

Context: `PoolManager.sol#L158-197`

Situation: The function `addTarget()` checks `targetParams` is not empty. However, `addSeries()` does not check for that and `zeroParams` and `lpTokenParams` are not empty.

As these parameters are set via `setParams()`, they might not be set yet. Combined this with the fact that anyone can call `addSeries()` and there is a possibility of a griefing attack.

```
function addTarget(address target, address adapter) external requiresTrust
↪   returns (address cTarget) {
    ...
    require(targetParams.irModel != address(0), Errors.TargetParamNotSet);
    ...
    bytes memory constructorData = abi.encode(
        ...
        targetParams.irModel,
        ...
    );
    uint256 err = ComptrollerLike(comptroller)._deployMarket(false,
↪   constructorData, targetParams.collateralFactor);
    ...
}

function addSeries(address adapter, uint48 maturity) external {
    ... // no checks on zeroParams
    bytes memory constructorDataZero = abi.encodePacked(
        ...
        zeroParams.irModel,
        ...
    );
    uint256 errZero = ComptrollerLike(comptroller)._deployMarket(false,
↪   constructorDataZero, zeroParams.collateralFactor);
    ... // no checks on lpTokenParams
    bytes memory constructorDataLpToken = abi.encodePacked(
        ...
        lpTokenParams.irModel,
        ...
    );
    uint256 errLpToken =
↪   ComptrollerLike(comptroller)._deployMarket(false,constructorDataLpToken,
↪   lpTokenParams.collateralFactor);
    ...
}
```

Recommendation: Verify that `zeroParams` and `lpTokenParams` are not empty within the function `addSeries()`.

Sense: Addressed in #156.

Spearbit: Acknowledged.

3.1.11 Use `safeMath` for `Space` contract

Severity: *High Risk*

Context:

- Space.sol#L132, Space.sol#L133, Space.sol#L182, Space.sol#L213,
- Space.sol#L214, Space.sol#L263, Space.sol#L264, Space.sol#L291,
- Space.sol#L300, Space.sol#L356, Space.sol#L366, Space.sol#L369,
- Space.sol#L373, Space.sol#L379, Space.sol#L401, Space.sol#L418,
- Space.sol#L423, Space.sol#L435, Space.sol#L439, Space.sol#L443,
- Space.sol#L447, Space.sol#L456, Space.sol#L493, Space.sol#L498,
- Space.sol#L503, Space.sol#L509, Space.sol#L510, Space.sol#L516,
- Space.sol#L523, Space.sol#L524.

Situation: `onJoinPool()` is high risk, as it could mint a large amount of tokens and other locations in `Space.sol`.

The safe contract builds on Balancer contracts and thus uses solidity 0.7.x.

However, the math operations in solidity 0.7.x can underflow and overflow. The safe contract doesn't have sufficient protection against this.

Recommendation: Use `safeMath` functions for all addition, subtraction, multiplication and divisions, from for example Open Zeppelin `SafeMath.sol` or Balancer `Math.sol` libraries for solidity 0.7.x.

Sense: Addressed in #3. Note in this PR that everywhere `safeMath` is not used, an explicit reason is given.

Spearbit: Acknowledged.

3.2 Medium Risk Issues

3.2.1 Attempted Out Of Bound Array Access On `TWAP` Result When `Zero Address >= Target Address`

Severity: *Medium Risk*

Context: `Zero.sol#L87-98`

```
BalancerOracleLike.OracleAverageQuery[] memory queries = new
↪  BalancerOracleLike.OracleAverageQuery[](1);
queries[0] = BalancerOracleLike.OracleAverageQuery({
    variable: BalancerOracleLike.Variable.PAIR_PRICE,
    secs: TWAP_PERIOD,
    ago: 0
});

uint256[] memory results = pool.getTimeWeightedAverage(queries);
// get the price of Zeros in terms of underlying
(uint8 zeroi, ) = pool.getIndices();
uint256 zeroPrice = results[zeroi];
```

Situation: `zeroi` can be either `0` or `1`, dependant upon the `zero` address being smaller or equal/larger than its corresponding target address. In the latter case, out of bounds array access is attempted on the results variable, which only returns 1 result. This matches the requested number of queries, which is 1. This out of bounds access leads to a panic code-path and will revert the transaction.

This means any `Zero` address that has a larger `bytes20` representation than its target would be DoS'd from accessing the `Zero` oracle. Thereby, a number of functionalities for it would never be able to work. In the case of non-deterministic address being used, the chance of this occurring is equivalent to a coin toss, i.e. 50/50.

If it had been deployed in the wild, this may not initially be noticed. Due to the probabilistic nature of this problem, it may go unnoticed, even with some initial successful deployment of `Zero` contracts. However, when it has been discovered, it would require a pause and redeploy with fix.

References Balancer v2 test snippet confirming behaviour of `getTimeWeightedAverage` returning 1 result per query: `PoolPriceOracle.test.ts#L521-L540`.

Recommendation: With array access or modification, always ensure it is kept within bounds (and that its bounds are known). The addition of unit tests and fuzzed tests, attempting different scenarios, could help avoid this.

In this case, the contract should only access the result at index`0`, and mathematically handle the pricing as needed, if it's reciprocal is required.

3.2.2 Force Claim Collection For Any Address

Severity: *Medium Risk*

Context: Divider.sol#L331-398, Claim.sol#L37-44

Situation: In this scenario, assume an actor does a `transferFrom` transaction on the `Claim` token contract with a specific `from` and an amount of `0` tokens. This transaction will succeed in the `ERC20` contract as `allowance == 0` and `amount <= allowance`, then `collect()` in `Divider.sol` will be called with `uBalTransfer == 0`. The `_collect()` function will be called where `uBal` and `uBalTransfer` are set to `uBal` (which is the balance of the `from`).

This scenario triggers the claim collection from the `from`. The `from` might not want to do the claim collection at that point in time. The claim itself will go to the `from`, so nothing is lost there, but the control over timing could be an issue.

```
contract Claim is Token {
    ...
    function transferFrom(...) public override returns (bool) {
        Divider(divider).collect(from, adapter, maturity, value, to);
        return super.transferFrom(from, to, value);   // No revert on
↪   super.transferFrom(from, to, 0);
    }
}
contract Divider is Trust, ReentrancyGuard, Pausable {
    ...
    function collect(address usr,address adapter, uint256 maturity, uint256
↪   uBalTransfer, // uBalTransfer == 0
        address to) external nonReentrant onlyClaim(adapter, maturity)
↪   whenNotPaused returns (uint256 collected) {
        uint256 uBal = Claim(msg.sender).balanceOf(usr);
        return _collect(usr, adapter, maturity, uBal, uBalTransfer > 0 ?
↪   uBalTransfer : uBal, to); //_collect is called with uBal as second last
↪   parameter
    }
}
```

This construction is probably created to support the calling of `collect()` in contract `Claim.sol`:

```
function collect() external returns (uint256 _collected) {
    return Divider(divider).collect(msg.sender, adapter, maturity, 0,
↪   address(0));
}
```

Recommendation: Differentiate between these two calls from the `Claim.sol` contract: `transferFrom()` and `collect()` and do not trigger a `collect()` from an empty `transferFrom()`.

Sense: Fixed in #168.

Spearbit: Acknowledged.

3.2.3 Disabled Adapters Should Stay Disabled

Severity: *Medium Risk*

Context: Divider.sol#L107-109

```
function addAdapter(address adapter) external whenPermissionless whenNotPaused
 ↪  {
    _setAdapter(adapter, true);
}
```

Situation: Anyone can enable an adapter again as soon as someone has disabled the adapter (in `whenPermissionless`). This circumvents the reason to disable the adapter.

Recommendation: Don't allow an adapter that has been removed (i.e. it was active and it is being disabled) to be re-added.

As discussed, this is the preferred solution of the Sense team.

Sense: Fixed here.

Spearbit: Acknowledged.

3.2.4 Composability of `GClaimManager`

Severity: *Medium Risk*

Context: GClaimManager.sol, Claim.sol, Divider.sol

Situation: When calling the function `join()` and `exit()` from contract `GClaimManager`, the function `_collect()` of `Divider` is called via `transfer()` or `transferFrom()` and `collect()`.

The function `_collect()` can revert in certain circumstances, which means `join()` and `exit()` will also revert.

As these functions are meant for composability, this can pose a problem because transactions will not be able to continue.

```
contract GClaimManager {
    function join(address adapter, uint48 maturity, uint256 uBal) external {
        ...
        ERC20(claim).safeTransferFrom(msg.sender, address(this), uBal);
        ...
    }
    function exit(address adapter, uint48 maturity, uint256 uBal) external {
        ...
        ERC20(claim).safeTransfer(msg.sender, uBal);
        ...
    }
}
contract Claim is Token {
    function transfer(address to, uint256 value) public override returns (bool)
↪   {
        Divider(divider).collect(msg.sender, adapter, maturity, value, to);
        ...
    }
    function transferFrom(address from, address to, uint256 value) public
↪   override returns (bool) {
        Divider(divider).collect(from, adapter, maturity, value, to);
        ...
    }
}

contract Divider is Trust, ReentrancyGuard, Pausable {
    function collect(...) external nonReentrant onlyClaim(adapter, maturity)
↪   whenNotPaused returns (uint256 collected) {
        ...
        return _collect(usr, adapter, maturity, uBal, uBalTransfer > 0 ?
↪   uBalTransfer : uBal, to);
    }

    function _collect(...) internal returns (uint256 collected) {
        ...
        if (_settled(adapter, maturity)) {
            ...
        } else {
            // If we're not settled and we're past maturity + the sponsor
↪   window,
            // anyone can settle this Series so revert until someone does
            if (block.timestamp > maturity + SPONSOR_WINDOW) {
                revert(Errors.CollectNotSettled);
↪   // in some situation revert occurs
        }
    }
}
```

Recommendation: Determine the goals for composability and possibly reconsider the revert.

Note: `GClaimManager` will most likely be deprecated. This issue is included for completeness.

3.2.5 Reentrancy Safeguards Needed In `GClaimManager`

Severity: *Medium Risk*

Context: GClaimManager.sol#L77-104

Situation: In function `exit()` of `GClaimManager.sol`, the burning on the tokens is done at the end.

This is not according to the Checks-Effects-Interactions pattern and no `nonReentrant` modifier is used.

Reentrancy is possible with a malicious adapter contract and is potential in other situations as well.

```
function exit(address adapter, uint48 maturity, uint256 uBal) external {
    ...
    // External call
    uint256 collected = Claim(claim).collect();
    ...
    // External call
    uint256 tBal = uBal.fdiv(gclaims[claim].totalSupply(), total);
    ...
    // External call
    ERC20(Adapter(adapter).target()).safeTransfer(msg.sender, tBal);
    ...
    // External call
    ERC20(claim).safeTransfer(msg.sender, uBal);
    // Burn the user's gclaims
    // Completed at the very end with potential reentrancy above
    gclaims[claim].burn(msg.sender, uBal);
    ...
}
```

Recommendation: To improve security, add a `nonReentrant` modifier on functions of `GClaimManager` that do external calls.

Note: `GClaimManager` will most likely be deprecated. This issue is included for completeness.

3.2.6 Do `collect()` First In `GClaimManager`

Severity: *Medium Risk*

Context: `GClaimManager.sol#L36-75`

Situation: The function `join()` of `GClaimManager.sol` pulls target to backfill for previous `collect()`s. However, if `collect()` hasn't been called for a long time (or not called at all), the user might not have enough target for the backfill.

```
function join(address adapter,uint48 maturity,uint256 uBal) external {
    ...
    /* Pull the amount of `Target` needed to
     backfill the `excess` back to issuance,
     retrieves previously `collect()`'ed `target`
    */
    ERC20(Adapter(adapter).target()).safeTransferFrom(msg.sender,
↪   address(this), tBal);
    ...
    // Pull Collect Claims to GClaimManager.sol
    ERC20(claim).safeTransferFrom(msg.sender, address(this), uBal);
    /* This will call `Divider.collect()`
    and send target to the `msg.sender` */
    ...
}
```

Recommendation: If the Sense Team agrees this is an issue, then call `Divider.collect()` at the beginning of function `join()`.

Note: `GClaimManager` will most likely be deprecated. This issue is included for completeness.

3.2.7 Do Not Use `abi.encodedPacked` With Multiple Dynamically-Sized Types

Severity: *Medium Risk*

Context: `Fuse.sol#L242-L252`, `Fuse.sol#L262-L272`

Situation: The `abi.encodePacked` method is called where there are multiple dynamically-sized types. Trying to pack these dynamically-sized types leads to ambiguity in unpacking. This makes it difficult to impossible to unpack, especially in a safe manner. It is made more difficult, in a case like this, because users have some limited input control over the contents passed via the name and symbol parameters. This could present a danger, when the contract becomes permissionless for adding custom `zero/claim` adapters.

```
bytes memory constructorDataZero = abi.encodePacked(
    zero,
    comptroller,
    zeroParams.irModel,
    ERC20(zero).name(),
    ERC20(zero).symbol(),
    cERC20Impl,
    "0x00",
    zeroParams.reserveFactor,
    adminFee
);
```

Recommendation: `abi.encode` should be used here instead and `abi.decode` to appropriately decode the externally called contract.

Sense: Addressed in #156.

3.2.8 Imported `Trust` Contract Has Poor Access Controls

Severity: *Medium Risk*

Context: `solmate/Trust.sol`

Situation: This contract gives blanket authority to any and every single account that is trusted. This means full access to any privileged contract functions. It also allows any trusted account to unilaterally set or remove trusted users.

Therefore, if any single trusted account were to ever be compromised, the malicious accounts could lock out all other trusted accounts, including the compromised account, leaving control only in the hands of the malicious actors.

Based on the work-in-progress deployment scripts inspected within the project, the `Divider` contract would be deployed by an EOA (externally owned account, i.e. regular Ethereum address behind a private key) named `deployer`, and subsequently another EOA named `Dev` would be added as an additional trusted user.

If either were to be compromised, which is very well likely to occur with an EOA, it would lead to a compromise of the `Divider` contract with its privileged functions. The more EOAs are added as trusted users, the greater the risk with this `Trust` contract.

Additionally, `TokenHandler` and `PoolManager` are insofar deployed by an EOA, which becomes their trusted account. `Periphery` will also depend on a trusted

account. These contracts expose a number of privileged actions to trusted users.

In some cases, the `Trust` contract is utilized to set the contract that deployed it as its trusted user. In many cases, this will be safer with the assumption that the contract does not have some ability or vulnerability of adding other trusted users.

Recommendation: Consider a more robust RBAC (Role Based Access Control) Trust and Authority contract to import for such functionality that allows for finer-grained control as to which users have which permissions, rather than a blanket `'root'` access among all users.

With such a contract, a secure multisig would be best to be utilized as `root`, with all permissions and able to add or remove permissions to other addresses. EOAs would be safer to utilize with this pattern, but one would ideally just utilize them still for lower privileged common calls, while leaving higher privileged calls in the hands of multisigs.

This `Trust` contract can be acceptable for contract-to-contract trust. However, this is true only if either the ability to add or remove new trusted users is removed or it is assured that the trusted contract has no methodologies of adding new trusted users in such a design.

If the team wishes to continue using this `Trust` library, it should only operate on it utilizing a multisig.

Initial deployment is acceptable by an EOA, however, before it gets deployed live, all access controls should be solely with secure multisigs from that point forward.

Sense: We've decided to stick with `Trust` and use a multisig.

3.2.9 Move The `safeApprove()` Stake to `BaseAdapter.sol` and Update the Inheritance in `Periphery.sol`

Severity: *Medium Risk*

Context: `BaseAdapter.sol`, `CropAdapter.sol`, `WstETHAdapter.sol`, `CAdapter.sol`, `Periphery.sol`

Situation: The inheritance of the `BaseAdapter.sol` isn't quite right, which results in a few issues:

`Periphery.sol` imports `Adapter` from `CropAdapter.sol`, while `WstETHAdapter.sol` inherits directly from `BaseAdapter.sol`.

The `safeApprove()` of stake in the `constructor` for `CropAdapter` isn't called for `WstETHAdapter.sol`.

This means that the `Divider` cannot transfer the stake, which it tries to do in `settleSeries()` and `backfillScale()`.

Recommendation: Move the `safeApprove()` of stake to `BaseAdapter.sol` (assuming the stake is relevant for all adapters).

In `Periphery.sol`, change the inheritance from `CropAdapter.sol` to `BaseAdapter.sol`.

Sense: We've changed here the `Periphery` to import `BaseAdapter` and not `CropAdapter`. Also, we have removed the `safeApprove()` there which fixes the mentioned issue.

Do you still think that it's better to merge `CropAdapter` into `BaseAdapter`? The idea was to keep them separate as not all the adapters need the crop functionality (this is the case of the `CAdapter` for example).

Spearbit: If it's useful to keep `CropAdapter` separate that is also fine.

However, the linked change does not seem to remove the `safeApprove()` call from the `CropAdapter` constructor as stated.

3.2.10 Checks In `issue()` And `redeemZero()`

Severity: *Medium Risk*

Context: `Divider.sol#L183-231`, `Divider.sol#L279-313`

Situation: The functions `issue()` in `Divider.sol#L183-231` and `redeemZero()` of `Divider.sol` have extra checks when the flags `level.issueRestricted()` or `level.redeemZeroRestricted()` are set.

In that case, the function can only be executed when called from an adapter.

The contract `Periphery`, specifically `Periphery.sol#L410`, `Periphery.sol#L514`, `Periphery.sol#L557`, calls these functions as well. However, these calls would not be allowed. The extra checks are potentially too strict.

```
function issue(...) external nonReentrant whenNotPaused returns (uint256 uBal)
↪ {
    ...
    if (level.issueRestricted()) {
        require(msg.sender == adapter, Errors.IssuanceRestricted);
    }
    ...
}

function redeemZero(...) external nonReentrant whenNotPaused returns (uint256
↪ tBal) {
    ...
    if (level.redeemZeroRestricted()) {
        require(msg.sender == adapter, Errors.RedeemZeroRestricted);
    }
}
```

Recommendation: Double check if calling from `Periphery` should also be allowed.

Sense: After seeing this issue, we've decided that if `redeemZeroRestricted()` in `Divider.sol#L289-291` is set, then when `_removeLiquidity()` in `Periphery.sol#L551-578` is called at or after maturity we will not be redeeming `zero`s and will transfer the `zero`s back to the user. You can see this change on PR#160.

Spearbit: Acknowledged.

3.2.11 Defining Bytes Literals With A String Literal Causes Malformed Bytes

Severity: *Medium Risk*

Context: `PoolManager.sol#L185`, `PoolManager.sol#L249`, `PoolManager.sol#L269`, `Periphery.sol#L375`, `Periphery.sol#L465`, `Periphery.sol#L472`

Situation: The bytes variable is set by a string literal. The variable's data in this case will be equivalent to the `ASCII` encoded data of the string, not a hex literal version of it.

This causes the variable to contain malformed bytes which the developer likely didn't expect. In the best case, it causes unnecessary extra gas cost. Worst case scenario, it could lead to undefined and unexpected effects during contract interaction, especially in the case of said bytes being passed onto other contracts as `calldata`.

e.g.PoolManager.sol#L185

```
bytes memory constructorData = abi.encode(
    target,
    comptroller,
    targetParams.irModel,
    ERC20(target).name(),
    ERC20(target).symbol(),
    cERC20Impl,
    "0x00", // calldata sent to becomeImplementation (currently unused) L185
    targetParams.reserveFactor,
    adminFee
);
```

The developer would appear to expect bytes as `0x00` to be sent along as `calldata`, but in fact this would send `0x30783030` as the `calldata`, the ASCII encoded bytes equivalent of `0x00`. The prospective contract this would call out to, for now, appears to essentially ignore these bytes. However, future implementations could end up depending on it which would lead to adverse effects and issues.

One of the on-chain contracts within this call-chain that may use this `calldata` is an unverified contract on chain, which could lead to issues, but cannot be confirmed due to lack of source code availability.

If this ought to be `0x00`, define it using a hex literal as `hex'00'`.

Periphery.sol#L465

```
userData: ""
```

It is an exception from the other mentioned context lines, since it's just an empty string literal and won't cause malformed bytes. However, it would still be best practice to define with a hex literal. The malformed bytes issue is why hex literals should be preferred over even hex escaped strings for bytes.

Recommendation: Use hex literals whenever declaring or setting a bytes type, via `hex'01234dead5678beef'` for most clarity that it is bytes.

If Sense prefers to utilize strings, appropriate hex escapes in the form of `"\x01\x23"` also work.

Sense: Addressed in #156.

3.2.12 Beware Of Malicious Adapters

Severity: *Medium Risk*

Context: `Periphery.sol`

Situation: The following functions allow for interaction with *any* adapter. These adapters could be malicious. No additional checks are made on the validity of the adapter.

Through the adapter, a re-entrant call could be made and no `nonReentrant` modifier is used in the `Periphery` contract.

```
function sponsorSeries(address adapter, ... )
function swapTargetForZeros(address adapter, ... )
function swapUnderlyingForZeros(address adapter, ... )
function swapTargetForClaims(address adapter, ... )
function swapUnderlyingForClaims(address adapter, ... )
function swapZerosForTarget(address adapter, ... )
function swapZerosForUnderlying(address adapter, ... )
function swapClaimsForTarget(address adapter, ... )
function swapClaimsForUnderlying(address adapter, ... )
function addLiquidityFromTarget(address adapter, ... )
function addLiquidityFromUnderlying(address adapter, ... )
function removeLiquidityToTarget(address adapter, ... )
function removeLiquidityToUnderlying(address adapter, ... )
function migrateLiquidity(address srcAdapter, address dstAdapter,...)
```

Within `Periphery.sol`, there are several locations where a `safeApprove()` is given to an adapter. Frequently, the token for which the `safeApprove()` is given is also derived from the adapter that can also be manipulated. These are the relevant lines:

```
ERC20(target).safeApprove(address(adapterClone), type(uint256).max);
ERC20(Adapter(adapter).underlying()).safeApprove(adapter, uBal); // approve
↪  adapter to pull uBal
ERC20(Adapter(adapter).underlying()).safeApprove(adapter, uBal); // approve
↪  adapter to pull underlying
ERC20(Adapter(adapter).target()).safeApprove(adapter, tBal); // approve adapter
↪  to pull target
underlying.safeApprove(adapter, uBal);
ERC20(Adapter(adapter).target()).safeApprove(adapter, tBal);
if (_allowance < amount) target.safeApprove(address(adapter),
↪  type(uint256).max);
```

A malicious adapter could therefore steal any tokens present in `Periphery` contract. This is true for both now and in the future, as the approval will persist.

Recommendation: Consider using a whitelist for the adapters. Add a `nonReentrant` modifier to the function that use the adapter. Make sure no tokens will be stored in the `Periphery` contract (now and in the future).

Sense: No tokens should be present in the Periphery contract so this reduces the potential impact.

Sense: Regarding:

> Make sure no tokens will be stored in the Periphery contract (now and in the future).

Yes, this is a great call and we're currently going through to verify this. On the note of malicious adapters, our thinking is that we will need to be very clear which adapters have been verified/audited, because unknown ones we can make no guarantees about.

Spearbit: Agreed.

3.2.13 Admin Can Always Update `lscales` Levels

Severity: *Medium Risk*

Context: Divider.sol#511-547, Divider.sol#L466-468

Situation: An administrator of the protocol can arbitrarily set the `lscales` levels of users at any moment. This directly impacts the amount that can be collected in the `collect()` function.

The update of `lscales` levels can be done by setting the adapter to `off`, which will allow the require in `backfillScale()` to continue. Following this, a call to `backfillScale()` should be done to set arbitrary values to the `lscales`. Finally, set the adapter back to `on`.

Although this is protected by `requiresTrust`, it is probably best to limit this possibility.

```
function setAdapter(address adapter, bool isOn) public requiresTrust {
    _setAdapter(adapter, isOn);
}

function backfillScale(..., address[] calldata _usrs, uint256[] calldata
↪   _lscales) external requiresTrust {
    ...
    // continues when adapters[adapter] == false
    require(!adapters[adapter] || block.timestamp > cutoff, ...);

    /* Set user's last scale values the Series
    (needed for the `collect` method) */
    for (uint256 i = 0; i < _usrs.length; i++) {
        lscales[adapter][maturity][_usrs[i]] = _lscales[i];
    }
    ...
}
```

Recommendation: Double check the circumstances under which an administrator can perform such updates.

Sense: We've decided to keep this ability for admins for edge cases. Instead of removing it, it'll be clearly mentioned in the docs and we'll run dis processes to ensure that the community has sight into our thought processes. That said, we are internally still discussing this.

3.3 Low Risk Issues

3.3.1 `Trust.sol` No Longer Present In `solmate` Library

Severity: *Low Risk*

Context:

- `Divider.sol#L7`, `CropAdapter.sol#L5`, `EmergencyStop.sol#L5`, `LP.sol#L6`,
- `Periphery.sol#L6`, `PoolManager.sol#L6`, `SpaceFactory.sol#L8`,
- `Token.sol#L6`, `Underlying.sol#L5`, `Zero.sol#L6`, `Target.sol#L5`.

Situation: The contract `Trust.sol` is no longer present in the latest version of the Rari-Capital `solmate library`. This means no (security) updates/fixes will be made on the `Trust.sol` contract.

Recommendation: Fork the latest version of `Trust.sol` or move to another

authorization library.

Sense: We've decided to bump `solmate` to v6 and maintain `Trust.sol` ourselves inside our `utils` package. Fixed here.

3.3.2 Change References to `SafeERC20.sol` to `SafeTransferLib.sol`

Severity: *Low Risk*

Context:

- `Divider.sol#L6`, `GClaimManager.sol#L5`, `Periphery.sol#L5`,
- `BaseAdapter.sol#L5`, `CAdapter.sol#L6` `CropAdapter.sol#L6`,
- `LP.sol#L5`, `Pool.sol#L4`, `Vault.sol#L4`, `Zero.sol#L5`,
- `BaseFactory.sol#L6`, `WstETHAdapter.sol#L9`.

Situation: The functions from `SafeERC20.sol` are moved to `SafeTransferLib.sol`, which means the code need changes to be able to compile with the latest version of the solmate library.

Recommendation: Change the references from `SafeERC20` to `SafeTransferLib`.

Sense: Fixed here.

3.3.3 Entry Checks `exit()` and `excess()`

Severity: *Low Risk*

Context: `GClaimManager.sol`

Situation: The function `exit()` and `excess()` in `GClaimManager` don't check `join()` has been completed. The function `excess()` doesn't check `claim` exists. Although the functions will either revert or do non-harmful updates, it is better to check and give a relevant error message.

Also for consideration: the comment `"VIEWS"` is misleading, as the function `excess()` stores data.

```
function exit(address adapter, uint48 maturity,uint256 uBal) external {
    (, address claim, , , , , , , ) = Divider(divider).series(adapter,
 ↪  maturity);
    require(claim != address(0), Errors.SeriesDoesntExists);
    ...                                  // Doesn't check join() has been done
    uint256 total = totals[claim] + collected;
    ...
    gclaims[claim].burn(msg.sender, uBal); // will revert if join() hasn't been
 ↪  done
}

function excess(address adapter,uint48 maturity,uint256 uBal) public returns
 ↪  (uint256 tBal) {
    (, address claim, , , , , , , ) = Divider(divider).series(adapter,
 ↪  maturity);
                                         // Doesn't check claim != address(0)
    uint256 initScale = inits[claim];  // Doesn't check initScale !=0, e.g.
 ↪  join() has been done
}
```

Recommendation: Do appropriate entry checks in the functions `exit()` and `excess()`. Remove the `"VIEWS"`comment.

Note: `GClaimManager` will most likely be deprecated. This issue is included for completeness.

3.3.4 Require `SeriesStatus` to be `NONE` Before Calling `queueSeries`

Severity: Low Risk

Context: `PoolManager.sol#L210`

```
require(sStatus[adapter][maturity] != SeriesStatus.QUEUED,
 ↪  Errors.DuplicateSeries);
```

Situation: Currently, this line checks that the `SeriesStatus` is not `QUEUED`. This means, after it is added, a trusted user could once more re-queue the series, allowing others to potentially re-call `addSeries`. This is so potentially because, from the submitted contract, it is possible, and, from the verified contracts, this would actually call out to it also seems possible. However, there is at least one unverified contract in the call-chain. This could entail a griefing attack on this contract and `Fuse`.

Recommendation: Explicitly check that the `SeriesStatus` to be queued is

NONE.

3.3.5 `block.timestamp` Can Be Manipulated By Miners

Severity: *Low Risk*

Context:

- `Periphery.sol`, `GClaimManager.sol`, `Space.sol#L160`, `Space.sol#L401`,
- `Space.sol#L435`, `Divider.sol#L30`, `Divider.sol#L33`, `Divider.sol#L142`,
- `Divider.sol#L368`, `Divider.sol#L522`, `Divider.sol#L564`, `Divider.sol#L566`,
- `Divider.sol#L572`.

Situation: On the protocol level, timestamps are a variable submitted by miners. The protocol only guarantees its monotonicity. Therefore, they are non-trivially manipulable by miners for purposes of MEV.

It is a risk for critical logic of the contract to be dependent on such a variable. However, such manipulation would widely affect the Ethereum ecosystem. Therefore, there is a low chance miners would take such a risk.

This serves as a notice of the possibility of miner's manipulating of the timestamp for the purposes of MEV.

Recommendation: The past general recommendation regarding `block.timestamp` has been to consider `block.number` as a better gauge of time. However, with the upcoming merge and forks, it may become unreliable for this in the mid- to long-term right now.

For a contract that aims to operate consistently following the merge, `block.timestamp` may be the less risky option compared to `block.number` in regards to references for elapsed time.

The rule of thumb regarding this usage is to make sure that it is for references to longer time frames, i.e. nothing under 1 hour.

3.3.6 Dependence On `symbol()` Value

Severity: *Low Risk*

Context: `CAdapter.sol#L176-178`

Situation: The evaluation of `_isCETH` depends on the `symbol()` of a token. With permissonless use of the protocol, this might not be reliable.

Recommendation: Consider checking for (whitelisted) addresses instead of token `symbol()`.

3.3.7 Refrain From Shadowing State Variables

Severity: *Low Risk*

Context: Periphery.sol#L351, CAdapter.sol#L133, CAdapter.sol#L156.

Situation: Shadowing variables (`factory` and `target`), i.e. by having function parameters or other local variables declared with the same name, could lead to unintended consequences; it may even indicate underhanded code design.

Variable shadowing could also lead to various mistakes on the dev side. If a state variable is shadowed in a function, then developers think they are accessing and/or modifying the state variable when it is actually a local variable.

Recommendation: Avoid shadowing of state variables with local variables. One way to avoid this is to consider exclusively prefixing or suffixing an underscore to state variable names, which should make shadowing impossible, if followed consistently.

Sense: This was fixed in #155 and fixed in #158.

3.3.8 Maximum Value of `tilt`

Severity: *Low Risk*

Context: Divider.sol#L279-297

Recommendation: Make sure `tilt` is not larger than `FixedMath.WAD`, this can be done in the `constructor` of BaseAdapter.sol#L78-109

3.3.9 Reward Has Two Meanings

Severity: *Low Risk*

Context: Divider.sol#L157-180, Divider.sol#L334-414

Situation: Reward has 2 meanings: (1) Sum of all the fees in `Divider.sol` and (2) `COMP` tokens in adapter

This might be confusing. *Consider the following code snippet:*

```
function _collect(...) internal returns (uint256 collected) {
    ...
    Adapter(adapter).notify(usr, collected, false); // Distribute reward tokens
 ↪  // COMP tokens
    ...
}

function settleSeries(address adapter, uint48 maturity) external nonReentrant
 ↪  whenNotPaused {
    ...
    // Reward the caller for doing the work of settling the Series at around
 ↪  the correct time
    ...
    ERC20(target).safeTransferFrom(adapter, msg.sender,
 ↪  series[adapter][maturity].reward); // fees
    ...
}
```

Recommendation: Change one of the two rewards to different names.

Sense: Changed one of the two rewards to different names.

3.3.10 `Space.sol` Contract Missing Necessary Logic To Function As A Balancer Oracle

Severity: *Low Risk*

Context: `Space.sol`, `Zero.sol#L15-45`, `WeightedPool2Tokens.sol`

Situation: The `Space` contract is missing the implementation of a number of dependencies and functions that necessitate it to work as a `BalancerOracleLike` with the `MasterOracle` and `PriceOracles` within the `fuse` portion of this project.

`Zero.sol` context notes some of the missing parts.

The Sense team has notified us that it hasn't yet implemented these as they are in the midst of working on more fully understanding its nuances with the Balancer team. Hence, this is set as low risk, because it just entails a portion of the architecture failing to work, if deployed at this stage, if they were to deploy, and they are unlikely to do this.

Recommendation: Ensure oracle functionality is properly implemented before deployment so all parts of the architecture work as intended.

3.3.11 Multiple Fixed Math Libraries

Severity: *Low Risk*

Context: `FixedMath.sol`, `Rari: FixedPointMathLib.sol`, `FixedPoint.sol`

Situation: Three different `FixedMath` libraries are used in the code. The more libraries are used with duplicate intended purpose the higher the risk of undetected issues arising from potential differences in how they may work.

The first library, `FixedMath.sol`, is used in the following: `BaseAdapter.sol#L6`, `CAdapter.sol#L5`, `CropAdapter.sol#L12`, `Divider.sol#L10`, `GClaimManager.sol#L6`, `Periphery.sol#L7`, `WstETHAdapter.sol#L5`

As:

```
import { FixedMath } from "../external/FixedMath.sol";
```

The second library, `FixedPointMathLib.sol` from Rari in the solmate library, is used in the following: `LP.sol#L8`, `Target.sol#L7`, `Underlying.sol#L7`, `Zero.sol#L8`

As:

```
import { FixedPointMathLib } from
↪  "@rari-capital/solmate/src/utils/FixedPointMathLib.sol";
```

The third library, `FixedPoint.sol` from Balancer Labs, is used in the following:

`Space.sol#L6`, `SpaceFactory.sol#L5`

As:

```
import { FixedPoint } from
↪  "@balancer-labs/v2-solidity-utils/contracts/math/FixedPoint.sol";
```

Recommendation: Reduce the number of libraries as much as possible. Also, avoid using different library implementations aiming to achieve the same goal. Choose the one that best suits the needs of your projects, is safe, and backed by audits.

Sense: Solmate's `FixedPointMathLib` removed here (would have gone with Solmate's exclusively but `mulDow` or `mulUp` is in next but not in the latest stable version just yet). `FixedPoint` from Balancer is kept because it's for `0.7.0` and it has and it can do `pow` functions with floating point numbers.

Spearbit: Acknowledged. Do note, you may wish to check the licenses for that code as it appears pulled from `yield utils v2`. Following the links to the files,

they appear to have `BUSL-1.1` as their license although their root `README` states all files within the repository are licensed as `MIT`. Just something for internal consideration.

3.3.12 Check For Master Oracle

Severity: *Low Risk*

Context: `LP.sol`, `Zero.sol`

Situation: The functions `_price()` in the contracts `ZeroOracle` and `LPOracle` do a call to `msg.sender`.

If `msg.sender` isn't the `master oracle`, then these calls will fail.

```
contract ZeroOracle is PriceOracle, Trust {
    function _price(address zero) internal view returns (uint256) {
        ...
        // assumes the caller is the maser oracle, which will have its own way
 ↪  to get the underlying price
        return zeroPrice.fmul(PriceOracle(msg.sender).price(underlying),
 ↪  FixedPointMathLib.WAD);  // call to msg.sender
    }
}
contract LPOracle is PriceOracle, Trust {
    function _price(address _pool) internal view returns (uint256) {
        ...
        uint256 value = PriceOracle(msg.sender).price(tokenA).fmul(balanceA,
 ↪  FixedPointMathLib.WAD) +  // call to msg.sender
            PriceOracle(msg.sender).price(tokenB).fmul(balanceB,
 ↪  FixedPointMathLib.WAD);                          // call to msg.sender
        ...
    }
}
```

Recommendation: Consider verifying the calling `msg.sender` is indeed the `master oracle`. However, since these functions are `view`, access control is required.

This just means that if someone is not a `master oracle` or other price oracle type is calling, then they are wasting their money on gas and it's ultimately the caller's issue.

3.3.13 Implement Checks For `g1` and `g2`

Severity: *Low Risk*

Context: SpaceFactory.sol, Space.sol#L105-144

Situation: The function `setParams()` does some validity checks on `g1` and `g2`. However these checks are not present in the constructors of `SpaceFactory` and `Space.sol`.

This might lead to situation where the values are set in an incorrect way.

```
contract SpaceFactory is Trust {
    constructor(...) Trust(msg.sender) {
        ...
        g1 = _g1;  // no checks for g1
        g2 = _g2;  // no checks for g2
    }
    function setParams(uint256 _ts,uint256 _g1, uint256 _g2) public
↪   requiresTrust {
        // g1 is for swapping Targets to Zeros and should discount the
↪   effective interest
        require(_g1 <= FixedPoint.ONE, "INVALID_G1");
        // g2 is for swapping Zeros to Target and should mark the effective
↪   interest up
        require(_g2 >= FixedPoint.ONE, "INVALID_G2");
        ...
    }
}
contract Space is IMinimalSwapInfoPool, BalancerPoolToken {
    constructor(...) BalancerPoolToken(...) {
        ...
        // Set Yieldspace config
        g1 = _g1; // fees are baked into factors `g1` & `g2`,
        // no checks for g1
        g2 = _g2; // see the "Fees" section of the yieldspace paper
        // no checks for g2
    }
}
```

Recommendation: Add validity checks for `g1` and `g2` in `constructor`s of `SpaceFactory` and `Space.sol`.

3.3.14 Check `reserves.length`

Severity: *Low Risk*

Context: Space.sol

Situation: Within the functions `onJoinPool()` and `onExitPool()`, there is no check on the length of reserves. As `onJoinPool()` and `onExitPool()` functions are called from the `Balancer` contracts, it is safer to verify the length. This is true because these contracts are unlikely to be exploited based on no checks on the length of reserves. Whereas, in old versions of Solidity (`0.4.x`), it was possible to send large parameters to an unbound memory array. This could overwrite local variables.

```
function onJoinPool(...,  uint256[] memory reserves, ...) external override
↪  onlyVault(poolId) returns (...) {
    ... // no checks on length of reserves
}
function onExitPool(...,  uint256[] memory reserves, ...) external override
↪  onlyVault(poolId) returns (...) {
    ... // no checks on length of reserves
}
```

Recommendation: Add the following to `onJoinPool()` and `onExitPool()`:

```
require(reserves.length == 2, ...);
```

3.3.15 Pause Functionality Risk

Severity: *Low Risk*

Context: EmergencyStop.sol, Divider.sol#L5

```
import { Pausable } from "@openzeppelin/contracts/security/Pausable.sol";
```

Situation: There are 2 ways to pause the protocol: (1) Via the modifier `whenNotPaused` in `Divider.sol`and (2) Via `stop()` in `EmergencyStop.sol`.

However, other parts of the code, like the `balancer/space` pool, are not pausable.

While protocols do have pause functionality in their balancers contracts, this is included .

Recommendation: Double check the `pause` functionality.

3.3.16 `downscaleUp` Function Does Not Handle `0` Input

Severity: *Low Risk*

Context: Space.sol#L502-525

Situation: The functions `_downscaleUp()` and `_downscaleUpArray()` will not work if `amount == 0`.

If a `safeMath` library is used, then an underflow or a revert will occur.

```
function _downscaleUp(uint256 amount, uint256 scalingFactor) internal returns
↪  (uint256) {
    return 1 + (amount - 1) / scalingFactor;
}
function _downscaleUpArray(uint256[] memory amounts) internal view {
    (uint8 _zeroi, uint8 _targeti) = getIndices();
    amounts[_zeroi] = 1 + (amounts[_zeroi] - 1) / _scalingFactor(true);
    amounts[_targeti] = 1 + (amounts[_targeti] - 1) / _scalingFactor(false);
}
```

Recommendation: Use `divUp` from the Math.sol#L88-96 Balancer Library.

3.3.17 Unsafe Use of `transfer()` and `transferFrom()`

Severity: *Low Risk*

Context: BaseAdapter.sol#L117-137

Situation: The function `flashLoan()` of `BaseAdapter.sol` uses the functions `transfer()` and `transferFrom()`. However, in the rest of source, `safeTransfer()` and `safeTransferFrom()` are used.

```
function flashLoan(...) external onlyPeriphery returns (bool, uint256) {
    require(ERC20(target).transfer(address(receiver), amount),
↪   Errors.FlashTransferFailed);
    ...
    require(ERC20(target).transferFrom(address(receiver), address(this),
↪   amount), Errors.FlashRepayFailed);
    ...
}
```

Recommendation: Using the safe versions is slightly safer and is more consistent. We recommend that the Sense team replace `transfer()` with `safeTransfer()` and replace `transferFrom()` with `safeTransferFrom()`.

Sense: Fixed here.

Spearbit: An addendum here for your consideration: there was a recent hack exploiting a `safeTransfer` implementation that differed from OpenZeppelin's. Technically it was less safe, in that calling it on the zero address, or EOAs, it

would not revert. OZ actually checks that some contract code exists, so removes the possibility that lead to this specific exploit.

I looked at the solmate implementation, it doesn't check for this either, so is technically a 'less safer' implementation like the one that was used leading to said exploit. Also tested similarly and confirmed.

Article regarding the exploit.

Do note other variables were at play leading to the exploit, and using solmate's version itself won't lead you to an exploit. Be aware of this when using it regarding the above and difference from OZ. However, many realizable exploits tend to come from a few holes that can be connected together.

3.3.18 Verify `decimals() <= 18`

Severity: *Low Risk*

Context: `Space.sol#L132-L133`, `Periphery.sol#L443`, `Periphery.sol#L625`

Situation: In a few locations in the code, it is assumed that the number of decimals of a token is smaller than or equal to 18. If the number of decimals happens to be larger than 18, an underflow or a revert will occur.

Note: Balancer also requires a maximum of 18 decimals, see `Balancer Labs v2's WeightedPool2Tokens.sol#L72`:

```
// These factors are always greater than or equal to one: tokens with more than
↪  18 decimals are not supported.
```

Recommendation: As the number of decimals of all used tokens are derived from the number of decimals from the Target token, it is sufficient to verify that the Target tokens has 18 decimals or less.

This can be checked in the function `deploy()` of contract `TokenHandler` in `Divider.sol` on line 672.

3.3.19 Preconditions Of `addSeries()` Unchecked

Severity: *Low Risk*

Context: `PoolManager.sol#L221-224`, `Zero.sol#L81`

Situation: The function `addSeries()` should only be executed when the yield space pool has filled its buffer.

Although retrieving oracle `_price()` will only work once the buffer is sufficiently filled, it is more defensive to explicitly check the preconditions in function `addSeries()`.

```
contract PoolManager is Trust {

    ...
    function addSeries(address adapter, uint48 maturity) external {
        require(sStatus[adapter][maturity] == SeriesStatus.QUEUED,
↪   Errors.SeriesNotQueued);
        ...
    }
}
contract ZeroOracle is PriceOracle, Trust {
    function _price(address zero) internal view returns (uint256) {
        ...
        (, , , , , , uint256 sampleTs) = pool.getSample(1023);
        if (sampleTs == 0) {
            ...
        }
    }
}
```

Recommendation: Consider explicitly checking all preconditions in `addSeries()`.

3.3.20 Contract Names Differ From File Names

Severity: *Low Risk*

Context: LP.sol#L52, Target.sol#L13, Underlying.sol#L13, Zero.sol#L52, IRModel.sol#L8

Situation: Several contracts have a file name that is different from the contract name. In addition, certain solidity tools expect the file name and the contract name to be the same. This can be confusing for developers or others diving into the code-base.

Recommendation: Make the file names and contract names the same.

3.4 Informational Issues:

3.4.1 `zero` Should Be `pool`

Severity: *Informational*

Context: LP.sol

Situation: The function `price()` in contract `LPOracle` references the variable `zero`. This is probably a copy-paste error and should be `pool`, just like in function `_price()`.

```
contract LPOracle is PriceOracle, Trust {
    ...
    function price(address zero) external view override returns (uint256) {
        // zero should be pool
        return _price(zero);
    }
    function _price(address _pool) internal view returns (uint256) {
        ...
    }
}
```

Recommendation: Rename `zero` to `pool`.

3.4.2 Bump `Space` Contracts to Compile with Latest Solidity 0.7.x

Severity: *Informational*

Context: `hardhat.config.js#L110`

Situation: The deployment scripts do not target the latest available minor version for the previous major Solidity version `0.7.0`. They are compiled with `0.7.5` while `0.7.6` is the latest.

It was mentioned this was done to match Balancer's contract versions, however, Balancer's deployed contracts on Ethereum mainnet appear to target `0.7.1`.

Recommendation: Bump to the latest Solidity version `0.7.6` to benefit from a number of bug fixes and underhanded coding that have been disallowed with the scanner.

If an earlier minor Solidity version of `0.7.x` is required, please note so and why.

References: `Vault`: earliest deployment, `Balancer Relayer`: one of their earliest deployments.

3.4.3 Use `pragma abicoder v2`

Severity: *Informational*

Context: `Space.sol#L3`, `SpaceFactory.sol#L3`

Situation: In the latest versions of Solidity 0.7.x, the `ABIEncoderV2` is no longer experimental.

It is best practice *not* to use experimental construction in production code.

Recommendation: Replace the following:

```
pragma experimental ABIEncoderV2;
```

with:

```
pragma abicoder v2;
```

3.4.4 Use Consistent Optimizer Runs for CI And Targeted Deployment

Severity: *Informational*

Context: justfile#L102-130, hardhat.config.js#L103-L111

Situation: The testing scripts utilize a different set of optimizer runs than the deployment scripts. The testing scripts are set to 20 runs, while the `Space` package uses the default at 200 and the other `0.8.11` contracts use 1000 runs.

This is problematic as the tests will essentially be checking bytecode that is different than what will end up being deployed. It can also make it so that the contract's deploy size for the tests meets the 24576 bytes maximum limit set during the spurious dragon fork, but the deployment scripts end up exceeding it, and therefore fail.

Runs does not indicate how many times the optimizer runs for, but how many runs the contract is expected to have and whether to optimize for runtime or deploy size. The 200 default should generally save on both. Setting runs higher or lower, should not affect the speed of the optimizer in a significant manner, so don't try and use lower runs for speed.

The documentation covering the runs parameter is here: Solidity Docs - Optimizer Parameter Runs.

Recommendation: Contracts and their backing tests should ideally use the same runs parameter, just as they should compile with the same version of solidity.

Each contract may have their own specific run parameter set as needed, in the case of an often re-deployed contract, but not run as often, lower runs may

make sense, while one that will be used many times may want to use higher runs.

3.4.5 Ensure CI Runs Tests For All Packages Of `monorepo`

Severity: *Informational*

Context: `package.json#L12-19`, `ci.yml`, CI for PR #153

Situation: While referencing the CI scripts to help with building the project, it was discovered the CI only runs on the `v1-core` portion of the project and ignores the others like `v1-fuse` and `v1-space`.

This is the case for the commit hash audited and latest commit as of `b888eda`.

Recommendation: Ensure the CI runs on all portions of the `monorepo`, as otherwise bugs may slip through if the team thinks the CI is testing the other packages, but in fact it is not.

3.4.6 Check Function Parameters To Be Non-Zero

Severity: *Informational*

Context: `setPeriphery()` in `Divider.sol#L487-492`

```
/// @notice Set periphery's contract
/// @param _periphery Target address
function setPeriphery(address _periphery) external requiresTrust {
    periphery = _periphery;
    emit PeripheryChanged(_periphery);
}
```

Situation: In several functions, no checks are done to verify the supplied parameters are non-zero.

Recommendation: Add zero checks to the `setPeriphery()`.

Sense: We decided to omit these checks for governance functions.

3.4.7 Don't Use Hard coded Values

Severity: *Informational*

Context: `Zero.sol#L74-L81`

```
function _price(address zero) internal view returns (uint256) {
    BalancerOracleLike pool = BalancerOracleLike(pools[address(zero)]);
    require(pool != BalancerOracleLike(address(0)), "Zero must have a pool
 ↪  set");
    ...
    (, , , , , , uint256 sampleTs) = pool.getSample(1023);
    ...
}
```

Situation: The function `_price()` in `Zero.sol` uses a hard code value of `1023`. Even though Balancer's `WeightedPool2Tokens` also states `1023`, it is not recommended to rely on hard coded values.

The value `1023` is equal to `Buffer.SIZE` in `Buffer.sol#L20`. This `Buffer.SIZE` can also be retrieved via `getTotalSamples()` in `PoolPriceOracle.sol#L76-78`.

```
function getTotalSamples() external pure override returns (uint256) {
    return Buffer.SIZE;
}
```

Recommendation: *Use the following code to not rely on hard coded values:*

```
(, , , , , , uint256 sampleTs) = pool.getSample(pool.getTotalSamples() - 1);
```

Note: For efficiency purposes, using hard coded values makes sense. However, it is perhaps safer to import that library and evaluate `Buffer.SIZE - 1` as needed in case of changes. This enhances code maintainability and keeps things safer in case of any changes on the Balancer side. If Balancer confirms to the Sense team that they will never end up changing it, at least with the version of Balancer Sense plans to use, they may consider keeping the hard coded approach for gas optimization.

3.4.8 Consider Separating `TokenHandler` Into It's Own Source File

Severity: *Informational*

Context: `Divider.sol#L655-L702`.

Situation: `TokenHandler` is a standalone contract that is also specifically deployed in the deployment scripts, but its source is currently buried at the bottom of the `Divider`'s source file.

Recommendation: In staying consistent with the rest of the project, this contract should be contained within its own source file.

3.4.9 Remove Any Lines With Unused Imports

Severity: *Informational*

Context: CropAdapter.sol#L5, CropAdapter.sol#L9-10

```
import { Periphery } from "../Periphery.sol";
import { Divider } from "../Divider.sol";
```

Situation: There are instances where contracts are imported, but never used by the importing contract in question. This can affect contract readability as you may expect it to have some used functionality or dependence on the import when it does not.

Recommendation: It is best practice to not have dead or unused code, including imports. Remove these occurrences and any other. The noted were quickly identified, but there may be other instances of this the team should explore.

Sense: Addressed here.

3.4.10 Some Functions Can Be Restricted to Pure or View

Severity: *Informational*

Context:

- Space.sol#L342, Space.sol#L389, Space.sol#L434,
- Space.sol#L497, Space.sol#L502

Situation: Declaring a function appropriately as `view` or `pure` helps clarify intent of the function to the end-reader. The more restricted a function is, the "safer" it can be considered and the intent of state modification/access become clearer. A number of functions that could be restricted, are not.

Recommendation: Declare the most restrictive `view`/`pure` specifier available for a function and only leave it out if that function is planned to have certain state modifications/access.

Sense.finance: Addressed here.

3.4.11 Ensure Comments Match Actual Code Logic

Severity: *Informational*

Context:

- GClaimManager.sol#L106, Space.sol#L433, BaseAdapter.sol#L67-69,
- Divider.sol#L538, Periphery.sol#L129

Situation: There are multiple instances where comments in the submitted code do not match the underlying specifications or functionality of the code block they refer to. This hurts code readability and is not best practice.

e.g. The following comment is not accurate:

```
/// @dev This can't be a view function b/c `Adapter.scale` is not a view
 ↪  function
```

`Adapter.scale` is never accessed, and function is indeed restrictable to `view`.

```
/// @notice The number this function returns will be used to determine its
 ↪  access by checking for binary
/// digits using the following scheme:
 ↪  <onRedeemZero(y/n)><collect(y/n)><combine(y/n)><issue(y/n)>
/// (e.g. 0101 enables `collect` and `issue`, but not `combine`)
```

The `0101` would in fact enable just `init` and `combine`, while restricting `issue` and `collect` along with the other parts of `Level` that are undefined here. `1` indicates restrict rather than enable as stated here.

```
// Determine where the stake should go depending on where we are relative to
 ↪  the maturity date
address stakeDst = adapters[adapter] ? cup : series[adapter][maturity].sponsor;
```

The related code never checks maturity in question, but the status of the adapter, which provides no measures of time, as the adapter can be disabled at any point.

```
uint256 tBal = Adapter(adapter).wrapUnderlying(uBal); // convert target to
 ↪  underlying
```

The code logic does the opposite of what is stated and is likely repeated copy/paste of similar lines and remnant of comment. It actually wraps underlying to target and comment should state that.

Recommendation: Keep comments updated and consistent alongside code logic.

3.4.12 Better Define SPONSOR_WINDOW or Document Difference From Other Window with Comments

Severity: *Informational*

Context:

- Divider.sol#L29, Divider.sol#L30,
- Divider.sol#L564, Divider.sol#L566.

Situation: A variable named SETTLEMENT_WINDOW provides a window equivalent to its set time of 2 hours for anyone to settle after some preconditions.

SPONSOR_WINDOW on the other hand, provides a window before and after some preconditions, effectively doubling it's set time from 4 hours to 8 hours, which is inconsistent with how SETTLEMENT_WINDOW is applied.

The intent of this is not particularly clear and could be misread as a mistake by consulting the code alone. There were tests within this project confirming this was intended.

Recommendation: We recommend changing the variable names (i.e. PRE_SPONSOR_WINDOW and POST_SPONSOR_WINDOW) to make the intent clear that one should apply before and the other after a sponsor window. This way their effective windows can be considered cumulatively.

In the case that this is not preferable, we recommend that the Sense team at least make it clear with comments, in both cases, of when and how these variables are applied, as they are applied in different circumstances.

3.4.13 Remove Or Address Unused Variables

Severity: *Informational*

Context: Periphery.sol#L32

```
uint32 public constant TWAP_PERIOD = 10 minutes;
// ideal TWAP interval^
```

LP.sol#L57

```
uint32 public constant TWAP_PERIOD = 1 hours;
```

Situation: The unused variables could indicate a developer error with either introducing unused variables or their accompanying logic being missing and or

unfinished. It may introduce unnecessary deployment or run cost overhead.

Recommendation: Remove the unused variables, if erroneously included and not intended to be used. Alternatively, supplement with appropriate backing logic that uses the variable.

Sense: Fixed in #155.

3.4.14 Document All Function Parameters And Return Values

Severity: *Informational*

Context: `Periphery.sol#L253-265`

Situation: Most of the time, function parameters are documented with `Natspec`.

The return parameters are documented far less frequent, especially with unnamed return values, like in `addLiquidityFromUnderlying()`. This makes the code more difficult to read.

Note: With the later versions of Solidity, you can also document multiple return values with NatSpec.

```
function addLiquidityFromUnderlying(...) external returns (
    uint256,
    uint256,
    uint256
) {
    ...
}
```

Recommendation: Document all the function parameters and return values with `Natspec`.

3.4.15 Simplify Token Ordering Code In `Space.sol`

Severity: *Informational*

Context: `Space.sol`

Situation: The contract `Space` has a lot of code to manage the order of the tokens in an array. This ordering is necessary because Balancer requires the tokens to be in a specific order: sorted by token address.

Recommendation: Use the same way to represent difference between the tokens, e.g. either `boolean` or `uint8`.

The following snippets could be used:

```
tokens[1- _zeroi] = IERC20(target);
```

```
function getIndices() public view returns (uint8 _zeroi, uint8 _targeti) {
    _zeroi = zeroi;
    _targeti  = 1 - _zeroi;
}
```

This can be used if `scalingFactor()` used `uint8` as a parameter.

```
function _upscaleArray(uint256[] memory amounts) internal view {
    amounts[0] *= scalingFactor(0);
    amounts[1] *= scalingFactor(1);
}
```

Consider using a fixed token order.

As the address for the zero token is generated by `deploy()` in contract `TokenHandler` in `Divider.sol#L682-L689`, it is possible to do this via `CREATE2`.

That way you can generate an address for token `zero` such that it is smaller than `target`.

With a `require(zero < target)`, you can enforce this. Trying out different salts for `CREATE2` can be done on-chain for a normal address of target, with less than 10 tries you should be able to find an address that is smaller than target. With `computeAddress()` of OpenZeppelin `Create2.sol#L57-64`, you can calculate the address without deploying.

However, if target happens to be a vanity address with a lot of leading zeros, this will not work. In that case, the Sense team should generated an appropriate salt and pass that via `sponsorSeries()` of `Periphery.sol#L66-76` to `initSeries()` of `Divider.sol#L117-130`. A tool to generate a vanity address can be found here: `ERADICATE2`.

This way the sort order is fixed and the code of `Space` can be less complicated.

3.4.16 Unexpected Functionality of `_swapTargetForClaims()`

Severity: *Informational*

Context: `Periphery.sol#L414-431`

Situation: The function `_swapTargetForClaims()` tries to swap `target` tokens for `claim` tokens. However, this cannot be done in one step.

First, it issues new `zero` and `claim` tokens and moves the supplied target to the adapter. Second, it swaps the (unwanted) `zero` tokens to `target` tokens. Third, it returns the resulting `claim` and `target` tokens to the caller of the function. From a functional point of view this is unexpected. The caller receives `target` tokens back, which is what the caller started with.

This will make the logic from the caller side more complicated.

```
function _swapTargetForClaims(...) internal returns (uint256) {
    ...
    // transfer claims & target to user
    ERC20(Adapter(adapter).target()).safeTransfer(msg.sender, tBal);
    ...
}
```

Recommendation: Document this feature of the function in a comment.

Alternatively, consider using a flashloan to loan extra `target` and return the extra `target` at the end of the flashloan.

Sense: We tried using a flashloan, but ended up doing it this way.

3.4.17 Keep Build Instructions And Scripts Up To Date

Severity: *Informational*

Context: `README.md`

Situation: The provided instructions within the README were not sufficient to successfully build the project. The instructions advises you to install `dapptools`, however, the project no longer builds with `dapptools` due to a change in Solidity `0.8.8`. This change in `dapptools` requires remappings not automatically whitelisted any longer; they need to be explicitly set in the `--allow-path` flag.

The project at the provided commit is set to build with `foundry/forge` whose installation instructions were missing and requires a specific commit installation to work.

There are still a number of project scripts with references to `dapptools` that fail, such as `build` and `test`.

Recommendation: Remove any project scripts no longer supported or working, and provide working installation instructions on how to successfully build with the existing commit within the `README`.

3.5 Gas Optimizations

3.5.1 Optimize `Levels` Library

Severity: *Gas Optimization*

Context: `Levels.sol`

Situation: This contract has optimization potential. By replacing the expressions throughout the code with hardcoded values, the compiler will be able to evaluate some of them at build-time.

Recommendation: Instead of calculating `BIT` on each run, simply set the bit to the correct decimal representation, in the case of `2**3`:

```
uint256 private constant _COLLECT_BIT = 8;
```

Refactor the corresponding function; remove the unnecessary exponentiation which can become costly:

```
function collectDisabled(uint256 level) internal pure returns (bool) {
    return level & _COLLECT_BIT != _COLLECT_BIT;
}
```

Repeat this for the other `BIT` variables and their functions. This helps the compiler omit unnecessary runtime computation through constant folding.

Initial profiling on Remix indicates potential gas savings of `272` to `521` gas with optimizer set on 200 runs (greater savings on higher order bits).

3.5.2 Determine Usage Of `WETH` Once

Severity: *Gas Optimization*

Context: `BaseAdapter.sol`, `CropAdapter.sol`, `CAdapter.sol`

Situation: Within the contract `CAdapter`, `_isCETH(_target)` / `_isCETH(target)` is used several times.

As `target` is stored in an `immutable` variable, it is also possible to store `_isCETH(target)` in an `immutable` variable.

This will save gas and simplify the code.

```
abstract contract BaseAdapter {
    address public immutable target;
    constructor(..., address _target, ...) {
        ...
        target = _target;
    }
}

abstract contract CropAdapter is BaseAdapter {
    constructor(...,  address _target, ...) BaseAdapter(..., _target, ...) { }
}
contract CAdapter is CropAdapter {
    constructor(..., address _target, ...) CropAdapter(..., _target, ...) { ...
↪   }
}
```

Occurrences of `isCETH()` in `CAdapter.sol`:

- CAdapter.sol#L90, CAdapter.sol#L119, CAdapter.sol#L123,
- CAdapter.sol#L127, CAdapter.sol#L151, CAdapter.sol#L172.

Additionally, `_isCETH()` is used to differentiate between `WETH` and the `underlying()` token in the following CAdapter.sol#L90 and CAdapter.sol#L119:

```
ERC20 u = ERC20(_isCETH(_target) ? WETH :
↪   CTokenInterface(_target).underlying());
```

```
return _isCETH(target) ? WETH : CTokenInterface(target).underlying();
```

This result can also be stored in an `immutable` variable to save some gas and simplify the code.

Recommendation: Store `_isCETH(target)` in an `immutable` variable and store the result of

```
ERC20(_isCETH(_target) ? WETH : CTokenInterface(_target).underlying())
```

in an `immutable` variable.

Sense: Fixed here.

3.5.3 Redundant Calls to `setPermissionless()`

Severity: *Gas Optimization*

Context: EmergencyStop.sol#L18-25

```
function stop(address[] memory adapters) external virtual requiresTrust {
    Divider(divider).setPermissionless(false);
    for (uint256 i = 0; i < adapters.length; i++) {
        Divider(divider).setPermissionless(false);
        Divider(divider).setAdapter(adapters[i], false);
        emit Stopped(adapters[i]);
    }
}
```

Situation: The function `stop()` is calling `setPermissionless(false)` multiple times. Calling it once should be enough.

Recommendation: Remove the redundant call to `setPermissionless(false)` in the `for` loop.

Sense: Addressed in #155.

3.5.4 Save with `safeTransferFrom` in `sponsorSeries`

Severity: *Gas Optimization*

Context: Divider.sol#L117-150, Periphery.sol#L66-L76

Situation: The function `sponsorSeries()` transfers stake tokens to the Periphery contract, then calls `initSeries`, which transfers these same tokens to the adapter.

Note that `initSeries` uses the modifier `onlyPeriphery`, so it can only be called from the `Periphery` contract.

This could also be done in one step, which saves some gas.

```
contract Periphery is Trust {
    ...
    function sponsorSeries(address adapter, uint48 maturity) external returns
↪   (address zero, address claim) {
        (, address stake, uint256 stakeSize) =
↪   Adapter(adapter).getStakeAndTarget();
        ...
        // Transfer stakeSize from sponsor into this contract
        uint256 stakeDecimals = ERC20(stake).decimals();
        ERC20(stake).safeTransferFrom(msg.sender, address(this),
↪   _convertToBase(stakeSize, stakeDecimals));
        // Approve divider to withdraw stake assets
        ERC20(stake).safeApprove(address(divider), stakeSize);
        (zero, claim) = divider.initSeries(adapter, maturity, msg.sender);
        ...
    }
}
contract Divider is Trust, ReentrancyGuard, Pausable {
    ...
    function initSeries(address adapter, ... ) external onlyPeriphery
↪   whenNotPaused returns (address zero, address claim) {
        ...
        (address target, address stake, uint256 stakeSize) =
↪   Adapter(adapter).getStakeAndTarget();
        ...
        ERC20(stake).safeTransferFrom(msg.sender, adapter, stakeSize);
        ...
    }
}
```

Recommendation: Consider moving the stake tokens directly to the adapter. Be careful to include all necessary checks.

Sense: We may be in favour of this change, but it "mixes" things. The `divider` is supposed to charge that stake from whoever calls `initSeries`.

3.5.5 Balancer Tokens Are Already Sorted

Severity: *Gas Optimization*

Context: Zero.sol#L74-L110

Situation: Balancer requires tokens to be sorted, so the result of `getPoolTokens()` is also sorted. This means Sense does not have to discover which token is the `zero` token.

```
function _price(address zero) internal view returns (uint256) {
    ...
    (ERC20[] memory tokens, , ) =
↪   BalancerVault(pool.getVault()).getPoolTokens(pool.getPoolId());
    address underlying;
    // tokens[] is sorted, so this check can be optimized
    if (address(zero) == address(tokens[0])) {
        underlying = address(tokens[1]);
    } else {
        underlying = address(tokens[0]);
    }
    ...
}
```

Recommendation: Sense could use the following construction, which is in line with the rest of the code:

```
// change this line to include _targeti
(uint8 _zeroi, uint8 _targeti) = getIndices();
underlying = address(tokens[_targeti]);
```

3.5.6 Use Custom Errors

Severity: *Gas Optimization*

Context: `Errors.sol`

Situation: Strings are used to encode error messages. With the current Solidity versions, it is possible to replace them with custom errors, which are more gas efficient.

Most errors are derived from `Errors.sol`, but several error messages are also hardcoded. See the examples below:

- `CAdapter.sol#L139`, `CAdapter.sol#L157`, `CAdapter.sol#L164`,
- `SpaceFactory.sol#L66`, `SpaceFactory.sol#L81`, `SpaceFactory.sol#L83`,
- `Target.sol#L37`, `Underlying.sol#L35`, `WstETHAdapter.sol#L139`,
- `Zero.sol#L76`, `Zero.sol#L84`, `Space.sol#L160`,
- `Space.sol#L424`, `PoolManager.sol#L290`

Recommendation: Implement custom errors as explained in the Solidity Language Blog: Custom Errors Explainer.

Sense: Fixed here.

3.5.7 Use Full-Sized Types For Minimal Gas Cost Overhead On `immutables` or `constants`

Severity: *Gas Optimization*

Context: `Space.sol`, `BaseAdapter.sol`, `Claim.sol`, `BaseFactory.sol`

Situation: `Immutable/Constant` variables do not yield any gas savings normally appropriated to smaller bit-sized types. This is because they do not use storage slots and, therefore, do not benefit from tight packing storage slots. Instead, they are more likely to introduce increased gas costs for end-users. These increased costs are due to the overhead of the EVM working on smaller bytes. The EVM itself is designed to operate on 32 bytes at a time. It introduces more instructions to appropriately handle these smaller types.

Recommendation: Unless utilizing the ability of the smaller types to wrap for some algorithmic purpose or other external requirements/constraints, you should use the fully-sized respective type available for highest efficiency when it comes to `immutables/constants`.

3.5.8 Getter for Only `Zero` And `Claim`

Severity: *Gas Optimization*

Context:

- `PoolManager.sol#L206`, `PoolManager.sol#L226`, `SpaceFactory.sol#L68`,
- `GClaimManager.sol#L43`, `GClaimManager.sol#L82`, `GClaimManager.sol#L114`,
- `Periphery.sol#L395`, `Periphery.sol#L407`, `Periphery.sol#L559`,
- `Divider.sol#L70-L96`

Situation: On several locations, the addresses for `zero` and/or `claim` are retrieved from the `Divider` contract using the `getter` function for the `series` array.

This uses a relative large amount of gas as 9 parameters are retrieved of which only 2 are used.

This also depends on the order of the elements in struct `Series`.

For reference, consider the following snippet in `Periphery.sol#L395`, `Periphery.sol#L407`, and `Periphery.sol#L559`:

```
(address zero, , , , , , , , ) = Divider(divider).series(adapter, maturity);
```

Recommendation: Create a function to only retrieve the `zero` and `claim` address from the contract `Divider`, i.e. have different functions for retrieving `zero` and `claim`.

Sense: Fixed here.

3.5.9 Consolidate Mappings Accessed by Same Key Into Struct

Severity: *Gas Optimization*

Context:

- `Divider.sol#L58`, `Divider.sol#L73`, `Divider.sol#L67`,
- `Divider.sol#L70`, `PoolManager.sol#L94-98`

Situation: Multiple mappings are accessed by the same key in both `Divider.sol` and `PoolManager.sol`.

Recommendation: Consolidating the multiple `mappings` into a `struct` could yield gains in code readability and gas optimization.

Gas optimization could arise from packing smaller types into a single storage slot, thereby benefiting from improved warm storage access. This also potentially benefits the code-base from singular dirty slots for multiple variables, potentially saving significant gas costs by avoiding repeated clean slot writes.

In the `Divider.sol` case, `AdapterID`s appear as a very viable candidate to be packed, if it's type size is reduced alongside the `bool` variable that identifies whether an adapter is enabled.

In the `PoolManager.sol` case, the `SeriesStatus enum` and `address` could be packed into a single storage slot with the use of a `struct`.

The team should research and consider if there are sufficient amount of variables that could be lowered in type size and packed via a `struct` in the appropriate order.

Sense: Addressed in #155

Spearbit: We advice to also look into this issue in `PoolManager.sol` with the optimization ideas appended to it.

3.5.10 Lower `issuance` And `tilt` In `Divider.sol` To `uint46` / `uint96`

Severity: *Gas Optimization*

Context: `Divider.sol#177`

Situation: This struct contains 3 address elements, which are less than 32 bytes. It does not end up using a whole storage slot.

The `issuance` is a timestamp-based entity, and, therefore, could likely be lowered to `uint48` at least. It could fit into one of the commonly accessed addresses.

The `tilt` may be lowered as well and is currently packed with `issuance`. However, if it can be lowered to a `uint96`, it could pack into another `address` element, thereby saving an additional storage slot. If it could be lowered to `uint48`, both `issuance` and `tilt` could be packed in the same slot as the `address` variable most accessed alongside them, yielding gas benefits from warm access and dirty slot sharing.

Recommendation: The Sense team should explore if both can be *safely* lowered for their logic, if they wish to realize these gains. The gains will only come in the case of them being able to go down to `uint48` / `uint96` at least respectively.

3.5.11 Redundant `fdivUp`

Severity: *Gas Optimization*

Context: `GClaimManager.sol#L109-L130`

Situation: In the function, `excess()` does a `fdivUp`, with a multiplication by `FixedMath.WAD` followed by a division by `10**18`.

As `FixedMath.WAD == 10**18`, this does not do anything.

```
function excess(...) public returns (uint256 tBal) {
    ...
    if (scale - initScale > 0) {
        tBal = ((uBal.fmul(scale, FixedMath.WAD)).fdiv(scale - initScale,
↪   FixedMath.WAD)).fdivUp(
                10**18,                    // same as FixedMath.WAD
                FixedMath.WAD
        );
    }
}
```

Recommendation: Spearbit recommends checking if the above call to `fdivUp()` can be removed.

Note: `GClaimManager` will most likely be deprecated. This issue is included for completeness.