yAudit Exit10 Review

Review Resources:

- Protocol documentation

Auditors:

- spalen
- engn33r

Guest Auditor:

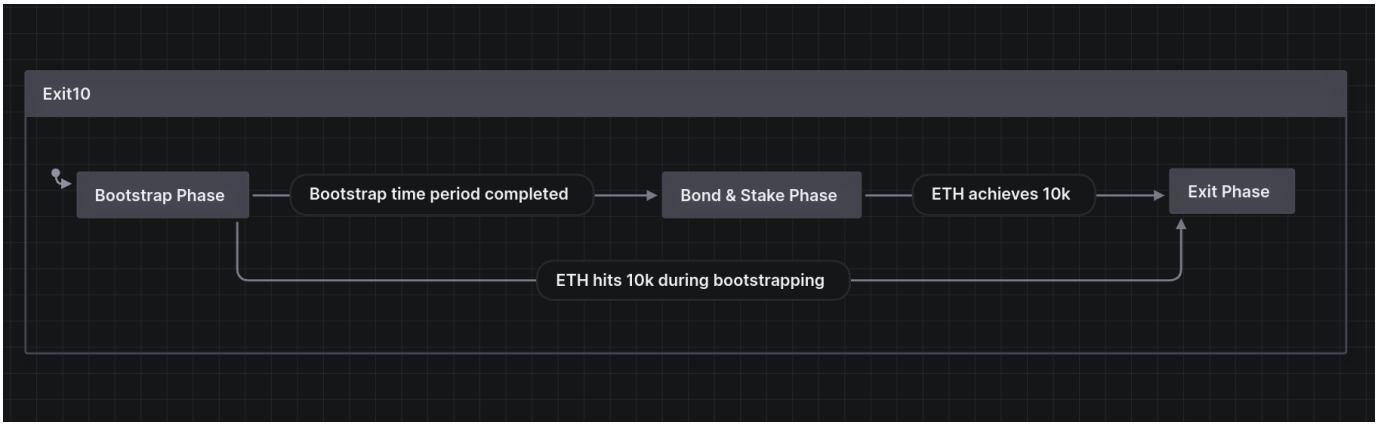
- securerodd (AfterDark Labs)

Table of Contents

Review Summary

Exit10

Exit10 provides a novel and gamified approach for users to gain exposure to and stack ETH as it increases to the price of $10K. This is done by depositing user funds into the WETH/USDC Uniswap v3 pool and holding the liquidity position in Exit10 to stack ETH over time. The fees received from the Uniswap v3 LP position fees are converted to WETH and stored in MasterChef rewards contracts to provide greater exposure to ETH price upside. The protocol will be deployed on Ethereum mainnet, Optimism, and Arbitrum.



The contracts of the Exit10 Repo were reviewed over 14 days. The code review was performed by 2 auditors between April 17 and April 30, 2023. Fellows from yAudit Block 5 additionally joined the review. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was

commit 0b3c2782c5a93d2218234bc70fee31ec32f9e337 for the Exit10 repo. One file from the uniswap-v3-swapper repository at commit 4a3d2c9af49285b0e5cbdbe819f264c13241d017 was also in scope.

Scope

The scope of the review consisted of the following contracts at the specific commit:

- All contracts in the primary exit10-protocol repository with the exception of:
  - Contracts related to STO token distribution (STOToken.sol, MerkleDistributor.sol)
  - Contracts implementing custom ERC20 and ERC721 tokens (BaseToken.sol, NFT.sol)
- Swapper.sol in the uniswap-v3-swapper repository

After the findings were presented to the Exit10 team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Exit10 and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Average	There are some functions that have access control modifiers for certain roles to perform special functions. Modifiers appear to be used properly and are not missing or overused. Some NFT.sol and FeeSplitter.sol call

Category	Mark	Description
		`renounceOwnership()` to remove any special owner privileges.
Mathematics	Good	There is no complex math in Exit10, only basic internal accounting math. Solidity 0.8.X is used so SafeMath is applied throughout.
Complexity	Low	The protocol design relies on some assumptions about the future of DeFi and some assumptions about how users wish to gain ETH exposure. There are design components borrowed from Chicken Bonds and used in a unique way that is arguably more complex that the original Chicken Bonds protocol. The mechanics of the 4 custom ERC20 tokens (BOOT, STO, BLP, EXIT) can be confusing, the accounting between the four liquidity buckets is complex, and the expected end result for early depositors and STO holders receive from accumulated fees is not transparently documented. Moving liquidity between buckets by modifying state variables right before using the updated liquidity values in calculations can cause fairness issues in cases where a cached liquidity value from before the update should be used instead. Some functions like `_getExitAmount()` could use refactoring and renaming.
Libraries	Average	Only OpenZeppelin external libraries were imported to Exit10. Additional code was borrowed from SushiSwap MasterChef and from ChickenBonds, but the borrowed code was also modified in ways that may potentially break assumptions that existed in the original code.
Decentralization	Good	There are no upgradeable proxies, the contracts are immutable, and after deployment the protocol does not have any centralized party that can control it.
Code stability	Average	The code was nearly ready to deploy on mainnet, but some last minute changes before the audit did introduce some bugs.

Category	Mark	Description
Documentation	Low	While there is a documentation website with high level docs, there is a severe lack of NatSpec in the contracts. This makes it difficult to understand developer assumptions and also makes the developer's job more difficult when edits are required.
Monitoring	Average	Events were emitted in all functions where they would be useful, though at least one event emit may not be implemented in the best way possible.
Testing and verification	Average	Tests were running on a fork of Ethereum mainnet, but there were no tests to run on Optimism or Arbitrum despite plans to deploy the Exit10 protocol on all 3 chains.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

Critical Findings

None.

High Findings

1. High - `_getPercentFromTarget()` called with incorrect argument

When `_getPercentFromTarget()` is defined, the input argument name is `_amountBootstrapped`. But when `_getPercentFromTarget()` is called from `_getExitAmount()`, the value passed is not an amount related to bootstrapping but instead is the liquidity that is left in the exit bucket when the user converts their bond to receive EXIT tokens. In other words, the wrong value is passed to `_getPercentFromTarget()`.

Technical Details

`_getPercentFromTarget()` calculates a return value that represents a percent value. The calculation can be summarized as `(x / bootstrapTargetLiquidity) * 100%`. This ratio and the context of this line of coe indicates that `x` should be the current bootstrap liquidity held by the protocol. The current code does not use the current bootstrap liquidity value and therefore needs modification.

Impact

High.

Recommendation

Make this change to `_getExitAmount()` to provide the current bootstrap liquidity as the argument to `_getPercentFromTarget()`.

```
- uint256 percentFromTaget = _getPercentFromTarget(_liquidity) <= 5000 ? 5000 :
_getPercentFromTarget(_liquidity);
+ uint256 percentFromTaget = _getPercentFromTarget(bootstrapBucket) <= 5000 ? 5000 :
_getPercentFromTarget(bootstrapBucket);
```

Developer Response

Fixed in commit 5bf6e76740d862c71a20feccd07ebb66872c27e7

2. High - Incorrect value in `_safeTokenClaim()`

A typo in `_safeTokenClaim()` uses the wrong variable in rewards calculations.

Technical Details

The ratio of rewards that a user can claim to the total rewards supply should be the same as the ratio of tokens that the user holds to the total token supply. This is how the `_safeTokenClaim()` logic works in the first step. The second step involves a correction to

the rewards that a user can claim if the total claimed rewards exceed the total supply of rewards. This is how the logic should work, but this line of code actually checks if the total claimed rewards exceed the total supply of EXIT, BOOT, or other tokens. The total supply of tokens should not be involved in this calculation, the total supply of rewards should be used.

Impact

High. The incorrect variable is used in reward calculations in `_safeTokenClaim()`.

Recommendation

Modify `_safeTokenClaim()` to fix the misused variable:

```
- _claim = (_claimed + _claim <= _supply) ? _claim : _supply - _claimed;
+ _claim = (_claimed + _claim <= _externalSum) ? _claim : _externalSum - _claimed;
```

Additionally, consider renaming the variables where `_safeTokenClaim()` is defined to clarify what each of them represents, for example:

- `_claim` -> `_claimableRewards`
- `_claimed` -> `_previouslyClaimedRewards`
- `_supply` -> `_finalTotalSupply`
- `_externalSum` -> `_rewardsFinalTotalSupply`

Developer Response

Applied the recommended change in L3 where the check is not needed since we are rounding down on every claim. Fixed in commit afc424cbe293201f4665e6c05e11c81359b7aa7

3. High - bootstrapBucket fees are double counted in `claimAndDistributeFees()`

`claimAndDistributeFees()` reinvests bootstrap bucket earned fees into the Uniswap v3 LP position. But the bootstrap bucket liquidity is still counted in the later calculations of `FeeSplitter.collectFees()` which gives the bootstrap bucket more fees than it deserves.

Technical Details

To summarize the fee collection logic that is currently implemented in `claimAndDistributeFees()`, assume:

- `amountCollected0` = 100
- `bootstrapFees0` = 10 (this implies bootstrap liquidity is 10% of the total Exit10 liquidity)
- `amountCollected0` passed to `FeeSplitter.collectFees()` = 90 (100 - 10 = 90)

The amount of fees passed to `FeeSplitter.collectFees()` is 90 (this is the `amountCollected0` value from the previous calculation). This value is then distributed by the ratio of pending bucket's liquidity over the total liquidity (the sum of liquidity of all buckets), which is 70%. If pendingBucket is 70% of liquidity while remainingBuckets are 30% of liquidity, then assuming the original 100 of fees, pendingBucket should get 70% of fees, in this example, 70 worth of fees. But because only 90 worth of fees is passed in `collectFees()`, pendingBucket instead gets 70% of 90, 63.

The result is that pendingBucket is penalized while the bootstrap bucket gets counted twice and effectively gets double fees.

Impact

High. The bootstrap bucket gets approximately double the fees that it should.

Recommendation

`bootstrapBucket` should be removed from the math on this line so that `bootstrapBucket` liquidity is excluded in the calculations that calculate the distribution of the remaining collected fees. The recommendation in the next high finding provides an improvement to fixing this issue individually.

Developer Response

Applied the fix together with H4 Fixed in commit 3d15e441fd5732facdd87b5407c15e0c60f0b303

4. High - Reinvested bootstrap fees are counted as fee-earning liquidity immediately

`claimAndDistributeFees()` reinvests bootstrap bucket fees into the Uniswap v3 LP position. After this liquidity is added, the liquidity will be counted towards the Exit bucket's liquidity. Immediately after the liquidity is added, `_exitBucket()` will use the newly increased Exit bucket liquidity to calculate how fees should be split. But the newly added liquidity did not contribute to generating the fees that will be split, therefore this newly added liquidity will bias the fee distribution unfairly towards the Exit bucket at the expense of other buckets.

Technical Details

`Exit10.claimAndDistributeFees()` collects and distributes the Uniswap LP fees between the different buckets in the Exit10 protocol. This distribution of rewards is calculated using the liquidity share owned by each bucket. The more liquidity that a bucket owns, the more fees that the bucket deserves. But the total Uniswap v3 LP liquidity is increased in this function and the newly added liquidity did not contribute to generating the fees that were just collected. Therefore the value of `Exit10._exitBucket()` is greater than the Exit bucket liquidity that is responsible for generating the collected fees. This means that the Exit bucket will receive a larger share of fees than it should and other buckets will be penalized.

Impact

High. Users expecting to receive rewards from the pending bucket will not receive the expected amount of value.

Recommendation

This recommended fix includes a mitigation to the previous high finding and includes a gas optimization finding.

1. Cache `_liquidityAmount()` in a temporary variable such as `_totalLiquidityBefore` before the bootstrap fees are reinvested into the Uniswap LP position.
2. Replace the 2nd argument passed to `FeeSplitter.collectFees()` with the following:

```
- bootstrapBucket + reserveBucket + _exitBucket(),
+ _totalLiquidityBefore - bootstrapBucket
```

Because the `remainingBuckets` argument in `FeeSplitter.collectFees()` will now equal the total Uniswap v3 LP liquidity before the reinvesting of bootstrap fees and excluding the bootstrap feed, the code in FeeSplitter.sol that handles the total liquidity value can be simplified. The new `remainingBuckets` value includes the `pendingBucket` liquidity, so this line which appears twice (1, 2) in `FeeSplitter.collectFees()` should be modified.

```
- pendingBucket + remainingBuckets,
+ remainingBuckets,
```

Developer Response

Applied the fix together with H3 Fixed in commit

3d15e441fd5732facdd87b5407c15e0c60f0b303

5. High - `_depositAndSwap()` should set non-zero `amountOutMinimum` values

To avoid MEV bot attacks, avoid using a zero value for the Uniswap `amountOutMinimum` values. The `params` variable returned by `_depositAndSwap()` is later used in `_addLiquidity()` to deposit liquidity into Uniswap.

Technical Details

Uniswap v3 docs have this to say about `amountOutMinimum`:

> `amountOutMinimum`: we are setting to zero, but this is a significant risk in production. For a real deployment, this value should be calculated using our SDK or an onchain price oracle - this helps protect against getting an unusually bad price for a trade due to a front running sandwich or another type of price manipulation

Exit10 sets a `amountOutMinimum` value of zero in several places, including in `_depositAndSwap()` and `claimAndDistributeFees()`. There is a risk of value loss due to MEV bots when the minimum value is set to zero.

Impact

High. Poorly chosen `amountOutMinimum` values can lead to loss of value from MEV bots.

Recommendation

Follow the recommendation from the Uniswap docs to use an on-chain price oracle or the value calculated by the SDK. If these options are too gas intensive, consider an approach using a hardcoded acceptable slippage value of roughly 0.3% to at least minimize the MEV losses.

Developer Response

Fixed in commit 4ba3a842a71bd4c9395c48c4f0b83c50aedd3332

6. High - Unconsumed allowance will break fee updates

The FeeSplitter.sol is responsible for redirecting rewards to two of the masterchef staking contracts. When updating rewards for these contracts, it uses Swapper.sol to swap USDC for WETH. There is a chance that during one of these swaps, Swapper.sol will not fully consume the allowance it sets using safeApprove. If this happens, the FeeSplitter will no

longer be able to distribute rewards that it collects.

Technical Details

When the FeeSplitter is updating the rewards for the masterchef contracts, it first swaps USDC for WETH. This in turn calls `swap()` in Swapper.sol. This function uses safeApprove to give the Uniswap V3 Router permission to spend all of the USDC the FeeSplitter has given the Swapper. Importantly, the Swapper also creates a price limit and adds this as parameter in its call to the Uniswap V3 Router.

The problem in this case combines two related truths:

1. safeApprove requires that the previous allowance be completely spent or that it is called with a 0 value. Under any other circumstance it will revert.
2. There is a non-zero chance that the call to the Uniswap V3 Router will not consume the entire allowance set by the call to safeApprove.

Uniswap V3 Router's exactInputSingle function, which is what was called by the Swapper, calls exactInputInternal which itself calls swap directly on the pool with the provided parameters. As mentioned earlier, a price limit was included among these parameters.

The swap function in Uniswap V3 then swaps in steps until it either has used the entire amountIn or it hits the price limit specified. If it hits the price limit specified before it hits the amountIn, there will be an unconsumed allowance and all subsequent calls to the Swapper's swap function will fail.

Impact

High. The system relies on the FeeSplitter to distribute rewards. Once an unconsumed allowance occurs, all fees collected and sent to the FeeSplitter will become inaccessible.

Recommendation

safeApprove is deprecated and not recommended for use in production since OpenZeppelin library version 3.1.0 because of the subtle risks it poses. Consider using safeIncreaseAllowance instead.

If you do wish to continue using safeApprove, ensure that you call safeApprove with a value of 0 each time before you call it with the desired allowance for the swap.

Developer Response

Fixed in commit e41100bcc401e8e0e9d4d417019e9d58b5a69f12

7. High - Exit10.sol ignores token order when sending fees to FeeSplitter.sol

The `claimAndDistributeFees()` function always assumes that the `token0` from the Uniswap pool matches the "token out" and that `token1` matches the "token in".

Technical Details

The call to `collect()` in the Uniswap pool will return collected fees as `amount0` and `amount1` which corresponds to the `token0` and `token1` of the pool.

These amounts are eventually forwarded to the call to `collectFees()` in the FeeSplitter.sol contract. The third and fourth arguments are `amountTokenOut` and `amountTokenIn`, `amountTokenOut` should correspond to the "token out" (USDC) while `amountTokenIn` should correspond to the "token in" (ETH). The order here is important because FeeSplitter.sol use `amountTokenOut` as transfer amount for USDC.

However, `claimAndDistributeFees()` ignores this order and simply sends `amountCollected0` as `amountTokenOut` and `amountCollected1` as `amountTokenIn`.

This will work on Ethereum mainnet as the address of USDC is lower than the address of WETH, meaning `token0` is "token out", but will fail on Optimism where the address of USDC is greater than the address of WETH.

The following test simulates an scenario where `token0` is WETH and `token1` is USDC by forking Optimism. The call to `claimAndDistributeFees()` will revert as FeeSplitter.sol will try to pull more funds than available from the Exit10.sol contract. Full test suite is available here.

```
function test_Exit10_claimAndDistributeFees_IncorrectTokenOrder() public {
    // We are on optimism where the token0/1 order in inverse to mainnet
    setUpOptimism();

    // Skip bootstrap phase
    _skipBootstrap();

    // Create a bond to provide some liquidity
    weth.approve(address(exit10), type(uint256).max);
    usdc.approve(address(exit10), type(uint256).max);
```

```
    deal(address(weth), address(this), 10 ether);
    deal(address(usdc), address(this), 10_000e6);
    _createBond(10 ether, 10_000e6);

    // Do some swaps to generate fees
    weth.approve(UNISWAP_V3_ROUTER, type(uint256).max);
    usdc.approve(UNISWAP_V3_ROUTER, type(uint256).max);
    _generateFees(usdc, weth, 1e12);

    // This function will claim fees from the pool and call collectFees() in
FeeSplitter.
    // FeeSplitter will try to pull the tokens from Exit10 but will fail since the
token order is wrong,
    // the call to collectFees() sends the USDC amount as the WETH amount and the
WETH amount as the
    // USDC amount. As these amounts represent different magnitudes, the transaction
will be reverted
    // since the Exit10 contract won't have those balances. For example, if fees are
500 USDC it will
    // try to pull 500 WETH from the Exit10 contract.
    vm.expectRevert("ERC20: transfer amount exceeds balance");
    exit10.claimAndDistributeFees();
}
```

Impact

High. Ignoring the token order in FeeSplitter.sol will cause the function `collectFees()` to revert on some chains.

Recommendation

Add this code before calling `collectFees()` in claimAndDistributeFees():

```
if (POOL.token0() != TOKEN_OUT) {
    uint256 temp = amountCollected0;
    amountCollected0 = amountCollected1;
    amountCollected1 = temp;
```

```
}
```

Developer Response

Fixed in commit d8df0f461cff3a9c6cb2f0876a5a61c25b648202

8. High - Potential loss of collected fees during the call to `claimAndDistributeFees()`

Fees collected during the call to `claimAndDistributeFees()` in Exit10.sol will be stuck in the contract when the call to `increaseLiquidity()` fails.

Technical Details

When `claimAndDistributeFees()` is called, the function will collect fees from the Uniswap pool, which will send the funds to the caller contract. The implementation will then try to compound the proportion corresponding to the bootstrap bucket using `increaseLiquidity()`:

https://github.com/open-bakery/exit10-protocol/blob/0b3c2782c5a93d2218234bc70fee31ec32f9e337/src/Exit10.sol#L439-L457

```
439:           try
440:             INPM(NPM).increaseLiquidity(
441:               INPM.IncreaseLiquidityParams({
442:                 tokenId: positionId,
443:                 amount0Desired: bootstrapFees0,
444:                 amount1Desired: bootstrapFees1,
445:                 amount0Min: 0,
446:                 amount1Min: 0,
447:                 deadline: DEADLINE
448:               })
449:             )
450:           returns (uint128, uint256 amountAdded0, uint256 amountAdded1) {
451:             unchecked {
452:               amountCollected0 -= amountAdded0;
453:               amountCollected1 -= amountAdded1;
454:             }
455:           } catch {
```

```
456:             return;
457:           }
```

The external call is wrapped in a try/catch statement. The main issue is that if the call to `increaseLiquidity()` fails then the `catch` block will simply return from the function, which means the call to `claimAndDistributeFees()` will succeed. However, funds from fees have been already claimed but won't be sent to the FeeSplitter.sol contract.

Additionally, this issue can be exploited by a bad actor using EIP-150 and the "rule of 1/64th". A bad actor can carefully choose the gas limit to make the call to `increaseLiquidity()` fail due out of gas, while still saving some gas in the main context to continue execution. The call to `increaseLiquidity()` will revert due to gas limits, and the `catch` block will be executed thanks to the saved gas.

The following test reproduces the issue. We set up the liquidity and fees, and mock the call to `increaseLiquidity()` to make it revert. The fees will never reach the FeeSplitter.sol and will be stuck in Exit10.sol.

Note: the following test requires a newer version of the Forge Standard Library in order to use `vm.mockCallRevert()`. It can be installed by executing `forge install foundry-rs/forge-std --no-commit`.

```
function test_Exit10_claimAndDistributeFees_LossOfFeesIfIncreaseLiquidityReverts()
public {
    // Generate liquidity and fees
    _bootstrapLock(10_000e6, 1 ether);
    _skipBootstrap();
    _createBond(100_000e6, 10 ether);
    _generateFees(address(token0), address(token1), 1000e6);

    // Assume call to nonfungiblePositionManager.increaseLiquidity reverts
    vm.mockCallRevert(
        nonfungiblePositionManager,
        abi.encodeWithSelector(INPM.increaseLiquidity.selector),
        ""
    );
```

```
    // Call function
    exit10.claimAndDistributeFees();

    // FeeSplitter is empty
    uint256 feesClaimed0 = token0.balanceOf(feeSplitter);
    uint256 feesClaimed1 = token1.balanceOf(feeSplitter);

    assertEq(feesClaimed0, 0);
    assertEq(feesClaimed1, 0);

    // Fees are stuck in exit10 contract
    assertGt(token0.balanceOf(address(exit10)), 0);
    assertGt(token1.balanceOf(address(exit10)), 0);
}
```

In this other test we demonstrate the griefing scenario. Even if the call to `increaseLiquidity()` would succeed, a bad actor can arbitrarily exercise the issue by choosing a gas limit such that the call reverts due out of gas. The entire call to `claimAndDistributeFees` takes about ~360k units of gas, before calling `increaseLiquidity()` the gas cost is a bit more than 150k and the call to `increaseLiquidity()` itself takes a bit more than 50k gas. By choosing 200k units of gas we can get to point of calling `increaseLiquidity()`, but make this call fail due to out of gas since the forwarded gas will be less than 50k.

```
function test_Exit10_claimAndDistributeFees_IntentionalRevert() public {
    // Generate liquidity and fees
    _bootstrapLock(10_000e6, 1 ether);
    _skipBootstrap();
    _createBond(100_000e6, 10 ether);
    _generateFees(address(token0), address(token1), 1000e6);

    // Call function and supply a gas limit such that the call to
"increaseLiquidity()" reverts due to OOG.
    // The function still continues execution since EIP150 will save 1/64 of
```

```
available gas, enough to
    // execute the return in the catch clause.
    exit10.claimAndDistributeFees{gas: 200_000}();

    // FeeSplitter is empty
    uint256 feesClaimed0 = token0.balanceOf(feeSplitter);
    uint256 feesClaimed1 = token1.balanceOf(feeSplitter);

    assertEq(feesClaimed0, 0);
    assertEq(feesClaimed1, 0);

    // Fees are stuck in exit10 contract
    assertGt(token0.balanceOf(address(exit10)), 0);
    assertGt(token1.balanceOf(address(exit10)), 0);
}
```

Impact

High. The collected fees will be lost in the contract. Normally, this would only happen if the call to `increaseLiquidity()` reverts. However, a bad actor can arbitrarily trigger the issue just by choosing the gas limit.

Recommendation

Use a two-step process to claim and re-invest bootstrap fees. 1) Store the fees earned by bootstrappers in some intermediate variables in `collectFees()`

```
        bootstrapFees0 += (bootstrapBucket * amountCollected0) / _liquidityAmount();
        bootstrapFees1 += (bootstrapBucket * amountCollected1) / _liquidityAmount();
```

2) Use another function to reinvest the fees into the Uniswap pool. Merely allowing the function continue execution in the catch clause is not enough, as that would incentivize attackers to intentionally trigger the issue to steal funds away from the bootstrap reserve.

```
function investBoostrapFees(AddLiquidity memory params) public {
    require(params.amount0Desired == boostrapFees0, "Invalid amount0Desired");
    require(params.amount1Desired == boostrapFees1, "Invalid amount1Desired");
```

```
    (, , uint256 amountAdded0, uint256 amountAdded1) = _addLiquidity(params);

    boostrapFees0 -= amountAdded0;
    boostrapFees1 -= amountAdded1;
}
```

We can adopt a similar approach as the MasterChef contract and reinvest liquidity whenever we withdraw from the MasterchefExit. As the rewards from the bootstrap reserve are directed towards the EXIT bucket, EXIT token holders are incentivized to reinvest their funds.

```
function withdraw(uint256 pid, uint256 amount,AddLiquidity memory params) external {
  withdraw(pid, amount);
  Exit10(owner()).investBoostrapFees(params);
}
```

Developer Response

Fixed in commit 71527c5ec2b1f186fd348bada65e0fd78ceb9bd9

Medium Findings

1. Medium - Using tick as price proxy is slightly inaccurate

The goal of Exit10 is to run the protocol until the price of ETH is $10k or greater, at which point `Exit10.exit10()` is called and the liquidity is withdrawn from Uniswap to distribute to users. In contrast to the stated goals, `_requireOutOfTickRange()` relies on the Uniswap pool tick and not the price provided by the pool. Because the ticks are located at discrete intervals, they are less precise than using the price value from the Uniswap TWAP. This can result in ETH reaching a price of $10k but the protocol won't enter exit state and withdrawals will still not be possible.

Technical Details

`_requireOutOfTickRange()` is the key function that determines whether the price of ETH has reached a price above $10k and whether the Exit10 protocol's mission is complete. This function works by checking the WETH/USDC tick value and determining whether it is in

the tick range or not.

The tick values set in the environment variables are `LOWER_TICK=184210` and `UPPER_TICK=214170`. We can do some math to check what ETH prices this corresponds to.

```
// Step 1. calculate 1.0001^tickValue
// Step 2. multiply by 10^-12 because of the USDC/WETH decimals conversion
// Step 3. invert the value because of the order of WETH and USDC addresses
1 / ((1.0001**184210)  * (10 ** -12)) = 10006.019136330182
1 / ((1.0001**214170)  * (10 ** -12)) = 500.24195658614434
```

The chosen tick value that is used for the lower tick corresponds to a price of roughly $10006, which means that if the price of ETH only reaches $10005 before dropping below $10k, the `exit10()` function cannot be called. A tick value of 184216 or relying on `slot0.sqrtPriceX96` instead of `slot0.tick` would be more precise, but using one of these values would then break the stated goal of holding only USDC when ETH arrives ta $10k because liquidity can only be removed at ticket intervals of 10 ticks. In summary, the protocol has 2 conflicting goals that are slightly mismatched: triggering `exit10()` when the price of ETH rises above $10k and holding only USDC at the same time that `exit10()` is called.

One possible improvement for price data that would also protect against the situation of a USDC depeg event causing the price of ETH to appear to rise above $10k is to use Chainlink or another decentralized oracle solution for price data. Even if Chainlink is not the primary source of price data, it would be a useful secondary data source to validate that the Uniswap TWAP price is correct and that the price of ETH has indeed reached above 10k to allow `exit10()` to be called.

Impact

Medium. Using tick values as a proxy for the price is less precise than using the Uniswap TWAP price data directly and can result in the protocol not functioning as designed.

Recommendation

Clarify the protocol goals and the accuracy of achieving those goals in the docs. While Exit10006 might not be a catchy name, such clarification would be useful to include in the docs.

Developer Response

Acknowledged, the contract will stay as is because the protocol requires all liquidity to be USDC in the pool once ETH reaches 10K.

2. Medium - Incorrect `EXIT_DISCOUNT` values

The .env environment file sets the `EXIT_DISCOUNT` value to 5, which when combined with the `PERCENT_BASE` constant value of 10_000, results in a 0.05% discount on EXIT tokens. This contradicts the documentation which states that the discount is 5%.

Technical Details

The docs page describing the EXIT token describes a 5% discount on the EXIT token. But this value is not used consistently in the code.

The value set in the .env file combined with the constant `PERCENT_BASE` value results in a much lower discount, only 0.05%.

Impact

Medium. The exit discount in the code does not match the docs.

Recommendation

Revise the documentation and the .env file so that the EXIT token discount implementation is consistent with the documentation.

Developer Response

Fixed in commit f15093751c1238b5c42f1351cb74b747d1329344

3. Medium - Problematic MasterchefExit rewards distribution to first EXIT staker

MasterChefExit.sol has a `deposit()` function for users to deposit their reward tokens. This function has issue that could lead to a MEV competition between users resulting in some users losing out on rewards.

Technical Details

There are three problems with `MasterchefExit.deposit()`:

1. Only the first staker to call `deposit()` receives rewards, which can lead to frontrunning of this function.
2. A zero amount is permitted in `deposit()`, meaning the caller does not need to have any

tokens.

3. Because a zero amount is permitted in `deposit()`, `pool.totalStaked` will not increase after `deposit()` is called the first time with a zero amount, so `deposit()` can keep getting called with a zero amount by anyone until there is a non-zero value staked.

Impact

Medium. There is a risk of frontrunning and the logic allows a zero amount.

Recommendation

Add a check to revert `deposit()` when it is called with a zero amount value.

Developer Response

Fixed in commit 0a05a38e2e049aedf1f6bcea1454bd934a46989f

4. Medium - Griefer can force bootstrap depositors to lose all funds if exit10() is triggered during the bootstrap phase

Technical Details

Within the `bootstrapLock(..)` function in the `Exit10` contract, there is no check which prevents a user from calling it even after the `exit10()` function has been called and `inExitMode` has been set to true. The call to `exit10()` will set `bootstrapBucketFinal` to the current bootstrap funds - however an attacker is still able to call `bootstrapLock(..)`, thus increasing the actual size of the bootstrap bucket to be greater than `bootstrapBucketFinal`.

If the attacker deposits an amount equal to `bootstrapBucketFinal`, they will be able to trap all existing bootstrapper's deposits in the contract. To do so, they first call `bootstrapLock(..)` with this amount, and then call `bootstrapClaim()`. This function will call `_safeTokenClaim()` which has the following check which only allows an amount up to `bootstrapBucketFinal` to be claimed: `_claim = (_claimed + _claim <= _supply) ? _claim : _supply - _claimed;`. Thus, the attacker will be able to withdraw their own funds, leaving normal bootstrap depositors locked out of claiming.

Impact

Medium. Griefer can force bootstrap depositors to lose all funds if `exit10()` is triggered during the bootstrap phase.

Recommendation

Add the following check at the start of the `bootstrapLock(..)` function in the `Exit10`

contract: `_requireNoExitMode();`

After this change is made, most likely the line `bootstrapBucket = 0` can be removed from `exit10()` for minor gas savings. Another option is to remove `bootstrapBucketFinal` entirely and replace it with `bootstrapBucket`.

Developer Response

Fixed in commit 4f0450f7727ec08c09bdfb57d45444793b683bc8

5. Medium - Updating fees with zero amount can be used to dilute rewards

The `updateFees()` function can be called with a zero amount to extend the reward period in Masterchef.sol without adding any reward.

Technical Details

The `updateFees()` function present in FeeSplitter.sol is used to swap fees to ETH and feed those as rewards in Masterchef.sol. Doing so recalculates the reward rate to include the new amounts and extends the reward period by the configured rewards duration.

This function can be abused by a malicious actor to dilute the reward process by calling it with a zero amount. This action won't increase the amount of rewards, but will extend the reward period, effectively lowering the reward rate. Current stakers will need to wait an additional period to claim their rewards.

In this test, the issue is triggered each day using a reward duration of 10 days, and produces the following log:

▶ Logs:

Impact

Medium. Technically, stakers won't lose their rewards but a griefer can extend the period indefinitely to dilute the reward process.

Recommendation

Avoid calling updateRewards() in Masterchef.sol if the amounts are zero or are below a certain threshold, so that only meaningful amounts extend the reward period.

Developer Response

Fixed in commit 68e6cc9303c104bcceb4deeb168877292ec0c925

6. Medium - Pool spot price manipulation allows to call `exit10()` before ETH reaches 10K

If a flash loan of sufficient size can be taken, it is possible to manipulate the spot price of the ETH/USDC 500 pool to beyond 10k USDC and trigger the `exit10()` function at will. This concerns in particular the chains where the ETH/USDC 500 pool liquidity is not very deep (e.g. Optimism).

Technical Details

In `Exit10.exit10()` it is checked inside `_requireOutOfTickRange()` whether the price of ETH has surpassed 10k by comparing the current tick to the upper or lower tick of the position range.

```
function _requireOutOfTickRange() internal view {
  if (TOKEN_IN > TOKEN_OUT) {
    require(_currentTick() <= TICK_LOWER, 'EXIT10: Current Tick not below
TICK_LOWER');
  } else {
    require(_currentTick() >= TICK_UPPER, 'EXIT10: Current Tick not above
TICK_UPPER');
  }
}
```

`_currentTick()` is taken from `slot0()`

```
function _currentTick() internal view returns (int24 _tick) {
  (, _tick, , , , , ) = POOL.slot0();
}
```

The tick value returned by `slot0()` is the current tick and moves as the liquidity inside the pool is used up during swaps. If the USDC cumulated liquidity from the current price up to 10k is less than what can be obtained with a flash loan, then it is possible to take a flashloan and make a swap which will push the price beyond 10k.

At this moment on Optimism there is:

- 3.24M of USDC liquidity on UniV3 WETH/USDC 500 pool

- 8M USDC available to borrow on AaveV3

The following PoC shows how a 4M USDC to ETH swap will manipulate the current tick to more than -184210 (corresponds to 10k on Optimism).

```
function testManipulate() public {
    console2.log("block number:", block.number);
    (uint160 sqrtPriceX96Init, int24 tickInit, , , , , ) = pool.slot0();
    // Take USDC from a large holder instead of flashloan for simplicity
    address atck = address(0xEbe80f029b1c02862B9E8a70a7e5317C06F62Cae);
    uint value = 4_000_000 * 1e6;
    vm.startPrank(atck);
    IERC20(pool.token1()).approve(address(router), value);
    ISwapRouter.ExactInputSingleParams memory params =
ISwapRouter.ExactInputSingleParams(
        pool.token1(), // USDC
        pool.token0(), // WETH
        500,
        atck,
        block.timestamp,
        value,
        0,
        0
    );
    router.exactInputSingle(params);
    vm.stopPrank();
    (uint160 sqrtPriceX96Fin, int24 tickFin, , , , , ) = pool.slot0();
    console2.log("tickInit:", tickInit);
    console2.log("tickFin:", tickFin);
}
```

which outputs:

```
Logs:
  block number: 95682224
```

```
  tickInit: -201109
  tickFin: -102138
```

Full PoC file here: https://gist.github.com/bahurum/9daac43a30cd67fe02453a58a52645b5

At any time, an attacker can take a flashloan, swap USDC to ETH, trigger `exit10()` and swap ETH back to USDC on the pool. Note that this may not yield any profit for the attacker (who would have to pay at least for flash loan fees), but the result is that the protocol ends sooner than expected.

Impact

Medium. Attacker can trigger `exit10()` sooner than expected at some cost. This will skip all the phases of the protocol and end the protocol.

Recommendation

In `_requireOutOfTickRange()`, consider checking that the current tick is not too far from the tick at the beginning of the block. This will make `exit10()` revert if the price has been manipulated prior to calling it.

```
function _requireOutOfTickRange() internal view {
+ int24 blockStartTick = OracleLibrary.getBlockStartingTickAndLiquidity(POOL);
+ int24 currentTick = _currentTick();
  if (TOKEN_IN > TOKEN_OUT) {
-   require(_currentTick() <= TICK_LOWER, 'EXIT10: Current Tick not below
TICK_LOWER');
+   require(currentTick <= TICK_LOWER, 'EXIT10: Current Tick not below TICK_LOWER');
  } else {
-   require(_currentTick() >= TICK_UPPER, 'EXIT10: Current Tick not above
TICK_UPPER');
+   require(currentTick >= TICK_UPPER, 'EXIT10: Current Tick not above TICK_UPPER');
  }
+ int24 tickDiff = blockStartTick > currentTick ? blockStartTick - currentTick :
currentTick - blockStartTick;
+ require(tickDiff < 100); // 100 ticks is about 1% in price difference
}
```

Developer Response

Fixed in commit f6978737682f202a7443f6c76ef22461bdcfad73

Low Findings

1. Low - Use `_collect()` return values

When Exit10.sol calls `_collect()`, it ignores the return values in all cases except one. Instead, the return values should be favored for later accounting purposes over the input arguments `amountRemoved0` and `amountRemoved1` that are provided to `_collect()`.

Technical Details

`_decreaseLiquidity()` is called before each `_collect()` call (except one) to return the amount of each token that is passed to `_collect()`. However, after `_collect()` is called, it is assumed that the values passed as arguments to `_collect()` are equivalent to the values that are returned by `_collect()`. Looking at the Uniswap v3 NatSpec for the underlying `NonfungiblePositionManager.collect()`, the arguments are described as indicating the maximum amount of tokens to collect, and the return values from `collect()` indicate the actual value that is returned by the function.

One example of this assumption is in `bootstrapLock()`. `amountRemoved0` and `amountRemoved1` are passed to `_collect()` as arguments and then are subtracted from `amountAdded0` and `amountAdded1`. But because the arguments passed to `_collect()` are the maximum values that could be collected, the return values from `_collect()` should be what is subtracted from `amountAdded0` and `amountAdded1`.

Impact

Low. If the assumption that maxAmount passed to `_collect()` is always the same from the return values of `_collect()` is ever broken in any edge case, the math in Exit10 will be inaccurate.

Recommendation

Rely on the return values from `_collect()` for internal accounting after `_collect()` is called.

Developer Response

Fixed in commit ade41fe76eeddd23864811e84120d6ff4a0c5958

2. Low - No emergency function(s) to handle edge cases

There are no emergency functions to handle unexpected edge cases. Such edge cases include ETH not reaching 10k before the protocol deadline or Uniswap activating the fee switch.

Technical Details

The Exit10 protocol is designed to be immutable, with `renounceOwnership()` calls in NFT.sol and FeeSplitter.sol. This immutability and the lack of emergency functions could result in locked funds due to uncontrollable external events. The funds that are locked until `Exit10.exit10()` is called include bootstrap funds, STO, and EXIT, which can only be withdrawn with `bootstrapClaim()`, `stoClaim()`, `exitClaim()`.

Uniswap fees are currently passed on to liquidity providers. The Uniswap DAO has long discussed the idea of turning on the fee switch in order to direct some of these fees to Uniswap token holders. This change could substantially impact the fees earned by liquidity providers and may therefore change the assumptions in the Exit10 protocol. A similar result could happen if other events, such as a Uniswap hack or a better AMM, causes the ETH-USDC pool to generate substantially lower fees than estimated. Regardless of what actually happens, the assumption that the underlying Uniswap LP positions will generate fees when held in Exit10 depends on external factors not under control of the protocol. If these assumptions change, it may be useful to allow for emergency withdrawal of funds even before ETH reaches 10k.

Another edge case that may occur is if ETH never reaches 10k. While this is a very pessimistic case, external factors like new laws banning certain cryptocurrencies, hacks, hard forks, or other factors could impact the price of ETH and when it might reach 10k. If the price of ETH does not reach 10k before the hardcoded deadline timestamp, which is in the year 2286, the funds will be locked.

Another edge case is if a USDC depeg event happens. This could cause the price of ETH in the Uniswap pool to appear to rise above 10k, but in actuality the price of ETH may only be at a lower value such as $8000.

A final edge case is that one of the Uniswap contracts behind a proxy, such as the Uniswap v3 Router, may undergo a change that is not backwards compatible. While this is unlikely to happen, a security vulnerability or unexpected DAO vote could cause this to happen.

Impact

Low. Unexpected edge cases can cause the protocol to have locked funds or reduced profits.

Recommendation

Consider mitigation strategies to avoid locked funds. This can include adding an emergency function or allowing the owner to override the call to `exit10()` even if the price point is not realized.

Developer Response

Acknowledged. This is a protocol risk and it is mentioned in the documentation.

3. Low - Inelegant rounding solution

The logic in `_safeTokenClaim()` implements a hacky solution to rounding imprecision. Unlike the ERC4626 standard which specifies when rounding up and rounding down should happen when converting between assets and shares for more consistent handling of all user interactions, the conversion between assets and rewards in `_safeTokenClaim()` could penalize late reward withdrawals.

Technical Details

The purpose of `_safeTokenClaim()` is to burn all tokens held by a user and to return the quantity of rewards the user can claim. The calculation of claimable rewards due to the user is a 2-step process.

In the first step, the user's claimable rewards is calculated as the fraction of final totalSupply that msg.sender owns. This calculation is similar to how ERC4626 asset <-> share conversion happens in functions such as `convertToShares()` and `convertToAssets()`. The second step is to modify the value of the user's claimable rewards if the total claimed rewards (including the amount that is claimed in the active transaction) exceeds the total supply of rewards that exist. The fact that this second step exists implies that the protocol could encounter a situation where the outstanding rewards exceed the rewards that the protocol can afford to payout, which means that at least one user owed rewards will not receive their full amount of rewards. The way the logic works could lead to a bank run scenario, where user(s) who are late to redeem their rewards don't receive the full value of their rewards.

While it is not clear under what circumstance this line of logic would be needed, a better

design is one like ERC4626 that manages the rounding up or down appropriately and never reaches a bank run type scenario where some users do not receive the expected amount of value due to rounding error accumulation.

Impact

Low. The exact scenario where rounding errors could be problematic is unclear, but the currently implemented solution is not ideal.

Recommendation

First determine whether this line of logic is needed, and then determine an improved solution that rounds up or down accordingly. Most likely the logic can be borrowed from ERC4626 `redeem()`, where shares (equivalent to reward tokens) are redeemed for underlying (USDC).

Developer Response

Same issue as H2 Fixed in commit afc424cbe293201f4665e6c05e11c81359b7aa71

4. Low - Incentive tokens EXIT, BOOT exposed to frontrunning

There is a capped supply of EXIT and BOOT tokens that are provided to early liquidity providers as additional incentives to deposit into Exit10 early. When the protocol is nearing or already at the capped supply of these incentive tokens, frontrunning can cause users to receive fewer incentive tokens than they were expecting to receive.

Technical Details

There is a limited supply of EXIT tokens. If a user converts a bond with `convertBond()` and `_mintExitCapped()` is called when `mintAmount` evaluates to zero or when the amount of EXIT to be minted updates to `MAX_EXIT_SUPPLY - EXIT.totalSupply()` instead of the original `amount` input argument, then the user may end up receiving less EXIT tokens than they expected (with a worst case of receiving zero EXIT). The process of increasing `EXIT.totalSupply()` to the `MAX_EXIT_SUPPLY` is at risk of being frontrun if a user is converting a bond in the same block as another user.

The same frontrun risk exists with BOOT tokens in `bootstrapLock()`. Once `bootstrapBucket` increases to the value of `BOOTSTRAP_CAP` or near this value, a user calling `bootstrapLock()` may receive less BOOT tokens than expected (with a worst case of receiving zero BOOT). This means that user calls to `bootstrapLock()` (and also calls to `swapAndBootstrapLock()`) could be frontrun with the user receiving less BOOT than expected. The risk of

frontrunning `bootstrapLock()` to claim BOOT is greater than the risk of frontrunning `convertBond()` to claim EXIT, because only users with existing bonds can call `convertBond()`.

Impact

Low. Normal users may get frontrun and lose out on these extra incentives but they will not lose any value from the tokens they choose to invest in the protocol. But because the user never actually owned these incentive tokens in the first place, there is technically no theft of rewards.

Recommendation

Consider recommending users to use MEV protection solution like CowSwap or MEV Blocker if they are concerned about the risk of receiving fewer incentive tokens than expected.

Developer Response

Acknowledged. Will add to the documentation.

5. Low - Incorrect `PROTOCOL_GUILD` address for multichain deployment

`PROTOCOL_GUILD` in Exit10.sol is a constant that is hardcoded to 0xF29Ff96aaEa6C9A1fBa851f74737f3c069d4f1a9. This is the proper address for Ethereum mainnet, but the Protocol Guild does not receive donations at this address on other chains. Exit10 will be deployed to Optimism and Arbitrum, where funds sent to this address will be locked unless the Protocol Guild changes their donation approach in the future.

Technical Details

`PROTOCOL_GUILD` is hardcoded to 0xF29Ff96aaEa6C9A1fBa851f74737f3c069d4f1a9 in this line. But checking the Optimism blockchain scanner and Arbitrum blockchain scanner shows there is no contract deployed to these addresses. Any funds sent to this address on chains other than mainnet will not be useful to the Protocol Guild. The Protocol Guild only takes donations on Ethereum mainnet.

Impact

Low. Donated funds will be locked. These funds did not belong to the users in the first place so it is only the donation funds that are impacted.

Recommendation

Change `PROTOCOL_GUILD` from a constant address to an immutable address that is set in the constructor on deployment. Identify a different address to donate funds to for Optimism and Arbitrum deployments.

Developer Response

Fixed in commit d16600dfcee4755d98942e488c94da701119e444

6. Low - High hardcoded slippage

FeeSplitter.sol has a constant slippage of 1%. This is a high slippage for a deep liquidity pool like USDC/ETH and could be prone to sandwich attacks.

Technical Details

The slippage can be changed to a lower value. It won't cause the protocol to become stuck because FeeSplitter.sol uses function `updateFees(uint256 swapAmountOut)` to swap which has an input parameter. If the liquidity gets too low, and defined slippage is too high, input `swapAmountOut` can be decreased to complete the swaps.

Impact

Low. Too high slippage can result in users receiving less value than expected.

Recommendation

Set slippage to lower value.

Developer Response

Fixed in commit 796c2143a2e43cd7613989e73f497ad52f7fccae

7. Low - Bootstrap rewards are shared

Early bootstrappers generate fees for the protocol that are shared across all bootstrappers.

Technical Details

When a user locks in an amount for bootstrapping, they transfer in USDC and WETH in order to add liquidity to the pool. BOOT tokens are then minted to the user 1:1 for the amount of liquidity they were able to add to the pool. Regardless of when a user locks into bootstrapping, they are minted the same ratio of BOOT as the original entrants.

After the protocol has performed an Exit10 and a user goes to claim their BOOT, they are given USDC based on the ratio of the BOOT they own to the size of the bootstrap bucket.

This means that fees generated during the bootstrap period are distributed to all bootstrappers, penalizing the early bootstrappers.

Impact

Low. Early bootstrappers are sharing fees they generate for the protocol with later bootstrappers. The impact is lowered due to the shortened time frame and upper capacity limits enforced during bootstrapping.

Recommendation

Consider incorporating the fees generated up until that time when calculating the number of BOOT to mint to newer bootstrappers.

Developer Response

Acknowledged. Will add to the documentation.

8. Low - Possible loss of funds with price limited swaps through `DepositHelper`

In `DepositHelper`, if a user calls `swapAndBootstrapLock()` and `swapAndCreateBond()` with `sqrtPriceLimitX96` != 0, the part of `amountIn` that is not swapped is not used for minting liquidity and is left in the contract.

Technical Details

In `DepositHelper._depositAndSwap()` a user can provide `TOKEN_IN` and `TOKEN_OUT` at a certain ratio and make the contract perform a swap with `_swapParams` in order to provide liquidity at a different ratio. Let's say that `_swapParams.tokenIn == TOKEN_0`. The issue is that the contract assumes that `_swapParams.amountIn` is fully used for the swap and computes the amount to provide as liquidity as `_initialAmount0 - _swapParams.amountIn` (DepositHelper.sol#L85-L91).

```
if (_swapParams.tokenIn == TOKEN_0) {
  _initialAmount0 -= _swapParams.amountIn;
  _initialAmount1 += amountOut;
} else {
  _initialAmount1 -= _swapParams.amountIn;
  _initialAmount0 += amountOut;
}
```

Here is a PoC that demonstrates the issue:

```
function testPriceLimitSwaps() public {
  _skipBootstrap();
  uint160 sqrtRatioAX96 = TickMath.getSqrtRatioAtTick(tickLower);
  uint160 sqrtRatioBX96 = TickMath.getSqrtRatioAtTick(tickUpper);
  (uint160 sqrtRatioX96, int24 tick, , , , , ) =
IUniswapV3Pool(vm.envAddress('POOL')).slot0();
  console.log("initial USDC balance of DepositHelper:",
ERC20(usdc).balanceOf(address(depositHelper)));
  console.log("sqrtRatioX96:", sqrtRatioX96);
  (uint256 amount0, uint256 amount1) = LiquidityAmounts.getAmountsForLiquidity(
      sqrtRatioX96, sqrtRatioAX96, sqrtRatioBX96, 1e15); // amounts to obtain 1e15
liquidity
  uint256 swapAmount0 = convert1ToToken0(sqrtRatioX96, amount1, 6);  // usdc amount
to swap
  ERC20(usdc).transfer(alice, amount0 + swapAmount0); // give usdc to alice
  IUniswapV3Router.ExactInputSingleParams memory swapParams = _getSwapParams(
      usdc, weth, swapAmount0);
  swapParams.sqrtPriceLimitX96 = TickMath.getSqrtRatioAtTick(tick - 1);
  console.log("alice sends", amount0 + swapAmount0, "USDC");
  console.log("and swaps", swapAmount0, "USDC for ETH");
  console.log("swapParams.sqrtPriceLimitX96 set to:", swapParams.sqrtPriceLimitX96);
  vm.startPrank(alice);
  ERC20(usdc).approve(address(depositHelper), amount0 + swapAmount0);
  depositHelper.swapAndCreateBond(amount0 + swapAmount0, 0, swapParams);
  vm.stopPrank();
  uint256 usdcLeft = ERC20(usdc).balanceOf(address(depositHelper));
  console.log("Only", swapAmount0 - usdcLeft, "USDC has been swapped and used to mint
bonds");
  console.log("USDC Left into DepositHelper:", usdcLeft);
}
```

which outputs:

```
Logs:
  initial USDC balance of DepositHelper: 0
  sqrtRatioX96: 1980704062856608439838598758400000
  alice sends 41638723701 USDC
  and swaps 24004813136 USDC for ETH
  swapParams.sqrtPriceLimitX96: 19805309121342075146510077392103 16
  Only 5701654427 USDC has been swapped and used to mint bonds
  USDC Left into DepositHelper: 18303158709
```

Full PoC file here: https://gist.github.com/bahurum/96a5a6c2082b81712392924cd2e673fd.

Impact

Low. Part of the swap amount can be lost when `swapAndBootstrapLock()` and `swapAndCreateBond()` are called by a user specifying a price limit for the swap.

Recommendation

When calculating the amounts for providing liquidity, account for the amount of `_swapParams.tokenIn` left into the contract after the swap:

```
if (_swapParams.tokenIn == TOKEN_0) {
- _initialAmount0 -= _swapParams.amountIn;
+ _initialAmount0 = IERC20(TOKEN_0).balanceOf(address(this));
  _initialAmount1 += amountOut;
} else {
- _initialAmount1 -= _swapParams.amountIn;
+ _initialAmount1 = IERC20(TOKEN_1).balanceOf(address(this));
  _initialAmount0 += amountOut;
}
```

Developer Response

Fixed in commit 87257c08584ca001d72b672914bf7b2c4a654b66

Gas Savings Findings

1. Gas - Relying on hardcoded values can save gas

Some logic that relies on unchanging immutable variables can be simplified for gas savings.

Technical Details

The variables `TOKEN_IN` and `POOL` are immutable in Exit10.sol. In UniswapV3Pool.sol the variables `token1` and `token0` are immutable. Because these values are not changing, branching logic like this if statement can be removed and simplified because the same logic path will always be followed because immutable variables cannot change after deployment. The same applies to places where `POOL.token0()` is used, it can be replaced with `TOKEN_IN` or `TOKEN_OUT` to remove an external call.

Impact

Gas savings.

Recommendation

Simplify code for gas savings by hardcoding logic based on immutable values. Be aware that the hardcoded logic would differ depending on the chain that Exit10 is deployed on.

2. Gas - Unnecessary 1e18 decimals multiplication

Reducing the decimals of `bondDuration` and `ACCRUAL_PARAMETER` in `_getAccruedLiquidity()` could save gas when it is called by `convertBond()` by removing a multiplication operation.

Technical Details

`_getAccruedLiquidity()` multiplies 1e18 by the value of `block.timestamp - _params.startTime` so that the resulting `bondDuration` variable has decimals of 1e18. But this is unnecessary because in the next line of code, the presence of `bondDuration` in the numerator and denominator means these decimals will cancel each other out. The code can be simplified and one multiplication operation removed by keeping the original decimals that `block.timestamp` has.

Impact

Gas savings.

Recommendation

Remove unnecessary 1e18 decimals on timestamp-related values. This involves changing the value that the immutable `ACCRUAL_PARAMETER` is set to and editing `_getAccruedLiquidity()` by removing the 1e18 multiplication.

```
- ACCRUAL_PARAMETER = params_.accrualParameter * DECIMAL_PRECISION;
+ ACCRUAL_PARAMETER = params_.accrualParameter;
```

```
- uint256 bondDuration = 1e18 * (block.timestamp - _params.startTime);
+ uint256 bondDuration = block.timestamp - _params.startTime;
accruedAmount = (_params.bondAmount * bondDuration) / (bondDuration +
ACCRUAL_PARAMETER);
```

3. Gas - Pass `_liquidityAmount()` to `collectFees()` for 2nd argument

The only use case for the `remainingBuckets` parameter in `FeeSplitter.collectFees()` is the sum this value with `pendingBucket`. This summation rederives the value of `_liquidityAmount()` but in a less efficient way.

Technical Details

The only place where the `remainingBuckets` argument is used in on lines of code like `pendingBucket + remainingBuckets`. This summation should return the same value as `Exit10._liquidityAmount()`, so pass `_liquidityAmount()` instead of `bootstrapBucket + reserveBucket + _exitBucket()` as the 2nd parameter to `FeeSplitter.collectFees()` to save gas. Even better, cache `_liquidityAmount()` instead of calling the internal function multiple times.

Impact

Gas savings.

Recommendation

Make this change in `Exit10.claimAndDistributeFees()` or cache `_liquidityAmount()` to a local variable at the start of the function:

```
FeeSplitter(FEE_SPLITTER).collectFees(
        pendingBucket,
-        bootstrapBucket + reserveBucket + _exitBucket(),
+        _liquidityAmount(),
        amountCollected0,
        amountCollected1
      );
```

And then make this change in `FeeSplitter.collectFees()` (and consider renaming the 2nd parameter from `remainingBuckets` to `totalLiquidityAmount`).

```
uint256 portionOfPendingBucketTokenIn = _calcPortionOfValue(
           pendingBucket,
-    pendingBucket + remainingBuckets,
+    remainingBuckets,
           amountTokenIn
           );
```

```
uint256 portionOfPendingBucketTokenOut = _calcPortionOfValue(
           pendingBucket,
-          pendingBucket + remainingBuckets,
+          remainingBuckets,
           amountTokenOut
           );
```

4. Gas - Remove function used only once

Some function are used only once and can be removed to save gas without sacrificing readability.

Technical Details

Functions `_getDiscountedExitAmount()` and `_getLiquidityForBootsrapTarget()` are used only once. They can be removed and their code can be inlined to the only place where they are used. This will save gas without sacrificing readability if the variable naming is in line. The same is true of `_transferAmountIn()` in AMasterchefBase, it is only used in `deposit()`.

Impact

Gas savings.

Recommendation

Remove functions `_getDiscountedExitAmount()` and `_getLiquidityForBootsrapTarget()` and inline their code to the only place where they are used.

5. Gas - Remove unused variables

Some state variables are not used and can be removed to save gas.

Technical Details

State variable `FACTORY` is defined and used only in `constructor()` and can be removed.

State variable `bootstrapDeposit` is not used at all and can be removed.

Impact

Gas savings.

Recommendation

Remove unused variables.

6. Gas - Remove unneeded variable

State variable `DEPLOYMENT_TIMESTAMP` can be removed without loss of any data or functionality.

Technical Details

There is no need to store deployment timestamp because this information can be retrieved from deployed block. This value is not used in any function and can be removed.

Small change is required in `Exit10.sol`.

```
- DEPLOYMENT_TIMESTAMP = block.timestamp;
- BOOTSTRAP_PERIOD = params_.bootstrapPeriod;
+ BOOTSTRAP_PERIOD = params_.bootstrapPeriod + block.timestamp;
```

Additional gas saving will be in dropping add operation at L554:

```
- return (block.timestamp < DEPLOYMENT_TIMESTAMP + BOOTSTRAP_PERIOD);
```

```
+ return (block.timestamp < BOOTSTRAP_PERIOD);
```

Impact

Gas savings.

Recommendation

Remove unneeded state variable `DEPLOYMENT_TIMESTAMP`.

7. Gas - Struct packing

The struct `PoolInfo` could benefit from struct packing to place multiple elements into a single slot.

Technical Details

The `allocPoint` and `lastUpdateTime` elements are currently uint256, however, `allocPoint` appears unlikely to exceed 100 and `lastUpdateTime` is only required to be as large as necessary to hold updated block.timestamp values. Given this, it may be reasonable to combine these two with the token address in order to have all three occupy only a single slot in storage.

Impact

Gas savings.

Recommendation

One example configuration may be:

```
struct PoolInfo {
  address token;
  uint32 allocPoint;
  uint64 lastUpdateTime;
  uint256 totalStaked;
  uint256 accRewardPerShare;
  uint256 accUndistributedReward;
}
```

8. Gas - Skip double fetching of the same value

The function `_updatePool(uint256 _pid)` receives a parameter `_pid` which is used to fetch

the same value from the storage. This can be avoided by passing the value directly to the function because basically all functions that call already fetch the value from the storage.

Technical Details

All functions that call `_updatePool()` already fetch the value from the storage, except `_massUpdatePools()` which can be easily changed to fetch the value from the storage. This will save gas on every call of `_updatePool()`.

Impact

Gas savings.

Recommendation

Change function signature to use `PoolInfo` instead of `_pid`:

```
- function _updatePool(uint256 _pid) internal {
+ function _updatePool(PoolInfo storage pool) internal {
```

9. Gas - Use Shift Left instead of Multiplication if possible

A multiplication by any number x, which is a power of 2, can be calculated by shifting log2(x) to the left. MUL opcode uses 5 gas, while the SHL opcode uses 3 gas.

Technical Details

In the file FeeSplitter.sol:125:

```
-uint256 mc0TokenIn = (pendingBucketTokenIn * 4) / 10; // 40%
+uint256 mc0TokenIn = (pendingBucketTokenIn << 2) / 10; // 40%
```

In the file Exit10.sol:608:

```
-_stoRewards = tenPercent * 2;
+_stoRewards = tenPercent << 1;
```

Impact

Gas savings.

Recommendation

Use Shift Left instead of Multiplication when possible.

10. Gas - Cache state variables

State variable that are accessed multiple times can be cached in a local variable to save on gas.

Technical Details

In `exitClaim()` and `stoClaim()` the state variables `EXIT` and `STO` (respectively) are loaded three times, rather than being cached in a local variable.

Impact

Gas savings.

Recommendation

Amend the code as follows: Exit10.sol - https://github.com/open-bakery/exit10-protocol/blob/0b3c2782c5a93d2218234bc70fee31ec32f9e337/src/Exit10.sol#L372

```
function stoClaim() external returns (uint256 claim) {
   BaseToken sto = STO;
   uint256 stoBalance = IERC20(sto).balanceOf(msg.sender);
   claim = _safeTokenClaim(
     sto,
     stoBalance,
     STOToken(address(sto)).MAX_SUPPLY(),
     teamPlusBackersRewards,
     teamPlusBackersRewardsClaimed
   );

   // STO tokens are only for early backers and team
   teamPlusBackersRewardsClaimed += claim;

   _safeTransferToken(TOKEN_OUT, msg.sender, claim);

   emit ClaimRewards(msg.sender, address(sto), stoBalance, claim);
 }
```

Exit10.sol - https://github.com/open-bakery/exit10-protocol/blob/0b3c2782c5a93d2218234bc70fee31ec32f9e337/src/Exit10.sol#L389

```
  function exitClaim() external returns (uint256 claim) {
    BaseToken exit = EXIT;
    uint256 exitBalance = IERC20(exit).balanceOf(msg.sender);
    claim = _safeTokenClaim(exit, exitBalance, exitTokenSupplyFinal,
exitTokenRewardsFinal, exitTokenRewardsClaimed);

    exitTokenRewardsClaimed += claim;

    _safeTransferToken(TOKEN_OUT, msg.sender, claim);

    emit ClaimRewards(msg.sender, address(exit), exitBalance, claim);
  }
```

The gas savings for the two changes above are as follow:

```
test_bootstrapClaim() (gas: -1 (-0.000%))
testScenario_3() (gas: -15 (-0.000%))
testScenario_0() (gas: -21 (-0.000%))
testFuzz_claims(uint256,uint256,uint256,uint256,uint256) (gas: -8 (-0.000%))
testScenario_1() (gas: -20 (-0.000%))
testScenario_2() (gas: -12 (-0.000%))
test_exitClaim() (gas: -8 (-0.000%))
test_stoClaim() (gas: -8 (-0.000%))
test_stoClaim_RevertIf_NotExited() (gas: 6 (0.001%))
test_exitClaim_RevertIf_NotExited() (gas: 6 (0.001%))
Overall gas change: -81 (-0.000%)
```

11. Gas - Redundant check in `cancelBond()`

`cancelBond()` does not need to perform a check on whether the `param.liquidity` parameter is equal to `bond.bondAmount` as long as it is then updated to use `bond.bondAmount`.

Technical Details

When a user cancels their bond, the amount of liquidity that they provided initially is removed from Uniswap, in order to pay them back. The `_decreaseLiquidity()` function requires a parameter of type `RemoveLiquidity`. In order to save gas, it is possible to remove the check `_requireEqualValues` and instead update the `memory param` parameter with `bond.bondAmount`.

Impact

This could lead to small gas savings:

```
test_cancelBond_withMoreBondsInTheSystem() (gas: -57 (-0.003%))
test_cancelBond_claimAndDistributeFees() (gas: -46 (-0.004%))
test_cancelBond() (gas: -46 (-0.006%))
test_cancelBond_RevertIf_StatusIsCanceled() (gas: -45 (-0.006%))
test_convertBond_RevertIf_StatusIsCanceled() (gas: -46 (-0.006%))
Overall gas change: -240 (-0.000%)
```

Recommendation

Update the code as follow to reduce gas usage for this function.

Exit.sol:cancelBond()

```
Line #243
- _requireEqualValues(bond.bondAmount, params.liquidity);
Line #249
+ params.liquidity = uint128(bond.bondAmount);
```

12. Gas - `>=` costs less gas than `>`.

There are 5 instances of this issue.

Technical Details

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>`, but only requires `LT` for `>=`, which saves 3 gas.

File: Exit10-code/exit10-protocol/src/AMasterchefBase.sol

```
215: if (_poolLastUpdateTime > periodFinish) return 0;
```

File: Exit10-code/exit10-protocol/src/Exit10.sol

```
177: if (bootstrapBucket > BOOTSTRAP_CAP) {
482: uint256 liquidityPerExit = actualLiquidityPerExit > projectedLiquidityPerExit
529: uint256 mintAmount = newSupply > MAX_EXIT_SUPPLY ? MAX_EXIT_SUPPLY -
EXIT.totalSupply() : amount;
566: if (TOKEN_IN > TOKEN_OUT) {
```

Impact

Gas savings.

Recommendation

Use `>=` instead if appropriate.

Informational Findings

1. Informational - Standardize ProcessEth implementation

`_processEth()` is implemented in DepositHelper.sol and UniswapBase.sol. There are differences between the implementations:

1. The DepositHelper implementation uses `else` instead of a more strict `else if`
2. The UniswapBase implementation is missing an event.

Technical Details

Compare the implementations of `_processEth()`:

1. DepositHelper implementation, without an event
2. UniswapBase implementation, without an event

Impact

Informational.

Recommendation

Replace the `else` clause in the DepositHelper implementation with `else if` and add

missing event to the UniswapBase implementation to keep the implementations identical.

2. Informational - Consider `else if` instead of `else` for stricter checks

There are several places where an `else` branch exists while an `else if` may increase the security assurances of the code.

Technical Details

Some `else` branches might be best changed to `else if` branches to ensure that no unexpected edge cases trigger the `else` case. For example:

- This `else` branch could be replaced with `else if (POOL.token0() == TOKEN_IN)`
- This `else` branch could be replaced with `else if (_swapParams.tokenIn == TOKEN_1)`

Impact

Informational.

Recommendation

Add stricter checks instead of using `else` branches.

3. Informational - Possibly unnecessary event emit

An event emit in `_mintExitCapped()` may be unnecessary in some cases.

Technical Details

The `MintExit` event is emitted in `_mintExitCapped()`. This is the only event emitted in an internal function of Exit10.sol. It is emitted even when no EXIT token is minted. Instead, consider expanding the if statement to determine if EXIT is minted to include this emit. Another side effect of this emit is that it is duplicating the emit in `ERC20._mint()`, so each minting event will have 2 emits. Consider whether this is the intended result for minting EXIT tokens.

Impact

Informational.

Recommendation

Consider changing how this emit is triggered or removing it entirely.

4. Informational - Revert on zero case

There are several logic paths that skip certain logic in the case where `amount == 0`. Instead of allowing the code to execute in this case, revert instead to return gas back to the user and to avoid an attacker from reaching unexpected code paths.

Technical Details

Consider the implementation of `_safeTransferToken()`:

```
function _safeTransferToken(address _token, address _recipient, uint256 _amount)
internal {
          if (_amount != 0) IERC20(_token).safeTransfer(_recipient, _amount);
}
```

If `_amount` is zero, `_safeTransferToken()` will not revert, but means any functions calling `_safeTransferToken()` will continue executing. This is similar behavior to phantom functions, an area of research that dedaub previously found a novel security issue with. Because there should be no changes when an amount of zero is involved, reverting will save the user gas and also prevent later logic from executing unexpectedly.

Impact

Informational.

Recommendation

Add more revert code paths for unexpected edge cases, such as `_amount == 0` in `_safeTransferToken()`.

5. Informational - Replace modifiers

Exit10.sol has many internal functions that could have been implemented as internal functions instead of modifiers. But two other contracts do implement modifiers. If a modifier is only used once, it may make more sense to inline the logic directly in the function where it is needed. But at a minimum, consistency within a codebase is generally preferred and the modifiers could be converted to internal functions to match the pattern use in Exit10.sol.

Technical Details

NFT.sol and FeeSplitter.sol implement an onlyAuthorized modifier. This modifier can be replaced with internal functions in the same way that `_requireExitMode()`,

`_isBootstrapOngoing()`, and similar internal functions are implemented in Exit10.sol.

Impact

Informational.

Recommendation

Replace the onlyAuthorized modifiers with internal functions or vice versa. The change may also provide some gas savings.

6. Informational - Missing NatSpec

Most functions in Exit10 contracts are missing detailed NatSpec and inline comments. This can make it more difficult for users of the code to determine whether the code is implemented in a way that matches what the docs advertise.

Technical Details

NatSpec is a good way to explain your code to other developers modifying or forking a project, to users who want to understand what the contracts are doing, and to auditors who are trying to determine whether the contract logic is implemented properly. The contracts of Exit10 have a severe lack of detailed NatSpec comments which makes it harder to understand the developer's intentions.

Impact

Informational.

Recommendation

Add NatSpec to all functions, or at a minimum, all public and external functions. Consider using GitHub Copilot or slither-docs-action to leverage AI to speed up the process.

7. Informational - Use abstract contract

The contract UniswapBase is never deployed and can be defined as `abstract` contract.

Technical Details

Defining contract that is not deployed as `abstract` is a good practice to avoid confusion.

Impact

Informational.

Recommendation

Define contract `UniswapBase` as `abstract`.

8. Informational - Solidity version

The contract Helper.sol uses Solidity versions below and above `0.8.0`.

Technical Details

Solidity version `0.8.0` introduced a lot breaking changes. It would be good to define the same Solidity version for all contracts. Version above `0.8.0` don't need imported SafeMath library and abicoder v2 is enabled by default.

Impact

Informational.

Recommendation

Define Solidity version below `0.8.0`.

9. Informational - Non-descriptive variable and function names

Some variable names do not clearly describe what the variable or function refers to.

Technical Details

Some variable names could be improved:

- `exitBucketFinal` set in `exit10()` is a very confusing name for a variable that contains the liquidity of the exit bucket and the bootstrap bucket. Consider changing the variable name to `exitBucketBootstrapBucketFinal`.
- `exitBucketRewards` is the amount of withdrawn USDC from the `exitBucketFinal` liquidity amount. This liquidity amount includes the exit bucket and bootstrap bucket liquidity so it would be better named `exitBucketBootstrapBucketRewards`.
- `_getActualLiquidityPerExit` is the estimated liquidity per EXIT token assuming `MAX_EXIT_SUPPLY` of EXIT is minted and redeemed. This is not always a valid assumption, because `MasterchefExit.stopRewards()` will burn the undistributedRewards if `periodFinish` is not reached. This function would be better named `_getActualLiquidityPerExitAfterPeriodFinish`.
- `BOOTSTRAP_TARGET` and `BOOTSTRAP_CAP` would be better described by the names `BOOTSTRAP_LIQUIDITY_TARGET` and `BOOTSTRAP_LIQUIDITY_CAP`.
- `percentFromTaget` would be better described with `percentFromBootstrapTarget`.

- `_getPercentFromTarget()` can return values that represent over 100%, so the word "percent" should be removed from the name of this function because a value representing 100% is meaningless. Consider instead `getDollarPerExit()` or `getValuePerExit()`.

Impact

Informational.

Recommendation

Give variables more accurate names to avoid confusion.

10. Informational - Outdated documentation

The documentation does not include details about the bootstrap bucket when describing the other conceptual buckets.

Technical Details

The documentation describing the Exit10 buckets could be improved by adding a section for the bootstrap bucket.

Impact

Informational.

Recommendation

Add bootstrap buckets details in docs.

11. Informational - Replace magic numbers with constants

Constant variables should be used in place of magic numbers to prevent typos. For one example, the magic number 5000 is used in `_getExitAmount()` and should be replaced with a constant. Using a constant adds a description using the variable name to explain what this value is for. This will not change gas consumption.

Technical Details

The value 5000 appears on this line of Exit10.sol but there is no explanation for what this value represents. It looks like it represents the value 50% with a PERCENT_BASE of 10_000, but it turns out the 5000 value has nothing to do with a percentage. Instead, it helps set the price floor of the EXIT token, and a percentage over 100% only signifies a higher price. A constant variable with a clear name would make it easier to understand what is

happening in this function. Arguably the

Impact

Informational.

Recommendation

Use constant variables instead of magic numbers.

12. Informational - Typos

There were typos found in the name of a function and a comment within Exit10.sol. Typos can reduce the readability of the codebase.

Technical Details

The `_getLiquidityForBootsrapTarget` function name was missing a `t` in `Bootstrap` and so was this comment.

In addition, variable `percentFromTaget` on `Exit10.sol` lines 479 and 480 has a typo, it should be `percentFromTarget` instead.

Impact

Informational.

Recommendation

Consider modifying the two examples to include the omitted `t`.

```
- function _getLiquidityForBootsrapTarget() ...
+ function _getLiquidityForBootstrapTarget() ...
```

```
-  // Total initial deposits that needs to be returned to bootsrappers
+  // Total initial deposits that needs to be returned to bootstrappers
```

13. Informational - Unclaimed rewards can be added to user's reward debt

Users who have staked into one of the masterchef contracts are eligible for reward distributions. If the reward distributions that a user is entitled to are greater than the total balance of rewards held in the masterchef contract, then a user is given the lesser amount.

In this scenario, the full eligible amount of rewards are still added to the user's debt. A user can therefore be "charged" more rewards than they have actually received. Even if more rewards are placed in the contract at a later point in time through gathered fees or otherwise, the user will not be able to access the unclaimed rewards.

Technical Details

The `_safeClaimRewards` function calculates the amount of underlying reward tokens a user may claim for their staked position. This function is called during deposit or withdraw operations. It sets the claimable amount to the lesser of the remaining rewards in the contract or the user's eligible rewards. It then transfers this amount to the user. In both the deposit and withdraw functions, the rewardDebt is then set to be eligible for 0 rewards regardless of the results of the claimed rewards.

Impact

Informational.

Recommendation

Consider either keeping track of potential unclaimed rewards and subtracting them from a user's reward debt or making it clearer to users that this may occur. The second option could be achieved by modifying the pendingReward view function to return the minimum of the contract's reward token balance and the eligible rewards:

```
- return _getUserPendingReward(user.amount, user.rewardDebt, accRewardPerShare);
+ return Math.min(IERC20(REWARD_TOKEN).balanceOf(address(this)),
_getUserPendingReward(user.amount, user.rewardDebt, accRewardPerShare));
```

14. Informational - Remove unnecessary address casting

Address type values doesn't have to be casted to address type.

Technical Details

In the file AMasterchefBase.sol there are multiple unnecessary address casting at L51 and L121.

Impact

Informational.

Recommendation

Remove unnecessary casting.

15. Informational - Remove unnecessary virtual marker

Remove the function `virtual` marker if the function is not overridden in the child contract.

Technical Details

The function `_updateUndistributedRewards()` can drop the `virtual` marker because it is not overridden in child contract.

Impact

Informational.

Recommendation

Remove the `virtual` marker if the function is not overridden in the child contract.

16. Informational - Potentially unnecessary line of code

There is a line of code in `MasterchefExit.stopRewards()` which may not be necessary. If this line of code is actually needed, it may indicate problems with the protocol design.

Technical Details

It is likely that this line of logic in MasterchefExit is never reached. In fact, if it is reached it means that the Masterchef contract has accumulated debt by paying out more rewards than it should have. If this scenario ever occurs, it means there is a problem with another part of the protocol.

Impact

Informational.

Recommendation

Check if the tests pass with this line of code removed. Consider removing this line of code entirely.

17. Informational - (Out of scope file) Ensure that Bond NFT contract is initialized with non-zero `TRANSFER_LOCKOUT_PERIOD_SECONDS`

Technical Details

Bond NFTs are tradeable ERC721s that represent the holder's share of the liquidity in the protocol's `pendingBucket`. When a user converts or cancels the bond, the NFT still exists

but has its status in Exit10's `idToBondData` mapping changed to converted/canceled, rendering it unusable for protocol functions. To avoid situations where a user converts/cancels their bond and immediately sells it to an unsuspecting buyer, the following check in `NFT.sol#beforeTokenTransfer()` exists:

```
require(
        status == uint8(Exit10.BondStatus.active) || block.timestamp >= endTime +
TRANSFER_LOCKOUT_PERIOD_SECONDS,
        'NFT: Cannot transfer during lockout period'
      );
```

In the current tests, the value of TRANSFER_LOCKOUT_PERIOD_SECONDS is initialized to zero. This would allow a malicious user to bypass the lockout period and sell their bond immediately.

Impact

Informational.

Recommendation

Initialize `TRANSFER_LOCKOUT_PERIOD_SECONDS` to a non-zero value.

18. Informational - Potential lock funds if USDC implements taking fee mechanism in the future

Business logic changes from USDC could cause users fund locked forever in system

Technical Details

USDC contract is now using proxy pattern, which indeed could change business logic in the future. In case USDC takes fee on transfers before ETH 10k, Exit10 users can not claim total amount of USDC from system, i.e. at least the latest user can not claim his full amount because contract balance deducted by fee from earlier claim

Impact

Informational. Since this is not bug in system but it is potential scenario can happen in the future and funds locked could be large even though the chance is minimum

Recommendation

Treat USDC as fee on transfer token and using below pattern

```
// for transferFrom
uint256 balanceBefore = ERC20(TOKEN).balanceOf(recipient);
ERC20(TOKEN).safeTransferFrom(sender, recipient, amount);
uint256 received = ERC20(TOKEN).balanceOf(recipient) - balanceBefore;
...
/// for transfer
uint256 balance = ERC20(TOKEN).balanceOf(address(this));
ERC20(TOKEN).safeTransfer(recipient, amount > balance ? balance : amount);
```

19. Informational - Bonds may convert less EXIT tokens than users expect

It is possible that bonds mint less or 0 EXIT tokens.

Technical Details

The summation of convertible liquidity can exceed the cap EXIT token cap:

$$\sum_{i=1}^n (\frac{I_{it}}{r_{it}}) \geq E_S$$

Where:

- I_{it} is the convertible liquidity of bond i at time t;
- r_{it} is the rate of liquidity per EXIT token that bond i will receive at time t;
- E_s is the EXIT token supply cap.

The rate of liquidity per token for a bond at time ${t}$ to receive r_{it} is the larger of two values: the projected liquidity per token r_p and actual liquidity per token r_a. The formula for r_p is:

$$r_p=p_t*L_{USDC}$$

Where:

- ${p_t}$ is the percentage of collected bootstrap liquidity to its target, floored to 50%. In other words, if less than 50% of the target was reached, $p_t=0.5$.
- L_{USDC} is a constant: the projected amount of liquidity per USDC;

For r_a the formula is:

$$r_a=\frac{e_l}{E_s}$$

Where:

- e_l is the available liquidity to be claimed by EXIT holders from the exit bucket (i.e. 70% of the exit bucket);
- E_s is the exit token supply cap.

To break the invariant and fall in the case where the token cap exceeds we assume:

- Less than or 50% of the bootstrap target was reached (i.e. $p_t=0.5$);
- Not enough liquidity accumulated in the exit bucket and $r_a<=r_p$.

So we have:

$$r_p=r_a=\frac{L_{USDC}}{2} = \frac{e_l}{E_s}=r_{it}$$

The amount of EXIT tokens a bond i is entitled to at time t (before the discount is added) e_i is:

$$e_i=\frac{l_{it}}{r_{it}}$$

The convertible liquidity cap (i.e. the point from which the entire EXIT supply is minted and all other bonds receive 0 EXIT tokens per liquidity) can be defined by setting the amount of tokens received to the supply $e_i=E_s$. Solving for r_p and r_a:

$$e_i=\frac{2*l_{it}}{L_{USDC}} =\frac{l_{it}}{e_l}*E_s=E_s$$

From here, we can make two observations:

1. If the ratio between the bond's convertible liquidity l_{it} to the liquidity in the exit bucket $e_l\geq 1$ or;
2. If the bond's convertible liquidity $l_{it} \geq \frac{E_s*L_{USDC}}{2}$, i.e. $l_{it} \geq 6437989144.5 * 10^7$ assuming 10mi EXIT.

The bond would mint the entire EXIT supply and all bonds will receive 0 EXIT on conversion.

> Note: To simplify modeling we are using one massive bond, but this behavior is equivalent if the liquidity is split in multiple bonds. In other words, the two

observations above still apply if we replace I_{it} with $\sum_{k=1}^n(I_{it})$ as what matters is that the aggregate convertible liquidity.

Note that as the code is, people can still create new bonds even after the supply cap is reached.

Impact

Once the EXIT supply is minted, all bonds current and new bonds will receive 0 EXIT tokens on conversion.

Recommendation

- Warn the users creating new bonds (if the supply is reached) that their bond won't be eligible to receive any EXIT;
- Disclose to the users in the UI and documentation that their bonds may end up generating less EXIT than expected if the remaining mintable supply is less than what their bond is entitled to. Display how far from the cap the current supply is so that they can properly assess the risk, taking into account front running.

20. Informational - Masterchef is vulnerable to reentrancy attacks

Masterchef.sol and MasterchefExit.sol are vulnerable to reentrancy attacks in the `deposit()` and `withdraw()` functions.

Technical Details

In both functions, rewards are sent to the caller before state is updated to reflect the claim. If the implementation of the reward token contains callbacks or hooks that would grant control to the caller, such as an ERC777, then it is possible to execute a reentrancy attack to steal the reward tokens from the contract.

Using the withdraw() as an example, we can see that tokens are transferred in line 101 and state is updated in lines 103-105.

```
094:   function withdraw(uint256 pid, uint256 amount) public {
095:     PoolInfo storage pool = poolInfo[pid];
096:     UserInfo storage user = userInfo[pid][msg.sender];
097:     _updatePool(pid);
098:
```

```
099:     amount = Math.min(user.amount, amount);
100:
101:     _safeClaimRewards(pid, _getUserPendingReward(user.amount, user.rewardDebt,
pool.accRewardPerShare));
102:
103:     user.amount -= amount;
104:     user.rewardDebt = (user.amount * pool.accRewardPerShare) / PRECISION;
105:     pool.totalStaked -= amount;
106:     _transferAmountOut(pool.token, amount);
107:
108:     emit Withdraw(msg.sender, pid, amount);
109:   }
```

If the caller receives control during the call to `_safeClaimRewards()` (which transfers the reward token), an attacker can reenter the function and execute the claim again, since the state hasn't been updated yet, in particular line 104 which tracks how many rewards have been already sent to the user.

A test with a proof of concept for this issue is available here.

Impact

Informational. The issue would allow a bad actor to steal all reward tokens from the contract, however, the protocol won't use tokens that grant control to the caller during transfers (Ethereum, Optimism and Arbitrum).

Recommendation

Follow the "Checks Effects Interactions" pattern and update internal state before transferring the tokens. Alternatively, use a reentrancy guard to protect the functions from being called again (see OpenZeppelin ReentrancyGuard.sol).

21. Informational - Event not emitted when adding a token.

There is no event emitted when calling `AMasterchefBase.add()`.

When adding a token in AMasterchefBase, an event is not emitted.

File: Exit10-code/exit10-protocol/src/AMasterchefBase.sol

```
50: function add(uint256 allocPoint, address token) external onlyOwner {
```

Emitting events allows monitoring activities with off-chain monitoring tools.

Impact

Informational.

Recommendation

Emit an appropriate event.

22. Informational - Centralization risk during protocol bootstrap

A malicious protocol owner can take advantage before protocol contracts are finally assembled in their intended configuration.

Technical Details

During bootstrap, different contracts need to be created separately by the deployer of the protocol before they are put together in their final configuration. During this period of time, the deployer is the owner of these contracts that are later transferred to the Exi10.sol contract.

BLP, STO, BOOT and EXIT tokens used in the protocol are created during the initialization. As these contracts grant the owner the ability to arbitrarily mint tokens, a malicious deployer can mint any number of tokens they want before transferring control to the main contract.

Another example attack would be to backdoor the Masterchef.sol or MasterchefExit.sol contracts by configuring another pool with a fake token, which can then be used to steal rewards.

Impact

Informational. The issue requires a malicious protocol owner and can be eventually detected by a third party given enough attention at the initial state of the contracts.

Recommendation

Bootstrap the whole set of contracts in the Exit10.sol constructor. If there's a dependency issue, use a factory contract to deploy the protocol.

Final remarks

The design of the Exit10 protocol pulls in different ideas from other protocols, such as Liquity's Chicken Bonds and Uniswap v3, in a way that initially can look unintuitive. However, when considered as a fraction of a user's overall ETH exposure, the protocol design can make more sense when seen as providing exposure to a specific set of outcomes as the price of 1 ETH approaches $10,000. The tokenomics of Exit10 are more complex than Chicken Bonds and requires deep understanding to best make sense of the protocol design. One recommended task is to develop and run tests for all chains that the protocol will be deployed on, which means the existing tests should be adapted to also work on forked Optimism and Arbitrum networks. This will allow any chain-specific differences to be tested for, such as whether `TOKEN_IN > TOKEN_OUT` is true or false and whether the Uniswap v3 slot0 tick value is positive or negative. Another difference between Exit10 and the protocols it takes inspiration from (Chicken Bonds and Uniswap v3) is that Exit10 is designed with the goal to bootstrap funds for future Open Bakery projects. Unlike Chicken Bonds, some tokens in Exit10 are not directly redeemable with the protocol until ETH hits the price of $10k, which then unlocks the ability to redeem these tokens. But one similarity with Chicken Bonds is that the protocol design and reward incentives favors early depositors, especially in the bootstrap phase.