Boot Finance contest
Findings & Analysis Report

2022-01-21

Table of contents

Overview

About C4

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

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

During the code contest outlined in this document, C4 conducted an analysis of Boot Finance contest smart contract system written in Solidity. The code contest took place between November 4—November 10 2021.

Wardens

28 Wardens contributed reports to the Boot Finance contest:

1. jonah1005
2. WatchPug (jtp and ming)
3. pants
4. pauliax
5. Reigada
6. MetaOxNull
7. cmichel
8. gpersoon
9. gzeon
10. leastwood
11. fr0zn
12. defsec
13. JMukesh
14. 0v3rf10w
15. hyh
16. nathaniel
17. elprofesor
18. Ruhum
19. mics
20. ye0lde
21. loop
22. rfa
23. TomFrenchBlockchain

24. tqts
25. pmerkleplant
26. 0x0x0x
27. PranavG
28. jah

This contest was judged by 0xean.

Final report assembled by itsmetechjay and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 55 unique vulnerabilities and 156 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 9 received a risk rating in the category of HIGH severity, 12 received a risk rating in the category of MEDIUM severity, and 34 received a risk rating in the category of LOW severity.

C4 analysis also identified 39 non-critical recommendations and 62 gas optimizations.

Scope

The code under review can be found within the C4 Boot Finance contest repository, and is composed of 27 smart contracts written in the Solidity programming language and includes 5588 lines of Solidity code and 107 lines of JavaScript.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

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

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

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (9)

[H-01] Contract BasicSale is missing an approve(address(vestLock), 2**256-1) call

Submitted by Reigada, also found by WatchPug

Impact

As we can see in the contracts `AirdropDistribution` and `InvestorDistribution`, they both have the following `approve() call:` `mainToken.approve(address(vestLock), 2\*\*256-1);`

- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/AirdropDistribution.sol#L499
- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/InvestorDistribution.sol#L80

This is necessary because both contracts transfer tokens to the vesting contract by calling its `vest()` function:

- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/AirdropDistribution.sol#L544
- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/AirdropDistribution.sol#L569

- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/InvestorDistribution.sol#L134
- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/InvestorDistribution.sol#L158

The code of the `vest()` function in the Vesting contract performs a transfer from `msg.sender` to Vesting contract address -> `vestingToken.transferFrom(msg.sender, address(this), \_amount);`
https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/Vesting.sol#L95

Same is done in the BasicSale contract: https://github.com/code-423n4/2021-11-bootfinance/blob/main/tge/contracts/PublicSale.sol#L225

The problem is that this contract is missing the `approve()` call. For that reason, the contract is totally useless as the function `\_withdrawShare()` will always revert with the following message: revert reason: ERC20: transfer amount exceeds allowance. This means that all the `mainToken` sent to the contract would be stuck there forever. No way to retrieve them.

How this issue was not detected in the testing phase? Very simple. The mock used by the team has an empty `vest()` function that performs no transfer call.
https://github.com/code-423n4/2021-11-bootfinance/blob/main/tge/contracts/helper/MockVesting.sol#L10

Proof of Concept

See below Brownie's custom output:

```
Calling -> publicsale.withdrawShare(1, 1, {'from': user2})
Transaction sent: 0x9976e4f48bd14f9be8e3e0f4d80fdb8f660afab96a7c
Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 5
BasicSale.withdrawShare confirmed (ERC20: transfer amount exceec

Call trace for '0x9976e4f48bd14f9be8e3e0f4d80fdb8f660afab96a7cbc
Initial call cost  \[21344 gas]
BasicSale.withdrawShare  0:3724  \[16114 / -193010 gas]
├── BasicSale.\_withdrawShare  111:1109  \[8643 / 63957 gas]
│   ├── BasicSale.\_updateEmission  116:405  \[53294 / 55739 gas
```

```
    │   └── BasicSale.getDayEmission  233:248  \[2445 gas]
    ├── BasicSale.\_processWithdrawal  437:993  \[-7726 / -616 g
    │   ├── BasicSale.getEmissionShare  484:859  \[4956 / 6919 g
    │   │   │
    │   │   └── MockERC20.balanceOf  \[STATICCALL]  616:738  \[1
    │   │           ├── address: mockerc20.address
    │   │           ├── input arguments:
    │   │           │   └── account: publicsale.address
    │   │           └── return value: 100000000000000000000
    │   │
    │   └── SafeMath.sub  924:984  \[191 gas]
    └── SafeMath.sub  1040:1100  \[191 gas]
    │
    ├── MockERC20.transfer  \[CALL]  1269:1554  \[1115 / 30109 gas]
    │   ├── address: mockerc20.address
    │   ├── value: 0
    │   ├── input arguments:
    │   │   ├── recipient: user2.address
    │   │   └── amount: 27272727272727272727
    │   └── return value: True
    │
    └── ERC20.transfer  1366:1534  \[50 / 28994 gas]
        └── ERC20.\_transfer  1374:1526  \[28944 gas]
    └── Vesting.vest  \[CALL]  1705:3712  \[-330491 / -303190 gas]
    │   ├── address: vesting.address
    │   ├── value: 0
    │   ├── input arguments:
    │   │   ├── \_beneficiary: user2.address
    │   │   ├── \_amount: 63636363636363636363
    │   │   └── \_isRevocable: 0
    │   └── revert reason: ERC20: transfer amount exceeds allowance
    │
    ├── SafeMath.add  1855:1883  \[94 gas]
    ├── SafeMath.add  3182:3210  \[94 gas]
    ├── SafeMath.add  3236:3264  \[94 gas]
    │
    └── MockERC20.transferFrom  \[CALL]  3341:3700  \[99923 / 27019
    │   ├── address: mockerc20.address
    │   ├── value: 0
    │   ├── input arguments:
    │   │   ├── sender: publicsale.address
    │   │   ├── recipient: vesting.address
    │   │   └── amount: 63636363636363636363
    │   └── revert reason: ERC20: transfer amount exceeds allowance
    │
    └── ERC20.transferFrom  3465:3700  \[-97648 / -72904 gas]
```

```
└── ERC20.\_transfer  3473:3625  \[24744 gas]
```

Tools Used

Manual testing

Recommended Mitigation Steps

The following `approve()` call should be added in the constructor of the BasicSale contract: `mainToken.approve(address(vestLock), 2\*\*256-1);`

chickenpie347 (Boot Finance) confirmed

[H-02] Can not update target price

Submitted by jonah1005, also found by WatchPug

Impact

The sanity checks in `rampTargetPrice` are broken SwapUtils.sol#L1571-L1581

```
        if (futureTargetPricePrecise < initialTargetPricePrecise
            require(
                futureTargetPricePrecise.mul(MAX_RELATIVE_PRICE_
                "futureTargetPrice_ is too small"
            );
        } else {
            require(
                futureTargetPricePrecise <= initialTargetPricePr
                "futureTargetPrice_ is too large"
            );
        }
```

If `futureTargetPricePrecise` is smaller than `initialTargetPricePrecise` 0.01 of `futureTargetPricePrecise` would never larger than `initialTargetPricePrecise`.

Admin would not be able to ramp the target price. As it's one of the most important features of the customswap, I consider this is a high-risk issue

Proof of Concept

Here's a web3.py script to demo that it's not possible to change the target price even by 1 wei.

```
p1, p2, _, _ =swap.functions.targetPriceStorage().call()
future = w3.eth.getBlock(w3.eth.block_number)['timestamp'] +

# futureTargetPrice_ is too small
swap.functions.rampTargetPrice(p1 -1, future).transact()
# futureTargetPrice_ is too large
swap.functions.rampTargetPrice(p1 + 1, future).transact()
```

Tools Used

None

Recommended Mitigation Steps

Would it be something like:

```
if (futureTargetPricePrecise < initialTargetPricePrecise
    require(
        futureTargetPricePrecise.mul(MAX_RELATIVE_PRICE_
        "futureTargetPrice_ is too small"
    );
} else {
    require(
        futureTargetPricePrecise <= initialTargetPricePr
        "futureTargetPrice_ is too large"
    );
}
```

I believe the dev would spot this mistake if there's a more relaxed timeline.

chickenpie347 (Boot Finance) confirmed

[H-03] `SwapUtils.sol` Wrong implementation

Submitted by WatchPug

Based on the context, the `tokenPrecisionMultipliers` used in price calculation should be calculated in realtime based on `initialTargetPrice`, `futureTargetPrice`, `futureTargetPriceTime` and current time, just like `getA()` and `getA2()`.

However, in the current implementation, `tokenPrecisionMultipliers` used in price calculation is the stored value, it will only be changed when the owner called `rampTargetPrice()` and `stopRampTargetPrice()`.

As a result, the `targetPrice` set by the owner will not be effective until another `targetPrice` is being set or `stopRampTargetPrice()` is called.

Recommendation

Consider adding `Swap.targetPrice` and changing the `_xp()` at L661 from:

https://github.com/code-423n4/2021-11-bootfinance/blob/f102ee73eb320532c5a7c1e833f225c479577e39/customswap/contracts/SwapUtils.sol#L661-L667

```
    function _xp(Swap storage self, uint256[] memory balances)
        internal
        view
        returns (uint256[] memory)
    {
        return _xp(balances, self.tokenPrecisionMultipliers);
    }
```

To:

```
    function _xp(Swap storage self, uint256[] memory balances)
        internal
        view
        returns (uint256[] memory)
    {
        uint256[2] memory tokenPrecisionMultipliers = self.tokenPrec
        tokenPrecisionMultipliers[0] = self.targetPrice.originalPrec
        return _xp(balances, tokenPrecisionMultipliers);
```

```
}
```

chickenpie347 (Boot Finance) confirmed

[H-04] Swaps are not split when trade crosses target price

Submitted by cmichel, also found by gzeon

The protocol uses two amplifier values A1 and A2 for the swap, depending on the target price, see `SwapUtils.determineA`. The swap curve is therefore a join of two different curves at the target price. When doing a trade that crosses the target price, it should first perform the trade partially with A1 up to the target price, and then the rest of the trade order with A2.

However, the `SwapUtils.swap / _calculateSwap` function does not do this, it only uses the "new A", see `getYC` step 5.

```
// 5. Check if we switched A's during the swap
if (aNew == a){      // We have used the correct A
    return y;
} else {     // We have switched A's, do it again with the new A
    return getY(self, tokenIndexFrom, tokenIndexTo, x, xp, aNew,
}
```

Impact

Trades that cross the target price and would lead to a new amplifier being used are not split up and use the new amplifier for the *entire trade*. This can lead to a worse (better) average execution price than manually splitting the trade into two transactions, first up to but below the target price, and a second one with the rest of the trader order size, using both A1 and A2 values.

In the worst case, it could even be possible to make the entire trade with one amplifier and then sell the swap result again using the other amplifier making a profit.

Recommended Mitigation Steps

Trades that lead to a change in amplifier value need to be split up into two trades using both amplifiers to correctly calculate the swap result.

chickenpie347 (Boot Finance) confirmed

[H-05] Claim airdrop repeatedly

Submitted by gpersoon, also found by elprofesor, fr0zn, and pauliax

Impact

Suppose someone claims the last part of his airdrop via `claimExact()` of `AirdropDistribution.sol` Then `airdrop\[msg.sender].amount` will be set to 0.

Suppose you then call validate() again. The check `airdrop\[msg.sender].amount == 0` will allow you to continue, because amount has just be set to 0. In the next part of the function, `airdrop\[msg.sender]` is overwritten with fresh values and `airdrop\[msg.sender]` .claimed will be reset to 0.

Now you can claim your airdrop again (as long as there are tokens present in the contract)

Note: The function `claim()` prevents this from happening via `assert(airdrop\[msg.sender].amount - claimable != 0);` , which has its own problems, see other reported issues.

Proof of Concept

// https://github.com/code-423n4/2021-11-bootfinance/blob/7c457b2b5ba6b2c887dafdf7428fd577e405d652/vesting/contracts/AirdropDistribution.sol#L555-L563

```
    function claimExact(uint256 \_value) external nonReentrant {
    require(msg.sender != address(0));
    require(airdrop\[msg.sender].amount != 0);`

        uint256 avail = _available_supply();
        uint256 claimable = avail * airdrop[msg.sender].fraction / 1
        if (airdrop[msg.sender].claimed != 0){
```

```
            claimable -= airdrop[msg.sender].claimed;
        }

        require(airdrop[msg.sender].amount >= claimable); // amount
        require(_value <= claimable);                          // _valu
        airdrop[msg.sender].amount -= _value;       // amount will be
```

// https://github.com/code-423n4/2021-11-bootfinance/blob/7c457b2b5ba6b2c887dafdf7428fd577e405d652/vesting/contracts/AirdropDistribution.sol#L504-L517

```
    function validate() external nonReentrant {
    ...
    require(airdrop\[msg.sender].amount == 0, "Already validated.");
    ...
    Airdrop memory newAirdrop = Airdrop(airdroppable, 0, airdroppabl
    airdrop\[msg.sender] = newAirdrop;
    validated\[msg.sender] = 1;   // this is set, but isn't checked
```

Recommended Mitigation Steps

Add the following to `validate() : require(validated\[msg.sender]== 0, "Already validated.");`

chickenpie347 (Boot Finance) confirmed and resolved:

> Addressed in issue #101

[H-06] Ideal balance is not calculated correctly when providing imbalanced liquidity

Submitted by jonah1005

Impact

When a user provides imbalanced liquidity, the fee is calculated according to the ideal balance. In saddle finance, the optimal balance should be the same ratio as in the Pool.

Take, for example, if there's 10000 USD and 10000 DAI in the saddle's USD/DAI pool, the user should get the optimal lp if he provides lp with ratio = 1.

However, if the `customSwap` pool is created with a target price = 2. The user would get 2 times more lp if he deposits DAI. [SwapUtils.sol#L1227-L1245](#) The current implementation does not calculates ideal balance correctly.

If the target price is set to be 10, the ideal balance deviates by 10. The fee deviates a lot. I consider this is a high-risk issues.

Proof of Concept

We can observe the issue if we initiates two pools DAI/LINK pool and set the target price to be 4.

For the first pool, we deposit more DAI.

```
swap = deploy_contract('Swap'
    [dai.address, link.address], [18, 18], 'lp', 'lp', 1, 85
link.functions.approve(swap.address, deposit_amount).transac
dai.functions.approve(swap.address, deposit_amount).transact
previous_lp = lptoken.functions.balanceOf(user).call()
swap.functions.addLiquidity([deposit_amount, deposit_amount
post_lp = lptoken.functions.balanceOf(user).call()
print('get lp', post_lp - previous_lp)
```

For the second pool, one we deposit more DAI.

```
swap = deploy_contract('Swap'
    [dai.address, link.address], [18, 18], 'lp', 'lp', 1, 85
link.functions.approve(swap.address, deposit_amount).transac
dai.functions.approve(swap.address, deposit_amount).transact
previous_lp = lptoken.functions.balanceOf(user).call()
swap.functions.addLiquidity([deposit_amount, deposit_amount
post_lp = lptoken.functions.balanceOf(user).call()
print('get lp', post_lp - previous_lp)
```

We can get roughly 4x more lp in the first case

Tools Used

None

Recommended Mitigation Steps

The current implementation uses `self.balances`

https://github.com/code-423n4/2021-11-bootfinance/blob/main/customswap/contracts/SwapUtils.sol#L1231-L1236

```
    for (uint256 i = 0; i < self.pooledTokens.length; i++) {
        uint256 idealBalance = v.d1.mul(self.balances[i]).div(v.d0);
        fees[i] = feePerToken
            .mul(idealBalance.difference(newBalances[i]))
            .div(FEE_DENOMINATOR);
        self.balances[i] = newBalances[i].sub(
            fees[i].mul(self.adminFee).div(FEE_DENOMINATOR)
        );
        newBalances[i] = newBalances[i].sub(fees[i]);
    }
```

Replaces `self.balances` with `_xp(self, newBalances)` would be a simple fix. I consider the team can take balance's weighted pool as a reference.
WeightedMath.sol#L149-L179

chickenpie347 (Boot Finance) confirmed

[H-07] `customPrecisionMultipliers` would be rounded to zero and break the pool

Submitted by jonah1005

Impact

`CustomPrecisionMultipliers` are set in the constructor:

```
    customPrecisionMultipliers[0] = targetPriceStorage.originalPreci
```

`originalPrecisionMultipliers` equal to 1 if the token's decimal = 18. The targe price could only be an integer.

If the target price is bigger than 10**18, the user can deposit and trade in the pool. Though, the functionality would be far from the spec.

If the target price is set to be smaller than 10**18, the pool would be broken and all funds would be stuck.

I consider this is a high-risk issue.

Proof of Concept

Please refer to the implementation. [Swap.sol#L184-L187]

We can also trigger the bug by setting a pool with target price = 0.5. (0.5 * 10**18)

Tools Used

None

Recommended Mitigation Steps

I recommend providing extra 10**18 in both multipliers.

```
customPrecisionMultipliers[0] = targetPriceStorage.originalPreci
customPrecisionMultipliers[1] = targetPriceStorage.originalPreci
```

The customswap only supports two tokens in a pool, there's should be enough space. Recommend the devs to go through the trade-off saddle finance has paid to support multiple tokens. The code could be more clean and efficient if the pools' not support multiple tokens.

chickenpie347 (Boot Finance) confirmed

[H-08] Unable to claim vesting due to unbounded timelock loop

Submitted by nathaniel, also found by WatchPug, leastwood, and pauliax

Impact

The timelocks for any *beneficiary* are unbounded, and can be vested by someone who is not the *beneficiary*. When the array becomes significantly big enough, the vestments will no longer be claimable for the *beneficiary*.

The `vest()` function in Vesting.sol does not check the *beneficiary*, hence anyone can vest for anyone else, pushing a new timelock to the `timelocks[_beneficiary]`. The `_claimableAmount()` function (used by `claim()` function), then loops through the `timelocks[_beneficiary]` to determine the amount to be claimed. A malicious actor can easy repeatedly call the `vest()` function with minute amounts to make the array large enough, such that when it comes to claiming, it will exceed the gas limit and revert, rendering the vestment for the beneficiary unclaimable. The malicious actor could do this to each *beneficiary*, locking up all the vestments.

Proof of Concept

- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/Vesting.sol#L81
- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/Vesting.sol#L195
- https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/Vesting.sol#L148

Tools Used

Manual code review

Recommended Mitigation Steps

- Create a minimum on the vestment amounts, such that it won't be feasible for a malicious actor to create a large amount of vestments.
- Restrict the vestment contribution of a *beneficiary* where `require(beneficiary == msg.sender)`

chickenpie347 (Boot Finance) confirmed

[H-09] addInvestor() Does Not Check Availability of investors_supply

Submitted by MetaOxNull

Impact

When add investor, `addInvestor()` does not check how many tokens is available from `investors_supply`. The total tokens allocated for Investors could more than `investors_supply`.

Possible Attack Scenario:

1. Attacker who have Admin Private key call `addInvestor()` and `Input \_amount >= investors_supply`.
2. Attacker can Claim All Available Tokens Now.

Proof of Concept

https://github.com/code-423n4/2021-11-bootfinance/blob/main/vesting/contracts/InvestorDistribution.sol#L85-L94

Tools Used

Manual Review

Recommended

1. Add `require(\_amount <= (investors_supply - Allocated_Amount))`
2. When Add an Investor add the amount to `Allocated_Amount` with SafeMath

chickenpie347 (Boot Finance) acknowledged:

> While this is true, the addInvestor would be a one-time routine at deployment which would precisely send the allocated number of tokens to the contract as per to the allocatations.

Medium Risk Findings (12)

[M-01] Unchecked transfers

Submitted by Reigada, also found by Ruhum, loop, cmichel, defsec, pauliax, WatchPug, and 0v3rf10w

Impact

Multiple calls to `transferFrom` and transfer are frequently done without checking the results. For certain ERC20 tokens, if insufficient tokens are present, no revert occurs but a result of “false” is returned. It’s important to check this. If you don’t, in this concrete case, some airdrop eligible participants could be left without their tokens. It is also a best practice to check this.

Proof of Concept

```
AirdropDistributionMock.sol:132:        mainToken.transfer(msg.s
AirdropDistributionMock.sol:157:        mainToken.transfer(msg.s
AirdropDistribution.sol:542:        mainToken.transfer(msg.sende
AirdropDistribution.sol:567:        mainToken.transfer(msg.sende

InvestorDistribution.sol:132:         mainToken.transfer(msg.senc
InvestorDistribution.sol:156:         mainToken.transfer(msg.senc
InvestorDistribution.sol:207:         mainToken.transfer(msg.senc

Vesting.sol:95:        vestingToken.transferFrom(msg.sender, adc

PublicSale.sol:224:            mainToken.transfer(\_member, v_va
```

Tools Used

Manual testing

Recommended Mitigation Steps

Check the result of `transferFrom` and transfer. Although if this is done, the contracts will not be compatible with non standard ERC20 tokens like USDT. For that reason, I would rather recommend making use of SafeERC20 library: `safeTransfer` and `safeTransferFrom`.

chickenpie347 (Boot Finance) confirmed

[M-02] Unchecked low-level calls

Submitted by 0v3rf10w, also found by Reigada

Impact

Unchecked low-level calls

Proof of Concept

Unchecked cases at 2 places :- `BasicSale.receive()` (2021-11-bootfinance/tge/contracts/PublicSale.sol#148-156) ignores return value by `burnAddress.call{value: msg.value}()` (2021-11-bootfinance/tge/contracts/PublicSale.sol#154)

BasicSale.burnEtherForMember(address) (2021-11-bootfinance/tge/contracts/PublicSale.sol#158-166) ignores return value by `burnAddress.call{value: msg.value}()` (2021-11-bootfinance/tge/contracts/PublicSale.sol#164)

Tools Used

Manual

Recommended Mitigation Steps

The return value of the low-level call is not checked, so if the call fails, the Ether will be locked in the contract. If the low level is used to prevent blocking operations, consider logging failed calls.

chickenpie347 (Boot Finance) confirmed

[M-03] Investor can't claim the last tokens (via claim())

Submitted by gpersoon

Impact

Suppose you are an investor and want to claim the last part of your claimable tokens (or your entire set of claimable tokens if you haven't claimed anything yet). Then you call the function `claim()` of `InvestorDistribution.sol`, which has the following statement: `require(investors\[msg.sender].amount - claimable != 0);` This statement will prevent you from claiming your tokens because it will stop execution.

Note: with the function `claimExact()` it is possible to claim the last part.

Proof of Concept

// https://github.com/code-423n4/2021-11-bootfinance/blob/7c457b2b5ba6b2c887dafdf7428fd577e405d652/vesting/contracts/InvestorDistribution.sol#L113-L128

```
function claim() external nonReentrant {
...
require(investors\[msg.sender].amount - claimable != 0);
investors\[msg.sender].amount -= claimable;
```

Tools Used

Recommended Mitigation Steps

Remove the require statement.

chickenpie347 (Boot Finance) commented:

> Duplicate of issue #130

chickenpie347 (Boot Finance) commented:

> I just noticed it's different files. The AirdropDistrbution.sol and InvestorDistribution.sol contracts were built on the same base, with slight changes.

0xean (judge) commented:

> Downgrading to medium risk as an alternative path does exist for claiming the drop. Funds are not lost, but the availability of them is compromised. Per Docs:

```
2 — Med: Assets not at direct risk, but the function of the prot
3 — High: Assets can be stolen/lost/compromised directly (or inc
```

[M-04] Get virtual price is not monotonically increasing

Submitted by jonah1005

Impact

There's a feature of `virtualPrice` that is monotonically increasing regardless of the market. This function is heavily used in multiple protocols. e.g.(curve metapool, mim, ...) This is not held in the current implementation of customSwap since `customPrecisionMultipliers` can be changed by changing the target price.

There are two issues here: The meaning of `virtualPrice` would be vague. This may damage the lp providers as the protocol that adopts it may be hacked.

I consider this is a medium-risk issue.

Proof of Concept

We can set up a mockSwap with extra `setPrecisionMultiplier` to check the issue.

```
    function setPrecisionMultiplier(uint256 multipliers) externa
        swapStorage.tokenPrecisionMultipliers[0] = multipliers;
    }
```

```
    print(swap.functions.getVirtualPrice().call())
    swap.functions.setPrecisionMultiplier(2).transact()
    print(swap.functions.getVirtualPrice().call())

# output log:
#   1000000000000000000
```

```
#    1499889859738721606
```

Tools Used

None

Recommended Mitigation Steps

Dealing with the target price with multiplier precision seems clever as we can reuse most of the existing code. However, the precision multiplier should be an immutable parameter. Changing it after the pool is set up would create multiple issues. This function could be implemented in a safer way IMHO.

The quick fix would be to remove the `getVirtualPrice` function. I can't come up with a safe way if other protocol wants to use this function.

chickenpie347 (Boot Finance) confirmed

[M-05] Stop ramp target price would create huge arbitrage space.

Submitted by jonah1005

Stop ramp target price would create huge arbitrage space.

Impact

`stopRampTargetPrice` would set the `tokenPrecisionMultipliers` to `originalPrecisionMultipliers[0].mul(currentTargetPrice).div(WEI_UNIT);` Once the `tokenPrecisionMultipliers` is changed, the price in the AMM pool would change. Arbitrager can sandwich `stopRampTargetPrice` to gain profit.

Assume the decision is made in the DAO, an attacker can set up the bot once the proposal to `stopRampTargetPrice` has passed. I consider this is a medium-risk issue.

Proof of Concept

The `precisionMultiplier` is set here: Swap.sol#L661-L666

We can set up a mockSwap with extra `setPrecisionMultiplier` to check the issue.

```
    function setPrecisionMultiplier(uint256 multipliers) external {
        swapStorage.tokenPrecisionMultipliers[0] = multipliers;
    }
```

```
    print(swap.functions.getVirtualPrice().call())
    swap.functions.setPrecisionMultiplier(2).transact()
    print(swap.functions.getVirtualPrice().call())

    # output log:
    #     1000000000000000000
    #     1499889859738721606
```

Tools Used

None

Recommended Mitigation Steps

Dealing with the target price with multiplier precision seems clever as we can reuse most of the existing code. However, the precision multiplier should be an immutable parameter. Changing it after the pool is setup would create multiple issues. This function could be implemented in a safer way IMHO.

A quick fix I would come up with is to ramp the `tokenPrecisionMultipliers` as the `aPrecise` is ramped. As the `tokenPrecision` is slowly increased/decreased, the arbitrage space would be slower and the profit would (probably) distribute evenly to lpers.

Please refer to `_getAPreceise` 's implementation SwapUtils.sol#L227-L250

chickenpie347 (Boot Finance) confirmed

[M-07] `MainToken.set_mint_multisig()` doesn't check that `_minting_multisig` doesn't equal zero

Submitted by pants

The function `MainToken.set_mint_multisig()` doesn't check that `_minting_multisig` doesn't equal zero before it sets it as the new `minting_multisig`.

Impact

This function can be invoked by mistake with the zero address as `_minting_multisig`, causing the system to lose its `minting_multisig` forever, without the option to set a new `minting_multisig`.

Tool Used

Manual code review.

Recommended Mitigation Steps

Check that `_minting_multisig` doesn't equal zero before setting it as the new `minting_multisig`.

chickenpie347 (Boot Finance) confirmed

[M-08] `LPToken.set_minter()` doesn't check that `_minter` doesn't equal zero

Submitted by pants

The function `LPToken.set_minter()` doesn't check that `_minter` doesn't equal zero before it sets it as the new minter.

Impact

This function can be invoked by mistake with the zero address as `_minter`, causing the system to lose its minter forever, without the option to set a new minter.

Tool Used

Manual code review.

Recommended Mitigation Steps

Check that `_minter` doesn't equal zero before setting it as the new minter.

chickenpie347 (Boot Finance) confirmed)

[M-09] NFT flashloans can bypass sale constraints

Submitted by pauliax

Impact

Public sale has a constraint that for the first 4 weeks only NFT holders can access the sale:

```
    if (currentEra < firstPublicEra) {
        require(nft.balanceOf(msg.sender) > 0, "You need NFT to part
    }
```

However, this check can be easily bypassed with the help of flash loans. You can borrow the NFT, participate in the sale and then return this NFT in one transaction. It takes only 1 NFT that could be flashloaned again and again to give access to the sale for everyone (`burnEtherForMember`).

Recommended Mitigation Steps

I am not sure what could be the most elegant solution to this problem. You may consider transferring and locking this NFT for at least 1 block but then the user will need to do an extra tx to retrieve it back. You may consider taking a snapshot of user balances so the same NFT can be used by one address only but then this NFT will lose its extra benefit of selling it during the pre-sale when it acts as a pre-sale token. You may consider checking that the caller is EOA but again there are ways to bypass that too.

chickenpie347 (Boot Finance) acknowledged

[M-10] Can't claim last part of airdrop

Submitted by gpersoon

Impact

Suppose you are eligible for the last part of your airdrop (or your entire airdrop if you haven't claimed anything yet). Then you call the function claim() of AirdropDistribution.sol, which has the following statement: `assert(airdrop\[msg.sender].amount - claimable != 0);` This statement will prevent you from claiming your airdrop because it will stop execution.

Note: with the function `claimExact()` it is possible to claim the last part.

Proof of Concept

// https://github.com/code-423n4/2021-11-bootfinance/blob/7c457b2b5ba6b2c887dafdf7428fd577e405d652/vesting/contracts/AirdropDistribution.sol#L522-L536

```
    function claim() external nonReentrant {
    ..
    assert(airdrop\[msg.sender].amount - claimable != 0);
    airdrop\[msg.sender].amount -= claimable;
```

Recommended Mitigation Steps

Remove the assert statement. Also add the following to `validate()` , to prevent claiming the airdrop again: `require(validated\[msg.sender]== 0, "Already validated.");`

chickenpie347 (Boot Finance) confirmed:

> Patched it with `assert(airdrop[msg.sender].amount - claimable >= 0);` the >=0 check is just to ensure the claimant does not end up claiming more than allocated due to any fringe case.

Oxean (judge) commented:

> Downgrading to medium risk as an alternative path does exist for claiming the drop. Funds are not lost, but the availability of them is compromised. Per Docs:

```
2 — Med: Assets not at direct risk, but the function of the prot
3 — High: Assets can be stolen/lost/compromised directly (or inc
```

[M-11] Overwrite benRevocable

Submitted by gpersoon, also found by pauliax, WatchPug, cmichel, hyh, and leastwood

Impact

Anyone can call the function `vest()` of `Vesting.sol`, for example with a smail `\_amount` of tokens, for any `\_beneficiary`.

The function overwrites the value of `benRevocable\[\_beneficiary]`, effectively erasing any previous value.

So you can set any `\_beneficiary` to Revocable. Although `revoke()` is only callable by the owner, this is circumventing the entire mechanism of `benRevocable`.

Proof of Concept

// https://github.com/code-423n4/2021-11-bootfinance/blob/7c457b2b5ba6b2c887dafdf7428fd577e405d652/vesting/contracts/Vesting.sol#L73-L98

```
function vest(address \_beneficiary, uint256 \_amount, uint256 \
    ...
if(\_isRevocable == 0){
    benRevocable\[\_beneficiary] = \[false,false];  // just over
}
else if(\_isRevocable == 1){
    benRevocable\[\_beneficiary] = \[true,false]; // just overwr
}
```

Recommended Mitigation Steps

Whitelist the calling of `vest()` Or check if values for `benRevocable` are already set.

chickenpie347 (Boot Finance) confirmed

[M-12] No Transfer Ownership Pattern

Submitted by defsec, also found by Ruhum, elprofesor, pauliax, Reigada, and mics

Impact

The current ownership transfer process involves the current owner calling `Swap.transferOwnership()` . This function checks the new owner is not the zero address and proceeds to write the new owner's address into the owner's state variable. If the nominated EOA account is not a valid account, it is entirely possible the owner may accidentally transfer ownership to an uncontrolled account, breaking all functions with the `onlyOwner()` modifier.

Proof of Concept

1. Navigate to "https://github.com/code-423n4/2021-11-bootfinance/blob/7c457b2b5ba6b2c887dafdf7428fd577e405d652/customswap/contracts/Swap.sol#L30"
2. The contract has many `onlyOwner` function.
3. The contract is inherited from the Ownable which includes `transferOwnership` .

Tools Used

None

Recommended Mitigation Steps

Implement zero address check and consider implementing a two step process where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of ownership to fully succeed. This ensures the nominated EOA account is a valid and active account.

chickenpie347 (Boot Finance) confirmed

Oxean (judge) commented:

> upgrading to med severity as this could impact availability of protocol

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

Low Risk Findings (34)

- [L-01] Missing Zero-check *Submitted by 0v3rf10w, also found by Reigada*
- [L-02] Reentrancy *Submitted by 0v3rf10w, also found by defsec*
- [L-03] No checking of admin*fee, wether it is <= max*admin_fee *Submitted by JMukesh*
- [L-04] No event was emitted while setting fees and admin_fees in constructor *Submitted by JMukesh*
- [L-05] wrong operator used in checking the fees, adminfee, withdrawfee *Submitted by JMukesh, also found by loop and yeOlde*
- [L-06] revoke() Does Not Check Zero Address for _addr *Submitted by MetaOxNull*
- [L-07] Incorrect event parameter used in emit *Submitted by Reigada*
- [L-08] Tokens not recoverable *Submitted by Reigada*
- [L-09] Vesting.revoke is missing a require statement *Submitted by Reigada*
- [L-10] Require statement missing in fallback and burnEtherForMember() functions *Submitted by Reigada*
- [L-11] Check ERC20 token `approve()` function return value *Submitted by Ruhum*
- [L-12] Vesting contract locks tokens for less time than expected *Submitted by TomFrench, also found by pauliax*
- [L-13] Tokens with decimals larger than 18 are not supported *Submitted by WatchPug*

- [L-14] `SwapUtils.sol` Inconsistent parameter value of `lpTokenSupply` among `Liquidity` related events *Submitted by WatchPug*
- [L-15] # Missing parameter validation *Submitted by cmichel*
- [L-16] `BasicSale` has unused ERC20 code *Submitted by cmichel*
- [L-17] `BasicSale` uses inaccurate `secondsPerDay` value *Submitted by cmichel*
- [L-18] can withdraw shares on behalf of anyone *Submitted by cmichel*
- [L-19] Lack of maximum and minimum vesting amount check on the vesting function *Submitted by defsec*
- [L-20] Airdrop Supply differs *Submitted by fr0zn, also found by pauliax*
- [L-21] Vesting.benVested storage variable can be simplified, while *claimableAmount's "s <= benTotal[addr]"* check is redundant and to be removed *Submitted by hyh*
- [L-22] Incorrect `require` Statement in `Vesting.claim()` *Submitted by leastwood*
- [L-23] Usage of deprecated safeApprove *Submitted by mics*
- [L-24] claimExact does not check claimable amount *Submitted by nathaniel, also found by fr0zn*
- [L-25] admin can override investor and stuck its funds in the system *Submitted by pants, also found by defsec*
- [L-26] Array out-of-bounds errors in `BTCPoolDelegator` *Submitted by pants*
- [L-27] Array out-of-bounds errors in `ETHPoolDelegator` *Submitted by pants*
- [L-28] Array out-of-bounds errors in `USDPoolDelegator` *Submitted by pants*
- [L-29] payable vest *Submitted by pauliax*
- [L-30] _recordBurn does not handle 0 _eth appropriately *Submitted by pauliax*
- [L-31] Usage of assert *Submitted by pauliax, also found by leastwood*
- [L-32] Validations *Submitted by pauliax*
- [L-33] Unclear Commented Out Code *Submitted by yeOlde, also found by loop*
- [L-34] Don't allow swapping the same token *Submitted by Ruhum, also found by rfa*

Non-Critical Findings (39)

- [N-01] claimExact() Missing Validation As In claim() *Submitted by MetaOxNull, also found by pauliax*
- [N-02] `Vesting.sol#calcClaimableAmount()` Claimed amount should be excluded in claimable amount *Submitted by WatchPug*
- [N-03] Invalid return value when calculating claimable amount *Submitted by frOzn*
- [N-04] Missing revert message *Submitted by mics*
- [N-05] `Token.transfer()` and `Token.transferFrom()` emit `Transfer` events when the transferred amount is zero *Submitted by pants*
- [N-06] `Token.transferFrom()` emits `Transfer` events when `_from` equals `_to` *Submitted by pants*
- [N-07] `Token.approve()` emits `Approval` events when the allowence hasn't changed *Submitted by pants*
- [N-08] `PoolGauge.deposit()` emits `Deposit` events when the deposited amount is zero *Submitted by pants*
- [N-09] `PoolGauge.withdraw()` emits `Withdraw` events when the withdrawn amount is zero *Submitted by pants*
- [N-10] `MainToken.transfer()` and `MainToken.transferFrom()` emit `Transfer` events when the transferred amount is zero *Submitted by pants*
- [N-11] `MainToken.transferFrom()` emits `Transfer` events when `_from` equals `_to` *Submitted by pants*
- [N-12] `MainToken.approve()` emits `Approval` events when the allowance hasn't changed *Submitted by pants*
- [N-13] `MainToken.mint()` and `MainToken.mint_dev()` emit `Transfer` events when the minted amount is zero *Submitted by pants*
- [N-14] `MainToken.burn()` emits `Transfer` events when the burned amount is zero *Submitted by pants*
- [N-15] `MainToken.set_minter()` emits `SetMinter` events when the minter hasn't changed *Submitted by pants*

- [N-16] `MainToken.set_admin()` emits `SetAdmin` events when the admin hasn't changed *Submitted by pants*
- [N-17] `MainToken.set_mint_multisig()` emits `SetMintMultisig` events when `minting_multisig` hasn't changed *Submitted by pants*
- [N-18] `MainToken.__init__()` emits `Transfer` events when the amount minted for `msg.sender` is zero (and it is always the case) *Submitted by pants*
- [N-19] `LPToken.__init__()` emits `Transfer` events when the amount minted for `msg.sender` is zero *Submitted by pants*
- [N-20] `LPToken.transfer()` and `LPToken.transferFrom()` emit `Transfer` events when the transferred amount is zero *Submitted by pants*
- [N-21] `LPToken.transferFrom()` emits `Transfer` events when `_from` equals `_to` *Submitted by pants*
- [N-22] `LPToken.approve()` emits `Approval` events when the allowance hasn't changed *Submitted by pants*
- [N-23] Missing emit of initial `SetAdmin` event in `MainToken.__init__()` *Submitted by pants*
- [N-24] Missing emit of initial `ApplyOwnership` event in `GaugeController.__init__()` *Submitted by pants*
- [N-25] `GaugeController.apply_transfer_ownership()` emits `ApplyOwnership` events when the admin hasn't changed *Submitted by pants*
- [N-26] `GaugeController.commit_transfer_ownership()` emits `CommitOwnership` events when the future admin hasn't changed *Submitted by pants*
- [N-27] block timestamp *Submitted by 0v3rf10w*
- [N-28] Unnecessary imports *Submitted by PranavG*
- [N-29] Code Style: consistency *Submitted by WatchPug*
- [N-30] Typos *Submitted by WatchPug*
- [N-31] Missing error messages in require statements *Submitted by WatchPug*
- [N-32] Renaming variables for clarity *Submitted by nathaniel*
- [N-33] burnAddress is not actually meant to burn anything *Submitted by pauliax*

- [N-34] Events should be written in CapWords *Submitted by pmerkleplant*
- [N-35] Functions should be written in mixedCase *Submitted by pmerkleplant*
- [N-36] Contract `Vesting` should inherit from interface `IVesting` *Submitted by pmerkleplant*
- [N-37] Constants should be written in UPPER_CASE *Submitted by pmerkleplant*
- [N-38] Function _getDayEmission can be simplified (PublicSale.sol) *Submitted by ye0lde*
- [N-39] use of floating pragma *Submitted by JMukesh, also found by loop*

Gas Optimizations (62)

- [G-01] Unnecessary require statement in vesting.claim() *Submitted by Reigada, also found by MetaOxNull*
- [G-02] uint256 is always >= 0 *Submitted by 0x0x0x, also found by Reigada, WatchPug, loop, and pauliax*
- [G-03] Packing of state variable *Submitted by JMukesh*
- [G-04] validate() to Verify Airdrop Address On Chain is Unnecessary *Submitted by MetaOxNull, also found by cmichel and pauliax*
- [G-05] State variables could be declared constant *Submitted by PranavG, also found by defsec, ye0lde, WatchPug, nathaniel, and pmerkleplant*
- [G-06] Use of uint256 parameter instead of bool *Submitted by Reigada, also found by fr0zn*
- [G-07] If statement in _updateEmission() can be removed *Submitted by Reigada, also found by cmichel and fr0zn*
- [G-08] No usage of immutable keyword leaves free gas savings on the table *Submitted by TomFrench, also found by WatchPug, jah, PranavG, Reigada, nathaniel, pants, pauliax, and pmerkleplant*
- [G-09] Remove unnecessary variables can save some gas *Submitted by WatchPug*
- [G-10] `SwapUtils.sol#getYD()` Remove redundant code can save gas *Submitted by WatchPug*
- [G-11] External call can be done later to save gas *Submitted by WatchPug*

- [G-12] `SwapUtils.sol#getD()` Remove unnecessary variable and internal call can make the code simpler and save some gas *Submitted by WatchPug, also found by hyh*
- [G-13] Use literal `2` instead of read from storage for `pooledTokens.length` can save gas *Submitted by WatchPug, also found by ye0lde, 0x0x0x, JMukesh, Ruhum, WatchPug, pauliax, gzeon, and loop*
- [G-14] Cache external call results can save gas *Submitted by WatchPug, also found by hyh*
- [G-15] `Vesting.sol#_claimableAmount()` Remove unnecessary storage variables can save gas *Submitted by WatchPug*
- [G-16] Gas: Unnecessary length check in `Swap.constructor` *Submitted by cmichel*
- [G-17] Gas: Unnecessary msg.sender != 0 check *Submitted by cmichel, also found by rfa, WatchPug, and pauliax*
- [G-18] Adding unchecked directive can save gas *Submitted by defsec, also found by mics, pants, and pauliax*
- [G-19] Upgrade pragma to at least 0.8.4 *Submitted by defsec*
- [G-20] Gas optimization on InvestorDistribution.sol *Submitted by fr0zn*
- [G-21] Duplicated code and usage of assert *Submitted by fr0zn, also found by nathaniel and pauliax*
- [G-22] Redundant check on claim *Submitted by fr0zn*
- [G-23] SwapUtils.getVirtualPrice double calling to storage reading function _xp(self) *Submitted by hyh*
- [G-24] SwapUtils's addLiquidity does multiple LP token total supply calls *Submitted by hyh*
- [G-25] SwapUtils.calculateTokenAmount does repetitive checks of static condition *Submitted by hyh*
- [G-26] SwapUtils's getD, getY, getYD functions do repetitive calculations of contant expression within the cycles *Submitted by hyh*
- [G-27] No need to initialize variables with default values *Submitted by jah, also found by ye0lde, MetaOxNull, Reigada, WatchPug, pants, and pauliax*
- [G-28] `Timelock` Struct Packing in `Vesting.sol` *Submitted by leastwood*

- [G-29] safeERC20 library imported but not used *Submitted by loop*
- [G-30] unnecessary variable y in getYD *Submitted by mics*
- [G-31] Use calldata instead of memory for function parameters *Submitted by mics*
- [G-32] Public functions can be external *Submitted by nathaniel, also found by pants, Reigada, hyh, leastwood, and defsec*
- [G-33] double reading of memory inside a loop without caching *Submitted by pants, also found by Ruhum, WatchPug, and mics*
- [G-34] Rearrange state variables *Submitted by pants*
- [G-63] optimizing for loops by caching array length *Submitted by pants, also found by pauliax, WatchPug, and hyh*
- [G-36] internal functions could be set private *Submitted by pants*
- [G-37] `PoolGauge.withdraw()` can be optimized when `_value` equals zero *Submitted by pants*
- [G-38] Unnecessary use of safeMath *Submitted by pants, also found by Ruhum, Reigada, TomFrench, loop, pauliax, and defsec*
- [G-39] too many bits to describe small quantities *Submitted by pants*
- [G-40] multiple reading of state variables without caching *Submitted by pants, also found by pauliax*
- [G-41] modifyInvestor does not need to check if _investor is not empty *Submitted by pauliax*
- [G-42] function claim optimizations *Submitted by pauliax*
- [G-43] Itteration over all the timelocks when revoking the user *Submitted by pauliax*
- [G-44] Optimize structs *Submitted by pauliax*
- [G-45] Useless nonReentrant *Submitted by pauliax*
- [G-46] _recordBurn _payer *Submitted by pauliax*
- [G-47] Remove unused variables *Submitted by pmerkleplant, also found by Reigada and pauliax*
- [G-48] Redundant check *Submitted by tqts*
- [G-49] Numerous gas optimizations in SwapUtils.sol *Submitted by tqts*

- [G-50] Cache Reference To State Variables “currentDay, currentEra, emission” in _updateEmission (PublicSale.sol) *Submitted by yeOlde, also found by WatchPug*
- [G-51] Unnecessary “else if” in function vest (Vesting.sol) *Submitted by yeOlde*
- [G-52] Long Revert Strings *Submitted by yeOlde, also found by WatchPug and pants*
- [G-53] Unused Named Returns (PublicSale.sol) *Submitted by yeOlde, also found by WatchPug and pants*
- [G-56] _withdrawShare Can Be Rewritten To Be More Efficient (PublicSale.sol) *Submitted by yeOlde*
- [G-57] _processWithdrawal Can Be Rewritten To Be More Efficient (PublicSale.sol) *Submitted by yeOlde*
- [G-58] getEmissionShare Can Be Rewritten To Be More Efficient (PublicSale.sol) *Submitted by yeOlde*
- [G-59] From’ and ‘to’ tokens are read from storage multiple times in SwapUtils’s swap function *Submitted by hyh*
- [G-60] Multiple double storage reading _xp(self) function calls *Submitted by hyh*
- [G-61] Redundant hardhat console import *Submitted by pants, also found by WatchPug*
- [G-62] Use of uint8 for counter in for loop increases gas costs *Submitted by pants, also found by Reigada and pauliax*
- [G-64] Use bytes32 instead of string when possible *Submitted by pants*
- [G-65] Using ++i consumes less gas than i++ *Submitted by Reigada*

Disclosures

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

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

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

Top