Sushi Trident contest phase 1 Findings & Analysis Report

2021-11-22

Table of contents

Overview

About C4

Code 423n4 (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 Sushi Trident smart contract system written in Solidity. The code contest took place between September 16—September 29 2021.

Wardens

18 Wardens contributed reports to the Sushi Trident code contest (phase 1):

1. broccoli (shw and jonah1005)
2. cmichel
3. WatchPug (jtp and ming)
4. 0xsanson
5. GreyArt (hickuphh3 and itsmeSTYJ)
6. 0xRajeev
7. hack3r-0m
8. pauliax
9. hrkrshnn
10. GalloDaSballo
11. gpersoon

12. JMukesh
13. defsec
14. nikitastupin
15. t11s

This contest was judged by Alberto Cuesta Cañada.

Final report assembled by moneylegobatman and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 65 unique vulnerabilities and 110 total findings. All of the issues presented here are linked back to their original findings

Of these vulnerabilities, 16 received a risk rating in the category of HIGH severity, 10 received a risk rating in the category of MEDIUM severity, and 39 received a risk rating in the category of LOW severity.

C4 analysis also identified 20 non-critical recommendations and 25 gas optimizations.

Scope

The code under review can be found within the C4 Sushi Trident contest repository, and is composed of 58 smart contracts written in the Solidity programming language and includes 5,556 lines of Solidity code.

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 (16)

[H-01] Flash swap call back prior to transferring tokens in `indexPool`

Submitted by broccoli, also found by 0xsanson and cmichel

Impact

In the `IndexPool` contract, `flashSwap` does not work. The callback function is called prior to token transfer. The sender won't receive tokens in the callBack function. `ITridentCallee(msg.sender).tridentSwapCallback(context);`

`Flashswap` is not implemented correctly. It may need a migration to redeploy all `indexPools` if the issue is found after main-net launch. I consider this a high-risk issue.

Proof of Concept

IndexPool.sol#L196-L223

```
ITridentCallee(msg.sender).tridentSwapCallback(context);
// @dev Check Trident router has sent `amountIn` for skim into p
unchecked { // @dev This is safe from under/overflow - only logg
    require(_balance(tokenIn) >= amountIn + inRecord.reserve, "N
    inRecord.reserve += uint120(amountIn);
    outRecord.reserve -= uint120(amountOut);
}
_transfer(tokenOut, amountOut, recipient, unwrapBento);
```

Recommended Mitigation Steps

```
    _transfer(tokenOut, amountOut, recipient, unwrapBento);
    ITridentCallee(msg.sender).tridentSwapCallback(context);
    // @dev Check Trident router has sent `amountIn` for skim into p
    unchecked { // @dev This is safe from under/overflow - only logg
        require(_balance(tokenIn) >= amountIn + inRecord.reserve, "N
        inRecord.reserve += uint120(amountIn);
        outRecord.reserve -= uint120(amountOut);
    }
```

maxsam4 (Sushi) commented:

> Duplicate of https://github.com/code-423n4/2021-09-sushitrident-findings/issues/157 and https://github.com/code-423n4/2021-09-sushitrident-findings/issues/80

[H-02] Index Pool always swap to Zero

Submitted by broccoli, also found by 0xsanson, cmichel, and WatchPug

Impact

When an Index pool is initiated with two tokens A: B and the weight rate = 1:2, then no user can buy token A with token B.

The root cause is the error in pow. It seems like the dev tries to implement Exponentiation by squaring. IndexPool.sol#L286-L291

```
    function _pow(uint256 a, uint256 n) internal pure returns (uint2
      output = n % 2 != 0 ? a : BASE;
      for (n /= 2; n != 0; n /= 2) a = a * a;
      if (n % 2 != 0) output = output * a;
    }
```

There's no bracket for `for`.

The `IndexPool` is not functional. I consider this is a high-risk issue.

Proof of Concept

When we initiated the pool with 2:1.

```
deployed_code = encode_abi(["address[]","uint136[]","uint256"],
    (link.address, dai.address),
    (2*10**18,  10**18),
    10**13
])
```

No one can buy dai with link.

```
# (address tokenIn, address tokenOut, address recipient, bool ur
previous_balance = bento.functions.balanceOf(dai.address, admin)
swap_amount =  10**18

bento.functions.transfer(link.address, admin, pool.address, swap
pool.functions.swap(
    encode_abi(
        ['address', 'address', 'address', 'bool', 'uint256'],
        [link.address, dai.address, admin, False, swap_amount]
    )
).transact()
current_balance = bento.functions.balanceOf(dai.address, admin).
token_received = current_balance - previous_balance
# always = 0
print(token_received)
```

Recommended Mitigation Steps

The brackets of `for` were missed.

```
function _pow(uint256 a, uint256 n) internal pure returns (uint2
    output = n % 2 != 0 ? a : BASE;
    for (n /= 2; n != 0; n /= 2) {
        a = a * a;
        if (n % 2 != 0) output = output * a;
    }
}
```

[H-03] `IndexPool` pow overflows when `weightRatio` > 10.

Submitted by broccoli

Impact

In the `IndexPool` contract, pow is used in calculating price. (`IndexPool.sol` L255-L266). However, Pow is easy to cause overflow. If the `weightRatio` is large (e.g. 10), there's always overflow.

Lp providers can still provide liquidity to the pool where no one can swap. All pools need to redeploy. I consider this a high-risk issue.

Proof of concept

It's easy to trigger this bug by deploying a 1:10 `IndexPool`.

```
    deployed_code = encode_abi(["address[]","uint136[]","uint256"],
        (link.address, dai.address),
        (10**18, 10 * 10**18),
        10**13
    ])
    tx_hash = master_deployer.functions.deployPool(index_pool_factor
```

Transactions would be reverted when buying `link` with `dai`.

Recommended Mitigation Steps

The `weightRatio` is an 18 decimals number. It should be divided by `(BASE)^exp`. The scale in the contract is not consistent. Recommend the dev to check all the scales/ decimals.

maxsam4 (Sushi) confirmed

[H-04] IndexPool's `INIT_POOL_SUPPLY` is not fair.

Submitted by broccoli, also found by WatchPug

Impact

The `indexPool` mint `INIT_POOL_SUPPLY` to address 0 in the constructor. However, the value of the burned lp is decided by the first lp provider. According to the formula in `IndexPool.sol` L106.

`AmountIn = first_lp_amount / INIT_POOL_SUPPLY` and the burned lp worth = `AmountIn * (INIT_POOL_SUPPLY) / (first_lp_amount + INIT_POOL_SUPPLY)`. If a pool is not initialized with optimal parameters, it would be a great number of tokens been burn. All lp providers in the pool would receive less profit.

The optimal parameter is `10**8`. It's likely no one would initialize with `10**8` wei in most pools. I consider this is a high-risk issue.

Proof of concept

There are two scenarios that the first lp provider can do. The lp provider provides the same amount of token in both cases. However, in the first scenario, he gets about `10 ** 18 * 10**18` lp while in the other scenario he gets `100 * 10**18` lp.

```
    deposit_amount = 10**18
    bento.functions.transfer(link.address, admin, pool.address, depo
    bento.functions.transfer(dai.address, admin, pool.address, depos
    pool.functions.mint(encode_abi(
        ['address', 'uint256'],
        [admin, 10**8] # minimum
    )).transact()
    pool.functions.mint(encode_abi(
        ['address', 'uint256'],
        [admin, 10000000000009999 * 10** 20]
    )).transact()
```

```
    deposit_amount = 10**18
    bento.functions.transfer(link.address, admin, pool.address, depo
    bento.functions.transfer(dai.address, admin, pool.address, depos
    pool.functions.mint(encode_abi(
        ['address', 'uint256'],
        [admin, deposit_amount * 100]
    )).transact()
```

Recommended Mitigation Steps

Recommend to handle `INIT_POOL_SUPPLY` in uniswap-v2's way. Determine an optimized parameter for the user would be a better UX design.

[H-05] hybrid pool uses wrong `non_optimal_mint_fee`

Submitted by broccoli

Impact

When an lp provider deposits an imbalance amount of token, a swap fee is applied. `HybridPool` uses the same `_nonOptimalMintFee` as `constantProductPool`; however, since two pools use different AMM curve, the ideal balance is not the same. ref: `StableSwap3Pool.vy` L322-L337

Stable swap Pools are designed for 1B+ TVL. Any issue related to pricing/fee is serious. I consider this is a high-risk issue

Proof of Concept

- StableSwap3Pool.vy#L322-L337
- HybridPool.sol#L425-L441

Recommended Mitigation Steps

Calculate the swapping fee based on the stable swap curve. refer to StableSwap3Pool.vy#L322-L337.

maxsam4 (Sushi) confirmed

[H-06] `IndexPool`: Poor conversion from Balancer V1's corresponding functions

Submitted by GreyArt

Impact

A number of functions suffer from the erroneous conversion of Balancer V1's implementation.

- `_compute()` (equivalent to Balancer's [`bpow()`](#))
  - `if (remain == 0) output = wholePow;` when a return statement should be used instead.
- `_computeSingleOutGivenPoolIn()` (equivalent to Balancer's [`_calcSingleOutGivenPoolIn()`](#))
  - `tokenOutRatio` should be calculated with `_compute()` instead of `_pow()`
  - `zaz` should be calculated with `_mul()` instead of the native `*`
- `_pow()` (equivalent to Balancer's [`bpowi()`](#))
  - Missing brackets `{}` for the for loop causes a different interpretation
  - `_mul` should be used instead of the native `*`

Recommended Mitigation Steps

The fixed implementation is provided below.

```
function _computeSingleOutGivenPoolIn(
  uint256 tokenOutBalance,
  uint256 tokenOutWeight,
  uint256 _totalSupply,
  uint256 _totalWeight,
  uint256 toBurn,
  uint256 _swapFee
) internal pure returns (uint256 amountOut) {
    uint256 normalizedWeight = _div(tokenOutWeight, _totalWeight
    uint256 newPoolSupply = _totalSupply - toBurn;
    uint256 poolRatio = _div(newPoolSupply, _totalSupply);
    uint256 tokenOutRatio = _compute(poolRatio, _div(BASE, norma
    uint256 newBalanceOut = _mul(tokenOutRatio, tokenOutBalance)
    uint256 tokenAmountOutBeforeSwapFee = tokenOutBalance - newB
    uint256 zaz = _mul(BASE - normalizedWeight, _swapFee);
    amountOut = _mul(tokenAmountOutBeforeSwapFee, (BASE - zaz));
}

function _compute(uint256 base, uint256 exp) internal pure retur
  require(MIN_POW_BASE <= base && base <= MAX_POW_BASE, "INVALII
      uint256 whole = (exp / BASE) * BASE;
      uint256 remain = exp - whole;
      uint256 wholePow = _pow(base, whole / BASE);

      if (remain == 0) return wholePow;

      uint256 partialResult = _powApprox(base, remain, POW_PRECISION
      output = _mul(wholePow, partialResult);
    }

    function _pow(uint256 a, uint256 n) internal pure returns (uint2
      output = n % 2 != 0 ? a : BASE;
      for (n /= 2; n != 0; n /= 2) {
                  a = _mul(a, a);
        if (n % 2 != 0) output = _mul(output, a);
            }
    }
```

maxsam4 (Sushi) acknowledged

[H-07] `IndexPool.mint` The first liquidity provider is forced to supply assets in the same amount, which may cause a significant amount of fund loss

Submitted by WatchPug, also found by broccoli

When `reserve == 0`, `amountIn` for all the tokens will be set to the same amount: `ratio`, regardless of the weights, decimals and market prices of the assets.

The first liquidity provider may not be aware of this so that it may create an arbitrage opportunity for flashbots to take a significant portion of the value of The first liquidity provider's liquidity.

`IndexPool.sol#L93` L105

```
    /// @dev Mints LP tokens - should be called via the router after
    /// The router must ensure that sufficient LP tokens are minted
    function mint(bytes calldata data) public override lock returns
        (address recipient, uint256 toMint) = abi.decode(data, (addr
        uint120 ratio = uint120(_div(toMint, totalSupply));

        for (uint256 i = 0; i < tokens.length; i++) {
            address tokenIn = tokens[i];
            uint120 reserve = records[tokenIn].reserve;
            // @dev If token balance is '0', initialize with `ratio`
            uint120 amountIn = reserve != 0 ? uint120(_mul(ratio, re
            require(amountIn >= MIN_BALANCE, "MIN_BALANCE");
            // @dev Check Trident router has sent `amountIn` for ski
            unchecked {
                // @dev This is safe from overflow - only logged amo
                require(_balance(tokenIn) >= amountIn + reserve, "NO
                records[tokenIn].reserve += amountIn;
            }
            emit Mint(msg.sender, tokenIn, amountIn, recipient);
        }
        _mint(recipient, toMint);
        liquidity = toMint;
    }
```

Proof of Concept

Given:

- A `IndexPool` of 99% USDT and 1% WBTC;
- Alice is the first liquidity provider.
- Alice transfers 1e18 WBTC and 1e18 USDT to mint 100e18 of liquidity;
- Bob can use 100e18 USDT (~$100) to swap out most of the balance of WBTC.

Impact

A significant portion (>90% in the case above) of the user's funds can be lost due to arbitrage.

Recommendation

Consider allowing the first liquidity provider to use custom `amountIn` values for each token or always takes the MIN_BALANCE of each token.

[H-08] `HybridPool`'s reserve is converted to "amount" twice

Submitted by cmichel, also found by 0xsanson and WatchPug

The `HybridPool`'s reserves are stored as Bento "amounts" (not Bento shares) in `_updateReserves` because `_balance()` converts the current share balance to amount balances. However, when retrieving the `reserve0/1` storage fields in `_getReserves`, they are converted to amounts a second time.

Impact

The `HybridPool` returns wrong reserves which affects all minting/burning and swap functions. They all return wrong results making the pool eventually economically exploitable or leading to users receiving less tokens than they should.

POC

Imagine the current Bento amount / share price being `1.5`. The pool's Bento *share* balance being `1000`. `_updateReserves` will store a reserve of `1.5 * 1000 = 1500`. When anyone trades using the `swap` function, `_getReserves()` is called and multiplies it by `1.5` again, leading to using a reserve of 2250 instead of 1500. A higher reserve for the output token leads to receiving more tokens as the swap output. Thus the pool lost tokens and the LPs suffer this loss.

Recommended Mitigation Steps

Make sure that the reserves are in the correct amounts.

maxsam4 (Sushi) confirmed

[H-09] Unsafe cast in `IndexPool` mint leads to attack

Submitted by cmichel, also found by cmichel and pauliax

The `IndexPool.mint` function performs an unsafe cast of `ratio` to the `uint120` type:

```
uint120 ratio = uint120(_div(toMint, totalSupply));
```

Note that `toMint` is chosen by the caller and when choosing `toMint = 2**120 * totalSupply / BASE`, the `ratio` variable will be `2**120` and then truncated to 0 due to the cast.

This allows an attacker to mint LP tokens for free. They just need to choose the `ratio` such that the `amountIn = ratio * reserve / BASE` variable passes the `require(amountIn >= MIN_BALANCE, "MIN_BALANCE");` check. For example, when choosing `ratio = 2**120 * totalSupply / BASE + 1e16`, an attacker has to pay 1/100th of the current reserves but heavily inflates the LP token supply.

They can then use the inflated LP tokens they received in `burn` to withdraw the entire pool reserves.

POC

I created [this POC](#) that implements a hardhat test and shows how to steal the pool tokens:

Impact

An attacker can inflate the LP token pool supply and mint themselves a lot of LP tokens by providing almost no tokens themselves. The entire pool tokens can be stolen.

Recommended Mitigation Steps

Even though Solidity 0.8.x is used, type casts do not throw an error. A [`SafeCast` library](#) must be used everywhere a typecast is done.

[maxsam4 (Sushi) confirmed](#)

[H-10] `IndexPool` initial LP supply computation is wrong

Submitted by cmichel

The `IndexPool.constructor` function already mints `INIT_POOL_SUPPLY = 100 * 1e18 = 1e20` LP tokens to the zero address.

When trying to use the pool, someone has to provide the actual initial reserve tokens in `mint`. On the first `mint`, the pool reserves are zero and the token amount required to mint is just this `ratio` itself: `uint120 amountIn = reserve != 0 ? uint120(_mul(ratio, reserve)) : ratio;`

Note that the `amountIn` is **independent of the token** which does not make much sense. This implies that all tokens must be provided in equal "raw amounts", regardless of their decimals and value.

POC

Issue 1

Imagine I want to create a DAI/WBTC pool. If I want to initialize the pool with 100$ of DAI, `amountIn = ratio` needs to be `100*1e18=1e20` as DAI has 18 decimals. However, I now also need to supply `1e20` of WBTC (which has 8 decimals) and I'd need to pay `1e20/1e8 * priceOfBTC`, over a quadrillion dollars to match it with the 100$ of DAI.

Issue 2

Even in a pool where all tokens have the same decimals and the same value, like `USDC <> USDT`, it leads to issues:

- Initial minter calls `mint` with `toMint = 1e20` which sets `ratio = 1e20 * 1e18 / 1e20 = 1e18` and thus `amountIn = 1e18` as well. The total supply increases to `2e20`.
- Second minter needs to pay **less** tokens to receive the same amount of `1e18` LP tokens as the first minter. This should never be the case. `toMint = 1e20` => `ratio = 1e20 * 1e18 / 2e20 = 0.5e18`. Then `amountIn = ratio * reserve / 1e18 = 0.5*reserve = 0.5e18`. They only pay half of what the first LP provider had to pay.

Impact

It's unclear why it's assumed that the pool's tokens are all in equal value - this is *not* a StableSwap-like pool.

Any pool that uses tokens that don't have the same value and share the same decimals cannot be used because initial liquidity cannot be provided in an economically justifiable way.

It also leads to issues where the second LP supplier has to pay **less tokens** to receive the exact same amount of LP tokens that the initial minter receives. They can steal from the initial LP provider by burning these tokens again.

Recommended Mitigation Steps

Do not mint the initial token supply to the zero address in the constructor.

Do it like Uniswap/Balancer and let the first liquidity provider provide arbitrary token amounts, then mint the initial pool supply. If `reserve == 0`, `amountIn` should just take the pool balances that were transferred to this account.

In case the initial mint to the zero address in the constructor was done to prevent the "Uniswap-attack" where the price of a single wei of LP token can be very high and price out LPs, send a small fraction of this initial LP supply (~1000) to the zero address **after** it was minted to the first supplier in `mint`.

maxsam4 (Sushi) confirmed

[H-11] `ConstantProductPool.burnSingle` swap amount computations should use balance

Submitted by cmichel

The `ConstantProductPool.burnSingle` function is basically a `burn` followed by a `swap` and must therefore act the same way as calling these two functions sequentially.

The token amounts to redeem (`amount0`, `amount1`) are computed on the **balance** (not the reserve). However, the swap amount is then computed on the **reserves** and not the balance. The `burn` function would have updated the `reserve` to the balances and therefore `balance` should be used here:

```
    amount1 += _getAmountOut(amount0, _reserve0 - amount0, _reserve1
```

> ⚠️ The same issue occurs in the `HybridPool.burnSingle`.

Impact

For a burn, usually the `reserve` should equal the `balance`, however if any new tokens are sent to the contract and `balance > reserve`, this function will return slightly less swap amounts.

Recommended Mitigation Steps

Call `_getAmountOut` with the balances instead of the reserves: `_getAmountOut(amount0, balance0 - amount0, balance1 - amount1)`

maxsam4 (Sushi) confirmed:

> Please bump this to High sev. This bug can actually lead to loss of funds from the pool. The author found the right issue but failed to analyze the full impact. Regardless, I think they deserve "High" for pointing this out.

alcueca (judge) commented:

> This is what we come to C4 for 👏👏👏

[H-12] absolute difference is not calculated properly when a > b in MathUtils

Submitted by hack3r-0m, also found by broccoli

the difference is computed incorrectly when a > b. `MathUtils.sol` L22

As it only used in `within1` function, scope narrows down to where `difference(a, b) <= 1;` is exploitable.

cases where `difference(a, b) <= 1` should be true but is reported false:

- where b = a-1 (returned value is `type(uint256).max`)

cases where `difference(a, b) <= 1` should be false but is reported true:

- where a = `type(uint256)`.max and b = 0, it returns 1 but it should ideally return `type(uint256).max`

`within1` is used at the following locations:

- `HybridPool.sol` L359
- `HybridPool.sol` L383
- `HybridPool.sol` L413

It is possible to decrease the denominator and increase the value of the numerator (when calculating y) using constants and input to make `within1` fail

Mitigation:

Add `else` condition to mitigate it.

```
unchecked {
    if (a > b) {
        diff = a - b;
    }
    else {
        diff = b - a;
    }
}
```

maxsam4 (Sushi) confirmed

[H-13] Overflow in the `mint` function of `IndexPool` causes LPs' funds to be stolen

Submitted by broccoli, also found by WatchPug

Impact

It is possible to overflow the addition in the balance check (i.e., `_balance(tokenIn) >= amountIn + reserve`) in the mint function by setting the `amountIn` to a large amount. As a result, the attacker could gain a large number of LP tokens by not even providing any liquidity. The attacker's liquidity would be much greater than any other LPs, causing him could effectively steal others' funds by burning his liquidity (since the funds he receives are proportional to his liquidity).

Proof of Concept

- mint_overflow.js

Referenced code:

- IndexPool.sol L110

Recommended Mitigation Steps

Consider removing the `uncheck` statement to prevent integer overflows from happening.

maxsam4 (Sushi) acknowledged:

> FWIW The problem here isn't that we used unchecked but that we didn't cast amountIn to uint256. It's possible to overflow uint120 but not uint256.

[H-14] Incorrect usage of `_pow` in `_computeSingleOutGivenPoolIn` of `IndexPool`

Submitted by broccoli

Impact

The `_computeSingleOutGivenPoolIn` function of `IndexPool` uses the `_pow` function to calculate `tokenOutRatio` with the exponent in `WAD` (i.e., in 18 decimals of precision). However, the `_pow` function assumes that the given exponent `n` is not in `WAD`. (for example, `_pow(5, BASE)` returns `5 ** (10 ** 18)` instead of `5 ** 1`). The misuse of the `_pow` function could causes an integer overflow in the `_computeSingleOutGivenPoolIn` function and thus prevent any function from calling it.

Proof of Concept

Referenced code: IndexPool.sol#L279

Recommended Mitigation Steps

Change the `_pow` function to the `_compute` function, which supports exponents in `WAD`.

maxsam4 (Sushi) confirmed

[H-15] Incorrect multiplication in `_computeSingleOutGivenPoolIn` of `IndexPool`

Submitted by broccoli

Impact

The `_computeSingleOutGivenPoolIn` function of `IndexPool` uses the raw multiplication (i.e., `*`) to calculate the `zaz` variable. However, since both `(BASE - normalizedWeight)` and `_swapFee` are in `WAD`, the `_mul` function should be used instead to calculate the correct value of `zaz`. Otherwise, `zaz` would be `10 ** 18` times larger than the expected value and causes an integer underflow when calculating `amountOut`. The incorrect usage of multiplication prevents anyone from calling the function successfully.

Proof of Concept

Referenced code: IndexPool.sol#L282

Recommended Mitigation Steps

Change `(BASE - normalizedWeight) * _swapFee` to `_mul((BASE - normalizedWeight), _swapFee)`.

maxsam4 (Sushi) confirmed

[H-16] Funds in the pool could be stolen by exploiting `flashSwap` in `HybridPool`

Submitted by broccoli

Impact

An attacker can call the `bento.harvest` function during the callback function of a flash swap of the `HybridPool` to reduce the number of input tokens that he has to pay to the pool, as long as there is any unrealized profit in the strategy contract of the underlying asset.

Proof of Concept

1. The `HybridPool` accounts for the reserve and balance of the pool using the `bento.toAmount` function, which represents the actual amount of assets that the pool owns instead of the relative share. The value of `toAmount` could increase or decrease if the `bento.harvest` function is called (by anyone), depending on whether the strategy contract earns or loses money.
2. Supposing that the DAI strategy contract of `Bento` has a profit not accounted for yet. To account for the profit, anyone could call `harvest` on `Bento` with the corresponding parameters, which, as a result, increases the `elastic` of the DAI token.
3. Now, an attacker wants to utilize the unrealized profit to steal funds from a DAI-WETH hybrid pool. He calls `flashSwap` to initiate a flash swap from WETH to DAI. First, the pool transfers the corresponding amount of DAI to him, calls the `tridentSwapCallback` function on the attacker's contract, and expects that enough DAI is received at the end.
4. During the `tridentSwapCallback` function, the attacker calls `bento.harvest` to realize the profit of DAI. As a result, the pool's `bento.toAmount` increases, and the amount of DAI that the attacker has to pay to the pool is decreased. The attacker could get the same amount of ETH but paying less DAI by exploiting this bug.

Referenced code:

- `HybridPool.sol` L218-L220

- `HybridPool.sol` [L249-L250]
- `HybridPool.sol` [L272-L285]
- `BentoBoxV1Flat.sol` [L1105]
- `BentoBoxV1Flat.sol` [L786-L792]
- `BentoBoxV1Flat.sol` [L264-L277]

Recommended Mitigation Steps

Consider not using `bento.toAmount` to track the reservers and balances, but use `balanceOf` instead (as done in the other two pools).

[maxsam4 (Sushi) confirmed]:

> Stableswap needs to use `toAmount` balances rather shares to work. This issue allows skimming yield profits from the pool. There's no user funds at risk but still an issue.

> We plan on resolving this by using a fixed toElastic ratio during the whole swap.

Medium Risk Findings (10)

[M-01] No bar fees for `IndexPools`?

Submitted by 0xsanson, also found by pauliax

Impact

`IndexPool` doesn't collect fees for `barFeeTo`. Since this Pool contains also a method `updateBarFee()`, probably this is an unintended behavior. Also without a fee, liquidity providers would probably ditch `ConstantProductPool` in favor of `IndexPool` (using the same two tokens with equal weights), since they get all the rewards. This would constitute an issue for the ecosystem.

Recommended Mitigation Steps

Add a way to send `barFees` to `barFeeTo`, same as the other pools.

maxsam4 (Sushi) confirmed

[M-02] `ConstantProductPool` & `HybridPool` : Adding and removing unbalanced liquidity yields slightly more tokens than swap

Submitted by GreyArt, also found by broccoli

Impact

A mint fee is applied whenever unbalanced liquidity is added, because it is akin to swapping the excess token amount for the other token.

However, the current implementation distributes the minted fee to the minter as well (when he should be excluded). It therefore acts as a rebate of sorts.

As a result, it makes adding and removing liquidity as opposed to swapping directly (negligibly) more desirable. An example is given below using the Constant Product Pool to illustrate this point. The Hybrid pool exhibits similar behaviour.

Proof of Concept

1. Initialize the pool with ETH-USDC sushi pool amounts. As of the time of writing, there is roughly 53586.556 ETH and 165143020.5295 USDC.
2. Mint unbalanced LP with 5 ETH (& 0 USDC). This gives the user `138573488720892 / 1e18` LP tokens.
3. Burn the minted LP tokens, giving the user 2.4963 ETH and 7692.40 USDC. This is therefore equivalent to swapping 5 - 2.4963 = 2.5037 ETH for 7692.4044 USDC.
4. If the user were to swap the 2.5037 ETH directly, he would receive 7692.369221 (0.03 USDC lesser).

Recommended Mitigation Steps

The mint fee should be distributed to existing LPs first, by incrementing `_reserve0` and `_reserve1` with the fee amounts. The rest of the calculations follow after.

`ConstantProductPool`

```
    (uint256 fee0, uint256 fee1) = _nonOptimalMintFee(amount0, amour
    // increment reserve amounts with fees
    _reserve0 += uint112(fee0);
    _reserve1 += uint112(fee1);
    unchecked {
        _totalSupply += _mintFee(_reserve0, _reserve1, _totalSupply)
    }
    uint256 computed = TridentMath.sqrt(balance0 * balance1);
    ...
    kLast = computed;
```

`HybridPool`

```
    (uint256 fee0, uint256 fee1) = _nonOptimalMintFee(amount0, amour
    // increment reserve amounts with fees
    _reserve0 += uint112(fee0);
    _reserve1 += uint112(fee1);
    uint256 newLiq = _computeLiquidity(balance0, balance1);
    ...
```

maxsam4 (Sushi) confirmed

[M-03] Router would fail when adding liquidity to index Pool

Submitted by broccoli

Impact

`TridentRouter` is easy to fail when trying to provide liquidity to an index pool.

Users would not get extra lp if they are not providing lp at the pool's spot price. It's the same design as uniswap v2. However, uniswap's v2 handle's the dirty part.

Users would not lose tokens if they use the router (`UniswapV2Router02.sol` L61-L76).

However, the router wouldn't stop users from transferring extra tokens (`TridentRouter.sol` L168-L190).

Second, the price would possibly change when the transaction is confirmed. This would be reverted in the index pool.

Users would either transfer extra tokens or fail. I consider this is a medium-risk issue.

Proof of Concept

TridentRouter.sol#L168-L190

A possible scenario:

There's a BTC/USD pool. BTC = 50000 USD.

1. A user sends a transaction to transfer 1 BTC and 50000 USD.
2. After the user send a transaction, a random bot buying BTC with USD.
3. The transaction at step 1 is mined. Since the BTC price is not 50000 USD, the transaction fails.

Recommended Mitigation Steps

Please refer to the uniswap v2 router in `UniswapV2Router02.sol` L61-L76

The router should calculate the optimal parameters for users.

maxsam4 (Sushi) confirmed

[M-04] Router's `complexPath` percentagePaths don't work as expected

Submitted by cmichel

The `TridentRouter.complexPath` function allows splitting a trade result into several buckets and trade them in a different pool each. The distribution is defined by the `params.percentagePath[i].balancePercentage` values:

```
    for (uint256 i; i < params.percentagePath.length; i++) {
        uint256 balanceShares = bento.balanceOf(params.percentagePat
        uint256 transferShares = (balanceShares * params.percentageP
        bento.transfer(params.percentagePath[i].tokenIn, address(thi
        isWhiteListed(params.percentagePath[i].pool);
        IPool(params.percentagePath[i].pool).swap(params.percentageP
    }
```

However, the base value `bento.balanceOf(params.percentagePath[i].tokenIn, address(this));` is recomputed after each iteration instead of caching it before the loop.

This leads to not all tokens being used even though the percentages add up to 100%.

POC

Assume I want to trade 50% DAI to WETH and the other 50% DAI to WBTC. In the first iteration, `balanceShares` is computed and then 50% of it is swapped in the first pool.

However, in the second iteration, `balanceShares` is updated again, and only 50% **of the remaining** (instead of the total) balance, i.e. 25%, is traded.

The final 25% are lost and can be skimmed by anyone afterwards.

Impact

Users can lose their funds using `complexPath`.

Recommended Mitigation Steps

Cache the `balanceShares` value once before the second `for` loop starts.

sarangparikh22 (Sushi) disputed:

> This is not the correct way to calculate the complexPath swap parameters. For instance, if we need to swap 50% DAI to WETH and the other 50% DAI to WBTC, we would keep percentages as 50 and 100, instead of 50-50 as described above. If the user enters wrong percentages, they would loose funds.

alcueca (judge) commented:

> The format for entering the percentages is not documented. Sustained as Sev 2 as the lack of documentation on this parameter could lead to loss of funds.

[M-05] `_depositToBentoBox` sometimes uses both ETH and WETH

Submitted by cmichel, also found by 0xRajeev

The `TridentRouter._depositToBentoBox` function only uses the `ETH` in the contract if it's higher then the desired `underlyingAmount` (`address(this).balance >= underlyingAmount)`).

Otherwise, the ETH is ignored and the function uses WETH from the user.

Impact

Note that the `underlyingAmount = bento.toAmount(wETH, amount, true)` is computed from the Bento share price and it might happen that it increases from the time the transaction was submitted to the time the transaction is included in a block. In that case, it might completely ignore the sent `ETH` balance from the user and in addition transfer the same amount of `WETH` from the user.

The user can lose their `ETH` deposit in the contract.

Recommended Mitigation Steps

Each batch must use `refundETH` at the end.

Furthermore, we recommend still depositing `address(this).balance` ETH into Bento and if it's less than `underlyingAmount` use `WETH` only for **the remaining token difference.**

[maxsam4 (Sushi) acknowledged]()

[M-06] `withdrawFromWETH` always reverts

Submitted by cmichel

The `TridentHelper.withdrawFromWETH` (used in `TridentRouter.unwrapWETH`) function performs a low-level call to `WETH.withdraw(amount)`.

It then checks if the return `data` length is more or equal to `32` bytes, however `WETH.withdraw` returns `void` and has a return value of `0`. Thus, the function always reverts even if `success == true`.

```
function withdrawFromWETH(uint256 amount) internal {
    // @audit WETH.withdraw returns nothing, data.length always
    require(success && data.length >= 32, "WITHDRAW_FROM_WETH_FA
}
```

Impact

The `unwrapWETH` function is broken and makes all transactions revert. Batch calls to the router cannot perform any unwrapping of WETH.

Recommended Mitigation Steps

Remove the `data.length >= 32` from the require and only check if `success` is true.

sarangparikh22 (Sushi) confirmed

[M-07] `HybridPool`'s `flashSwap` sends entire fee to `barFeeTo`

Submitted by cmichel, also found by 0xsanson

The `HybridPool.flashSwap` function sends the entire trade fees `fee` to the `barFeeTo`. It should only send `barFee * fee` to the `barFeeTo` address.

Impact

LPs are not getting paid at all when this function is used. There is no incentive to provide liquidity.

Recommended Mitigation Steps

The `flashSwap` function should use the same fee mechanism as `swap` and only send `barFee * fee / MAX_FEE` to the `barFeeTo`. See `_handleFee` function.

- [maxsam4 (Sushi) confirmed]()

[M-08] Rounding errors will occur for tokens without decimals

Some rare tokens have 0 decimals: https://etherscan.io/token/0xcc8fa225d80b9c7d42f96e9570156c65d6caaa25

For these tokens, small losses of precision will be amplified by the lack of decimals.

Consider a constant product pool with 1000 of token0 (with no decimals), and 1000 of token1 (also with no decimals). Suppose I swap n= 1,2,3,4 of token0 to token1. Then my output amount of token1 will be 0,1,2,3.

If token0/1 are valuable than I will be losing 100%, 50%, 33%, 25% of my trade to rounding. Currently there is no valuable token with 0 decimals, but there may be in the future.

Rounding the final `getAmountOut` division upwards would fix this.

[maxsam4 (Sushi) commented]():

> Acceptable risk. We can't do anything if the token itself doesn't have decimals. We don't create synthetic assets and fractionalize such tokens ourselves.

[M-09] Approximations may finish with inaccurate values

Submitted by 0xsanson, also found by broccoli

Impact

In `HybridPool.sol`, functions `_computeLiquidityFromAdjustedBalances`, `_getY` and `_getYD` may finish before approximation converge, since it's limited by

`MAX_LOOP_LIMIT` iterations. In this situation the final estimated value will still be treated as correct, even though it could be relatively inaccurate.

Recommended Mitigation Steps

Consider reverting the transactions if this doesn't occur. See https://blog.openzeppelin.com/saddle-contracts-audit/ issue [M03], with their relative fix.

[M-10] Users are susceptible to back-running when depositing ETH to `TridenRouter`

Submitted by broccoli

Impact

The `_depositToBentoBox` and `_depositFromUserToBentoBox` allow users to provide ETH to the router, which is later deposited to the `bento` contract for swapping other assets or providing liquidity. However, in these two functions, the input parameter does not represent the actual amount of ETH to deposit, and users have to calculate the actual amount and send it to the router, causing a back-run vulnerability if there are ETH left after the operation.

Proof of Concept

1. A user wants to swap ETH to DAI. He calls `exactInputSingleWithNativeToken` on the router with the corresponding parameters and `params.amountIn` being 10. Before calling the function, he calculates `bento.toAmount(wETH, 10, true) = 15` and thus send 15 ETH to the router.
2. However, at the time when his transaction is executed, `bento.toAmount(wETH, amount, true)` becomes to `14`, which could happen if someone calls `harvest` on `bento` to update the `elastic` value of the `wETH` token.
3. As a result, only 14 ETH is transferred to the pool, and 1 ETH is left in the router. Anyone could back-run the user's transaction to retrieve the remaining 1 ETH from the router by calling the `refundETH` function.

Referenced code: [TridentRouter.sol#L318-L351]()

Recommended Mitigation Steps

Directly push the remaining ETH to the sender to prevent any ETH left in the router.

maxsam4 (Sushi) confirmed:

> I think it’s low risk because it’s basically arbitrage and we have protection for the user in terms of “minOutputAmount”. I will be reworking ETH handling to avoid this issue completely.

alcueca (judge) commented:

> It’s a loss of funds, not arbitrage. It should be prevented or documented. Sustained.

Low Risk Findings (39)

- [L-01] unchecked use of optional function “decimals” of erc20 standards *Submitted by hack3r-0m*
- [L-02] `ConstantProductPool` lacks zero check for maserDeployer *Submitted by hack3r-0m*
- [L-03] HybridPool.sol lacks zero check for maserDeployer *Submitted by hack3r-0m*
- [L-04] `TridentERC20.sol` Possible replay attacks on `permit` function in case of a future chain split *Submitted by nikitastupin, also found by 0xRajeev, 0xsanson, broccoli, and t11s*
- [L-05] Reset cachedPool ? *Submitted by gpersoon*
- [L-06] Missing validation of recipient argument could indefinitely lock owner role *Submitted by defsec*
- [L-07] # Hybrid Pool underflow when a < 100 *Submitted by broccoli*
- [L-08] `HybridPool` : SwapCallback should be done regardless of data.length *Submitted by GreyArt*
- [L-09] Missing invalid token check against pool address *Submitted by 0xRajeev*

- [L-10] `barFee` handled incorrectly in `flashSwap` (or swap) *Submitted by OxRajeev*
- [L-11] Strict bound in reserve check of Hybrid Pool *Submitted by OxRajeev*
- [L-12] Unlocked Solidity compiler pragma is risky *Submitted by OxRajeev, also found by broccoli, hrkrshnn, and JMukesh*
- [L-13] Missing zero-address checks *Submitted by OxRajeev*
- [L-14] Missing timelock for critical contract setters of privileged roles *Submitted by OxRajeev*
- [L-15] Unconditional setting of boolean/address values is risky *Submitted by OxRajeev*
- [L-16] Timelock between new owner transfer+claim will reduce risk *Submitted by OxRajeev*
- [L-17] Allowing direct single-step ownership transfer even as an option is risky *Submitted by OxRajeev*
- [L-18] Missing contract existence check may cause silent failures of token transfers *Submitted by OxRajeev*
- [L-19] `IndexPool` should check that tokens are supported *Submitted by cmichel*
- [L-20] Several low-level calls don't check the success return value *Submitted by cmichel*
- [L-21] `ConstantProductPool` bar fee computation seems wrong *Submitted by cmichel*
- [L-22] `ConstantProductPool` mint liquidity computation should include fees *Submitted by cmichel*
- [L-23] `HybridPool` 's `flashSwap` does not always call callback *Submitted by cmichel*
- [L-24] The functions `refundETH` and `unwrapWETH` is generalized-front-runnable *Submitted by hrkrshnn*
- [L-25] Lack of checks for address and amount in `TridentERC20._mint` *Submitted by GalloDaSballo*
- [L-26] Lack of checks for address and amount in `TridentERC20._mint` *Submitted by GalloDaSballo*

- [L-27] Lack of address validation in `MasterDeployer.setMigrator` *Submitted by GalloDaSballo*
- [L-28] Consider using solidity 0.8.8
- [L-29] lack of input validation in `Transfer()` and `TransferFrom()` *Submitted by JMukesh*
- [L-30] `_powApprox` : unbounded loop and meaning *Submitted by Oxsanson*
- [L-31] `HybridPool` 's wrong amount to balance conversion *Submitted by Oxsanson*
- [L-32] `_computeLiquidityFromAdjustedBalances` order of operations can be improved *Submitted by Oxsanson*
- [L-33] Wrong initialization of `blockTimestampLast` in `ConstantProductPool` *Submitted by broccoli*
- [L-34] Oracle Initialization
- [L-35] Docs disagrees with index pool code
- [L-36] Division by zero in `_computeLiquidityFromAdjustedBalances` of `HybridPool` *Submitted by broccoli*
- [L-37] `_getY` and `_getYD` math operations can be reordered *Submitted by Oxsanson*
- [L-38] Division and division in `_getY` of `HybridPool` *Submitted by broccoli*
- [L-39] Incorrect comparison in the `_updateReserves` function of `HybridPool` *Submitted by broccoli*

Non-Critical Findings (20)

- [N-01] Events not emitted while changing state variables in constructor *Submitted by hack3r-0m*
- [N-02] `ConstantProductPool` : Move minting of MIN_LIQUIDITY after checks *Submitted by GreyArt*
- [N-03] Unused constants could indicate missing logic or redundant code *Submitted by 0xRajeev, also found by GreyArt*
- [N-04] `TridentRouter.isWhiteListed()` Misleading name *Submitted by WatchPug*

- [N-05] TridentERC20 does not emit Approval event in `transferFrom` *Submitted by cmichel*
- [N-06] `ConstantProductPool.getAmountOut` does not verify token *Submitted by cmichel*
- [N-07] `HybridPool` missing positive token amount checks for initial mint *Submitted by cmichel*
- [N-08] `MAX_FEE_SQUARE` dependency on `MAX_FEE` *Submitted by pauliax*
- [N-09] Emit events when setting the values in constructor *Submitted by pauliax*
- [N-10] Inclusive check of `type(uint128).max` *Submitted by pauliax*
- [N-11] Consider avoiding low level calls to MasterDeployer *Submitted by hrkrshnn*
- [N-12] Functions that can be made external *Submitted by hrkrshnn*
- [N-13] Style issues *Submitted by pauliax*
- [N-14] Lack of address validation in `MasterDeployer.addToWhitelist` *Submitted by GalloDaSballo*
- [N-15] Using interfaces instead of selectors is best practice
- [N-16] Follow Curve's convention: `_getYD` and `_getY` *Submitted by 0xsanson*
- [N-17] View functions in Hybrid Pool Contract Pool need better documentation
- [N-18] Inconsistent tokens sent to `barFeeTo` *Submitted by 0xsanson*
- [N-19] Unnecessary condition on `_processSwap` of `HybridPool` *Submitted by broccoli*
- [N-20] Similarly initialized weight thresholds may cause unexpected deployment failures *Submitted by 0xRajeev*

Gas Optimizations (25)

- [G-01] Safe gas on `_powApprox` *Submitted by gpersoon*
- [G-02] Use parameter `_blockTimestampLast` in `_update()` *Submitted by gpersoon, also found by 0xsanson*
- [G-03] Consider unlocking pool only upon initial mint *Submitted by GreyArt*

- [G-04] `ConstantProductPool` : Unnecessary mod before casting to uint32 *Submitted by GreyArt*
- [G-05] `IndexPool` : Redundant MAX_WEIGHT *Submitted by GreyArt*
- [G-06] `TridentOwnable` : `pendingOwner` should be set to address(1) if direct owner transfer is used *Submitted by GreyArt*
- [G-07] Replace multiple calls with a single new function call *Submitted by OxRajeev*
- [G-08] Unused code can be removed to save gas *Submitted by OxRajeev*
- [G-09] Use of unchecked can save gas where computation is known to be overflow/underflow safe *Submitted by OxRajeev*
- [G-10] Gas: `HybridPool._computeLiquidityFromAdjustedBalances` should return early *Submitted by cmichel, also found by OxRajeev*
- [G-11] Avoiding initialization of loop index can save a little gas *Submitted by OxRajeev*
- [G-12] Caching in local variables can save gas *Submitted by OxRajeev*
- [G-13] Gas: `HybridPool` unnecessary `balance` computations *Submitted by cmichel*
- [G-14] Use of `ecrecover` is susceptible to signature malleability *Submitted by OxRajeev*
- [G-15] Caching the length in for loops *Submitted by hrkrshnn*
- [G-16] Consider using custom errors instead of revert strings *Submitted by hrkrshnn*
- [G-17] Consider changing the `_deployData` architecture *Submitted by hrkrshnn*
- [G-18] Caching the storage read to `tokens.length` *Submitted by hrkrshnn*
- [G-19] Caching a storage load in TridentERC20 *Submitted by hrkrshnn*
- [G-20] Unused state variable `barFee` and _barFeeTo in `IndexPool` *Submitted by hrkrshnn*
- [G-21] Consider putting some parts of `_div` in unchecked *Submitted by hrkrshnn*

- [G-22] Use `calldata` instead of `memory` for function parameters *Submitted by hrkrshnn*
- [G-23] Cache array length in for loops can save gas *Submitted by WatchPug*
- [G-24] Cache storage variable in the stack can save gas *Submitted by WatchPug*
- [G-25] Using 10**X for constants isn't gas efficient

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.