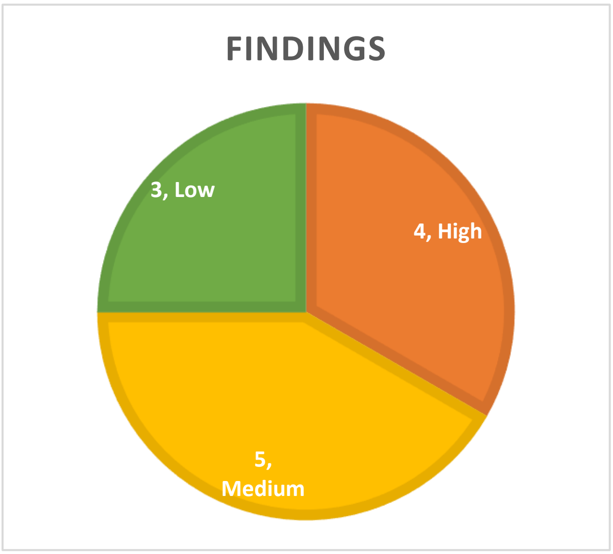
Trust Security

Smart Contract Audit

Baseline Protocol

27/02/2024

Executive summary

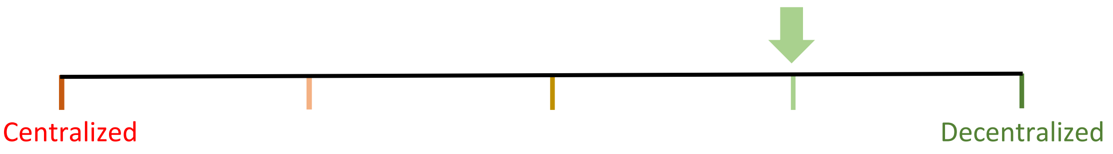


Category	Financial assets
Audited file count	10
Lines of Code	1657
Auditor	Jeiwan, 100proof
Time period	05-16/02/2024

Findings

Severity	Total	Fixed	Acknowledged
High	4	4	-
Medium	5	5	-
Low	3	3	-

Centralization score



Signature

Digitally signed by Trust
Date: 2024.02.27 21:12:15 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	19/02/2024	Client report
0.2	27/02/2024	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- BaselineFactory.sol
- Baseline.sol
- bAsset.sol
- BlastClaimer.sol
- Core.sol
- CreditFacility.sol
- IBlast.sol
- LiquidityManager.sol
- MarketMaking.sol
- preAsset.sol

Repository details

- **Repository URL:** https://github.com/0xBaseline/baseline-protocol
- **Commit hash:** 07193a58c3fba14dafac8528713fe2aa48aa0d5c
- **Mitigation review commit hash:** cb75fd91450b8bb10eb0840c428bca1cdb290bad

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

After spending many years working as a web developer and studying blockchain technologies in his free time, Jeiwan started his full-time smart contracts security journey in September 2022. Since then, he has participated in more than 50 auditing contests on Code4rena and Sherlock, where he took multiple Top 5 places competing with the best auditors in the field.

Jeiwan is the author of Uniswap V3 Development Book. Thanks to his deep knowledge of Uniswap, Jeiwan specializes in projects that integrate or extend Uniswap, as well as any other AMM.

100proof transitioned into smart contract security after many years as a software developer. He is interested in the application of formal methods to software correctness but believes a solid practical understanding of security is necessary for them to have any meaningful impact. Since finding a critical bug in Kyber Network's Elastic Pools, he has specialized in Uniswap V3-like concentrated liquidity AMMs.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	**Rating**	**Comments**
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks
Documentation	**Moderate**	Project is mostly very well documented; the code, however, lacks documentation.
Best practices	**Good**	Project uses common best practices to reduce security risks.
Centralization risks	**Moderate**	Project is mostly decentralized and permissionless.

Findings

High severity findings

TRST-H-1 Debt can be forced on arbitrary users

- **Category:** Access control
- **Source:** Baseline.sol#L78
- **Status:** Fixed

Description

The *Baseline.borrow()* function allows borrowing for an arbitrary user who has approved the bAsset to the contract: the function transfers bAssets from the caller-specified address (CreditFacility.sol#L161). Since approving the maximal token amount is a common practice, any user who has previously approved their bAssets to the contract (e.g. to take a credit) can be forced into a debt of up to their bAsset balance or the approve amount. Since borrowing happens at a price below the floor price (due to the interest paid when borrowing), taking a credit on behalf of another user will force them to lose some amount of funds.

Recommended mitigation

In the *Baseline.borrow()*, consider disallowing borrowing for an arbitrary address. The function should create a credit only for the caller.

Team response

Fixed in commit 9804df5.

Mitigation review

Fixed by transferring bAssets from the caller, not from the specified address. Due to how borrowing is implemented in the protocol, borrowing on behalf of another user (given that the collateral is paid by the caller) bears no risks for the user.

TRST-H-2 Repaying for an arbitrary user can cause a loss of funds

- **Category:** Access control
- **Source:** Baseline.sol#L86
- **Status:** Fixed

Description

The *Baseline.repay()* function allows repaying a credit for an arbitrary user address. This is a common practice in lending protocols to allow repaying debts for someone else. However, the function pulls the reserve funds from the credit owner, not the caller (CreditFacility.sol#L208) .

As a result, the function can be used to force a repaying of anyone’s credit, which can result in a loss profit and/or a partial loss of funds.

Recommended mitigation

In the *Baseline.repay()*, consider pulling the reserve funds only from the caller, while returning the collateral funds to the credit owner.

Team response

Fixed in commit 8faabe3.

Mitigation review

Fixed as recommended.

TRST-H-3 *BaselineFactory.deploy()* allows deployment of malicious Uniswap pools

- **Category:** Integration issues
- **Source:** BaselineFactory.sol#L35
- **Status:** Fixed

Description

When a new *Baseline* contract is deployed via the *BaselineFactory.deploy()*, a Uniswap V3 pool is created via a factory (Baseline.sol#L64-L67). However, the address of the factory contract that deploys the pool is provided by the user to the *BaselineFactory.deploy()* function (BaselineFactory.sol#L35), and it can be an arbitrary address. The protocol doesn't guarantee that a provided factory address is an authentic Uniswap factory that deploys authentic Uniswap V3 pools.

As a result, a malicious actor can deploy (via *BaselineFactory*) an authentic *Baseline* contract that will integrate with a malicious Uniswap V3 pool. Such pool, for example, can act as an authentic pool and have a function that will allow the malicious actor to withdraw all reserves from the contract to their address. Since the *Baseline* is deployed via the official *BaselineFactory* it'll have the authenticity of an official *Baseline* contract that uses an authentic Uniswap V3 pool.

Recommended mitigation

Consider making the Uniswap V3 factory address immutable in *BaselineFactory*. The address should be chosen by the protocol team to guarantee that the Uniswap V3 pool deployed with *Baseline* is not malicious.

Team response

Fixed in commit dbf008c.

Mitigation review

Fixed as recommended.

TRST-H-4 Credit duration is reduced during credit extension

- **Category:** Logical flaws
- **Source:** **CreditFacility.sol#L369**

- **Status:** Fixed

Description

When borrowing, there are multiple scenarios how the credit expiration date can be computed:

1. If it's a new credit, its expiration timestamp is computed starting from **block.timestamp** (CreditFacility.sol#L362).
2. If an existing credit is increased (without increasing its duration), the expiration timestamp remains unchanged (CreditFacility.sol#L365-L366).
3. If duration is increased for an existing credit, the old expiration timestamp is increased by the specified number of days (CreditFacility.sol#L368-L370).

In the latter case, 1 day is subtracted from the old expiration timestamp. However, this shouldn't be done since the old credit duration already includes the final day: its timeslot points at the last second of the day (CreditFacility.sol#L403-L405). Thus, subtracting 1 day causes an overlapping of the old and the new durations, making the final duration shorter by 1 day.

As a result, extended credits will expire earlier than expected, which might not allow their owners to repay or re-extend them in time, causing a loss of collateral.

Recommended mitigation

When increasing the duration of an existing credit, consider not subtracting 1 day from the starting timeslot. The interest for the subtracted 1 day has already been paid when the credit was created.

Team response

Fixed in commit 7ac067b.

Mitigation review

Fixed as recommended.

Medium severity findings

TRST-M-1 Loss of precision in the current price calculation

- **Category:** Arithmetic errors
- **Source:** MarketMaking.sol#L288
- **Status:** Fixed

Description

In the *MarketMaking._calculateCapacity()* function, the current spot price in the underlying Uniswap V3 pool is obtained by converting the current tick to a square-root ratio (MarketMaking.sol#L288). However, ticks cannot be converted to prices without losing precision because ticks are integers and prices are decimals with high precision. With each tick being a 0.01% (or 1 basis point) price movement, this results in a price calculation error of up to 0.01%.

The amount of tokens that can be absorbed by the floor and anchor positions can be slightly miscalculated, which will impact the rebalancing process, which plays a crucial role in maintaining the stability of the protocol.

Recommended mitigation

Instead of converting **activeTick_** to a price, consider passing and using the current square root price as returned by the *LiquidityManager._getActivePriceTick()* function.

Team response

Fixed in commit 316c077.

Mitigation review

Fixed as recommended.

TRST-M-2 Inaccurate borrow estimation for an account with a credit

- **Category:** Logic flaws
- **Source:** CreditFacility.sol#L78-L91
- **Status:** Fixed

Description

The *CreditFacility.estimateBorrow()* allows users to estimate their credit before borrowing. However, the function only works for new credits, while actual borrowing allows to increase and/or extend a credit.

As a result, *CreditFacility.estimateBorrow()* will return wrong credit principal and interest when called by a user who already has an open credit. Specifically, it returns wrong values when used to compute the extension or an increase of a credit.

Recommended mitigation

In the *CreditFacility.estimateBorrow()* function, consider taking into account the callers credit account and copying the principal and interest computations from the *CreditFacility._borrow()*

function. To avoid the code duplication, *CreditFacility._borrow()* can call *CreditFacility.estimateBorrow()* for the credit calculations.

Team response

Fixed in commit e898bb8.

Mitigation review

Fixed as recommended.

TRST-M-3 *BAsset* address cannot be reliably precomputed

- **Category:** Address derivation
- **Source: BaselineFactory.sol#L46, Baseline.sol#L56**
- **Status:** Fixed

Description

The distribution of BAssets in return for the funds collected during the pre-sale phase requires the precomputing of the BAsset address before the *Baseline* contract is deployed (preAsset.sol#L164-L168). However, the address cannot be reliably precomputed because its derivation includes a CREATE opcode address derivation.

The BAsset token is deployed when the *Baseline* is deployed (Baseline.sol#L55-L56). This deployment uses the CREATE2 opcode with a salt to guarantee a deterministic address derivation process. As per EIP-1014, the preimage includes the deployer's address, which is the address of the *Baseline* contract. However, the contract's address is not deterministic: it's deployed via *BaselineFactory.deploy()* using the CREATE opcode, which address derivation process depends on the deployment nonce of the deployer. Since anyone is allowed to deploy *Baseline* contracts via *BaselineFactory.deploy()*, there's no way to guarantee that the precomputed BAsset address will remain correct by the time the distribution of BAssets happens.

As a result, deploying and initializing a *Baseline* contract with a pre-sale phase may result in a revert or cause a distribution of wrong BAssets.

Recommended mitigation

When deploying *Baseline* contract via *BaselineFactory.deploy()*, consider using the CREATE2 opcode with the same salt that's used to deploy BAsset.

Team response

Fixed in commit dbf008c.

Mitigation review

Fixed as recommended.

TRST-M-4 *BAsset* deployment salt collision allows hijacking of Baseline contract

- **Category:** Frontrunning attacks

- **Source:** Baseline.sol#L56
- **Status:** Fixed

Description

When deploying a *Baseline* contract via the BaselineFactory.deploy(), a salt parameter is specified by the caller (BaselineFactory.sol#L34) to derivate a unique and pre-computed address for the BAsset contract (Baseline.sol#L56). Since the *BaselineFactory.deploy()* can be called by anyone and since the salt in the BAsset deployment is used as is, there's a possibility of salt collisions.

When a salt collision happens accidentally, the *Baseline* constructor will revert because there'll already be a Uniswap V3 pool for the pair of tokens (Baseline.sol#L64-L67).

However, there's also a possibility of a salt being leaked or intercepted from a well-intentioned deployer (e.g. a team that deploys a *Baseline* contract with a presale phase that has collected some user funds). The salt can also be hijacked directly in public mempool deployments. In this scenario, a malicious actor can:

1. set their address as the fee receiver (Baseline.sol#L58) or the Blast yield claimer (Baseline.sol#L70);
2. set a different initial tick (MarketMaking.sol#L35-L40) and/or a different initial supply of bAssets (MarketMaking.sol#L42-L44) to disrupt the contract;
3. specify a malicious router address to steal the presale funds during the swapping (MarketMaking.sol#L50-L63);

Recommended mitigation

In the *BaselineFactory.deploy()* function, consider concatenating the user-specified salt with the address of the sender, and using the resulting salt in the BAsset deployment. This will make off-chain BAsset address derivation slightly more complicated (it'll need to concatenate the deployer's address) but it'll protect from salt collisions.

Team response

Fixed in commit dbf008c.

Mitigation review

Fixed as recommended.

TRST-M-5 Credit interest is increased due to a miscalculation

- **Category:** Arithmetic errors
- **Source:** **CreditFacility.sol#L99**
- **Status:** Fixed

Description

When computing the interest amount of a credit, the interest percentage is applied to the entire credit amount (CreditFacility.sol#L99). However, this amount is later split into the principal and the interest of the credit (CreditFacility.sol#L138). As a result, the interest amount, in addition to the principal, is subject to the interest fee, which causes an overcharging of credit holders. The actual annual interest is higher (~4.16%) than the expected one (~4%).

Recommended mitigation

In the *CreditFacility.getInterest()* function, consider applying **INTEREST_PER_DIEM** as the reverse percentage, to compute the correct interest amount. The **credit_** argument should be seen as the after-interest credit size (which is the credit principal + the interest). The interest should be taken only on the principal part and should make the difference between **credit_** and the before-interest credit size (which is the credit principal).

Additionally, consider improving the tests to ensure that this invariant holds true: the ratio of the interest over the principal of a credit should always be equal the ~4% annual interest rate.

Team response

Fixed in commit 5b3cf46.

Mitigation review

Code documentation was updated to reflect the actual interest rate. The interest is still charged on the entire credit amount.

Low severity findings

TRST-L-1 *bAsset* shouldn't be allowed to change in *preAsset*

- **Category:** Validation issues
- **Source:** preAsset.sol#L166
- **Status:** Fixed

Description

The *preAsset.setBAsset()* allows setting and changing the BAsset address at any time. However, this shouldn't be allowed because the address is used during claiming: using the *preAsset.claim()* function, pre-sale buyers can claim their share of BAsset. If the address has been changed to a different one, or has been set to a wrong address, pre-sale buyers will receive a wrong token or won't be able to claim their share of the correct BAsset.

Recommended mitigation

In the *preAsset.setBAsset()* function, consider reverting if the **bAsset** address is not the zero address. This change needs to be implemented after solidifying *Baseline* and *BAsset* deployments as outlined in other findings in the report. There should never be the need to change the address once it was set.

Team response

Fixed in commit dbf008c.

Mitigation review

Fixed as recommended.

TRST-L-2 *setFee()* should have an upper bound

- **Category:** Input validation
- **Source:** BaselineFactory.sol#L73-77
- **Status:** Fixed

Description

Setting the fee to greater than 10000 will cause function *shift()* to revert due to an arithmetic underflow on MarketMaking.sol#L50. At the very least the new value should be validated to be less than 10000, and perhaps be set to a much lower value. (See recommendation TRST-R-3).

Recommended mitigation

Validate input.

Team response

Fixed in commit dbf008c.

Mitigation review

Fixed by disallowing setting a fee greater than *MAX_FEE*.

TRST-L-3 Small precision loss in *_repay()*

- **Category:** Precision loss errors
- **Source:** CreditFacility.sol#L186-187
- **Status:** Fixed

Description

A small precision loss occurs to a repayer's detriment because of division before multiplication in the line described.

Recommended mitigation

Remove local variable *proportion* and modify code to be

```
uint256 bAssetsReturned = currentAccount.collateral.mulWad(reservesIn_).divWad(totalOwed);
```

Team response

Fixed in commit ae05f13.

Mitigation review

Fixed as recommended.

Additional recommendations

TRST-R-1 Allow tokens with decimals other than 18

Given that *BaselineFactory* is permissionless it is recommended that Baseline contracts are modified to handle ERC-20 tokens with decimals other than 18 decimals.

TRST-R-2 Remove unused variables/dead code from contracts

There were a number of unused variables and dead code in the contracts audited which could be removed.

- MarketMaking.sol#L225, MarketMaking.sol#L242-L245 - **bAssetsFLOAT** is unused.
- MarketMaking.sol#L124 - Duplicate line.
- Core.sol#L102 – *getCirculatingSupply()* is marked as **internal**.
- preAsset.sol#L30 - Unused event.
- LiquidityManager.sol#L138 – *removeUsingBAssets()* is unused.
- LiquidityManager.sol#L262-L274 - *_getLiquidity()* unused

TRST-R-3 Only allow fee to be set to fraction of 100%

If *BaselineFactory.setFee* is used to set the fee to 100% then then **brs** for the *Baseline* instance will receive zero fee (see MarketMaking.sol#L150). Consider setting a maximum fee that is some fraction less than 100% so that deployers of *Baseline* know the minimum fees they receive.

Centralization risks

TRST-CR-1 *preAsset* owner can steal pre-sale funds via a malicious Uniswap router

During *Baseline* contract deployment and initialization, the pre-sale funds are swapped for the bAsset tokens via the caller-provided router address (BaselineFactory.sol#L39, MarketMaking.sol#L50-L52). This allows the deployer (who's also the owner of *preAsset*, and thus the party that runs the pre-sale campaign), to specify the address of a malicious router that, instead of swapping funds, transfers them to the malicious deployer.

This allows malicious actors run preliminary sales of bAssets, promising to deploy a *Baseline* contract and let pre-sale buyer be first buyers, eventually steal all funds collected during the pre-sale phase.

Team response

Fixed in commit dbf008c.

Mitigation review

The risk was mitigated by making the router address immutable in the *BaselineFactory* contract. The address is chosen by the Baseline protocol team and cannot be changed by a *Baseline* contract deployer.

TRST-CR-2 Protocol fee can be as big as 100%

The Baseline protocol team can set a protocol fee via the BaselineFactory.setBrs() and BaselineFactory.setFee() functions. The fee is collected from all *Baseline* contracts deployed via the official *BaselineFactory* contract when shifting happens and is subtracted from the fee collected by *Baseline* contract deployers (MarketMaking.sol#L137-L148). Since the maximal protocol fee is set to 100% (BaselineFactory.sol#L15), the Baseline protocol team can take the entire liquidity surplus fee from all deployed *Baseline* contracts, leaving their deployers without the profit generated by the fees.