Blend Capital

Security Assessment

February 29th, 2024 — Prepared by OtterSec

Nicola Vella nick0ve@osec.io

Andreas Mantzoutas andreas@osec.io

Table of Contents

01 — Executive Summary

Overview

Blend Capital engaged OtterSec to assess the `blend-protocol` programs. This assessment was conducted between January 29th and February 29th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 10 findings throughout this audit engagement.

In particular, we identified a critical vulnerability where the state of the user-to-be-liquidated may be outdated in case the user is also the filler, resulting in incorrect position updates and potential storage overwrite (OS-BCL-ADV-00), and another issue that enables the initial depositor to execute an inflation attack, manipulating the share token's price (OS-BCL-ADV-01, OS-BCL-ADV-02). Additionally, we emphasized the absence of slippage protection during the deposit and withdrawal processes in the pool and backstop contracts.

We also made recommendations regarding modifications to the code to enhance robustness and readability (OS-BCL-SUG-04), and we advised modifying the rounding directions followed during the calculation of the health factor and utilization rate (OS-BCL-SUG-01). Furthermore, we made suggestions to increase the overall efficiency of the code base (OS-BCL-SUG-05).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/blend-capital/blend-contracts. This audit was performed against commit 4035456.

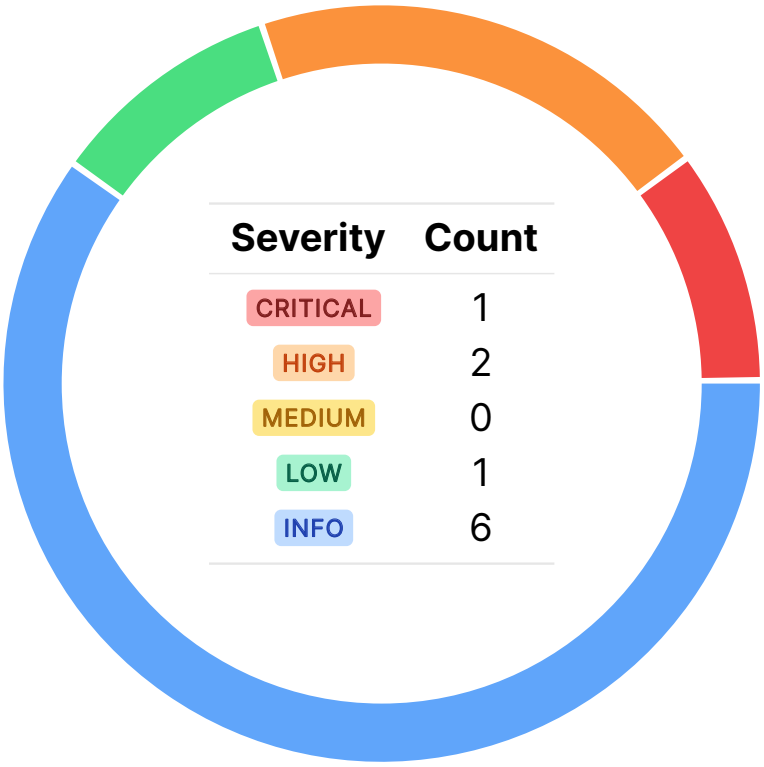
A brief description of the programs is as follows:

Name	Description
blend-protocol	A modular and ungoverned liquidity protocol primitive designed to facilitate the creation of isolated, permissionless lending pools.

03 — Findings

Overall, we reported 10 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	1
HIGH	2
MEDIUM	0
LOW	1
INFO	6

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-BCL-ADV-00	CRITICAL	RESOLVED	In `fill_user_liq_auction`, the state of the user-to-be-liquidated may be outdated if the user is also the filler, resulting in incorrect position updates and potential storage overwrite.
OS-BCL-ADV-01	HIGH	RESOLVED	The first depositor may manipulate the price by minting a single share and making a large token donation to the pool. This affects subsequent depositors, as their minted `b_tokens` may floor to zero.
OS-BCL-ADV-02	HIGH	RESOLVED	The initial depositor may perform an inflation attack, manipulating the share token's price.
OS-BCL-ADV-03	LOW	RESOLVED	Pool deployment via `Factory::deploy` may be front-un, disrupting the deployment process of another user by using the same `salt` with a different `admin` address.

Incorrect Implementation Of Self Liquidation CRITICAL OS-BCL-ADV-00

Description

There exists a vulnerability in `user_liquidation_auction`'s `fill_user_liq_auction` due to an inconsistency in the state of the liquidated user (`user_state`). The problem emerges when the user acts simultaneously as both the one being liquidated and the filler. `fill_user_liq_auction` incorrectly assumes that loading the user's state at the outset (`let mut user_state = User::load(e, user);`) provides a precise and current representation of the user's state.

>_ *auctions/user_liquidation_auction.rs* rust

```rust
pub fn fill_user_liq_auction(
    e: &Env,
    pool: &mut Pool,
    auction_data: &AuctionData,
    user: &Address,
    filler_state: &mut User,
) {
    let mut user_state = User::load(e, user);
    [...]
}
```

However, during the self-liquidation process, when the user's state is altered in-memory (cached in `from_state`) but not yet written to storage, retrieving from storage may fetch an outdated version, resulting in an inconsistency between the expected and actual user positions post self-liquidation.

Proof of Concept

In the provided scenario, the user initially possesses a specific amount of collateral in `Token0` and has liabilities in `Token1`. Concurrently, there is an active auction (`UserAuction`) featuring bids (`bid`) and lots (`lot`) associated with these tokens.

>_ *example.rs* rust

```
UserPosition: {
    collateral: { Token0: x1 },
    liabilities: { Token1: y1 },
},
UserAuction: {
    bid: { Token1: y2 },
    lot: { Token0: x2 },
}
```

Following self-liquidation, the expected user position should reflect the changes in collateral and liabilities directly, as shown below:

```rust
>_ example.rs
UserPosition: {
    collateral: { Token0: x1 + x2 },
    liabilities: { Token1: y1 + y2 },
}
```

Due to the vulnerability, the actual updated user position incorrectly combines the existing collateral and liabilities with self-liquidation results.

```rust
>_ example.rs
UserPosition: {
    collateral: { Token0: x1 },
    liabilities: { Token1: y1 },
}
```

In this example, the issue lies in the incorrect aggregation of collateral (`Token0: x1 + x2`) and liabilities (`Token1: y1 + y2`). Thus, the actual state deviates from the expected state.

Remediation

Ensure that the state loaded from storage (`user_state`) accurately reflects any in-memory changes made during the self-liquidation process.

Patch

Fixed in a10f49d by preventing self-liquidations.

Price Inflation Attack HIGH OS-BCL-ADV-01

Description

The first depositor may inflate the prices in a specific reserve, exploiting subsequent depositors. This may be achieved by making an initial supply, minting a single share (`b_token`) via `to_b_token_down` , and a large token donation to the pool. Subsequent depositors may experience unexpected behavior when depositing into the same reserve.

>_ pool/actions.rs rust

```rust
pub fn build_actions_from_request(
    e: &Env,
    pool: &mut Pool,
    from: &Address,
    requests: Vec<Request>,
) -> (Actions, User, bool) {
    [...]
    for request in requests.iter() {
        // verify the request is allowed
        require_nonnegative(e, &request.amount);
        pool.require_action_allowed(e, request.request_type);
        match RequestType::from_u32(e, request.request_type) {
            RequestType::Supply => {
                let mut reserve = pool.load_reserve(e, &request.address, true);
                let b_tokens_minted = reserve.to_b_token_down(request.amount);
                [...]
            }
            [...]
        }
    [...]
}
```

The vulnerability revolves around calculating `b_tokens_minted` for successive deposits through `to_b_token_down` . As a result of the initial supply and a substantial donation, the value of `b_tokens_minted` for subsequent deposits may be truncated to zero. This creates an unintended and potentially exploitable scenario where subsequent depositors may not obtain the anticipated amount of minted shares. Moreover, this problem may be exploited in coordination with a front-running attack, wherein the attacker strategically times the substantial token donation just before the second deposit to maximize the impact of the flooring issue.

Proof of Concept

1. The attacker mints one share (`b_token`) with a small amount.
2. The attacker then donates a large amount of tokens, artificially increasing the total reserve.
3. Another user attempts to deposit a significant amount into the same reserve after the attacker's actions.
4. The calculation of `b_tokens_minted` for this deposit is affected by the inflated total reserve, potentially flooring it to zero.
5. The subsequent depositor receives little to no minted shares (`b_tokens`), which is unexpected.
6. This may result in financial losses for the unsuspecting user, as they may not receive the proportionate share of the deposited tokens.

Remediation

Permanently lock a portion of the initial deposit and limit donations to prevent manipulation. Additionally, establish a mechanism to guarantee that the minted amount is never zero.

Patch

Fixed in 34d5e51 and 8a04ebc by not accepting donations and ensuring that the minted amount exceeds 0.

Backstop Deposit Inflation HIGH OS-BCL-ADV-02

Description

The vulnerability involves the first depositor in `backstop`. This initial depositor may exploit the price of the share token. The attacker initiates the attack by calling `execute_deposit` to mint a single share. The attacker influences the share token's price calculation by minting just one share.

>_ *backstop/src/backstop/pool.rs* rust

```rust
pub fn convert_to_shares(&self, tokens: i128) -> i128 {
    if self.shares == 0 {
        return tokens;
    }
    tokens
        .fixed_mul_floor(self.shares, self.tokens)
        .unwrap_optimized()
}
```

After minting a single share, the attacker executes a substantial donation of `backstop` tokens to the `backstop` pool using `execute_donate`. This combination of minting a single share and making a large donation of `backstop` tokens artificially inflates the calculated price of the share token. During the deposit process, subsequent depositors may observe their share tokens rounding down to zero, where the share token price is computed based on the product of the number of shares (`self.shares`) and the number of tokens (`self.tokens`). As indicated above, the `fixed_mul_floor` performs fixed-point multiplication with flooring.

The rounding down occurs when the computed number of shares falls below one, and the flooring operation adjusts it to zero. Consequently, depositors lose funds as their planned share allocation becomes zero, and the deposited funds fail to contribute to their share balance as anticipated.

Remediation

Implement thresholds and permanently lock a portion of the initial deposit, or restrict donations, to prevent manipulation.

Patch

Fixed in ea26698 by restricting donations.

Frontrunning Deployment Process LOW OS-BCL-ADV-03

Description

In `Factory::deploy`, a new pool contract is deployed utilizing a deployer. The `admin` and `salt` parameters are provided as function arguments. On observing a deployment of a transaction with a specific `salt`, another user may front-run by deploying their pool contract with the same `salt` but a different `admin` address, disrupting the original user's deployment process.

Remediation

Ensure that the same `salt` with a different `admin` results in a different deployer address.

Patch

Fixed in 8556fc0 by deriving the `salt` from the `admin`.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-BCL-SUG-00	The issue involves the potential for double-burning tokens in a liquidation auction initiated against the `backstop` address.
OS-BCL-SUG-01	Incorrect rounding directions are followed when calculating the health factor and utilization rate.
OS-BCL-SUG-02	Recommendations aimed at enhancing the protocol's resilience against vulnerabilities and threats, ensuring a higher level of security and stability.
OS-BCL-SUG-03	It is possible to manipulate the `usdc_per_token` metric in a lending pool to influence auction prices for bad debt and interest.
OS-BCL-SUG-04	Recommendations regarding modifications to the code to ensure increased robustness and readability.
OS-BCL-SUG-05	Suggestions to increase the overall efficiency of the code base.

Backstop Liquidation Safeguards OS-BCL-SUG-00

Description

`new_liquidation_auction` initiates a liquidation auction by calling `auctions::create_liquidation` against a specified user address. in theory, initiating a liquidation auction against the `backstop` address is possible. Since `backstop` does not hold collateral, attempting to create a liquidation auction against it will result in a panic in `create_user_liq_auction_data`.

>_ *pool/src/contract.rs* rust

```rust
fn new_liquidation_auction(e: Env, user: Address, percent_liquidated: u64) -> AuctionData {
    let auction_data = auctions::create_liquidation(&e, &user, percent_liquidated);
    e.events().publish(
        (Symbol::new(&e, "new_liquidation_auction"), user),
        auction_data.clone(),
    );
    auction_data
}
```

If two auctions (`bad_debt` and `liquidation`) simultaneously target the same address (potentially the `backstop` address), there may be a scenario where tokens are burned twice. Currently, this exploit is not feasible due to the lack of collateral in `backstop`.

Remediation

Explicitly panic in cases where a liquidation auction is attempted against an address unsuitable for liquidation.

Patch

Fixed in 9b9bfce.

Rounding Errors

OS-BCL-SUG-01

Description

The health factor calculation in `as_health_factor` should use the floor operation to round down the result to benefit the pool. Likewise, within `utilization`, use ceiling rounding (`ceil`) instead of floor rounding for calculating the utilization rate.

Remediation

Modify the rounding directions as mentioned above.

Patch

Fixed in b87b545 and 31b45c3.

Risk Management Improvements

OS-BCL-SUG-02

Description

There is an absence of deposit and borrow limits on the reserves. Without these limits, a user (potentially a whale with significant assets) may deposit an unusually large amount of a specific asset as collateral into the reserve and, consequently, borrow a substantial amount of a different asset, potentially utilizing the entire available liquidity of the reserve.

Furthermore, deposit and borrow limits may be used to implement better risk management, preventing single transactions from significantly impacting the overall health and stability of the reserve.

Remediation

Implement deposit and borrow limits on the reserves.

Manipulation Of Auction Metrics OS-BCL-SUG-03

Description

Within the protocol, the pools `usdc_per_token` metric is calculated by taking Comet's USDC balance, multiplying it by a constant (`SCALAR_7`), and then dividing by Comet's total supply. This determines bid or lot sizes for bad debt and interest auctions. An attacker may execute a large swap on the Comet token, updating `usdc_per_token` via `update_tkn_val` and consequently, reverse the initial swap, incurring only swap fees. The attacker may utilize this manipulated `usdc_per_token` value to determine bid or lot sizes for bad debt and interest auctions within the pool.

>_ *backstop/src/contract.rs* rust

```rust
fn update_tkn_val(e: Env) -> (i128, i128) {
    storage::extend_instance(&e)
    let backstop_token = storage::get_backstop_token(&e);
    let blnd_token = storage::get_blnd_token(&e);
    let usdc_token = storage::get_usdc_token(&e)
    backstop::execute_update_comet_token_value(&e, &backstop_token, &blnd_token, &usdc_token)
}
```

Remediation

Implement a mechanism that prevents or limits rapid and significant changes in the value of `usdc_per_token` .

Code Refactoring OS-BCL-SUG-04

Description

1. `set_drop_status` and `get_drop_status` utilize instance storage for managing the drop status associated with a `backstop`. The key for storage is generated with `EmitterDataKey::Dropped(Address)`. The `Dropped` data is exclusively employed in `drop`. If there is a requirement for this data to persist beyond the scope of a single contract instance, it would be more appropriate to relocate it to persistent storage.
2. `pool::initilaize` lacks an authentication check for the administrator. This requirement is already present in `pool_factory::deploy`. However, the pool may also be deployed separately from the factory contract. Hence, it is prudent to verify the `admin` (`admin.require_auth();`) within `pool::initilaize`.

>_ *pool/src/contract.rs* rust

```rust
fn initialize(
    [...]
) {
    storage::extend_instance(&e);
    pool::execute_initialize(
        &admin,
        [...]
    );
}
```

3. In `execute_queue_set_reserve`, a new reserve initialization is queued for a specific asset. However, there is no check to see if an existing queue exists for the same asset. This may result in a new queue overwriting an existing one.

>_ *pool/src/pool/config.rs* rust

```rust
pub fn execute_queue_set_reserve(e: &Env, asset: &Address, metadata: &ReserveConfig) {
    [...]
    storage::set_queued_reserve_set(
        &e,
        &QueuedReserveInit {
            new_config: metadata.clone(),
            unlock_time,
        },
        &asset,
    );
}
```

Remediation

1. Ensure to move `Dropped` data to persistent storage instead of keeping it in instance storage.
2. Include a check to verify the administrator within `pool::initilaize`.
3. Introduce a verification step within `execute_queue_set_reserve` to confirm the absence of an existing queue for the same asset. In the event of an existing queue, the function should trigger a panic, necessitating the `admin` to explicitly eliminate the old queue before establishing a new one.

Patch

1. Fixed in 6785933 by relocating `Dropped` data to persistent storage.
2. Fixed in 0a88dad by requiring auth for the admin.
3. Fixed in fca48c0 by preventing the queuing of identical reserves.

Code Optimization OS-BCL-SUG-05

Description

1. In its current implementation, `get_persistent_default` creates the default object even when it may not be needed. For example, in `get_user_positions`, the default `Positions` structure is created each time the function is called, regardless of whether the user already has positions in storage. This results in inefficiency, especially if creating the default object involves expensive operations, such as multiple guest-to-host transitions.

>_ *pool/src/storage.rs* rust

```rust
/// Fetch an entry in persistent storage that has a default value if it doesn't exist
fn get_persistent_default<K: IntoVal<Env, Val>, V: TryFromVal<Env, Val>>(
    e: &Env,
    key: &K,
    default: V,
    bump_threshold: u32,
    bump_amount: u32,
) -> V {
    if let Some(result) = e.storage().persistent().get::<K, V>(key) {
        e.storage()
            .persistent()
            .extend_ttl(key, bump_threshold, bump_amount);
        result
    } else {
        default
    }
}
```

2. `require_pool_above_threshold` may be optimised by eliminating the final multiplication by `SCALAR_7` after the saturating multiplication of `bal_blnd` and `bal_usdc`. Furthermore, the comparison condition may be modified from `saturating_pool_pc / threshold_pc >= SCALAR_7` to `saturating_pool_pc >= threshold_pc`.

Remediation

1. Optimize the default value by incorporating a lazy approach, achieved through passing a function or closure. This ensures that the construction of the default value occurs when required, only executing if the storage entry is absent.
2. Remove the unnecessary multiplication by `SCALAR_7` to reduce computational cost and modify the comparison condition as described above so that it checks directly whether the saturated pool product constant is greater than or equal to the threshold.

Patch

1. Fixed in 5102917 by lazy evaluating default values.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.