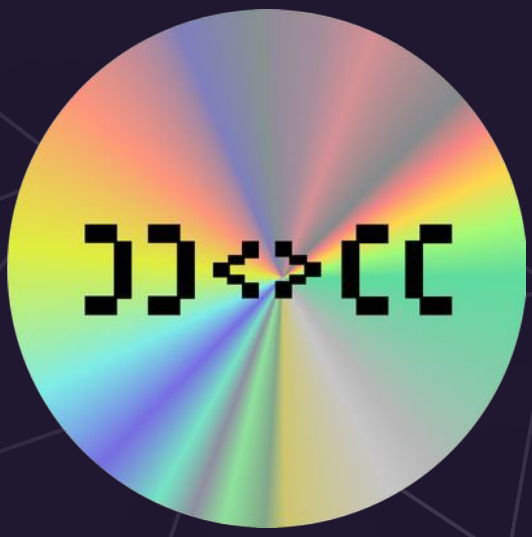
SB SECURITY

LiquidMint

Security Review



Jan 26, 2025

Conducted by:
Blckhv, Lead Security Researcher
Slavcheww, Lead Security Researcher

Contents

1. About SBSecurity

SBSecurity is a duo of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

- **High** - almost **certain** to happen, easy to perform, or highly incentivized.
- **Medium** - only **conditionally possible**, but still relatively likely.
- **Low** - requires specific state or **little-to-no incentive**.

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

LiquidMint contracts have been audited through the Hyacinth platform.

Overview

Project	LiquidMint
Repository	Private
Commit Hash	beramarket-contracts - 62548a3c84e0d91f6c6bad6923c39fc110248e1c crate - e9aa07ed9429eb743d169989243460a9073c9998
Resolution	e3eda878a0c54b007ef8ff0091289051e7f45acf
Timeline	Audit: December 5 - December 13, 2024 Mitigation: January 11, 2025 - January 26, 2025

Scope

contracts/ Crate721NTLC.sol
contracts/RouterVaults.sol
diamond/BaseFacet.sol
diamond/ DiamondTreasury.sol
utils/KeycodeUtils.sol
vaults/DummyVault.sol
vaults/ ReserveVaultBEX.sol
contracts/Core.sol
contracts/ERC721Crate.sol

contracts/extensions/ blacklist/Asset.sol
contracts/extensions/ blacklist/BlacklistExt.sol
contracts/extensions/lists/ MintlistExt.sol
contracts/extensions/ referral/ReferralExt.sol
contracts/extensions/ royalty/RoyaltyExt.sol
contracts/metadata/ CoreMetadata.sol
contracts/metadata/ CoreMetadata721.sol

Issues Found

Critical Risk	4
High Risk	8
Medium Risk	4
Low/Info Risk	17

5. Findings

5.1. Critical severity

5.1.1. withdrawFunds always uses target vault to withdraw instead to iterate

Severity: Critical Risk

Description: `RouterVaults::withdrawFunds` iterates over vaults in `listVaults` but `unzap` from the `targetVault`.

This will lead to reverts in `BaseFacet::redeemItem` when the balance of the vault with the highest reserve balance is not enough, so unzapping from other treasury vaults is needed. If we are at point where `targetVault` doesn't have enough reserves to fulfill the withdrawal, `unzap` performed in the else statement will revert since we already know that there aren't enough tokens.

```
function withdrawFunds(uint256 amount_) external {
  ...
      } else {
        // unzap a bit from every vault (in order of initialization for simplicity)
        uint256 reserveAvailable;
        for (uint i; i < rl.vaultSplit.length; i++) {
          reserveAvailable = balance(rl.vaultSplit[i]);
          // TODO: consider emitting revert if amountZap > 2^128
          _unzap(targetVault, uint128(reserveAvailable > remainingAmount ? remainingAmount :
reserveAvailable));//ISSUE always uses targetVault
          remainingAmount -= reserveAvailable > remainingAmount ? remainingAmount : reserveAvailable;

          if (remainingAmount == 0) break;
        }
      }
    }
  }
```

Recommendation: Extract vaults from `vaultSplit` and unzap from them:

```
function withdrawFunds(uint256 amount_) external {
  ...
      } else {
        // unzap a bit from every vault (in order of initialization for simplicity)
        uint256 reserveAvailable;
        for (uint i; i < rl.vaultSplit.length; i++) {
          reserveAvailable = balance(rl.vaultSplit[i]);
          // TODO: consider emitting revert if amountZap > 2^128
-          _unzap(targetVault, uint128(reserveAvailable > remainingAmount ? remainingAmount :
reserveAvailable));//ISSUE always uses targetVault
+          _unzap(rl.vaultSplit[i], uint128(reserveAvailable > remainingAmount ? remainingAmount :
reserveAvailable));//ISSUE always uses targetVault
          remainingAmount -= reserveAvailable > remainingAmount ? remainingAmount : reserveAvailable;

          if (remainingAmount == 0) break;
        }
      }
    }
  }
```

Resolution: Fixed

5.1.2. `syncReserves` are calculated wrong

Severity: Critical Risk

Description: In `syncReserves` function `loanedHeld` shouldn't be deducted from the `diff` calculation, since it's always being increased with the `reserves`, the only exception is in `itemLoanExpired`, where `loanedHeld` is deducted with the loan amount + fee, whereas `reserves` will be deducted only with the fee, since amount of collateral is already in the treasury.

Example:

NFT is loaned and cost + defaultFee is 200, so `reserves = 200`, `loanedHeld = 200`, now if we send 100 ETH, which will call `syncReserves` from the `receive` function, it will underflow, simply because it assumes **tracked balance** of the treasury to be 400, while in reality it's 200:

```
function syncReserves() public payable {
  BaseStorage.Layout storage bs = BaseStorage.layout();
  IWETH(tERC20.unwrap(bs._weth)).deposit{ value: address(this).balance }();

  uint256 balance = bs.collateral.balanceOf(address(this));
  if (balance > bs.reserves) {
    uint256 diff = balance - bs.reserves - bs.feesToWithdraw - bs.ownerFees - bs.loanedHeld;//300 - 200 - 0
- 0 - 200
    bs.reserves += diff;
    emit BackingAdded(diff, bs.reserves);
  }
}
```

As a result there will be idling tokens that will not increase the backing until balance doesn't become greater than reserves + loanedHeld.

Recommendation: Remove `loanedHeld` from the diff calculation.

Resolution: Fixed

5.1.3. `crocRouter` and `crocQuery` are not provided in `RouterVault::initializeRouter`

Severity: Critical Risk

Description: `RouterVaults::initializeRouter` doesn't provide `crocRouter` and `crocQuery` to `ReserveVaultBEX`, as a result all the `zap/unzap` functions will be reverting and none of the treasury actions will be possible.

```
function initializeRouter(Keycode reserveType_) external override onlyOwner {
...
  } else {
    //ISSUE: both _data are empty, no crocMultiSwap and crocQuery will be set
    initializeVault(rl.reserveVault, reserveType_, rl.reserveToken, bytes(""));
  }
}
```

Recommendation: Pass both contracts as a function arguments.

Resolution: Fixed

5.1.4. `_split()` is wrongly updating `feesToWithdraw`

Severity: Critical Risk

Description: `BaseFacet::_split`, when the recipient is `THIS_TREASURY` it wrongly assumes these funds will be able to be withdrawn and adds them to the `feesToWithdraw`. But in reality they increase the reserves and add to the backing of the collection, so they must stay in the contract.

The issue is that `total` is not decreased with these tokens and each split will be locking tokens until the point when `syncReserves` starts underflowing because `feesToWithdraw` + `ownerFees` > `reserves`.

Example when only treasury is fee recipient:

```
1. amount = 100
fee for treasury = 20
other recipients = 30
2. amount -= 20 - 30 = 50
3. feesToWithdraw = total - amount = 100 - 50 = 50  (this is wrong, since only 30 are for withdraw and 20
are transfered to reserves in the first if)
4. ownerFees = 50

Total Fees = 100, but in reality should be 80
```

Recommendation: Deduct `fee` amount from `total` when the recipient is the `treasury` itself.

Resolution: Fixed

5.2. High severity

5.2.1. Excess ETH is not refunded on mint

Severity: High Risk

Description: When minting, funds flow is the following:

1. Whole `msg.value` is distributed among `feeRecipients` - `_handlePayments`

2. After that `_canMint` calculates the total due based on the price and number of tokens.

That means minters are not protected from forwarding more ETH than needed, and the excess will also be utilized in the form of fees, instead of being refunded to them.

Recommendation: Add a check for `price * amount_ > msg.value` in `Crate721NTCL._handlePayments()` to first refund the amount if more Eth is sent, and then process the fees. Could also include the same check in `Core._canMint()`, that way it won't wait for the above contracts like `Crate721NTCL._handlePayments()` to always mitigate this issue, as `ERC721Crate` needs to be a separate contract and work properly as can be inherited from anyone.

Resolution: Acknowledged

5.2.2. If there is no owner anyone can withdraw all ETH in the 721Crate

Severity: High Risk

Description: `ERC721Crate` is self-sufficient implementation and doesn't need to be inherited by `Crate271NTLC` in order to be used. But if it's deployed like that we will have the following implementation of `_withdraw` in `Core`:

```
function _withdraw(address recipient_, uint256 amount_) internal virtual {
    if (recipient_ == address(0)) revert NotZero();

    // Cache owner address to save gas
    address owner = owner();
    bool forfeit = owner == address(0);

    // If contract is owned and caller isn't them, revert.
    if (!forfeit && owner != msg.sender) revert Unauthorized();

    uint256 balance = address(this).balance;
    // Instead of reverting for overage, simply overwrite amount with balance
    if (amount_ > balance || forfeit) amount_ = balance;

    // Process withdrawal
    (bool success,) = payable(recipient_).call{value: amount_}("");
    if (!success) revert TransferFailed();

    emit Withdraw(recipient_, amount_);
}
```

If the owner has revoked, anyone will be able to sweep all the ETH, because `Unauthorized` check will pass.

Recommendation: Refactor the `_withdraw` function not to allow everyone to take the funds.

Resolution: Acknowledged

5.2.3. poolIdx of Vault should be passed dynamically

Severity: High Risk

Description: `poolIdx` is the fee tier of CrocSwap DEX and currently `36_000` is hardcoded in the swap params, which is value **only used in testnet.** Once DEX reaches mainnet it's not certain the same fee tier identificators will be retained, since docs explicitly mention these are `testnet` values - https://docs.bex.berachain.com/developers/type-conventions#bartio-testnet-pool-indices.

If that happens `zap/unzap` functions of `ReserveVaultBEX` won't be working, because pool is computed by base, quote tokens and poolIdx.

Recommendation: Pass the `poolIdx` dynamically as `initializeVault` function argument, instead of hardcoding it.

Resolution: Fixed

5.2.4. 0 byte/payable selector bug

Severity: High Risk

Description: `LibDiamond.sol` is an old version of the `diamond-2` implementation that suffers from a issue with it selectors. When adding selectors to `LibDiamond.sol`, if you add `0x00000000` (payable selector) to the 1st slot in a new row and then remove it, it will cause the last function in the previous row to not work.

You can read more about this issue here - https://github.com/mudgen/diamond-2-hardhat/pull/11

Recommendation: Solution is to use `selectorInSlotIndex` instead of `_selectorSlot` on line 171.

Resolution: Fixed

5.2.5. When reserves are less than the `defaultFee`, `backingLoanExpired()` will reverting

Severity: High Risk

Description: In `backingLoanExpired`, there is a ternary operator in the `_split` function that caps the fee taken to the max reserves if needed:

```
function backingLoanExpired(uint256 loanId_) public {
...
  _split(loan.defaultFee > bs.reserves ? bs.reserves : loan.defaultFee);
  bs.reserves -= loan.defaultFee;

  emit BackingLoanExpired(loanId_, bs.reserves, bs.loanedOut, bs.itemsTreasuryOwns);
}
```

But that check is missing in the next line and as a result when `defaultFee` is greater than the `reserves`, execution will revert with underflow.

One exception is when some of the fees taken are allocated to treasury which increases the `reserves`, but since owners are not forced to set any non-zero percentage, we can't rely on that to bring the reserves above the defaultFee.

Recommendation: Perform the exact same check for reserves deduction in case they're less than `defaultFee`.

Resolution: Fixed

5.2.6. vaultSplit is used for balance checks instead of listVaults

Severity: High Risk

Description: `RouterVaults`'s owner can manipulate the `vaultsFunds` since it iterates only over `vaultSplit` records. This mapping is updated from `setVaultSplit` and is used as a target vault for new funds. Changing the `vaultSplit,` the owner can manipulate the balance returned from the `vaultFunds` function by adding or removing vaults with higher balances and vice-versa, depending on his desired outcome. Whereas if `listVaults` is being used, we will always receive the entire treasury balance without a way to be modified.

Recommendation: In `vaultFunds` replace `vaultSplit` with `listVaults,` however based on comment for `listVaults` `RSVR0` vault must be excluded from the calculation.

Resolution: Fixed

5.2.7. `_handleUnexpectedNFT` is increasing `itemsTreasuryOwns` wrongly when loan is expired

Severity: High Risk

Description: There is `rescueERC721()` dedicated to cases where an NFT is transferred with a native ERC721 transfer functions to the treasury. This function must first check whether this NFT has been loaned and whether the loan has expired. If the loan has not expired, they call `sendLoanedItemBack()`, simulating the repayment of the loan and the collateral associated with the loan will be sent to the address that the owner will specify. Otherwise, if the loan has expired, they call `itemLoanExpired()`, which will expire the loan and decrement `itemsTreasuryOwns`.

But this is not correct, as `rescueERC721()` does not check anywhere whether the NFT is actually in the contract, and if a user has loaned the NFT and the loan has expired, but the NFT is in the user, the owner can call `rescueERC721()` and `itemsTreasuryOwns` will be incremented.

```
function _handleUnexpectedNFT(address from_, uint256 tokenId_) internal {
  BaseStorage.Layout storage bs = BaseStorage.layout();
  uint256 loanId = bs.itemLoaned[tokenId_];
  if (loanId != 0) {
    ItemLoan memory loan = bs.itemLoanDetails[loanId];
    if (block.timestamp > loan.end) {
      itemLoanExpired(loanId);
      //ISSUE here we must return
    } else {
      sendLoanedItemBack(loanId, from_);
      return;
    }
  }
  // handle itemTreasuryOwns here
  ++bs.itemsTreasuryOwns;
  bs.treasuryOwned[tokenId_] = true;

  emit ItemReceived(tokenId_, bs.itemsTreasuryOwns);
}
```

Recommendation: Before updating `itemsTreasuryOwns` in `_handleUnexpectedNFT()` check if the NFT's owner is actually `address(this)`, otherwise it should revert.

Resolution: Fixed

5.2.8. Collateral can be stolen by frontrunning `sendLoanedItemBack`

Severity: High Risk

Description: `sendLoanedItemBack` function has a check if the NFT is send to treasury beforehand, if the owner has sent the token with normal transfer `onERC721Received` won't be triggered and now NFT will be in the contract, but collateral still won't be refunded. Here attacker can frontrun with call to `sendLoanedItemBack` by passing the `loanId` and his own `recipient`, which will steal all the `collateral - outstandingInterest`.

Recommendation: Cannot just remove the owner check, as that would break `_handleUnexpectedNFT()`. Instead, add a `recipient` parameter to the `ItemLoan` structure, and when the NFT is borrowed, the user will specify the `recipient` address.

Resolution: Acknowledged

5.3. Medium severity

5.3.1. Missing slippage protection in `zap/unzap`

Severity: Medium Risk

Description: Both `zap` and `unzap` functions in `ReserveVaultBEX` give `minOut = 0` to the swap step. That in addition to using the spot price of the asset in `exchangeRate` makes the swaps susceptible to sandwich attacks.

```
function zap(Keycode idVault_, uint128 amountReserve_) external payable returns (uint128 out_) {
  out_ = IMultiSwap(rvl._crocMultiSwap).multiSwap(swap, amountReserve_, 0);
}

function unzap(Keycode idVault_, uint128 amountReserve_) external returns (uint128 out_) {
  out_ = IMultiSwap(rvl._crocMultiSwap).multiSwap(swap, unzapAmount, 0);
}
```

Recommendation: Implement TWAP and set deviation threshold which you will be using to calculate the `minOut` dynamically based on a historical data.

Resolution: Fixed

5.3.2. Owner can steal all the tokens in DummyVault

Severity: Medium Risk

Description: `DummyVault`'s init function does max approval to the owner. This opens the risk whenever there are funds in the vault to be stolen by him.

Recommendation: Remove the max reserve token approval.

```
function initializeVault(Keycode idVault_, address reserve_, address collateral_, bytes calldata) external
onlyOwner {
    // Dummy vault should be used to give a uniform interface to deposit into
    if (reserve_ != collateral_) revert InvalidUsageDummy();

    DummyVaultLayout storage dvl = layoutVault(idVault_);

    if (dvl._initialized) {
      revert VaultAlreadyInitialized()w;
    }

    dvl._initialized = true;
    dvl._token = reserve_;
-   tERC20.wrap(reserve_).approve(owner(), type(uint256).max);
  }
```

Resolution: Fixed

5.3.3. fee is based on `rfv`, not on `rfvWithVaultGains`, but removed from `rfvWithVaultGains`

Severity: Medium Risk

Description: In `redeemItem()`, the fee is based on `rfv`, but then removed from `rfvWithVaultGains`.

```
function redeemItem(uint256 id_) external {
    BaseStorage.Layout storage bs = BaseStorage.layout();

    if (isLoaned(id_)) {
      itemLoanExpired(bs.itemLoaned[id_]);
    }

    if (bs.treasuryOwned[id_]) revert NotOwner();

    uint256 rfv_ = realFloorValue();
    uint256 rfvWithVaultGains = float() > 0 ? IRouter(bs._diamond).vaultsFunds() / float() : rfv_;
    // TODO: double check fee computation
    uint256 fee_ = FPML.fullMulDivUp(bs.royalty, rfv_, _DENOMINATOR_BPS);   <-------------------------
    uint256 toGive = rfvWithVaultGains - fee_;
```

Recommendation: Consider making the fee based on `rfvWithVaultGains`, since then that's the amount the user will receive.

Resolution: Acknowledged

5.3.4. Active loaned NFT can be arbitraged as a new collateral

Severity: Medium Risk

Description: If a user loans an NFT via `loanItem()`, they have a period of time in which they can return it and get their collateral back. Otherwise, if that period expires, the loan is considered expired. Then `itemLoanExpired()` can be called even by them or others to make that NFT expired (aka that the treasury no longer owns it). The idea is that this function cannot be called while the loan is active, but the loan originator can do so because of the check.

```
if (block.timestamp <= loan.end && bs.collection.ownerOf(loan.tokenId) != msg.sender) revert ActiveLoan();
```

There is no incentive for him to just call `itemLoanExpired()` as that would make his loan expire early. But he can use the other flow of sending NFTs for collateral via `receiveLoan()`, there is a check that the NFT was received by `loanItem()` and it will only continue if the loan has expired (do it again with `itemLoanExpired()`). But as we mentioned above, this can be skipped by the loan owner and that way he can borrow the NFT and then send it for loan and there may be cases where he will be able to get more collateral on profit.

The opposite is also true of first use `receiveLoan()` and then `loanItem()` with the same NFT.

Recommendation: To mitigate this, remove the check for `msg.sender` in `itemLoanExpired()` and `backingLoanExpired()` and only allow them to be called if `loan.end` has passed.

Resolution: Acknowledged

5.4. Low/Info severity

5.4.1. Lows, Informational issues and code suggestions

Description:

1. Max 255 lists can be created in `MintlistExt`, after that old ones will start to be overridden because `listId` is of type uint8. If you plan to have more than 255, choose a bigger number.

2. external mint functions of the NFT are missing `nonReentrant` modifiers, only `_handleMint` has it, if the recipient is a wallet he can eventually reenter the execution before the modifier has been entered. If needed move the modifiers to the external mint functions **and remove it from `_handleMint`**.

3. `_requireNotPaused` is used in `Core.sol` instead of the appropriate modifiers, such as `whenNotPaused`. Remove the function call and use the modifier.

4. `RouterVaults::withdrawFunds` is missing a check when either one of the `listVaults` has zero balance to be skipped. Currently, zap will be performed even when there is no balance, but this can block the whole execution if the collateral token reverts on zero transfers. Even more likely DEX will revert when a 0 tokenIn is provided. Add a check to continue the for loop when the given vault has a 0 balance.

5. in `withdrawFunds` the following ternary check can be extracted into new variable:

```
+ uint128 swappedAmount = uint128(reserveAvailable > remainingAmount ? remainingAmount : reserveAvailable);
- _unzap(targetVault, uint128(reserveAvailable > remainingAmount ? remainingAmount : reserveAvailable));
+ _unzap(targetVault, swappedAmount);
- remainingAmount -= reserveAvailable > remainingAmount ? remainingAmount : reserveAvailable;
+ remainingAmount -= swappedAmount;
```

6. duplicate vaults can be passed in `setVaultSplit`, this will allow owner to bypass the `_MAX_ALLOCATION_BPS` and allocate more funds than allowed.

7. missing function to `delete/disable` Vault

8. in `BaseFacet::setFees`, `Crate721NTLC::_setFees` and `BaseFacet::initialize` there are missing array length equality checks. Add check for feeRecipients and fees length.

9. `BaseFacet::receiveLoan` rename `id_` argument to `ids_` since this is an array.

10. more interfaces such as `ERC165` must be added as `supportedInterfaces` in DiamondTreasury::_appFacets.

11. if collateral can be token ≠ weth, `BaseFacet::rescueERC20` should be extended to be able to retrieve mistakenly send WETH as well, remove the `weth` check as the collateral check is sufficient to protect when `weth` is collateral.

12. `BlacklistExt::_enforceBlacklist` will be reverting if the `_blacklist` set grows bigger, since it's being iterated on each mint.

13. `MintlistExt::_setList` is missing check for passing empty `root_`.

14. Comment saying `RouterVaults::listVaults` shouldn't include the `RSVR0` but it is still added in `initializeRouter`.

15. in `Crate721NTLC::_handlePayments`, each one of the recipients can block the minting by reverting in his `receive` function. Force sending is available to be used: https://github.com/Vectorized/solady/blob/main/src/utils/SafeTransferLib.sol#L84

16. `BaseFacet::migrateTreasury` owner can steal all the collateral, weth and tokens.

17. in `BaseFacet::_split` add check if the `amount_` is less than zero to return early, if given NFT is flash loaned, no interest will be paid, but all the fee recipients will be iterated.

Resolution: Partially Fixed - 4, 8