Obsidian

AUDITS

Liquid Loot Security Review

Auditors

0xjuaan

0xSpearmint

14th July, 2025

Introduction

Obsidian Audits

Obsidian audits is a team of top-ranked security researchers, with a publicly proven track record, specialising in DeFi protocols across EVM chains and Solana.

The team has achieved 10+ top-2 placements in audit competitions, placing #1 in competitions for Yearn Finance, Pump.fun, and many more.

Find out more: obsidianaudits.com

Audit Overview

Liquid Loot is an NFT marketplace built on the HyperEVM.

Liquid Loot engaged Obsidian Audits to perform a security review on their NFT lending smart contracts. The 5-day review took place from the 9th to the 13th of July, 2025.

Scope

Files in scope

Repo	Files in scope
`LiquidLoot_Lending` **Commit hash:** c773a12	contracts/**/*.sol

Summary of Findings

Severity Breakdown

A total of **16** issues were identified and categorized based on severity:

- **3 Critical severity**
- **4 High severity**
- **5 Medium severity**
- **4 Informational**

NOTE: Due to the number of critical/high findings, the code is not deemed deployment ready and it is recommended to undergo a follow-up security review.

Findings Overview

ID	Title	Severity	Status
C-01	A malicious lender can prevent repayment and refinancing, to steal the borrower's NFT	Critical	Fixed
C-02	Lack of constraint on NFT collection allows draining of all lender funds	Critical	Fixed
C-03	acquireLend does not update critical fields of the loan's BorrowData struct, causing liquidations to fail	Critical	Fixed
H-01	Repaying will not be possible due to a strict check on `msg.value`	High	Fixed
H-02	Incorrect logic in `acquireLend()` causes lenders to miss out on all interest earned	High	Fixed
H-03	Existing accrued interest is wiped when a new loan is created	High	Fixed
H-04	Upgradeability is impossible due to setting incorrect ownership of `LootDiamond`	High	Fixed
M-01	`acquireLend()` does not refund the lender	Medium	Fixed

ID	Title	Severity	Status
M-02	Incorrect `protocolFees` accounting enables admins to drain 100% of HYPE stored in the vault	Medium	Fixed
M-03	New loan is created with incorrect APY in `acquireLend()`	Medium	Fixed
M-04	The linearly increasing auction APYs are calculated incorrectly	Medium	Fixed
M-05	`LootBase.setMaxApy()` is not implemented	Medium	Fixed
I-01	LootLib.computeAuctionId can be used instead of computing auctionId manually	Informational	Fixed
I-02	redeem() does not follow the CEI Pattern	Informational	Fixed
I-03	Missing sanity check when setting APR configs	Informational	Fixed
I-04	The `liquidate()` function is `payable` but does not use native tokens	Informational	Fixed

Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Info

Impact

- **High -** leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium -** leads to a moderate material loss of assets in the protocol or moderately harms a group of users. Alternatively, breaking a core aspect of the protocol's intended functionality
- **Low -** leads to a minor material loss of assets in the protocol or harms a small group of users.

Likelihood

- **High -** attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- **Medium -** the attack/vulnerability requires minimal or no preconditions, but there is limited or no incentive to exploit it in practice
- **Low -** requires highly unlikely precondition states, or requires a significant attacker capital with little or no incentive.

Findings

[C-01] A malicious lender can prevent repayment and refinancing, to steal the borrower's NFT

Description

A lender can close their position by starting a refinancing auction, in which another lender can assume the loan.

If the auction expires without being settled, and the borrower has not repaid their loan, their position can be liquidated (the NFT will be transferred to the lender, and the debt removed).

Both `repay()` and `acquireLoan()` transfer native HYPE to the lender:

```
payable(borrow.lender).transfer(lend.hypeAmount + requiredApy);
```

```
payable(lender).transfer(lend.hypeAmount + apy);
```

The issue is that these native token transfers invoke an external call on the lender, so they can cause it to revert (by reverting in the fallback function).

Attack path

1. Lender creates loan via smart contract
2. Borrower collateralises their NFT to borrow from the lender
3. Lender triggers a refinance, and sets their fallback function to revert
4. `acquireLoan()` will revert due to transferring to the user, and `repay()` will also revert
5. Once the auction expires, since the repayment has not occured, the lender can call `liquidate()` to steal the borrower's collateral NFT

Recommendation

Rather than transferring HYPE directly to the lender, consider transferring it to the vault and accounting it to the lender, letting them redeem the tokens on their own.

[C-02] Lack of constraint on NFT collection allows draining of all lender funds

Description

When lending, lenders create a lend request with the following information:

```
struct LendRequest {
  uint256 hypeAmount;
  address collection;
  int256 tokenId;
  uint256 apy;
  IPortfolio.RequestedTrait[] traits;
}
```

They specify the `collection` and `tokenId` of the collateral NFT which will be used to borrow their lent funds.

The issue is that in `borrow()`, the borrower passes in their own collection via the `BorrowRequest` struct, and this is not validated to be the same as the `LendRequest.collection`. Only the `tokenId` is checked:

```
require(
    request.tokenId == uint256(lend.tokenId),
    ENotRequestedNft()
);
```

This allows malicious borrowers to borrow from lenders using an NFT from an arbitrary collection, effectively stealing their HYPE since the collateral is worthless.

Recommendation

In `borrow()`, consider using `lend.collection` rather than the user-provided `request.collection`, in all instances. The collection field of the BorrowRequest can be removed.

[C-03] acquireLend does not update critical fields of the loan's BorrowData struct, causing liquidations to fail

Description

The `Auction.acquireLend()` function works as a refinance. A new lender takes over the loan by repaying the current lender, this will insert a new `LendData` entry under their own portfolio. After the refinance, the active loan should now reference the new lender and its new start time.

However, the code does not update the `lender`, `lendId` and `borrowedAt` fields of the `BorrowData` struct.

The impact of this is as follows:

1. Liquidation fails entirely: when getting the `LendData` inside `Liquidate()`, the function uses the stale `borrow.lender` field. This corresponds to the previous lender, who's `LendData` status is `Acquired`. This causes the following require statement to fail when trying to liquidate the loan. (the same issue arises from not updating the `lendId`)

```
        IPortfolio.LendData storage lend = state
            .portfolio[borrow.lender]
            .collaterals[borrow.lendId];

        require(
            lend.status == IPortfolio.LendStatus.AuctionStarted,
            IAuction.EAuctionDidNotStart()
        );
```

2. Incorrect interest is charged on repayment: the borrower ends up paying interest at the new rate, from the original `borrowedAt` timestamp, even though before the loan was refinanced it may have had a higher/lower rate. This over/under charges the borrower for time that falls outside the current loan term.

The `BorrowData.hypeAmount` should also be updated to include the interest till the current time, since `BorrowData.hypeAmount` is not used anywhere else in the codebase this has minimal on chain impact.

Recommendation

Consider updating the relevant fields in `BorrowData` during the refinance in `acquireLend()` to reflect the new loan state:

```
borrow.lender = msg.sender;
borrow.borrowedAt = block.timestamp;
borrow.lendId = state.portfolio[msg.sender].totalCollaterals;
```

[H-01] Repaying will not be possible due to a strict check on `msg.value`

Description

Upon repayment, the following constraint on `msg.value` is applied:

```
require(
    msg.value == lend.hypeAmount + requiredApy + fee,
    ILend.ENotEnoughHypeSent()
);
```

The issue with this is that `requiredApy` is calculated via `calculateInterestBps()`, which is proportional to the elapsed time of the loan (in seconds):

```
function calculateInterestBps(
    uint256 principal,
    uint256 rateBps,
    uint256 timeElapsed
) public pure returns (uint256) {
    return (principal * rateBps * timeElapsed) / SECONDS_IN_YEAR /
10_000;
}
```

Since `block.timestamp` cannot be exactly predicted upon signing the transaction, the `msg.value` passed in will not pass the strict constraint, causing repayment to fail. This can potentially lead to liquidation of the borrower if a refinance auction is triggered.

Recommendation

Consider making the following change, to allow the user to pass in extra HYPE, avoiding the revert:

```
require(
-   msg.value == lend.hypeAmount + requiredApy + fee,
+   msg.value >= lend.hypeAmount + requiredApy + fee,
    ILend.ENotEnoughHypeSent()
);
```

After making this change, also add logic at the end of the function, to refund the user any excess HYPE that they passed in.

[H-02] Incorrect logic in `acquireLend()` causes lenders to miss out on all interest earned

Description

When a refinancing auction is settled, the lender should receive their principal + accrued interest while the loan was active.

```
payable(lender).transfer(lend.hypeAmount + apy);
```

The issue is that the `apy` variable (which is meant to represent the earned interest), is calculated using the auction's APY, and start date (rather than using the original loan APY, and loan start date).

```
uint256 apy = LootLib.calculateInterestBps(
    lend.hypeAmount,
    apyMicroBps / 100,
    block.timestamp - auction.auctionStartedAt
);
```

This means that once the auction starts, another lender can assume the loan, paying the original lender with effectively zero interest (so they only receive their principal).

A borrower can take advantage of this by immediately acquiring their own loan once the auction starts, as this absolves them of all their interest owed to the lender.

Recommendation

Ensure that the `apy` (which can be renamed to `interestAccrued` which is more suitable), is calculated using the `lend.apy` and the `borrowData.borrowedAt`, rather than the auction APY and start date.

[H-03] Existing accrued interest is wiped when a new loan is created

Description

In `acquireLend()`, the new `LendData` is created via the following code:

```
state.portfolio[msg.sender].collaterals[
    state.portfolio[msg.sender].totalCollaterals
] = IPortfolio.LendData({
    id: newLendId,
    borrowId: borrow.id,
    borrower: lend.borrower,
    status: IPortfolio.LendStatus.Accepted,
    hypeAmount: lend.hypeAmount, <--------
    apy: auctionApyBps,
    collection: lend.collection,
    traits: lend.traits,
    postedAt: block.timestamp,
    tokenId: lend.tokenId
});
```

The issue is that the `hypeAmount` of the loan does not include the accrued interest so far. This means that once the loan is refinanced, the borrower's interest is wiped, and they can repay their loan with no interest.

Recommendation

Ensure that the accrued interest is added to the `hypeAmount` when creating the new `LendData`, to ensure that the borrower will pay the correct amount of interest upon repayment.

[H-04] Upgradeability is impossible due to setting incorrect ownership of `LootDiamond`

Description

When the `LootDiamond` is created, it's ownership is transferred to the `LootAdmin`:

```
// Execute all cuts in one transaction
diamondCut.diamondCut(cuts, address(0), "");


// Transfer ownership to admin using the transferOwnership function in
DiamondCutFacet
(bool success, ) = address(diamond).call(
    abi.encodeWithSignature("transferOwnership(address)", admin) //
`admin` is the `LootAdmin`
);
require(success, "Ownership transfer failed");
```

However, the `LootAdmin` contract does not contain any functionality to call the admin restricted functions on the `DiamondCutFacet`, like `transferOwnership()` or `diamondCut()`.

This means that no changes to the facets or owner can be made after deployment.

Recommendation

Consider either implementing functions in the `LootAdmin` which call `diamondCut()` and `transferOwnership()`, or otherwise transferring ownership to the EOA owner of `LootAdmin` instead of the `LootAdmin` itself.

[M-01] `acquireLend()` does not refund the lender

Description

`acquireLend()` requires that `msg.value` exceeds a certain threshold:

```
require(
    msg.value >= lend.hypeAmount + apy,
    ENotEnoughHypeSentForAuction()
);
```

This means that users may pass in a HYPE amount greater than `lend.hypeAmount + apy`

However, the excess amount is not refunded to the user at the end of execution, causing the funds to be stuck in the diamond contract, and not redeemable.

Recommendation

At the end of `acquireLend()`, consider adding logic to refund any excess HYPE back to the user.

[M-02] Incorrect `protocolFees` accounting enables admins to drain 100% of HYPE stored in the vault

Description

Upon repayment of loans, `updateFee()` is called to account the accrued protocol fees:

```
function updateFees(uint256 feeAmount) public onlyValidLoot {
    accumulatedFees += feeAmount;
    protocolFees += feeAmount;

    emit FeesUpdated(feeAmount, protocolFees, accumulatedFees);
}
```

These can later be withdrawn via `withdrawFees()`:

```
function withdrawFees(
    address destination,
    uint256 amount
) public onlyOwner {
    require(amount <= protocolFees, ETooBigProtocolFees());
    Vault v = Vault(payable(vault));
    v.withdraw(destination, amount);

    emit FeesWithdrawn(amount, destination);
}
```

In `withdrawFees()`, the withdrawn amount is limited to not exceed `protocolFees`. However, the value of `protocolFees` is not decreased by the withdrawn amount. This would allow a malicious/compromised admin to steal 100% of the stored HYPE in the vault (stealing from lenders with unborrowed HYPE), by repeatedly calling `withdrawFees()`.

Recommendation

```
function withdrawFees(
    address destination,
    uint256 amount
) public onlyOwner {
    require(amount <= protocolFees, ETooBigProtocolFees());
+   protocolFees -= amount;
    Vault v = Vault(payable(vault));
    v.withdraw(destination, amount);

    emit FeesWithdrawn(amount, destination);
}
```

[M-03] New loan is created with incorrect APY in `acquireLend()`

Description

In `acquireLend()`, a new loan is created for the caller, and the APY is meant to be equal to the auction APY.

However, when assigning values for the loan struct here, the APY is set to `lend.apy` which corresponds to the original loan's APY (prior to refinancing).

```
apy: lend.apy,
```

This is incorrect, as the purpose of the auction is to determine the interest rate of the refinanced loan.

Recommendation

When storing the data of the new refinanced loan, consider using the auction's APY rather than the original loan's APY

[M-04] The linearly increasing auction APYs are calculated incorrectly

Description

`calculateAuctionAPY` calculates the auction APY in the following way:

```
uint256 currentTimestamp = block.timestamp;

if (auctionStartedAt + auctionDuration < currentTimestamp) {
    return endApy;
}
uint256 bpsPerSecond = (endApy - startApy) / auctionDuration;

uint256 secondsElapsed = currentTimestamp - auctionStartedAt;

uint256 apyMicroBps = secondsElapsed * bpsPerSecond;

return apyMicroBps;
```

It calculates the delta in APY from the starting APY, but returns this delta without adding it to the `startApy`.

This leads to the auction APY always being smaller than the true APY, by a magnitude of `startAPY`.

Recommendation

Consider the following change:

```
-return apyMicroBps;
+return apyMicroBps + startApy
```

[M-05] `LootBase.setMaxApy()` is not implemented

Description

The `LootAdmin` contract is meant to be able to set the `state.maxApy` by calling `lootBase.setMaxApy(maxApy);`

However `LootBase.setMaxApy()` has not been implemented, only `LootBase.setMinApy()` has.

As a result, the admin cannot update the max APY.

Recommendation

Apply the following implementation:

```
-function setMaxApy(uint64 maxApy) external override onlyEntryAdmin {}
+function setMaxApy(uint64 maxApy) external override onlyEntryAdmin {
+       LootLib.LootState storage state = LootLib.lootStorage();
+       state.maxApy = maxApy;
+}
```

[I-01] LootLib.computeAuctionId can be used instead of computing auctionId manually

Description

In the `startAuction()` function, the `auctionId` is calculated using the following code:

```
bytes32 auctionId = bytes32(
    keccak256(
        abi.encode(msg.sender, lend.borrower, lend.id, lend.borrowId)
    )
 );
```

However, there is already a function in `LootLib` which performs the same operation:

```
function computeAuctionId(
    address lender,
    address borrower,
    uint256 lendId,
    uint256 borrowId
) public pure returns (bytes32) {
    bytes32 auctionId = bytes32(
        keccak256(abi.encode(lender, borrower, lendId, borrowId))
    );
    return auctionId;
}
```

This function can be used instead of re-writing the hashing logic.

Recommendation

Consider using `computeAuctionId` instead of rewriting the hashing logic

```
- bytes32 auctionId = bytes32(
-     keccak256(
-         abi.encode(msg.sender, lend.borrower, lend.id, lend.borrowId)
-     )
- );
+ bytes32 auctionId = LootLib.computeAuctionId(msg.sender, lend.borrower,
lend.id, lend.borrowId);
```

[I-02] redeem() does not follow the CEI Pattern

Description

The `redeem()` function updates the lender's lendData status after transferring HYPE via `withdrawToUser`. This violates the Checks-Effects-Interactions pattern.

Note that `withdrawToUser` uses `.transfer` (limited to 2300 gas), and is thus not vulnerable to reentrancy in this case. This could become an issue if the transfer logic is upgraded to `.call`.

Recommendation

Follow CEI for future-proofing by updating the lend status before the external call

[I-03] Missing sanity check when setting APR configs

Description

`setMaxApr` and `setMinApr` lacks input validation, allowing `minApr` > `maxApr` (if an admin enters incorrect values), which would cause a DoS of lending.

Consider implementing the following sanity checks in the respective functions to prevent a state where `minApr` > `maxApr`.

```
require(maxApr >= state.minApr, EInvalidApr());

require(minApr <= state.maxApr, EInvalidApr());
```

[I-04] The `liquidate()` function is `payable` but does not use native tokens

Description

The `liquidate()` function is `payable` but in practice does not require any native tokens to be passed.

```
function liquidate(
        address borrower,
        uint256 borrowId
    ) external payable override {
```

The `payable` specifier can be removed.