Obsidian AUDITS

Hyperdrive Security Review

Auditors

0xjuaan

0xSpearmint

May 21, 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, with notable results in competitions for Uniswap v4, Pump.fun, Sentiment, and many more.

Find out more: https://github.com/ObsidianAudits

Audit Overview

Hyperdrive (https://x.com/HyperdriveDefi) is the premier stablecoin money market on Hyperliquid.

Hyperdrive engaged Obsidian Audits to perform a security review on the smart contracts in the `hyperdrive-contracts` repo. The review was conducted from the 15th to the 19th of May.

Scope

Source code repo: https://github.com/ambitfi/hyperdrive-contracts

Commit hash: 4a69ae7ac999f471fb2b0a6e3d9ea1bd0c242834

Files in scope

- packages/lending/contracts/protocol/Market.sol
- packages/lending/contracts/protocol/CollateralLib.sol
- packages/lending/contracts/protocol/LiquidationLib.sol
- packages/lending/contracts/protocol/MarketHooksLib.sol
- packages/lending/contracts/protocol/MarketLib.sol
- packages/lending/contracts/protocol/MarketFactory.sol
- packages/lending/contracts/protocol/MarketImplementationLib.sol

Summary of Findings

A total of **10** issues were identified and categorized based on severity:

- **1 High severity**
- **1 Medium severity**
- **3 Low severity**
- **5 Informational**

Findings Overview

ID	Title	Severity	Status
H-01	Anyone can add/remove the liquidator role in the `Market` contract	High	Fixed
M-01	The `forceLiquidate()` function will always revert	Medium	Fixed
L-01	`maxWithdraw()` and `maxRedeem()` do not take into account the available liquidity	Low	Fixed
L-02	Decreasing LLTV via `addOrUpdateAsset()` can cause unexpected liquidations	Low	Acknowledged
L-03	The `absorb()` function does not enforce that the account has zero collateral	Low	Fixed
I-01	The flash loan function does not support collateral tokens	Informational	Acknowledged
I-02	Redundant computation in the `isLiquidatable` function	Informational	Fixed
I-03	Incorrect transient storage slot for `REENTRANCY_GUARD`	Informational	Fixed
I-04	Collateral and debt comparison logic in liquidation can be simplified	Informational	Fixed
I-05	In rare situations, malicious borrowers can avoid paying interest by repeatedly calling `accrue()`	Informational	Acknowledged

Severity Classification

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

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

[H-01] Anyone can add/remove the liquidator role in the `Market` contract

Description

The `grantRole()` and `revokeRole()` functions in the `RoleBasedAccessControlUpgradable` contract lack any access control checks, allowing anyone to assign or remove roles. They are meant to be overridden in the `Market` to apply the admin access control, but this is not done.

This means any user can grant themselves the `LIQUIDATOR_ROLE` in the `Market` and revoke the role from legitimate liquidators, bypassing the restriction that the liquidator role is intended only for protocol approved actors.

Bad actors can also prevent liquidations by repeatedly revoking the role from legitimate liquidators, potentially leading to bad debt in the protocol.

Recommendation

This issue is fixed by overriding `grantRole()` and `revokeRole()` in the `Market` contract and applying the `onlyAdmin` modifier to the functions, to enforce that only protocol admins can call these functions.

A long term solution for future use cases of the `RoleBasedAccessControlUpgradable` (and the non-upgradeable counterpart) would be to include a `_authorize()` internal virtual function (with no implementation) which is called inside `grantRole()` and `revokeRole()`

This would ensure that compilation does not work until the `_authorize()` function is overriden with an implementation (in a parent contract), ensuring access control is implemented in these critical functions.

Discussion

@Hyperdrive:

> Fixed here;
> 014b553f1f65cb19f10d54d1a98f692c8fdd086e

[M-01] The `forceLiquidate()` function will always revert

Description

The `liquidate()` function in the `LiquidationLib` includes the following check:

```
require(ReentrancyGuardLib.enabled());
```

This enforces that the reentrancy guard has been triggered, so the execution is required to be within a `nonReentrant` modifier.

In the `Market` contract, the `forceLiquidate()` function calls the `LiqudationLib.liquidate()` function, but does not use the `nonReentrant` modifer. This will cause the mentioned check in `liquidate()` to revert.

As a result, `Market.forceLiquidate()` will always revert.

Recommendation

Consider adding a `nonReentrant` modifer to `forceLiquidate()` to ensure that the check does not fail.

Discussion

@Hyperdrive:

> Fixed here;
> fc9dbb04ce746bfa27d51f5ea2e623dd70fb7f71

[L-01] maxWithdraw() and `maxRedeem()` do not take into account the available liquidity

Description

Upon withdrawing supplied loan tokens, the withdrawal amount is limited by the amount of available liquidity, which is returned by the `MarketLib.availableLiquidity()` function.

`Market::_withdraw()`:

```
uint256 availableLiquidity = MarketLib.availableLiquidity($);
require(assets <= availableLiquidity, InsufficentLiquidity(assets,
availableLiquidity));
```

The `maxWithdraw(address owner)` function in the `ERC4626Upgradeable` implementation is inherited without override in the `Market` contract. It returns the amount of assets a user could withdraw based solely on their share balance, without accounting for actual available liquidity in the pool.

In situations of high utilisation, or for a large withdrawal, the return value of `maxWithdraw()` for a certain user can be larger than the available liquidity, which is incorrect.

Note that the same issue exists for `maxRedeem()`

Recommendation

Override `maxWithdraw()` in the `Market` to return the minimum of:

- the assets represented by the user's shares, and
- the actual available liquidity in the market.

`maxRedeem()` should be overriden to return the minimum of:

- the user's shares balance, and
- the shares corresponding to the amount of available liquidity in the market

Discussion

@Hyperdrive:

> Fixed here;
> 64be29eb7d5c1b0e06de4351c12dde0e78bb0166

@ObsidianAudits:

> Due to rounding up in `previewWithdraw()`, the `maxRedeem()` function can sometimes return a number of shares that is more than what can actually be withdrawn (off by 1).
>
> Here is an example:
>
> ```
> totalSupplyAssets: 1.0011107978396092144e21 assets
> totalSupplyShares: 1e21 shares
>
> upon maxRedeem():
> availableLiquidity: 1e20 assets
> previewWithdraw(availableLiquidity): 9.988904346631698418 2e19 shares
>
> upon redeeming:
> converting to assets: 1.00000000000000000001e20 assets
> ```
>
> While the available assets is `1e20`, the amount of shares being withdrawn will convert to `1.000..001e20` assets, causing an underflow when attempting to transfer those assets to the user.
>
> To ensure that `maxRedeem()` always works in 100% of the cases, consider the following change:
>
> ```diff
> -   return Math.min(super.maxRedeem(owner),
> previewWithdraw(MarketLib.availableLiquidity($)));
> +   return Math.min(super.maxRedeem(owner),
> _convertToShares(MarketLib.availableLiquidity($),
> Math.Rounding.Floor));
> ```
>
> This ensures the conversion to shares is rounded down (while the original `previewWithdraw()` rounds up)
>
> Note: the tradeoff is that this will sometimes lead to `maxRedeem()` leaving 1 wei of assets in the market (only when redemption is constrained by the available liquidity)

@Hyperdrive:

> Fixed here;
> 69ded74c056bc1caced4067f373467d37a48761e

@ObsidianAudits:

> fix confirmed, `maxWithdraw()` and `maxRedeem()` are now correctly limited when available liquidity is less than the user's withdrawable liquidity

[L-02] Decreasing LLTV via `addOrUpdateAsset()` can cause unexpected liquidations

Description

The CollateralLib.addOrUpdateAsset() function allows immediate updates to the `liquidationLTV` value of a collateral asset. If the protocol admin decreases the `liquidationLTV`, users who were previously above the threshold may instantly become liquidatable without warning.

As a result, users can be unexpectedly liquidated due to retroactive tightening of the liquidation threshold.

Recommendation

Consider adding a time delay before `liquidationLTV` reductions take effect (giving borrowers more time to act), or disallow decreasing this value entirely.

Discussion

@Hyperdrive:

> Accepted as is.
>
> This will be handled through governance.

[L-03] The `absorb()` function does not enforce that the account has zero collateral

Description

The `Market.absorb()` function is meant to be used on accounts which have a debt position but no collateral. It clears the debt of the user, without seizing any collateral (since it assumes that there is no collateral left).

However, the function does not enforce that the account has no collateral, so can be executed on accounts which have collateral, clearing their debt and socializing the loss among lenders.

While this is an admin restricted function, it is recommended to implement onchain constraints to reduce the trust placed on the admins.

Recommendation

Consider enforcing that the account has no collateral, by implementing the following check in `Market.absorb()`:

```
require($.userCollateralTokens[account].length() == 0 &&
$.userBorrowShares[account] > 0)
```

Discussion

@Hyperdrive:

> The external absorb method has now been removed in favour of just using flashLiquidate as flashLiquidate with a 0 repay amount can achieve the same result.
>
> 1b42bf9e584f233ecda31ae7aa22198de2d7eca1

[I-01] The flash loan function does not support collateral tokens

Description

The Market.flashLoan() function only supports the market's asset token (`$.asset`), and does not allow flash loans of collateral tokens. Most lending protocols also allow flash loans of collateral tokens, not just the borrowable asset.

Recommendation

To enable a wider range of tokens that can be flashloaned, consider allowing the caller to specify a `token` parameter, then perform transfer and repayment logic based on the provided `token`, rather than hardcoding `$.asset`.

Discussion

@Hyperdrive:

> Wont fix.
>
> The preference here is that only the supplied loan token asset can be flash loaned and the collateral assets are not.

[I-02] Redundant computation in the `isLiquidatable` function

Description

The `isLiquidatable` function redundantly calls `previewLiabilities(account)` twice:

```
function isLiquidatable(address account) public view returns (bool) {
    uint256 liabilities = previewLiabilities(account);

    MarketStorage storage $ = getMarketStorage();
    return liabilities > 0 && $.isLiquidatable(account,
previewLiabilities(account));
}
```

This is an unnecessary computation., instead the result of the first call can be reused.

Recommendation

Consider passing the stored `liabilities` value directly:

```diff
function isLiquidatable(address account) public view returns (bool) {
    uint256 liabilities = previewLiabilities(account);

    MarketStorage storage $ = getMarketStorage();
-    return liabilities > 0 && $.isLiquidatable(account,
previewLiabilities(account));
+    return liabilities > 0 && $.isLiquidatable(account, liabilities);
}
```

Discussion

@Hyperdrive:

> Fixed here;
> d6b8077258c3dfba9de5b726210a1a5b1df9507c

[I-03] Incorrect transient storage slot for `REENTRANCY_GUARD` in `ReentrancyGuardLib`

Description

The `ReentrancyGuardLib` is used in the `Market` to prevent reentrancy.

The transient storage slot assigned to the `REENTRANCY_GUARD` variable is `0xe26c696bf0e34aaf444e67b257b0ce1f00d161ab27c76bcd8c8582bed8ddd000`

```
// keccak256(abi.encode(uint256(keccak256("hyperdrive.Reentrancy")) - 1))
& ~bytes32(uint256(0xff))
bytes32 private constant REENTRANCY_GUARD =
0xe26c696bf0e34aaf444e67b257b0ce1f00d161ab27c76bcd8c8582bed8ddd000;
```

This does not match the computed value of: `keccak256(abi.encode(uint256(keccak256("hyperdrive.Reentrancy")) - 1)) & ~bytes32(uint256(0xff))`

The above hash is actually equal to:
`0xc3b90f16c7045ba87b77eea20827663be0cdecaeb7b69dc2cf64a593a3917f00`

Unless the current slot value is used or written to somewhere else in the contract, this has no impact, so this is submitted as an informational finding

Recommendation

Update `REENTRANCY_GUARD` to the correct transient storage slot:
`0xc3b90f16c7045ba87b77eea20827663be0cdecaeb7b69dc2cf64a593a3917f00`

Discussion

@Hyperdrive:

> This is actually a typo in the comment, it should be `hyperdrive.ReentrancyGuard` instead of `hyperdrive.Reentrancy`
>
> Comment fixed here;
> 9d91545a31b05a82f382f48f98a7f6825531e487

[I-04] Collateral and debt comparison logic in liquidation can be simplified

Description

The `isLiquidatable()` function and `isHealthy()` functions both perform a similar type of comparison in a few areas:

```
    return (limit * 100) / debt < 100;
```

This comparison is equivalent to `limit < debt`, so the logic can be simplified.

Instances:

```
(limit * 100) / debt < 100;
```

- LiquidationLib.sol#L32
- LiquidationLib.sol#L52
- MarketLib.sol#L175

```
(limit * 100) / debt >= 100;
```

- isHealthy()- MarketLib.sol#L193

Recommendation

The first version can be replaced with:

```
limit < debt
```

The second version can be replaced with:

```
limit >= debt
```

Discussion

@Hyperdrive:

> Fixed here; a5a5a1769545e338553d84f4d990e47d8e83463b

[I-05] In rare situations, malicious borrowers can avoid paying interest by repeatedly calling `accrue()`

Description

The `accrueLiabilities()` function calculates how much interest has accumulated since the last updated time:

```
function accrueLiabilities(uint256 assets, uint256 rate, uint256 elapsed)
private pure returns (uint256 accrued) {
    uint256 fractionalRate = (MathLib.WAD + MathLib.wadDiv(rate,
SECONDS_PER_YEAR_IN_WAD));
    uint256 compoundRate = MathLib.wadPow(fractionalRate, elapsed);
    accrued = assets == 0 ? 0 : MathLib.wadMul(assets, compoundRate) -
assets;
}
```

When `assets` (i.e. `$.totalBorrowAssets`) is small, and `elapsed` is 1 second, `compoundRate` remains very close to 1 (e.g. `1.0000000316... * 1e18`). However, due to rounding in `wadMul` (to the nearest integer), the result can be the same as the original `assets`, resulting in `accrued == 0`.

Example:

```
assets:             1e7
compoundRate:       1.0000000316880878 14e18
wadMul result:      1e7
accrued:            1e7 - 1e7 = 0
```

This means interest does not accrue even though time has passed and a non-zero rate is applied.

This problem occurs primarily in low-liquidity markets where `totalBorrowAssets` is small, especially with low decimal, high-value tokens like uBTC.

An attacker can repeatedly call `accrue()` every second to reset `$.lastUpdate`, preventing interest from accumulating as long as `$.totalBorrowAssets` remains small (e.g. < 1e7).

Proof of Concept

Running both of the tests provided yields the below result.

> The tests are ommited in the public report to reduce report size

```
Ran 2 tests for test/POC.t.sol:POC
[PASS] test_borrow_accrue_interest() (gas: 1180182)
Logs:
  liabilities: 1e7
  liabilities after 1000 seconds: 1.0000317e7

[PASS] test_borrow_accrue_no_interest() (gas: 23074498)
Logs:
  liabilities: 1e7
  liabilities after 1000 seconds (with accrual every second): 1e7

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 59.07ms
(59.43ms CPU time)
```

This demonstrates that by calling `accrue()` every second, interest payment can be avoided (approx $28/day total). In the example, the total borrowed assets is equal to 0.1 uBTC (~$10k).

Recommendation

There is no simple fix to this issue. If the calculation changed to only round up (instead of to the nearest integer), then lenders would be able to repeatedly `accrue()` to earn a higher APY than what they are meant to earn.

The only loan token that would make this issue economically feasible to occur is uBTC (which has 8 decimals, and a value of ~$100k per 1e8 tokens), and it requires the interest rate and total borrowed amount to be idealistic values (such that `wadMul(assets, compoundRate)` rounds down to `assets`). It also requires the unpaid interest to exceed the gas cost per transaction, in order to be profitable.

Due to the above constraints, we are documenting this as a potential vector to be aware of, but do not believe a mitigation is required.

Discussion

@Hyperdrive:

> This is a good find, but it wont be an issue for us as the markets we are planning will be HYPE as the loan token which is 18 decimals, and then the others are all likely to be stablecoins as the loan tokens.