SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Swivel
Prepared by:	Sherlock
Lead Security Expert:	IIIIIII
Dates Audited:	January 9 - January 12, 2023
Prepared on:	January 26, 2023

Introduction

Iluminate is a fixed-rate lending protocol designed to aggregate fixed-yield Principal Tokens and provide Illuminate's users and integrators a guarantee of the best rate in DeFi.

Scope

~ 2115 nSLOC

- `Lender.sol`
- `MarketPlace.sol`
- `Redeemer.sol`
- `Converter.sol`
- `Creator.sol`
- `ERC5095.sol`
- `Maturities.sol`

For more information about the remeidations and modifications since the previous audit, see this document.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
2	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

IIIIIII	cccz

Issue H-1: Illuminate's PT doesn't respect users' slippage specifications for underlyings

Source: https://github.com/sherlock-audit/2023-01-illuminate-judging/issues/16

Found by

IllIllI

Summary

Illuminate's PT doesn't respect users' slippage specifications for underlyings, and allows more slippage than is requested

Vulnerability Detail

`ERC5095.withdraw()`/`redeem()`'s code adds extra underlying slippage on top of what the user requests

Impact

At the end of withdrawal/redemption, the user will end up receiving less underlying than they asked for, due to slippage. If the user had used a external PT to mint the Illuminate PT, they will have lost part of their principal.

Code Snippet

```
// File: src/tokens/ERC5095.sol : ERC5095.withdraw()   #1

271                     uint128 returned =
 ↪  IMarketPlace(marketplace).sellPrincipalToken(
272                         underlying,
273                         maturity,
274                         shares,
275 @>                      Cast.u128(a - (a / 100))
276:                    );
```

https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L271-L276

```
// File: src/tokens/ERC5095.sol : ERC5095.withdraw()   #2

302                     uint128 returned =
 ↪  IMarketPlace(marketplace).sellPrincipalToken(
303                          underlying,
304                          maturity,
305                          Cast.u128(shares),
306 @>                       Cast.u128(a - (a / 100))
307:                     );
```

https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L302-L307

`redeem()` has the same issue

Tool used

Manual Review

Recommendation

This is the same issue that I described in the last contest. In the original issue, the finding was disputed because there wasn't a clean solution for slippage protection on the number of shares burned in order to satisfy the input underlying amount. During the discussion of the issue, it became clear that the `ERC5095` contract was supposed to be cross-compatable with the `ERC4626` standard, and that standard has this to say:

`If implementors intend to support EOA account access directly, they should consider adding an additional function call for deposit/mint/withdraw/redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits, since they have no other means to revert the transaction if the exact output amount is not achieved.` https://eips.ethereum.org/EIPS/eip-4626

In other words, it's not up to the `ERC4626`/`ERC5095` contract implementation itself to determine whether too many shares needed to be burned in order to satisfy the request for exactly the provided number of underlying - it's up to the caller to have extra code to determine whether the number is satsifactory itself. Note though that both standards are very clear that the *exact* number of underlying *must* be provided back, and the implmentation as it stands does *not* do this.

Discussion

sourabhmarathe

We believe this issue and its duplicates are related to the same underlying problem of complying with ERC4626.

We will address this issue and its duplicates altogether in one PR.

Issue H-2: The Notional version of `lend()` can be used to lock iPTs

Source: https://github.com/sherlock-audit/2023-01-illuminate-judging/issues/15

Found by

IIIIIII

Summary

The Notional version of `lend()` can be used to lock extra iPTs in the `Lender` contract

Vulnerability Detail

The Notional version of `lend()` has no checks to ensure that the principal value, `p`, passed in is for Notional, and therefore the Illuminate principal value can be passed in and used, which will allow callers to buy iPTs to the `Lender` contract (rather than Notional PTs), and then mint a second one to themselves when the iPT is minted at the end of the function.

Impact

When the underlying is sold in the marketplace, the resulting iPT is given to the `Lender` contract, and there is no supported way to have those iPTs redeemed and their underlying released, which means when users try to redeem their own iPTs, there will be less underlying available than there should be, and they will have lost principal.

Code Snippet

In the code block below, there are no checks that `p` is for notional, and the market-provided token for that `p` is used directly for depositing, and at the end of the function, more iPTs are minted:

```
// File: src/Lender.sol : Lender.lend()   #1

875      function lend(
876          uint8 p,
877          address u,
878          uint256 m,
879          uint256 a,
880          uint256 r
881      ) external nonReentrant unpaused(u, m, p) matured(m) returns (uint256) {
882          // Instantiate Notional princpal token
```

```
883 @>       address token = IMarketPlace(marketPlace).markets(u, m, p);
884
885          // Transfer funds from user to Illuminate
886          Safe.transferFrom(IERC20(u), msg.sender, address(this), a);
...
894          // Swap on the Notional Token wrapper
895 @>       uint256 received = INotional(token).deposit(a - fee, address(this));
896
897          // Convert decimals from principal token to underlying
898          received = convertDecimals(u, token, received);
...
908          // Mint Illuminate zero coupons
909:@>       IERC5095(principalToken(u, m)).authMint(msg.sender, received);
```

https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/Lender.sol#L875-L909

The `deposit()` function sells the underlying for iPTs, using the marketplace:

```
// File: src/tokens/ERC5095.sol : ERC5095.deposit()   #2

191        // consider the hardcoded slippage limit, 4626 compliance requires no
 ↪  minimum param.
192        uint128 returned = IMarketPlace(marketplace).sellUnderlying(
193            underlying,
194            maturity,
195            Cast.u128(a),
196            shares
197:        );
```

https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L191-L197

NOTE: `INotional` is an ERC4626 token, and the `deposit()` function comes from that interface. While the Illuminate PT's `deposit()` function has a different signature (the argument order is flipped), it's clear that this is a mistake that will be corrected as a part of this audit, since comments within the `deposit()` function itself refer to the need to be compliant with the ERC4626 standard, and discussions in the prior audit extensively mention that compliance was in fact necessary, and the `deposit()` function is not a part of the EIP-5095 standard.

Tool used

Manual Review

Recommendation

Revert if `p` is not `MarketPlace.Principals.Notional`

Issue M-1: ERC5095 has not approved MarketPlace to spend tokens in ERC5095

Source: https://github.com/sherlock-audit/2023-01-illuminate-judging/issues/23

Found by

cccz

Summary

ERC5095 requires approving MarketPlace to spend the tokens in ERC5095 before calling MarketPlace.sellUnderlying/sellPrincipalToken

Vulnerability Detail

MarketPlace.sellUnderlying/sellPrincipalToken will call transferFrom to send tokens from msg.sender to pool, which requires msg.sender to approve MarketPlace. However, before calling MarketPlace.sellUnderlying/sellPrincipalToken in ERC5095, there is no approval for MarketPlace to spend the tokens in ERC5095, which causes functions such as ERC5095.deposit/mint/withdraw/redeem functions fail, i.e. users cannot sell tokens through ERC5095.

```
    function sellUnderlying(
        address u,
        uint256 m,
        uint128 a,
        uint128 s
    ) external returns (uint128) {
        // Get the pool for the market
        IPool pool = IPool(pools[u][m]);

        // Get the number of PTs received for selling `a` underlying tokens
        uint128 expected = pool.sellBasePreview(a);

        // Verify slippage does not exceed the one set by the user
        if (expected < s) {
            revert Exception(16, expected, 0, address(0), address(0));
        }

        // Transfer the underlying tokens to the pool
        Safe.transferFrom(IERC20(pool.base()), msg.sender, address(pool), a);
...
    function sellPrincipalToken(
        address u,
        uint256 m,
        uint128 a,
        uint128 s
    ) external returns (uint128) {
        // Get the pool for the market
        IPool pool = IPool(pools[u][m]);

        // Preview amount of underlying received by selling `a` PTs
        uint256 expected = pool.sellFYTokenPreview(a);

        // Verify that the amount needed does not exceed the slippage parameter
        if (expected < s) {
            revert Exception(16, expected, s, address(0), address(0));
        }

        // Transfer the principal tokens to the pool
        Safe.transferFrom(
            IERC20(address(pool.fyToken())),
            msg.sender,
            address(pool),
            a
        );
```

In the test file, `vm.startPrank(address(token))` is used and approves the MarketPlace, which cannot be done in the mainnet

```
vm.startPrank(address(token));
IERC20(Contracts.USDC).approve(address(marketplace), type(uint256).max);
IERC20(Contracts.YIELD_TOKEN).approve(
    address(marketplace),
    type(uint256).max
);
```

Impact

It makes functions such as ERC5095.deposit/mint/withdraw/redeem functions fail, i.e. users cannot sell tokens through ERC5095.

Code Snippet

https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/MarketPlace.sol#L396-L414 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/MarketPlace.sol#L319-L342 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L188-L197 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L230-L244 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L267-L276 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ER

C5095.sol#L372-L385 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L267-L307 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/tokens/ERC5095.sol#L372-L409 https://github.com/sherlock-audit/2023-01-illuminate/blob/main/test/fork/ERC5095.t.sol#L72-L77

Tool used

Manual Review

Recommendation

Approve MarketPlace to spend tokens in ERC5095 in ERC5095.setPool.

```
    function setPool(address p)
        external
        authorized(marketplace)
        returns (bool)
    {
        pool = p.fyToken();
+       Safe.approve(IERC20(underlying), marketplace, type(uint256).max);
+       Safe.approve(IERC20(p.), marketplace, type(uint256).max);

        return true;
    }

        pool = address(0);
    }
```

Issue M-2: Protocol fees not taken on premium

Source: https://github.com/sherlock-audit/2023-01-illuminate-judging/issues/22

Found by

IIIIIII

Summary

Protocol fees not taken on premium

Vulnerability Detail

The Swivel version of `lend()` allows the user to use any extra underlying premium from their Swivel orders, to buy more iPTs via a swap or minting directly, but no fee is taken from this premium.

Impact

Rather than using the Illuminate version of `lend()`, which charges a fee, users could use the Swivel version, and ensure the fee portion is small, and the premium non-fee portion is large, so that Illuminate misses out on fees.

Code Snippet

The fee is calculated based on the amount listed in the orders:

```
// File: src/Lender.sol : Lender.lend()   #1

488              // Lent represents the total amount of underlying to be lent
489 @>           uint256 lent = swivelAmount(a);
490
491              // Get the underlying balance prior to calling initiate
492              uint256 starting = IERC20(u).balanceOf(address(this));
493
494              // Transfer underlying token from user to Illuminate
495              Safe.transferFrom(IERC20(u), msg.sender, address(this), lent);
496
497              // Calculate fee for the total amount to be lent
498:@>           uint256 fee = lent / feenominator;
```

https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/Lender.sol#L488-L498

But the premium is the balance change after the orders have executed (can be thought of as positive slippage):

```
// File: src/Lender.sol : Lender.lend()   #2
525                   // Calculate the premium
526 @>                uint256 premium = (IERC20(u).balanceOf(address(this)) -
 ↪  starting) -
527:                      fee;
```

https://github.com/sherlock-audit/2023-01-illuminate/blob/main/src/Lender.sol#L525-L527

And no fee is charged on this premium, either when swapping in the yield pool, or when minting iPTs directly.

Tool used

Manual Review

Recommendation

Calculate the fee after the order, on the full balance change

This is similar to a finding from the previous contest, but the mitigation was to remove the amount fee from the premium, but didn't address the fee for the premium itself