SHERLOCK

SHERLOCK SECURITY REVIEW FOR

SHERLOCK

Prepared for: Swivel
Prepared by: Sherlock
Lead Security Expert: IllIllI
Dates Audited: October 24 - November 7, 2022
Prepared on: November 21, 2022

Introduction

Illuminate 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, while also deepening liquidity across the fixed-rate space.

Scope

~ 1847 nSLOC

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

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
15	18

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

IIIIIII	ctf_sec	Holmgren
0x52	kenzo	rvierdiiev
hyh	cccz	Ruhum

HonorLt
Bnke0x0
Tomo
Jeiwan
neumo
cryptphi
bin2chen
hansfriese
pashov
windowhan_kalosec
minhtrng
ak1
__141345__
ayeslick
Nyx
JohnSmith
John

Issue H-1: Unlimited mint of Illuminate PTs is possible whenever any market is uninitialized and unpaused

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/238

Found by

ctf_sec, ayeslick, cccz, Jeiwan, pashov, Nyx, kenzo, Bnke0x0, IllIIII, HonorLt, hyh

Summary

If some market isn't defined, but not paused, which is true by default, an unlimited mint of Illuminate PTs is possible as `Safe.transferFrom(address(0),...)` will be successful due to zero address check isn't performed for the token used, while low-level call to zero address is a success.

Vulnerability Detail

Lender's mint do not check that principal obtained from `marketPlace` is a viable token. It can be zero address if `(u,m,p)` isn't yet defined for a particular `p`. In the same time `unpaused(u,m,p)` is true by default corresponding to the uninitialized state.

This way once the Lender contract enters the state where `marketPlace` is defined, but some market for some particular `p` isn't yet (this is what setPrincipal() is for, i.e. it's a valid use case), and it is not paused (which is by default, as pausing is a manual pause() call), Lender's mint() can be used to issue unlimited number of Illuminate PTs to the attacker.

Bob the attacker can setup a script to track such situations for a new Lender contracts. I.e. he can track setMarketPlace() calls and if there is a principal token `IMarketPlace(marketPlace).token(u,m,0)` created, but some market for `p>0` from `Principals` is undefined, but isn't paused, Bob runs Lender's mint and obtains any number of `IMarketPlace(marketPlace).token(u,m,0)` for free.

Impact

An attacker can obtain unlimited Illuminate PTs, subsequently stealing all the funds of any other users with Redeeder's Illuminate redeem(). Such overmint can be unnoticed as it doesn't interfere with any other operations in the system.

The situation described can be a part of normal system setup workflow, unless being specifically handled. The impact itself is full insolvency for a given `(u,m)`. This way setting the severity to be high.

Code Snippet

If mint() be called with `p` such that `paused[u][m][p]==false` and Marketplace's `markets[u][m][p]==0`, i.e. both are uninitialized yet, it unlimitedly mints Illuminate PTs for free:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L264-L288

```
/// @notice mint swaps the sender's principal tokens for Illuminate's ERC5095
↪ tokens in effect, this opens a new fixed rate position for the sender on
↪ Illuminate
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a amount being minted
/// @return bool true if the mint was successful
function mint(
    uint8 p,
    address u,
    uint256 m,
    uint256 a
) external unpaused(u, m, p) returns (bool) {
    // Fetch the desired principal token
    address principal = IMarketPlace(marketPlace).token(u, m, p);

    // Transfer the users principal tokens to the lender contract
    Safe.transferFrom(IERC20(principal), msg.sender, address(this), a);

    // Mint the tokens received from the user
    IERC5095(principalToken(u, m)).authMint(msg.sender, a);

    emit Mint(p, u, m, a);

    return true;
}
```

`principalToken(u,m)` is the Illuminate PT for the `(u,m)` pair:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L1047-L1053

```
/// @notice retrieves the ERC5095 token for the given market
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @return address of the ERC5095 token for the market
function principalToken(address u, uint256 m) internal returns (address) {
    return IMarketPlace(marketPlace).token(u, m, 0);
```

```
}
```

`IMarketPlace(marketPlace).token(u,m,p)` returns `markets[u][m][p]`:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L601-L611

```
/// @notice provides an interface to receive principal token addresses from
↪   markets
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param p principal value according to the MarketPlace's Principals Enum
function token(
    address u,
    uint256 m,
    uint256 p
) external view returns (address) {
    return markets[u][m][p];
}
```

`markets[u][m][p]` might be set or not set by createMarket() and setPrincipal(), there is no control for setup to be full in either of the functions:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L120-L243

```
/// @notice creates a new market for the given underlying token and maturity
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param t principal token addresses for this market
/// @param n name for the Illuminate token
/// @param s symbol for the Illuminate token
/// @param e address of the Element vault that corresponds to this market
/// @param a address of the APWine router that corresponds to this market
/// @return bool true if successful
function createMarket(
    address u,
    uint256 m,
    address[8] calldata t,
    string calldata n,
    string calldata s,
    address e,
    address a
) external authorized(admin) returns (bool) {
    {
        // Get the Illuminate principal token for this market (if one exists)
        address illuminate = markets[u][m][
            (uint256(Principals.Illuminate))
```

```
        ];

        // If illuminate PT already exists, a new market cannot be created
        if (illuminate != address(0)) {
            revert Exception(9, 0, 0, illuminate, address(0));
        }
    }

    // Create an Illuminate principal token for the new market
    address illuminateToken = address(
        new ERC5095(
            u,
            m,
            redeemer,
            lender,
            address(this),
            n,
            s,
            IERC20(u).decimals()
        )
    );

    {
        // create the principal tokens array
        address[9] memory market = [
            illuminateToken, // Illuminate
            t[0], // Swivel
            t[1], // Yield
            t[2], // Element
            t[3], // Pendle
            t[4], // Tempus
            t[5], // Sense
            t[6], // APWine
            t[7] // Notional
        ];

        // Set the market
        markets[u][m] = market;

        ...

        emit CreateMarket(u, m, market, e, a);
    }
    return true;
}

/// @notice allows the admin to set an individual market
/// @param p principal value according to the MarketPlace's Principals Enum
```

```
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a address of the new principal token
/// @return bool true if the principal set, false otherwise
function setPrincipal(
    uint8 p,
    address u,
    uint256 m,
    address a
) external authorized(admin) returns (bool) {
    // Get the current principal token for the principal token being set
    address market = markets[u][m][p];

    // Verify that it has not already been set
    if (market != address(0)) {
        revert Exception(9, 0, 0, market, address(0));
    }

    // Set the principal token in the markets mapping
    markets[u][m][p] = a;

    ...

    emit SetPrincipal(u, m, a, p);
    return true;
}
```

Bob can track setMarketPlace() calls:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L249-L262

```
/// @notice sets the address of the marketplace contract which contains the
↪   addresses of all the fixed rate markets
/// @param m the address of the marketplace contract
/// @return bool true if the address was set
function setMarketPlace(address m)
    external
    authorized(admin)
    returns (bool)
{
    if (marketPlace != address(0)) {
        revert Exception(5, 0, 0, marketPlace, address(0));
    }
    marketPlace = m;
    return true;
}
```

Observing that `marketPlace` and `IMarketPlace(marketPlace).token(u,m,0)` are set, Bob can call mint() to obtain Illuminate PTs for a given `(u,m)`.

To check that current Safe does call zero address successfully please see the POC (it's basically tenderly start script with `IERC20` and `Safe` copied over):

https://sandbox.tenderly.co/dmitriia/safe-transfer-zero

Tool used

Manual Review

Recommendation

As Lender's mint() might be not the only instance where the absence of token existence check opens up this attack surface, consider requiring token address to have code in all Safe operations.

Also, controlling that PT address obtained isn't zero is advised in all the instances where is it used, for example:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L264-L288

```
    /// @notice mint swaps the sender's principal tokens for Illuminate's ERC5095
↪   tokens in effect, this opens a new fixed rate position for the sender on
↪   Illuminate
    /// @param p principal value according to the MarketPlace's Principals Enum
    /// @param u address of an underlying asset
    /// @param m maturity (timestamp) of the market
    /// @param a amount being minted
    /// @return bool true if the mint was successful
    function mint(
        uint8 p,
        address u,
        uint256 m,
        uint256 a
    ) external unpaused(u, m, p) returns (bool) {
        // Fetch the desired principal token
        address principal = IMarketPlace(marketPlace).token(u, m, p);

+   if (principal == address(0)) revert Exception(1, p, 0, address(0),
↪   address(0));  // same Exception as paused, as an example

        // Transfer the users principal tokens to the lender contract
        Safe.transferFrom(IERC20(principal), msg.sender, address(this), a);

        // Mint the tokens received from the user
        IERC5095(principalToken(u, m)).authMint(msg.sender, a);
```


SHERLOCK

```
        emit Mint(p, u, m, a);

        return true;
    }
```

Issue H-2: Sense redeem is unavailable and funds are frozen for underlyings whose decimals are smaller than the corresponding IBT decimals

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/228

Found by

0x52, hyh

Summary

Sense version of Redeemer's redeem() compares `amount` of Sense principal token Lender had on its balance vs `redeemed` amount of underlying as a slippage check, requiring that the latter be equal or greater than the former.

As these numbers have different decimals this check blocks the redeem altogether for the tokens whose decimals are smaller than decimals of the corresponding interest bearing token, freezing the funds.

Vulnerability Detail

Sense version of redeem() assumes that Sense PT has the same decimals as underlying, performing slippage check by directly comparing the amounts.

Sense principal has decimals of the corresponding interest bearing tokens, not the decimals of the underlying. In the compound case IBT decimals are `8` and can be greater or less than underlying's.

For example, `1stJuly2023cUSDCSensePrincipalToken` has `8` decimals, as cUSDC does (instead of 6 as USDC):

https://etherscan.io/token/0x869a70c198c937801b26d2701dc8e4e8c4de354a

In this case the slippage check reverts the operation. Sense PT cannot be turned to underlying and will remain on Lender's balance this way.

On the other hand, when underlying decimals are greater than IBT decimals the slippage check becomes a noop.

Impact

Protocol users can be subject to market manipulations as Sense AMM result isn't checked for the underlyings whose decimals are higher than decimals of the corresponding IBT, say in the cDAI (8) and DAI (18) case. I.e. sandwich attacks have high possibility in this case whenever amounts are big enough.

Sense redeem will be unavailable and funds frozen for the underlyings whose decimals are smaller than decimals of the corresponding IBT, say in the cUSDC (8) and USDC (6) case.

As without working redeem() the whole Sense PT funds be frozen for all the users as it deals with the cumulative holdings of the protocol, setting the severity to be high.

Code Snippet

Sense redeem() compares `amount` of Sense PT to `redeemed` amount of underlying in order to `Verifythatunderlyingarereceived1:1`:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L335-L394

```
/// @notice redeem method signature for Sense
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param s Sense's maturity is needed to extract the pt address
/// @param a Sense's adapter for this market
/// @return bool true if the redemption was successful
function redeem(
    uint8 p,
    address u,
    uint256 m,
    uint256 s,
    address a
) external returns (bool) {
    // Check the principal is Sense
    if (p != uint8(MarketPlace.Principals.Sense)) {
        revert Exception(6, p, 0, address(0), address(0));
    }

    // Get Sense's principal token for this market
    IERC20 token = IERC20(IMarketPlace(marketPlace).token(u, m, p));

    // Cache the lender to save on SLOAD operations
    address cachedLender = lender;

    // Get the balance of tokens to be redeemed by the user
    uint256 amount = token.balanceOf(cachedLender);

    // Transfer the user's tokens to the redeem contract
    Safe.transferFrom(token, cachedLender, address(this), amount);

    // Get the starting balance to verify the amount received afterwards
    uint256 starting = IERC20(u).balanceOf(address(this));
```

```
    ...

    // Redeem the compounding token back to the underlying
    IConverter(converter).convert(
        compounding,
        u,
        IERC20(compounding).balanceOf(address(this))
    );

    // Get the amount received
    uint256 redeemed = IERC20(u).balanceOf(address(this)) - starting;

    // Verify that underlying are received 1:1 - cannot trust the adapter
    if (redeemed < amount) {
        revert Exception(13, 0, 0, address(0), address(0));
    }

    // Update the holdings for this market
    holdings[u][m] = holdings[u][m] + redeemed;
```

This way, for example, 8 decimals amount of `1stJuly2023cUSDCSensePrincipalToken`, say `1e3*1e8` for `1000PT`, is checked to be greater than `1e3*1e6` for `1000USDC`, which basically is never true. Same holds for any Sense USDC PT.

On the other hand, for example DAI, having 18 decimals, will always pass this check as Sense cDAI PT has cDAI decimals of 8, for example (from https://docs.sense.finance/developers/deployed-contracts/):

https://etherscan.io/token/0xcfA7B126c680007D0367d0286D995c6aEE53e087

Tool used

Manual Review

Recommendation

In order to verify `redeemed=IERC20(u).balanceOf(address(this))-starting` vs initial `IERC20(IMarketPlace(marketPlace).token(u,m,p))`'s balance of Lender, consider introducing the decimals adjustment multiplier, i.e. read Sense PT decimals, underlying decimals, and multiply the smaller decimals amount to match the bigger decimals one in order to compare.

Issue H-3: No returning of premium if there is no swap to PT

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/211

Found by

hyh, cccz

Summary

Swivel version of Lender's lend() has optional premium conversion parameter. If it is set, the underlying funds are swapped to Swivel PTs and Illuminate PTs are minted to the caller. If it is not set, the underlying funds resulted from filling of the Swivel orders are left on the Lender's balance and are lost to the caller.

Vulnerability Detail

There is no code covering the case of not swapping the underlying funds originated from execution of the Swivel orders. All such funds become irretrievable for the user after lend() with `e==false` call. Subsequent lend() calls will record current balance, that will include the previously realized underlying premium, as a staring point.

Notice that this is not a user mistake as lend() call without swap of the resulting premium to PT if a valid option, i.e. a user might want to obtain some Swivel PTs from the orders execution and have the premium back as a separate matter, there is no internal link between the two. lend() description also states that swapping is just an option.

Impact

Net impact for the user is underlying fund freeze. The funds can be retrieved thereafter via administrative withdraw(), but as volume of operations will grow over time such manual accounting become less and less feasible, up to be operationally impossible, i.e. up to loss of these funds to the user.

As there is no low probability assumptions, i.e. the funds are being frozen as a part of the ordinary use case, setting the severity to be high.

Code Snippet

Swivel lend() allows for optionally swapping the net underlying funds resulting from the filled Swivel orders to Swivel PTs:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L349-L370

```
/// @notice lend method signature for Swivel
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a array of amounts of underlying tokens lent to each order in the
 ↪  orders array
/// @param y Yield Space Pool for the Illuminate PT in this market
/// @param o array of Swivel orders being filled
/// @param s array of signatures for each order in the orders array
/// @param e flag to indicate if returned funds should be swapped in Yield Space
 ↪  Pool
/// @param premiumSlippage slippage limit, minimum amount to PTs to buy
/// @return uint256 the amount of principal tokens lent out
function lend(
    uint8 p,
    address u,
    uint256 m,
    uint256[] memory a,
    address y,
    Swivel.Order[] calldata o,
    Swivel.Components[] calldata s,
    bool e,
    uint256 premiumSlippage
) external unpaused(u, m, p) returns (uint256) {
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L407-L435

```
uint256 received;
{
    // Get the starting amount of principal tokens
    uint256 startingZcTokens = IERC20(
        IMarketPlace(marketPlace).token(u, m, p)
    ).balanceOf(address(this));

    // Fill the given orders on Swivel
    ISwivel(swivelAddr).initiate(o, a, s);

    if (e) {
        // Calculate the premium
        uint256 premium = IERC20(u).balanceOf(address(this)) -
            starting;

        // Swap the premium for Illuminate principal tokens
        swivelLendPremium(u, m, y, premium, premiumSlippage);
    }
```


SHERLOCK

```
        // Compute how many principal tokens were received
        received =
            IERC20(IMarketPlace(marketPlace).token(u, m, p)).balanceOf(
                address(this)
            ) -
            startingZcTokens;
    }

    // Mint Illuminate principal tokens to the user
    IERC5095(principalToken(u, m)).authMint(msg.sender, received);
```

However, if this swap doesn't take place, i.e. when `e` being false, the corresponding underlying amount is left on Lender's balance and becomes inaccessible for the user.

It can only be rescued manually with admin's withdraw():

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L831-L852

```
/// @notice allows the admin to withdraw the given token, provided the holding
↪  period has been observed
/// @param e Address of token to withdraw
/// @return bool true if successful
function withdraw(address e) external authorized(admin) returns (bool) {
    uint256 when = withdrawals[e];
    if (when == 0) {
        revert Exception(18, 0, 0, address(0), address(0));
    }

    if (block.timestamp < when) {
        revert Exception(19, 0, 0, address(0), address(0));
    }

    delete withdrawals[e];

    delete fees[e];

    IERC20 token = IERC20(e);
    Safe.transfer(token, admin, token.balanceOf(address(this)));

    return true;
}
```

Tool used

Manual Review

Recommendation

Consider returning the funds originated from Swivel orders execution back to the caller if no underlying to PT swap is requested:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L417-L424

```diff
+                // Calculate the premium
+                uint256 premium = IERC20(u).balanceOf(address(this)) -
+                    starting;
+                if (premium > 0)
                    if (e) {
-                        // Calculate the premium
-                        uint256 premium = IERC20(u).balanceOf(address(this)) -
-                            starting;

                        // Swap the premium for Illuminate principal tokens
                        swivelLendPremium(u, m, y, premium, premiumSlippage);
-                    }
+                    } else {
+               // Return the premium if not swapping
+               Safe.transferFrom(IERC20(u), address(this), msg.sender, premium);
+                    }
```


SHERLOCK

Issue H-4: Lend or mint after maturity

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/208

Found by

Jeiwan, 0x52, Holmgren, kenzo, HonorLt

Summary

The protocol does not forbid lending or minting after the maturity leaving the possibility to profit from early users.

Vulnerability Detail

Let's take the mint function as an example:

```
function mint(
    uint8 p,
    address u,
    uint256 m,
    uint256 a
) external unpaused(u, m, p) returns (bool) {
    // Fetch the desired principal token
    address principal = IMarketPlace(marketPlace).token(u, m, p);

    // Transfer the users principal tokens to the lender contract
    Safe.transferFrom(IERC20(principal), msg.sender, address(this), a);

    // Mint the tokens received from the user
    IERC5095(principalToken(u, m)).authMint(msg.sender, a);

    emit Mint(p, u, m, a);

    return true;
}
```

It is a simple function that accepts the principal token and mints the corresponding ERC5095 tokens in return. There are no restrictions on timing, the user can mint even after the maturity. Malicious actors can take this as an advantage to pump their bags on behalf of legitimate early users.

Scenario:

1) Legitimate users lend and mint their ERC5095 tokens before maturity.

2) When the maturity kicks in, lender tokens are redeemed and holdings are updated.
3) Legitimate users try to redeem their ERC5095 for the underlying tokens. The formula is `(amount*holdings[u][m])/token.totalSupply();`
4) A malicious actor sandwiches legitimate users, and mints the ERC5095 thus increasing the totalSupply and reducing other user shares. Then redeem principals again and burn their own shares for increased rewards.

Example with concrete values:

1) userA deposits `100` tokens, user B deposits `200` tokens. The total supply minted is `300` ERC5095 tokens.
2) After the maturity the redemption happens and now let's say `holdings[u][m]` is `330` (+30).
3) userA tries to redeem the underlying. The expected amount is: `100*330/300 =110`. However, this action is frontrunned by userC (malicious) who mints yet another `500` tokens post-maturity. The total supply becomes `800`. The real value userA now receives is: `110*330/800=45.375`.
4) After that the malicious actor userC invokes the redemption again, and the `holdings[u][m]` is now `330-45.375+550=834.625`.
5) userC redeems the underlying: `500*834.625/700 =596.16` (expected was `550`).
6) Now all the remaining users will also slightly benefit, e.g. in this case userB redeems what's left: `200*238.46/200=238.46` (expected was `220`).

Impact

The amount legitimate users receive will be devaluated, while malicious actor can increase their ROI without meaningfully contributing to the protocol and locking their tokens.

Code Snippet

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L264-L288

Tool used

Manual Review

Recommendation

Lend/mint should be forbidden post-maturity.

Issue H-5: There are no Illuminate PT transfers from the owner in ERC5095's withdraw and redeem before maturity

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/195

Found by

cryptphi, bin2chen, Holmgren, hansfriese, hyh

Summary

Illuminate PT's withdraw() and redeem() sell the PTs via pool and return the obtained underlying to the receiver `r`, but do not transfer these shares from the owner, i.e. selling PTs from the contract balance, if there is any. There is no accounting for the owner side that something was sold on his behalf, i.e. any owner can sell all Pts from the contract balance without spending anything.

Normal operation, on the other hand, is inaccessible, if there are not enough PTs on the Illuminate PT contract balance, withdraw and redeem will be reverting, i.e. an owner is not able to provide anything.

Vulnerability Detail

`IMarketPlace(marketplace).sellPrincipalToken` transfers the PT to be sold from Illuminate PT contract via Marketplace's `Safe.transferFrom(IERC20(address(pool.fyToken())),msg.sender,address(pool),a)`, but these aren't owner's PTs as nothing was transferred from the owner before and the owner's ERC5095 record aren't updated anyhow.

I.e. `o==msg.sender` check does virtually nothing as `o` record neither checked nor changed as a result of withdraw() and redeem(). Say `o` might not own anything at all at this Illuminate PT contract, the calls succeed anyway as long as the contract has PTs on the balance.

Impact

Anyone can empty the total holdings of Illuminate PTs of the ERC5095 contract by calling withdraw() or redeem(). This is fund stealing impact for the PTs on the balance.

Valid withdraw() or redeem() from real owners will be reverted as long as there will not be enough Illuminate PTs on the balance, i.e. withdrawal before maturity functionality will not be available at all. Although it can be done directly via Marketplace, there also are downstream systems integrated specifically with Illuminate PT contract and execution delays overall do have monetary costs.

Setting the severity to be high as this is the violation of core business logic with total impact from temporal fund freezing to fund loss.

Code Snippet

withdraw() sells the PTs held by Illuminate PT contract, if any, reverting if there are not enough funds on the balance, not transferring PTs from the owner:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L204-L249

```
/// @notice At or after maturity, Burns `shares` from `owner` and sends exactly
↪  `assets` of underlying tokens to `receiver`. Before maturity, sends `assets`
↪  by selling shares of PT on a YieldSpace AMM.
/// @param a The amount of underlying tokens withdrawn
/// @param r The receiver of the underlying tokens being withdrawn
/// @param o The owner of the underlying tokens
/// @return uint256 The amount of principal tokens burnt by the withdrawal
function withdraw(
    uint256 a,
    address r,
    address o
) external override returns (uint256) {
    // Pre maturity
    if (block.timestamp < maturity) {
        uint128 shares = Cast.u128(previewWithdraw(a));
        // If owner is the sender, sell PT without allowance check
        if (o == msg.sender) {
            uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                underlying,
                maturity,
                shares,
                Cast.u128(a - (a / 100))
            );
            Safe.transfer(IERC20(underlying), r, returned);
            return returned;
            // Else, sell PT with allowance check
        } else {
            uint256 allowance = _allowance[o][msg.sender];
            if (allowance < shares) {
                revert Exception(
                    20,
                    allowance,
                    shares,
                    address(0),
```

```
                    address(0)
                );
            }
            _allowance[o][msg.sender] = allowance - shares;
            uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                underlying,
                maturity,
                Cast.u128(shares),
                Cast.u128(a - (a / 100))
            );
            Safe.transfer(IERC20(underlying), r, returned);
            return returned;
        }
    }
```

redeem() sells the PTs held by Illuminate PT contract, if any:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L279-L319

```
/// @notice At or after maturity, burns exactly `shares` of Principal Tokens from
↪  `owner` and sends `assets` of underlying tokens to `receiver`. Before
↪  maturity, sends `assets` by selling `shares` of PT on a YieldSpace AMM.
/// @param s The number of shares to be burned in exchange for the underlying
↪  asset
/// @param r The receiver of the underlying tokens being withdrawn
/// @param o Address of the owner of the shares being burned
/// @return uint256 The amount of underlying tokens distributed by the redemption
function redeem(
    uint256 s,
    address r,
    address o
) external override returns (uint256) {
    // Pre-maturity
    if (block.timestamp < maturity) {
        uint128 assets = Cast.u128(previewRedeem(s));
        // If owner is the sender, sell PT without allowance check
        if (o == msg.sender) {
            uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                underlying,
                maturity,
                Cast.u128(s),
                assets - (assets / 100)
            );
            Safe.transfer(IERC20(underlying), r, returned);
            return returned;
            // Else, sell PT with allowance check
        } else {
```

```
            uint256 allowance = _allowance[o][msg.sender];
            if (allowance < s) {
                revert Exception(20, allowance, s, address(0), address(0));
            }
            _allowance[o][msg.sender] = allowance - s;
            uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                underlying,
                maturity,
                Cast.u128(s),
                assets - (assets / 100)
            );
            Safe.transfer(IERC20(underlying), r, returned);
            return returned;
        }
        // Post-maturity
    } else {
```

sellPrincipalToken() both functions above invoke transfers `IERC20(address(pool.fyToken()))` from `msg.sender`, which is the calling Illuminate PT contract:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L279-L314

```
/// @notice sells the PT for the underlying via the pool
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a amount of PTs to sell
/// @param s slippage cap, minimum amount of underlying that must be received
/// @return uint128 amount of underlying bought
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);

    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
    );

    // Execute the swap
    uint128 received = pool.sellFYToken(msg.sender, uint128(expected));
    emit Swap(u, m, address(pool.fyToken()), u, received, a, msg.sender);

    return received;
}
```

Notice that after maturity there is no need to transfer, and the burning is correctly performed by authRedeem():

For example redeem() calls authRedeem():

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L318-L344

```
    // Post-maturity
} else {
    if (o == msg.sender) {
        return
            IRedeemer(redeemer).authRedeem(
                underlying,
                maturity,
                msg.sender,
                r,
                s
            );
    } else {
        uint256 allowance = _allowance[o][msg.sender];
        if (allowance < s) {
            revert Exception(20, allowance, s, address(0), address(0));
        }
        _allowance[o][msg.sender] = allowance - s;
        return
            IRedeemer(redeemer).authRedeem(
                underlying,
                maturity,
                o,
                r,
                s
            );
    }
}
```

authRedeem() burns the shares from the owner `f`:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L443-L470

```
function authRedeem(
    address u,
    uint256 m,
    address f,
    address t,
    uint256 a
)
    external
    authorized(IMarketPlace(marketPlace).token(u, m, 0))
    returns (uint256)
{
    // Get the principal token for the given market
    IERC5095 pt = IERC5095(IMarketPlace(marketPlace).token(u, m, 0));

    // Make sure the market has matured
    uint256 maturity = pt.maturity();
    if (block.timestamp < maturity) {
        revert Exception(7, maturity, 0, address(0), address(0));
    }

    // Calculate the amount redeemed
    uint256 redeemed = (a * holdings[u][m]) / pt.totalSupply();

    // Update holdings of underlying
    holdings[u][m] = holdings[u][m] - redeemed;

    // Burn the user's principal tokens
    pt.authBurn(f, a);
```

Tool used

Manual Review

Recommendation

Consider performing PT transfer before selling them via Marketplace, for example:

withdraw():

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L204-L249

```
    /// @notice At or after maturity, Burns `shares` from `owner` and sends
↪   exactly `assets` of underlying tokens to `receiver`. Before maturity, sends
↪   `assets` by selling shares of PT on a YieldSpace AMM.
    /// @param a The amount of underlying tokens withdrawn
    /// @param r The receiver of the underlying tokens being withdrawn
    /// @param o The owner of the underlying tokens
    /// @return uint256 The amount of principal tokens burnt by the withdrawal
    function withdraw(
        uint256 a,
        address r,
        address o
    ) external override returns (uint256) {
        // Pre maturity
        if (block.timestamp < maturity) {
            uint128 shares = Cast.u128(previewWithdraw(a));
+           _transfer(o, address(this), shares);
            // If owner is the sender, sell PT without allowance check
            if (o == msg.sender) {
                uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                    underlying,
                    maturity,
                    shares,
                    Cast.u128(a - (a / 100))
                );
                Safe.transfer(IERC20(underlying), r, returned);
                return returned;
                // Else, sell PT with allowance check
            } else {
                uint256 allowance = _allowance[o][msg.sender];
                if (allowance < shares) {
                    revert Exception(
                        20,
                        allowance,
                        shares,
                        address(0),
                        address(0)
                    );
                }
                _allowance[o][msg.sender] = allowance - shares;
                uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                    underlying,
                    maturity,
                    Cast.u128(shares),
                    Cast.u128(a - (a / 100))
                );
                Safe.transfer(IERC20(underlying), r, returned);
```

```
            return returned;
        }
    }
```

redeem():

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L279-L319

```
    /// @notice At or after maturity, burns exactly `shares` of Principal Tokens
↪   from `owner` and sends `assets` of underlying tokens to `receiver`. Before
↪   maturity, sends `assets` by selling `shares` of PT on a YieldSpace AMM.
    /// @param s The number of shares to be burned in exchange for the underlying
↪   asset
    /// @param r The receiver of the underlying tokens being withdrawn
    /// @param o Address of the owner of the shares being burned
    /// @return uint256 The amount of underlying tokens distributed by the
↪   redemption
    function redeem(
        uint256 s,
        address r,
        address o
    ) external override returns (uint256) {
        // Pre-maturity
        if (block.timestamp < maturity) {
+       _transfer(o, address(this), s);
            uint128 assets = Cast.u128(previewRedeem(s));
            // If owner is the sender, sell PT without allowance check
            if (o == msg.sender) {
                uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                    underlying,
                    maturity,
                    Cast.u128(s),
                    assets - (assets / 100)
                );
                Safe.transfer(IERC20(underlying), r, returned);
                return returned;
                // Else, sell PT with allowance check
            } else {
                uint256 allowance = _allowance[o][msg.sender];
                if (allowance < s) {
                    revert Exception(20, allowance, s, address(0), address(0));
                }
                _allowance[o][msg.sender] = allowance - s;
                uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                    underlying,
                    maturity,
                    Cast.u128(s),
```

```
                    assets - (assets / 100)
                );
                Safe.transfer(IERC20(underlying), r, returned);
                return returned;
            }
            // Post-maturity
        } else {
```

Discussion

Evert0x

@sourabhmarathe I would like to know why you disagree with severity, seems like a valid high to me.

IIIIIIOOO

I believe the ERC5095 doesn't normally hold any balance. The tests in https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/71 use `deal()` to give it a balance, which wouldn't happen in reality. The bug here looks to be that `withdraw()` will fail before maturity since `MarketPlace.sellPrincipalToken()` won't have any balance to pull out of the ERC5095. As the submitter mentions, `MarketPlace` can be called by the holder directly in order to sell before maturity, so the function is broken, but there's a workaround. @sourabhmarathe please correct me if I'm mistaken

sourabhmarathe

Originally, I felt that this did not put user funds at risk but certainly agreed with the report (I believe we have a fix for it). However, it seems like this is a serious enough problem where we can keep the severity level High. I removed the dispute label.

Issue H-6: Yield, Swivel, Element, APWine and Sense lend() are subject to reentracy resulting in Illuminate PT over-mint

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/179

Found by

cryptphi, Jeiwan, windowhan_kalosec, Holmgren, kenzo, HonorLt, hyh, minhtrng

Summary

Lender's lend() versions for Yield, Swivel, Element, APWine and Sense use balance difference for the net result calculation, i.e. how much Illuminate PTs to mint for the caller, and call user-provided contract to perform the swapping. The functions aren't protected from reentrancy.

This opens up an attack surface when the functions are being called repetitively, and, while the first call result is accounted once, nested calls, dealing with the same type of PTs, are accounted multiple times, leading to severe Illuminate PT over-mint.

Vulnerability Detail

Taking Yield version as an example, Bob the attacker can provide custom-made contract `y` instead of Yield Space Pool. `y` do call the real pool, but before that it calls the same lend() with the same parameters (apart from amount), so `y` got called again.

Let's say it happens 2 extra times. Let's say the first call is done with `10DAI`, the second with `100DAI`, the third with `10^6DAI`, i.e. Bob needs to provide `10^6+10^2+10^1DAI`. Let's say it is done right before maturity and there is no discounting remaining, i.e. `1DAI=1PT`.

The result of the first yield() call will be accounted once, as designed. The result of the second, nested, call, will be accounted twice as it mints to the user according to the yield() call performed and increases the Yield PT balance, which is counted in the first lend(). The result of the third call will be accounted in all lend() functions.

This way first lend() will mint `1*10^6+1*10^2+1*10^1` as it will be the total Yield PT balance difference from the three yield() calls it performed directly and nested, i.e. the balance will be counted before the swapping started, the second time it will be counted when all three swaps be completed. The second lend() will mint `1*10^6+1*10^2` as it be finished before first yield() do its swap. The third lend() will mint `1*10^6`, having no further calls nested.

Bob will get `3*10^6+2*10^2+1*10^1` Illuminate PT minted for the `10^6+10^2+10^1` DAI provided.

SHERLOCK

Impact

The impact is massive Illuminate PTs over-mint that result in attacker being able to steal the funds of all other users by redeeming first the whole underlying amount due to the type of Illuminate PTs he obtained.

As there are no low probability prerequisites, setting the severity to be high.

Code Snippet

Similar in all: Bob creates a wrapper that calls the same version of lend() with the same parameters, then calls the correct pool. In each version of lend() there are a user-provided contract that is called to perform the operation, allowing for reentracy.

Yield lend() calls yield() with user-provided contract `y`, that is called in-between balance recording:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L290-L347

```
/// @notice lend method for the Illuminate and Yield protocols
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a amount of underlying tokens to lend
/// @param y Yield Space Pool for the principal token
/// @param minimum slippage limit, minimum amount to PTs to buy
/// @return uint256 the amount of principal tokens lent out
function lend(
    uint8 p,
    address u,
    uint256 m,
    uint256 a,
    address y,
    uint256 minimum
) external unpaused(u, m, p) returns (uint256) {
    // Check that the principal is Illuminate or Yield
    if (
        p != uint8(MarketPlace.Principals.Illuminate) &&
        p != uint8(MarketPlace.Principals.Yield)
    ) {
        revert Exception(6, 0, 0, address(0), address(0));
    }

    // Get principal token for this market
    address principal = IMarketPlace(marketPlace).token(u, m, p);

    // Extract fee
    fees[u] = fees[u] + a / feenominator;
```


SHERLOCK

```
    // Transfer underlying from user to the lender contract
    Safe.transferFrom(IERC20(u), msg.sender, address(this), a);

    if (p == uint8(MarketPlace.Principals.Yield)) {
        // Make sure the Yield Space Pool matches principal token
        address fyToken = IYield(y).fyToken();
        if (IYield(y).fyToken() != principal) {
            revert Exception(12, 0, 0, fyToken, principal);
        }
    }

    // Swap underlying for PTs to lender
    uint256 returned = yield(
        u,
        y,
        a - a / feenominator,
        address(this),
        principal,
        minimum
    );

    // Mint Illuminate PTs to msg.sender
    IERC5095(principalToken(u, m)).authMint(msg.sender, returned);

    emit Lend(p, u, m, returned, a, msg.sender);

    return returned;
}
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L919-L957

```
/// @notice swaps underlying premium via a Yield Space Pool
/// @dev this method is only used by the Yield, Illuminate and Swivel protocols
/// @param u address of an underlying asset
/// @param y Yield Space Pool for the principal token
/// @param a amount of underlying tokens to lend
/// @param r the receiving address for PTs
/// @param p the principal token in the Yield Space Pool
/// @param m the minimum amount to purchase
/// @return uint256 the amount of tokens sent to the Yield Space Pool
function yield(
    address u,
    address y,
    uint256 a,
    address r,
```

```
    address p,
    uint256 m
) internal returns (uint256) {
    // Get the starting balance (to verify receipt of tokens)
    uint256 starting = IERC20(p).balanceOf(r);

    // Get the amount of tokens received for swapping underlying
    uint128 returned = IYield(y).sellBasePreview(Cast.u128(a));

    // Send the remaining amount to the Yield pool
    Safe.transfer(IERC20(u), y, a);

    // Lend out the remaining tokens in the Yield pool
    IYield(y).sellBase(r, returned);

    // Get the ending balance of principal tokens (must be at least starting +
 ↪  returned)
    uint256 received = IERC20(p).balanceOf(r) - starting;

    // Verify receipt of PTs from Yield Space Pool
    if (received <= m) {
        revert Exception(11, received, m, address(0), address(0));
    }

    return received;
}
```

Similarly, Swivel lend() calls yield() with user-supplied Yield Space Pool `y` via swivelLendPremium():

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L349-L449

```
/// @notice lend method signature for Swivel
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a array of amounts of underlying tokens lent to each order in the
 ↪  orders array
/// @param y Yield Space Pool for the Illuminate PT in this market
/// @param o array of Swivel orders being filled
/// @param s array of signatures for each order in the orders array
/// @param e flag to indicate if returned funds should be swapped in Yield Space
 ↪  Pool
/// @param premiumSlippage slippage limit, minimum amount to PTs to buy
/// @return uint256 the amount of principal tokens lent out
function lend(
    uint8 p,
```

```
    address u,
    uint256 m,
    uint256[] memory a,
    address y,
    Swivel.Order[] calldata o,
    Swivel.Components[] calldata s,
    bool e,
    uint256 premiumSlippage
) external unpaused(u, m, p) returns (uint256) {
    {
        // Check that the principal is Swivel
        if (p != uint8(MarketPlace.Principals.Swivel)) {
            ...
        }

        // Lent represents the total amount of underlying to be lent
        uint256 lent = swivelAmount(a);

        // Transfer underlying token from user to Illuminate
        Safe.transferFrom(IERC20(u), msg.sender, address(this), lent);

        // Get the underlying balance prior to calling initiate
        uint256 starting = IERC20(u).balanceOf(address(this));

        // Verify and collect the fee
        {
            ...
        }

        uint256 received;
        {
            // Get the starting amount of principal tokens
            uint256 startingZcTokens = IERC20(
                IMarketPlace(marketPlace).token(u, m, p)
            ).balanceOf(address(this));

            // Fill the given orders on Swivel
            ISwivel(swivelAddr).initiate(o, a, s);

            if (e) {
                // Calculate the premium
                uint256 premium = IERC20(u).balanceOf(address(this)) -
                    starting;

                // Swap the premium for Illuminate principal tokens
                swivelLendPremium(u, m, y, premium, premiumSlippage);
            }
```


SHERLOCK

```
            // Compute how many principal tokens were received
            received =
                IERC20(IMarketPlace(marketPlace).token(u, m, p)).balanceOf(
                    address(this)
                ) -
                startingZcTokens;
        }

        // Mint Illuminate principal tokens to the user
        IERC5095(principalToken(u, m)).authMint(msg.sender, received);

        {
            emit Lend(
                ...
            );
        }
        return received;
    }
}
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L959-L979

```
/// @notice lends the leftover underlying premium to the Illuminate PT's Yield
↪ Space Pool
function swivelLendPremium(
    address u,
    uint256 m,
    address y,
    uint256 p,
    uint256 slippageTolerance
) internal {
    // Lend remaining funds to Illuminate's Yield Space Pool
    uint256 swapped = yield(
        u,
        y,
        p,
        address(this),
        IMarketPlace(marketPlace).token(u, m, 0),
        slippageTolerance
    );

    // Mint the remaining tokens
    IERC5095(principalToken(u, m)).authMint(msg.sender, swapped);
}
```

This way both Yield and Swivel call yield() with user-supplied pool `y` and mint the difference obtained with the `y` call to a user.

Element lend calls elementSwap() with user-supplied pool `e` and mints the balance difference:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L451-L511

```
/// @notice lend method signature for Element
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a amount of underlying tokens to lend
/// @param r slippage limit, minimum amount to PTs to buy
/// @param d deadline is a timestamp by which the swap must be executed
/// @param e Element pool that is lent to
/// @param i the id of the pool
/// @return uint256 the amount of principal tokens lent out
function lend(
    uint8 p,
    address u,
    uint256 m,
    uint256 a,
    uint256 r,
    uint256 d,
    address e,
    bytes32 i
) external unpaused(u, m, p) returns (uint256) {
    // Get the principal token for this market for Element
    address principal = IMarketPlace(marketPlace).token(u, m, p);

    // Transfer underlying token from user to Illuminate
    Safe.transferFrom(IERC20(u), msg.sender, address(this), a);

    // Track the accumulated fees
    fees[u] = fees[u] + a / feenominator;

    uint256 purchased;
    {
        ...

        // Conduct the swap on Element
        purchased = elementSwap(e, swap, fund, r, d);
    }

    // Mint tokens to the user
    IERC5095(principalToken(u, m)).authMint(msg.sender, purchased);
```

```
    emit Lend(p, u, m, purchased, a, msg.sender);
    return purchased;
}
```

elementSwap() similarly calls user-supplied `e` to perform the swapping and mints the balance difference:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L1000-L1028

```
/// @notice executes a swap for and verifies receipt of Element PTs
function elementSwap(
    address e,
    Element.SingleSwap memory s,
    Element.FundManagement memory f,
    uint256 r,
    uint256 d
) internal returns (uint256) {
    // Get the principal token
    address principal = address(s.assetOut);

    // Get the intial balance
    uint256 starting = IERC20(principal).balanceOf(address(this));

    // Conduct the swap on Element
    IElementVault(e).swap(s, f, r, d);

    // Get how many PTs were purchased by the swap call
    uint256 purchased = IERC20(principal).balanceOf(address(this)) -
        starting;

    // Verify that a minimum amount was received
    if (purchased < r) {
        revert Exception(11, 0, 0, address(0), address(0));
    }

    // Return the net amount of principal tokens acquired after the swap
    return purchased;
}
```

APWine lend() in the same manner calls user-supplied pool `x` and mints the balance difference `received`:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L562-L621

```
/// @notice lend method signature for APWine
/// @param p principal value according to the MarketPlace's Principals Enum
```

```
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a amount of underlying tokens to lend
/// @param r slippage limit, minimum amount to PTs to buy
/// @param d deadline is a timestamp by which the swap must be executed
/// @param x APWine router that executes the swap
/// @param pool the AMM pool used by APWine to execute the swap
/// @return uint256 the amount of principal tokens lent out
function lend(
    uint8 p,
    address u,
    uint256 m,
    uint256 a,
    uint256 r,
    uint256 d,
    address x,
    address pool
) external unpaused(u, m, p) returns (uint256) {
    address principal = IMarketPlace(marketPlace).token(u, m, p);

    // Transfer funds from user to Illuminate
    Safe.transferFrom(IERC20(u), msg.sender, address(this), a);

    uint256 lent;
    {
        // Add the accumulated fees to the total
        uint256 fee = a / feenominator;
        fees[u] = fees[u] + fee;

        // Calculate amount to be lent out
        lent = a - fee;
    }

    // Get the starting APWine token balance
    uint256 starting = IERC20(principal).balanceOf(address(this));

    // Swap on the APWine Pool using the provided market and params
    IAPWineRouter(x).swapExactAmountIn(
        pool,
        apwinePairPath(),
        apwineTokenPath(),
        lent,
        r,
        address(this),
        d,
        address(0)
    );
```

```
    // Calculate the amount of APWine principal tokens received after the swap
    uint256 received = IERC20(principal).balanceOf(address(this)) -
        starting;

    // Mint Illuminate zero coupons
    IERC5095(principalToken(u, m)).authMint(msg.sender, received);

    emit Lend(p, u, m, received, a, msg.sender);
    return received;
}
```

Sense lend() also directly calls user-supplied AMM x and mints the balance difference to a caller:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L681-L741

```
/// @notice lend method signature for Sense
/// @dev this method can be called before maturity to lend to Sense while minting
↪  Illuminate tokens
/// @dev Sense provides a [divider] contract that splits [target] assets
↪  (underlying) into PTs and YTs. Each [target] asset has a [series] of
↪  contracts, each identifiable by their [maturity].
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a amount of underlying tokens to lend
/// @param r slippage limit, minimum amount to PTs to buy
/// @param x AMM that is used to conduct the swap
/// @param s Sense's maturity for the given market
/// @param adapter Sense's adapter necessary to facilitate the swap
/// @return uint256 the amount of principal tokens lent out
function lend(
    uint8 p,
    address u,
    uint256 m,
    uint128 a,
    uint256 r,
    address x,
    uint256 s,
    address adapter
) external unpaused(u, m, p) returns (uint256) {
    // Retrieve the principal token for this market
    IERC20 token = IERC20(IMarketPlace(marketPlace).token(u, m, p));

    // Transfer funds from user to Illuminate
    Safe.transferFrom(IERC20(u), msg.sender, address(this), a);
```

```
    // Determine the fee
    uint256 fee = a / feenominator;

    // Add the accumulated fees to the total
    fees[u] = fees[u] + fee;

    // Determine lent amount after fees
    uint256 lent = a - fee;

    // Stores the amount of principal tokens received in swap for underlying
    uint256 received;
    {
        // Get the starting balance of the principal token
        uint256 starting = token.balanceOf(address(this));

        // Swap those tokens for the principal tokens
        ISensePeriphery(x).swapUnderlyingForPTs(adapter, s, lent, r);

        // Calculate number of principal tokens received in the swap
        received = token.balanceOf(address(this)) - starting;

        // Verify that we received the principal tokens
        if (received < r) {
            revert Exception(11, 0, 0, address(0), address(0));
        }
    }

    // Mint the Illuminate tokens based on the returned amount
    IERC5095(principalToken(u, m)).authMint(msg.sender, received);

    emit Lend(p, u, m, received, a, msg.sender);
    return received;
}
```

Tool used

Manual Review

Recommendation

Consider adding reentracy guard modifier to Yield, Swivel, Element, APWine and Sense lend() functions of the Lender.

Notice that although Pendle, Tempus and Notional versions of lend() look to be resilient to the attack as they use either internal address (Pendle and Notional) or verify the supplied address (Tempus, https://github.com/tempus-finance/fixed-income-protocol/blob/master/contracts/TempusController.sol#L63) the same reentracy guard modifier can be used there as well as a general approach as these functions still mint the recorded balance difference to a user and there might exist yet unnoticed possibility to game it.

In all these cases either direct removal of the attack surface or precautious control for it do justify the reentracy guard gas cost.

Issue H-7: Lender#lend for Sense has mismatched decimals

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/164

Found by

0x52

Summary

The decimals of the Sense principal token don't match the decimals of the ERC5095 vault it mints shares to. This can be abused on the USDC market to mint a large number of shares to steal yield from all other users.

Vulnerability Detail

```
uint256 received;
{
    // Get the starting balance of the principal token
    uint256 starting = token.balanceOf(address(this));

    // Swap those tokens for the principal tokens
    ISensePeriphery(x).swapUnderlyingForPTs(adapter, s, lent, r);

    // Calculate number of principal tokens received in the swap
    received = token.balanceOf(address(this)) - starting;

    // Verify that we received the principal tokens
    if (received < r) {
        revert Exception(11, 0, 0, address(0), address(0));
    }
}

// Mint the Illuminate tokens based on the returned amount
IERC5095(principalToken(u, m)).authMint(msg.sender, received);
```

Sense principal tokens for DIA and USDC are 8 decimals to match the decimals of the underlying cTokens, cUSDC and cDAI. The decimals of the ERC5095 vault matches the underlying of the vault. This creates a disparity in decimals that aren't adjusted for in Lender#lend for Sense, which assumes that the vault and Sense principal tokens match in decimals. In the example of USDC the ERC5095 will be 6 decimals but the sense token will be 8 decimals. Each 1e6 USDC token will result in ~1e8 Sense tokens being received. Since the contract mints based on the difference in the number of sense tokens before and after the call, it will mint ~100x the number of vault shares than it should. Since the final yield is distributed pro-rata to the number of shares, the user who minted with sense will be entitled to much more yield than they should be and everyone else will get substantially less.

Impact

User can mint large number of shares to steal funds from other users

Code Snippet

Lender.sol#L693-L741

Tool used

Manual Review

Recommendation

Query the decimals of the Sense principal and use that to adjust the decimals to match the decimals of the vault.

Issue H-8: Swivel redeem function parameter signature mismatch in Redeemer.sol

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/145

Found by

ctf_sec

Summary

Swivel redeem function signature mismatch in Redeemer.sol

Vulnerability Detail

The redeem function for Swivel is implemented below:

```
if (p == uint8(MarketPlace.Principals.Swivel)) {
    // Redeems principal tokens from Swivel
    if (!ISwivel(swivelAddr).redeemZcToken(u, maturity, amount)) {
        revert Exception(15, 0, 0, address(0), address(0));
    }
```

We can look into the redeemZcToken interface:

```
function redeemZcToken(
    address u,
    uint256 m,
    uint256 a
) external returns (bool);
```

And we compare with the redeemZcToken implementation for Swivel's github:

https://github.com/Swivel-Finance/swivel/blob/3cc31302f84c2b1777a53c11b22c58ec6ef17888/contracts/v3/src/Swivel.sol#L1007

```
/// @notice Allows zcToken holders to redeem their tokens for underlying tokens
↪   after maturity has been reached (via MarketPlace).
/// @param p Protocol Enum value associated with this market pair
/// @param u Underlying token address associated with the market
/// @param m Maturity timestamp of the market
/// @param a Amount of zcTokens being redeemed
function redeemZcToken(
    uint8 p,
    address u,
    uint256 m,
```

```
    uint256 a
) external returns (bool) {
```

We miss-matched the parameter and function signature!

Impact

Redeem for swivel will not work.

Code Snippet

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L277-L282

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/interfaces/ISwivel.sol#L6-L19

Tool used

Manual Review

Recommendation

We recommend making the interface align with the implementation:

We change from

```
function redeemZcToken(
    address u,
    uint256 m,
    uint256 a
) external returns (bool);
```

to

```
function redeemZcToken(
    uint8 p,
    address u,
    uint256 m,
    uint256 a
) external returns (bool);
```

and we change from

```
if (!ISwivel(swivelAddr).redeemZcToken(u, maturity, amount)) {
    revert Exception(15, 0, 0, address(0), address(0));
```

```
}
```

to

```
if (!ISwivel(swivelAddr).redeemZcToken(p, u, maturity, amount)) {
    revert Exception(15, 0, 0, address(0), address(0));
}
```

Discussion

IllIllI000

@sourabhmarathe This is the version that matches the address in Contracts.sol, and it has the parameters used in the contest code: https://github.com/Swivel-Finance/swivel/blob/3cc31302f84c2b1777a53c11b22c58ec6ef17888/contracts/v2/swivel/Swivel.sol#L482

sourabhmarathe

Yes, that's right. However, Swivel v3 will be launched well in advance of Illuminate's launch. In lieu of a fork-mode test, we have manually confirmed the viability of Swivel up to this point. We will make sure to update the fork-mode test to v3 as part of this review process prior to launch.

SHERLOCK

Issue H-9: Users can mint free Illuminate PTs if underlying decimals don't match external PTs

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/120

Found by

IllIllI

Summary

Users can mint free Illuminate PTs if underlying decimals don't match external PTs

Vulnerability Detail

The Illuminate PTs always match the decimals of the underlying, but when external PTs are used for minting Illuminate PTs, the amount minted is not adjusted for the differences in decimals.

Impact

Users can inflate away the value of Illuminate PTs by minting using external PTs with different decimals than the underlying

Code Snippet

There are no conversions based on decimals - one input external PT results in one Illuminate PT:

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

      function mint(
          uint8 p,
          address u,
          uint256 m,
          uint256 a
      ) external unpaused(u, m, p) returns (bool) {
          // Fetch the desired principal token
          address principal = IMarketPlace(marketPlace).token(u, m, p);

          // Transfer the users principal tokens to the lender contract
          Safe.transferFrom(IERC20(principal), msg.sender, address(this),
 ↪  a);

          // Mint the tokens received from the user
          IERC5095(principalToken(u, m)).authMint(msg.sender, a);
```

```
        emit Mint(p, u, m, a);

        return true;
    }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L270-L288

For example, Swivel tokens are all locked at 18 decimals, Pendle uses the decimals of the yield token (e.g. cDai) rather than the decimals of the underlying, and (Notional)[https://github.com/notional-finance/wrapped-fcash/blob/ad5c145d9988eeee6e36cf93cc3412449e4e7eba/contracts/wfCashBase.sol#L103] locks the decimals to 8.

Tool used

Manual Review

Recommendation

Convert the decimals of the PT to those of the underlying, and adjust the number of Illuminate PTs minted based on that conversion

Issue H-10: Wrong Illuminate PT allowance checks lead to loss of principal

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/118

Found by

IIIIIII

Summary

Wrong Illuminate PT allowance checks lead to loss of principal

Vulnerability Detail

The `ERC5095.withdraw()` function, when called after maturity by a user with an allowance, incorrectly uses the amount of underlying rather than the number of shares the underlying is worth, when adjusting the allowance.

Impact

Direct theft of any user funds, whether at-rest or in-motion, other than unclaimed yield

If each underlying is worth less than a share (e.g. if there were losses due to Lido slashing, or the external PT's protocol is paused), then a user will be allowed to take out more shares than they have been given allowance for. If the user granting the approval had minted the Illuminate PT by providing a external PT, in order to become an LP in a pool, the loss of shares is a principal loss.

Code Snippet

The amount of *underlying* is being subtracted from the allowance, rather than the number of *shares required to retrieve that amount of underlying*:

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

                       uint256 allowance = _allowance[o][msg.sender];
   @>                  if (allowance < a) {
                           revert Exception(20, allowance, a, address(0),
 ↪ address(0));
                       }
   @>                  _allowance[o][msg.sender] = allowance - a;
                       return
                           IRedeemer(redeemer).authRedeem(
```

```
                                underlying,
                                maturity,
                                o,
                                r,
                                a
                            );
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L262-L274

Redemptions of Illuminate PTs for underlyings is based on shares of each Illuminate PT's `totalSupply()` of the *available* underlying, not the expect underlying total, and there is no way for an admin to pause this withdrawal since the `authRedeem()` function does not use the `unpaused` modifier: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L464

Tool used

Manual Review

Recommendation

Calculate how many shares the the amount of underlying is worth (e.g. call `previewWithdraw()`) and use that amount when adjusting the allowance

Discussion

sourabhmarathe

This report will be addressed. The documentation should be updated to reflect what `withdraw` will do for the user after maturity. Given the nature of the redemption process post-maturity, we'll instead have the user specify how many PTs they will burn post-maturity.

Issue H-11: Sense PTs can never be redeemed

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/117

Found by

Ruhum, IIIIIII, 0x52, neumo

Summary

Sense PTs can never be redeemed

Vulnerability Detail

Most of the protocols that require the user of the `Converter` contract have code that approves the `Converter` for that protocol, but there is no such approval for Sense.

Impact

Permanent freezing of funds

Users will be able to lend and mint using Sense, but when it's time for Illuminate to redeem the Sense PTs, the call will always revert, meaning the associated underlying will be locked in the contract, and users that try to redeem their Illuminate PTs will have lost principal.

While the Illuminate project does have an emergency `withdraw()` function that would allow an admin to rescue the funds and manually distribute them, this would not be trustless and defeats the purpose of having a smart contract.

Code Snippet

The Sense flavor of `redeem()` requires the use of the `Converter`:

```
// File: src/Redeemer.sol : Redeemer.redeem()   #1

            // Get the starting balance to verify the amount received
 afterwards
            uint256 starting = IERC20(u).balanceOf(address(this));

            // Get the divider from the adapter
            ISenseDivider divider = ISenseDivider(ISenseAdapter(a).divider());

            // Redeem the tokens from the Sense contract
            ISenseDivider(divider).redeem(a, s, amount);

            // Get the compounding token that is redeemed by Sense
```

```
            address compounding = ISenseAdapter(a).target();

            // Redeem the compounding token back to the underlying
@>          IConverter(converter).convert(
                compounding,
                u,
                IERC20(compounding).balanceOf(address(this))
            );

            // Get the amount received
            uint256 redeemed = IERC20(u).balanceOf(address(this)) - starting;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L366-L386

But there is no code that approves the `Converter` to be able to withdraw from the `Redeemer`. The only function available is required to have been called by the `MarketPlace`, and is thus not callable by the admin:

```
// File: src/Redeemer.sol : Redeemer.approve()   #2

        function approve(address i) external authorized(marketPlace) {
            if (i != address(0)) {
@>              Safe.approve(IERC20(i), address(converter), type(uint256).max);
            }
        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L203-L207

Redemptions of Illuminate PTs for underlyings is based on shares of each Illuminate PT's `totalSupply()` of the *available* underlying, not the expect underlying total: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L422 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L464 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L517

There is a fork test that tests the converter functionalty, but is uses `vm.startPrank()` to hack the approval, which wouldn't be available in real life.

Also note that if the admin ever deploys and sets a new converter, that all other redemptions using the converter will break

Tool used

Manual Review

Recommendation

Add the sense yield token to the `Redeemer`'s `Converter` approval during market creation/setting of principal

Issue H-12: Fee-on-transfer underlyings can be used to mint Illuminate PTs without fees

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/116

Found by

Bnke0x0, IllIllI, Tomo

Summary

Fee-on-transfer underlyings can be used to mint Illuminate PTs without fees

Vulnerability Detail

Illuminate's `Lender` does not confirm that the amount of underlying received is the amount provided in the transfer call. If the token is a fee-on-transfer token (e.g. USDT which is currently supported), then the amount may be less. As long as the fee is smaller than Illuminate's fee, Illuminate will incorrectly trust that the fee has properly been deducted from the contract's balance, and then will swap the funds and mint an Illuminate PT.

Impact

Theft of unclaimed yield

Attackers can mint free PT at the expense of Illuminate's fees.

Code Snippet

This is one example from one of the `lend()` functions, but they all have the same issue:

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

    function lend(
        uint8 p,
        address u,
        uint256 m,
        uint256 a,
        uint256 r
    ) external unpaused(u, m, p) returns (uint256) {
        // Instantiate Notional princpal token
        address token = IMarketPlace(marketPlace).token(u, m, p);

        // Transfer funds from user to Illuminate
```

```
@>        Safe.transferFrom(IERC20(u), msg.sender, address(this), a);

          // Add the accumulated fees to the total
          uint256 fee = a / feenominator;
          fees[u] = fees[u] + fee;

          // Swap on the Notional Token wrapper
@>        uint256 received = INotional(token).deposit(a - fee, address(this));

          // Verify that we received the principal tokens
          if (received < r) {
              revert Exception(16, received, r, address(0), address(0));
          }

          // Mint Illuminate zero coupons
@>        IERC5095(principalToken(u, m)).authMint(msg.sender, received);

          emit Lend(p, u, m, received, a, msg.sender);
          return received;
      }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L750-L780

And separately, if any of the external PTs ever become fee-on-transfer (e.g. CTokens, which are upgradeable), users would be able to mint Illuminate PT directly without having to worry about the FOT fee being smaller than the illuminate one, and the difference would be made up by other PT holders' principal, rather than Illuminate's fees:

```
// File: src/Lender.sol : Lender.mint()   #2

      function mint(
          uint8 p,
          address u,
          uint256 m,
          uint256 a
      ) external unpaused(u, m, p) returns (bool) {
          // Fetch the desired principal token
          address principal = IMarketPlace(marketPlace).token(u, m, p);

          // Transfer the users principal tokens to the lender contract
@>        Safe.transferFrom(IERC20(principal), msg.sender, address(this), a);

          // Mint the tokens received from the user
@>        IERC5095(principalToken(u, m)).authMint(msg.sender, a);
```

```
            emit Mint(p, u, m, a);

            return true;
        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L270-L288

POC

Imagine that the Illuminate fee is 1%, and the fee-on-transfer fee for USDT is also 1%

1. A random unaware user calls one of the `lend()` functions for 100 USDT
2. `lend()` does the `transferFrom()` for the user and gets 99 USDT due to the USDT 1% fee
3. `lend()` calculates its own fee as 1% of 100, resulting in 99 USDT remaining
4. `lend()` swaps the 99 USDT for a external PT
5. the user is given 99 IPT and only had to spend 100 USDT, and Illuminate got zero actual fee, and actually has to make up the difference itself in order to withdraw *any* fees (see other issue I've filed about this).

Tool used

Manual Review

Recommendation

Check the actual balance before and after the transfer, and ensure the amount is correct, or use the difference as the amount

Discussion

sourabhmarathe

Set label to `high` because based on what the report indicated.

IIIIIII000

@sourabhmarathe can you elaborate on what aspect of the report made this a high? https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/104 describes a separate way of how to mint IPT using protocol fees

sourabhmarathe

I was just updating the issue to reflect what the Watson had put on the report. To me, it appeared mislabeled as the original report had a high level severity at the top of the report.

sourabhmarathe

Re #104: It should not be marked as a duplicate. It's a separate issue in it's own right. That said, it doesn't put user funds at risk, so I think it should remain at a Medium.

Issue H-13: Illuminate's PTs burn more tokens than are necessary

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/115

Found by

IIIIIII

Summary

Illuminate's PTs burn more tokens than are necessary to get a specific number of underlying, leading to users getting fewer underlying than they deserve

Vulnerability Detail

`ERC5095.withdraw()/redeem()`'s code relies on `Redeemer.authRedeem()` to do the redemption, but this function always burns the specific amount of PT passed to it, regardless of whether the of whether the PT is worth more than one underlying, which may be the case if there is positive slippage/rewards when the external PT is redeemed, or if the underlying is a rebasing token.

While there are no currently-rebasing underlying tokens listed in the `Contracts.sol` test file, USDC is listed and is an upgradeable contract, which means it may have such functionality in the future for markets that already have users.

Impact

Theft of unclaimed yield

Users will get less underlying than they are owed, even though the code is attempting to track funds on a per-share basis, rather than a one-for-one basis.

Code Snippet

The NatSpec says `Burns’shares’from’owner’andsendsexactly’assets’ofunderlying tokensto’receiver’`, and `shares` is the output parameter (since no other argument has this name), so it's clear that the intention is to provide an exact number of assets, not more. In spite of this, the function calls `Redeemer.authRedeem()`...:

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

                if (o == msg.sender) {
                    return
@>                      IRedeemer(redeemer).authRedeem(
                            underlying,
```

```
256                             maturity,
257                             msg.sender,
258                             r,
259                             a
260                         );
261                 } else {
262                     uint256 allowance = _allowance[o][msg.sender];
263                     if (allowance < a) {
264                         revert Exception(20, allowance, a, address(0),
 ↪  address(0));
265                     }
266                     _allowance[o][msg.sender] = allowance - a;
267                     return
268 @>                      IRedeemer(redeemer).authRedeem(
269                             underlying,
270                             maturity,
271                             o,
272                             r,
273                             a
274                         );
275                 }
276             }
277:        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L252-L277

...which always burns the full amount of PTs, even if that results in too many underlying being transferred:

```
// File: src/Redeemer.sol : Redeemer.authRedeem()   #2

463             // Calculate the amount redeemed
464 @>          uint256 redeemed = (a * holdings[u][m]) / pt.totalSupply();
465
466             // Update holdings of underlying
467             holdings[u][m] = holdings[u][m] - redeemed;
468
469             // Burn the user's principal tokens
470 @>          pt.authBurn(f, a);
471
472             // Transfer the original underlying token back to the user
473             Safe.transfer(IERC20(u), t, redeemed);
474:
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L463-L474

Tool used

Manual Review

Recommendation

Change the behavior of `authRedeem()` to calculate how many PTs are required to send the right number of underlying, and only burn that many

Discussion

sourabhmarathe

I was unable to reconcile the slippage across the preview methods to make this work in fork-mode testing.

In addition, `authRedeem` always will burn the amount of shares (PTs) passed to it. Also, given that the users could lose at most their slippage, I believe this warrants a `medium` label.

JTraversa

Ah actually I think this might be a bit different than that ticket in 114 (though I do think there were other duplicates).

I think the question really is whether our keepers are lazy / imprecise enough to provide time for there to be additional yield generation which would warrant additional marginal calculations on redemption.

E.g. depending on the integrated protocol, those calculations can be quite gas heavy. For a protocol like pendle <-> compound, you have to do a read of exchangeRateCurrent which mutates state and costs 77k gas, in addition to any other storage reads.

Would that 77-100k gas be worth it for our users? Prooobably not outside of maybe insane future success and a user with 8 figures deposited in a single market, and even then it would be close.

Issue H-14: Illuminate's PT doesn't respect users' slippage specifications

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/114

Found by

IIIIIII

Summary

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

Vulnerability Detail

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

Impact

Direct theft of any user funds, whether at-rest or in-motion, other than unclaimed yield Miner-extractable value (MEV)

At the end of withdrawal/redemption, the user will end up losing more underlying than they wished to, 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

The NatSpec says `Beforematurity,sends'assets'bysellingsharesofPTonaYieldSpaceAMM.`, so it's clear that the intention is to send back the amount of tokens specified in the input argument. In spite of this, extra slippage is allowed for the amount:

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

                    uint128 returned = IMarketPlace(marketplace).sellPrincipalToken(
                        underlying,
                        maturity,
                        shares,
@>                      Cast.u128(a - (a / 100))
                    );
                    Safe.transfer(IERC20(underlying), r, returned);
                    return returned;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L219-L226

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

240                     uint128 returned =
↪   IMarketPlace(marketplace).sellPrincipalToken(
241                         underlying,
242                         maturity,
243                         Cast.u128(shares),
244 @>                      Cast.u128(a - (a / 100))
245                     );
246                     Safe.transfer(IERC20(underlying), r, returned);
247:                    return returned;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L240-L247

(`redeem()` has the same issue

and `IMarketPlace.sellPrincipalToken()` also considers the amount as an amount that already includes slippage:

```
// File: src/MarketPlace.sol : MarketPlace.a   #3

279         /// @notice sells the PT for the underlying via the pool
280         /// @param u address of an underlying asset
281         /// @param m maturity (timestamp) of the market
282         /// @param a amount of PTs to sell
283 @>      /// @param s slippage cap, minimum amount of underlying that must be
↪   received
284         /// @return uint128 amount of underlying bought
285         function sellPrincipalToken(
286             address u,
287             uint256 m,
288             uint128 a,
289 @>          uint128 s
290:        ) external returns (uint128) {
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L285-L298

Tool used

Manual Review

Recommendation

Pass `Cast.u128(a)` to the two calls instead

Discussion

sourabhmarathe

Unfortunately, there isn't a clean solution here for users. When using the preview methods as suggested in the recommendation, an invalid slippage is used for the fourth parameter to `sellPrincipalToken`. As a result, we are finding that this does not work on fork-mode tests. For now, we're going to keep the method in place with the knowledge that users should have other avenues available to them (via using the pool directly) to reduce their slippage risk.

In addition, I would disagree with the severity of this issue on that basis as well, and I am open to hearing other ideas the judges have.

IllIllI000

Even if there are other paths that the user can take, the presence of a path where they lose principal means there's still a high-severity issue. I believe the problem you're facing is that convert* is using preview*, when it's supposed to be a flash-resistant method of getting the value. The ERC5095 spec specifically only requires that convertToUnderlying() only work at maturity, because according to one of the spec authors, it's an 'open question' about how to get a valid price before then.

sourabhmarathe

As an alternative solution, should we have user provide the slippage then? Given that `withdrawPreview()` tells us how many `shares` will be required and the fact that it could change depending on where in the block the swap occurs, I think that's the only way around this (other than providing the 1% slippage built-in value we provide)

IllIllI000

The 5095 standard has specific arguments for `withdraw()`, so I don't think you should add another argument. The standard also mentions `Notethatsomeimplementationswillrequirepre-requestingtotheprincipaltokencontractbeforeawithdrawalmaybeperformed.Thosemethodsshouldbeperformedseparately.` so you could have a function that pre-specifies the slippage before each withdrawal, or you could have a completely separate pre-maturity withdrawal function, and have the normal `withdraw()` revert before maturity, as is done in the sample contract in the EIP

JTraversa

Yeah unfortunately we also intended some backwards compatability with 4626 (specifically for some integrations like the aztec 4626 bridge as this product targets their sort of batched design in particular). W/ that context we cant quite remove that sort of integration compatibility with pre-maturity redemptions, nor can add any params without breaking those integrations.

So its difficult to find a great solution other than writing overrides that *do* include additional parameters for slippage protection. IIRC we had issues with bytecode size limits and didnt want to add them but perhaps through other efforts we reduced some headroom there?

Issue H-15: Illuminate redemptions don't account for protocol pauses/temporary blocklistings

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/113

Found by

ak1, __141345__, ctf_sec, cccz, rvierdiiev, Jeiwan, Holmgren, kenzo, IllIllI, HonorLt

Summary

Illuminate redemptions don't account for protocol pauses/temporary blocklistings

Vulnerability Detail

By the time that Illuminate PTs have reached maturity, it's assumed that all external PTs will have been converted to underlying, so that the pool of combined underlying from the various protocols can be split on a per-Illuminate-PT-share basis. Unfortunately this may not be the case. Some of the protocol PTs that Illuminate supports as principals allow their own admins to pause [activity](# https://docs.notional.finance/developer-documentation/on-chain/notional-governance-reference#pauseability), and Illuminate has no way to protect users from redeeming while these protocol pauses are in effect. Unredeemed external PTs contribute zero underlying to the Illuminate PT's underlying balance, and when a user redeemes an Illuminate PT, the PT is burned for its share of what's available, not the total of what could be available in the future.

Impact

Permanent freezing of funds

If a external PT is paused, or its PT is otherwise unable to be redeemed for the full amount when the user requests it, that unredeemed amount of underlying is not claimable (since the user's Illuminate PT is burned), and the user loses that amount of principal. If the external PT is later able to be redeemed, the remaining users will be given the principal that should have gon to the original user.

Code Snippet

Holdings only increase when external PTs are redeemed successfully:

```
// File: src/Redeemer.sol : Redeemer.redeem()   #1

            // Calculate how much underlying was redeemed
            uint256 redeemed = IERC20(u).balanceOf(address(this)) - starting;
```

```
327
328             // Update the holding for this market
329:            holdings[u][m] = holdings[u][m] + redeemed;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L325-L329

```
// File: src/Redeemer.sol : Redeemer.redeem()   #2

385             // Get the amount received
386             uint256 redeemed = IERC20(u).balanceOf(address(this)) - starting;
387
388             // Verify that underlying are received 1:1 - cannot trust the
↪   adapter
389             if (redeemed < amount) {
390                 revert Exception(13, 0, 0, address(0), address(0));
391             }
392
393             // Update the holdings for this market
394:            holdings[u][m] = holdings[u][m] + redeemed;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L385-L394

And user redemptions of Illuminate PTs does not rely on external PT balances, only on the shares of what's available in the currently stored holdings balance at *any* point after maturity:

```
// File: src/Redeemer.sol : Redeemer.redeem()   #3

413             // Verify the token has matured
414             if (block.timestamp < token.maturity()) {
415                 revert Exception(7, block.timestamp, m, address(0),
↪   address(0));
416             }
417
418             // Get the amount of tokens to be redeemed from the sender
419             uint256 amount = token.balanceOf(msg.sender);
420
421             // Calculate how many tokens the user should receive
422 @>          uint256 redeemed = (amount * holdings[u][m]) /
↪   token.totalSupply();
423
424             // Update holdings of underlying
425             holdings[u][m] = holdings[u][m] - redeemed;
426
427             // Burn the user's principal tokens
```

```
428:            token.authBurn(msg.sender, amount);
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L413-L428

```
// File: src/Redeemer.sol : Redeemer.authRedeem()   #4

457             // Make sure the market has matured
458             uint256 maturity = pt.maturity();
459             if (block.timestamp < maturity) {
460                 revert Exception(7, maturity, 0, address(0), address(0));
461             }
462
463             // Calculate the amount redeemed
464 @>          uint256 redeemed = (a * holdings[u][m]) / pt.totalSupply();
465
466             // Update holdings of underlying
467             holdings[u][m] = holdings[u][m] - redeemed;
468
469             // Burn the user's principal tokens
470:            pt.authBurn(f, a);
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L449-L470

```
// File: src/Redeemer.sol : Redeemer.autoRedeem()   #5

485         function autoRedeem(
486             address u,
487             uint256 m,
488             address[] calldata f
489         ) external returns (uint256) {
490             // Get the principal token for the given market
491             IERC5095 pt = IERC5095(IMarketPlace(marketPlace).token(u, m, 0));
492
493             // Make sure the market has matured
494             uint256 maturity = pt.maturity();
495             if (block.timestamp < maturity) {
496                 revert Exception(7, maturity, 0, address(0), address(0));
497             }
...
514                 uint256 amount = pt.balanceOf(f[i]);
515
516                 // Calculate how many tokens the user should receive
517 @>              uint256 redeemed = (amount * holdings[u][m]) /
 ↪  pt.totalSupply();
518
```

```
519                 // Calculate the fees to be received (currently .025%)
520                 uint256 fee = redeemed / feenominator;
521
522                 // Verify allowance
523                 if (allowance < amount) {
524                     revert Exception(20, allowance, amount, address(0),
 ↪  address(0));
525                 }
526
527                 // Burn the tokens from the user
528:                pt.authBurn(f[i], amount);
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L485-L528

The Illuminate admin has no way to pause/disable redemption for users that try to redeem via `ERC5095.redeem()/withdraw()` or via `autoRedeem()`.

`autoRedeem()` doesn't use the `unpaused` modifier, and does not rely on the normal `redeem()` for redemption:

```
// File: src/Redeemer.sol : Redeemer.u   #6

485        function autoRedeem(
486            address u,
487            uint256 m,
488            address[] calldata f
489 @>     ) external returns (uint256) {
490            // Get the principal token for the given market
491            IERC5095 pt = IERC5095(IMarketPlace(marketPlace).token(u, m, 0));
492
493            // Make sure the market has matured
494            uint256 maturity = pt.maturity();
495            if (block.timestamp < maturity) {
496:               revert Exception(7, maturity, 0, address(0), address(0));
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L485-L496

The ERC5095 also does not use the `unpaused` modifier. It uses `authRedeem()` for its post-maturity redemptions (the pre-maturity redemptions also are not pausable)...:

```
// File: src/tokens/ERC5095.sol : ERC5095.redeem()   #7

284        function redeem(
285            uint256 s,
286            address r,
287            address o
```

```
@>      ) external override returns (uint256) {
...
                // Post-maturity
            } else {
                if (o == msg.sender) {
                    return
@>                      IRedeemer(redeemer).authRedeem(
                            underlying,
                            maturity,
                            msg.sender,
                            r,
                            s
                        );
                } else {
                    uint256 allowance = _allowance[o][msg.sender];
                    if (allowance < s) {
                        revert Exception(20, allowance, s, address(0), address(0));
                    }
                    _allowance[o][msg.sender] = allowance - s;
                    return
@>                      IRedeemer(redeemer).authRedeem(
                            underlying,
                            maturity,
                            o,
                            r,
                            s
                        );
                }
            }
        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L284-L345

...and `authRedeem()` does not use the modifier either:

```
// File: src/Redeemer.sol : Redeemer.authRedeem()   #8

        function authRedeem(
            address u,
            uint256 m,
            address f,
            address t,
            uint256 a
        )
            external
@>          authorized(IMarketPlace(marketPlace).token(u, m, 0))
```

```
        returns (uint256)
    {
        // Get the principal token for the given market
        IERC5095 pt = IERC5095(IMarketPlace(marketPlace).token(u, m, 0));

        // Make sure the market has matured
        uint256 maturity = pt.maturity();
        if (block.timestamp < maturity) {
            revert Exception(7, maturity, 0, address(0), address(0));
        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L443-L461

Tool used

Manual Review

Recommendation

This is hard to solve without missing corner cases, because each external PT may have its own idosyncratic reasons for delays, and there may be losses/slippage involved when redeeming for underlying. I believe the only way that wouldn't allow griefing, would be to track the number of external PTs of each type that were deposited for minting Illuminate PTs on a per-market basis, and `require()` that the number of each that have been redeemed equals the minting count, before allowing the redemption of any Illuminate PTs for that market. You would also need an administrator override that bypasses this check for specific external PTs of specific maturities. All of this assumes that none of the external PTs have rebasing functionality. Also, add the `unpaused` modifier to both `Redeemer.autoRedeem()` and `Redeemer.authRedeem()`.

Issue H-16: Sense PT redemptions do not allow for known loss scenarios

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/111

Found by

IIIIIII

Summary

Sense PT redemptions do not allow for known loss scenarios, which will lead to principal losses

Vulnerability Detail

The Sense PT redemption code in the `Redeemer` expects any losses during redemption to be due to a malicious adapter, and requires that there be no losses. However, there are legitimate reasons for there to be losses which aren't accounted for, which will cause the PTs to be unredeemable. The Lido FAQ page lists two such reasons:

```
- Slashing risk

ETH 2.0 validators risk staking penalties, with up to 100% of staked funds at
↪  risk if validators fail. To minimise this risk, Lido stakes across multiple
↪  professional and reputable node operators with heterogeneous setups, with
↪  additional mitigation in the form of insurance that is paid from Lido fees.

- stETH price risk

Users risk an exchange price of stETH which is lower than inherent value due to
↪  withdrawal restrictions on Lido, making arbitrage and risk-free market-making
↪  impossible.

The Lido DAO is driven to mitigate above risks and eliminate them entirely to the
↪  extent possible. Despite this, they may still exist and, as such, it is our
↪  duty to communicate them.
```

https://help.lido.fi/en/articles/5230603-what-are-the-risks-of-staking-with-lido

If Lido is slashed, or there are withdrawal restrictions, the Sense series sponsor will be forced to settle the series, regardless of the exchange rate (or miss out on their rewards). The Sense `Divider` contract anticipates and properly handles these losses, but the Illuminate code does not.

Lido is just one example of a Sense token that exists in the Illuminate code base -

there may be others added in the future which also require there to be allowances for losses.

Impact

Permanent freezing of funds

There may be a malicious series sponsor that purposely triggers a loss, either by DOSing Lido validators, or by withdrawing enough to trigger withdrawal restrictions. In such a case, the exchange rate stored by Sense during the settlement will lead to losses, and users that hold Illumimate PTs (not just the users that minted Illuminate PTs with Sense PTs), will lose their principal, because Illuminate PT redemptions are an a share-of-underlying basis, not on the basis of the originally-provided token.

While the Illuminate project does have an emergency `withdraw()` function that would allow an admin to rescue the funds and manually distribute them, this would not be trustless and defeats the purpose of having a smart contract.

Code Snippet

The Sense adapter specifically used in the Illuminate tests is the one that corresponds to wstETH:

```
// File: test/fork/Contracts.sol    #1
36    // (sense adapter)
37    // NOTE for sense, we have to use the adapter contract to verify the
↪   underlying/maturity
38    // NOTE also we had to use the wsteth pools.... (maturity: 1659312000)
39:     address constant SENSE_ADAPTER =
↪   0x880E5caBB22D24F3E278C4C760e763f239AccA95;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/test/fork/Contracts.sol#L36-L39

The code for the redemption of the Sense PTs assumes that one PT equals at least one underlying, which may not be the case:

```
// File: src/Redeemer.sol : Redeemer.redeem()    #2
360            // Get the balance of tokens to be redeemed by the user
361            uint256 amount = token.balanceOf(cachedLender);
...
379            IConverter(converter).convert(
380                compounding,
381                u,
382                IERC20(compounding).balanceOf(address(this))
383            );
384
385            // Get the amount received
```

```
386             uint256 redeemed = IERC20(u).balanceOf(address(this)) - starting;
387
388 @>          // Verify that underlying are received 1:1 - cannot trust the adapter
389 @>          if (redeemed < amount) {
390 @>              revert Exception(13, 0, 0, address(0), address(0));
391             }
392
393             // Update the holdings for this market
394             holdings[u][m] = holdings[u][m] + redeemed;
395
396             emit Redeem(p, u, m, redeemed, msg.sender);
397             return true;
398:        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L360-L398

Redemptions of Illuminate PTs for underlyings is based on shares of each Illuminate PT's `totalSupply()` of the *available* underlying, not the expect underlying total: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L422 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L464 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L517

Tool used

Manual Review

Recommendation

Allow losses during redemption if Sense's Periphery.verified() returns `true`

Discussion

sourabhmarathe

I agree with the stated problem from this report, the only thing I would change about the Recommendation is that we can check is if the Lender contract has approved the periphery.

SHERLOCK

Issue H-17: Notional PT redemptions do not use flash-resistant prices

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/110

Found by

IllIllI

Summary

Notional PT redemptions do not use the correct function for determining balances, which will lead to principal losses

Vulnerability Detail

EIP-4626 states the following about `maxRedeem()`:

```
MUST return the maximum amount of shares that could be transferred from `owner`
↪  through `redeem` and not cause a revert, which MUST NOT be higher than the
↪  actual maximum that would be accepted (it should underestimate if necessary).

MUST factor in both global and user-specific limits, like if redemption is
↪  entirely disabled (even temporarily) it MUST return 0.
```

https://github.com/ethereum/EIPs/blob/12fb4072a8204ae89c384a5562dedfdac32a3bec/EIPS/eip-4626.md?plain=1#L414-L416

The above means that the implementer is free to return less than the actual balance, and is in fact *required* to return zero if the token's backing store is paused, and Notional's can be paused. While neither of these conditions currently apply to the existing wfCashERC4626 implementation, there is nothing stopping Notional from implementing the MUST-return-zero-if-paused fix tomorrow, or from changing their implementation to one that requires `maxRedeem()` to return something other than the current balance.

Impact

Permanent freezing of funds

If `maxRedeem()` were to return zero, or some other non-exact value, fewer Notional PTs would be redeemed than are available, and users that `redeem()`ed their shares, would receive fewer underlying (principal if they minted Illuminate PTs with Notional PTs, e.g. to be an LP in the pool) than they are owed. The Notional PTs that weren't redeemed would still be available for a subsequent call, but if a user already redeemed their Illuminate PTs, their loss will already be locked in, since their Illuminate PTs will have been burned. This would affect *ALL* Illuminate PT holders of a specific market, not just the ones that provided the Notional PTs, because Illuminate PT redemptions are an a share-of-underlying basis, not on the basis of the originally-provided token. Markets that are already live with Notional set cannot be protected via a redemption pause by the Illuminate admin, because redemption of `Lender`'s external PTs for underlying does not use the `unpaused` modifier, and does have any access control.

Code Snippet

```
// File: src/Redeemer.sol : Redeemer.redeem()   #1

309                 // Retrieve the pool for the principal token
310                 address pool = ITempusToken(principal).pool();
311
312                 // Redeems principal tokens from Tempus
313                 ITempus(tempusAddr).redeemToBacking(pool, amount, 0,
 ↪ address(this));
314             } else if (p == uint8(MarketPlace.Principals.Apwine)) {
315                 apwineWithdraw(principal, u, amount);
316             } else if (p == uint8(MarketPlace.Principals.Notional)) {
317                 // Redeems principal tokens from Notional
318                 INotional(principal).redeem(
319 @>                  INotional(principal).maxRedeem(address(this)),
320                     address(this),
321                     address(this)
322                 );
323             }
324
325             // Calculate how much underlying was redeemed
326             uint256 redeemed = IERC20(u).balanceOf(address(this)) - starting;
327
328             // Update the holding for this market
329:            holdings[u][m] = holdings[u][m] + redeemed;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L309-L329

Redemptions of Illuminate PTs for underlyings is based on shares of each Illuminate PT's `totalSupply()` of the *available* underlying, not the expect underlying total: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L422 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L464 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L517

Tool used

Manual Review

Recommendation

Use `balanceOf()` rather than `maxRedeem()` in the call to `INotional.redeem()`, and make sure that Illuminate PTs can't be burned if `Lender` still has Notional PTs that it needs to redeem (based on its own accounting of what is remaining, not based on balance checks, so that it can't be griefed with dust).

SHERLOCK

Issue H-18: APWine PT redemptions can be blocked forever

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/109

Found by

IIIIIII

Summary

APWine PT redemptions can be blocked, causing Illuminate IPTs to be unredeemable

Vulnerability Detail

APWine requires both a PT and a FYT to be provided in order to withdraw funds, and the `IRedeemer` may not be able to acquire the right number of both. The code assumes that FYTs will be available because the PT will have been rolled into the next period, generating new FYTs. However, the code does not account for malicious users sending extra PTs after the roll, to the `Lender`, which would mean there is no corresponding FYT available, and the redemption of all APWine PTs for that maturity/underlying combination will fail.

Impact

Permanent freezing of funds

Users that provided their APWine PTs to mint Illuminate PTs (e.g. in order to be an LP in the pool) will have those tokens (their principal) locked forever. Because those users get Illuminate PTs, when it comes time to redeem a specific market, *ALL* Illuminate PT holders of that market will receieve less than they lent, regardless of whether the original token was an underlying, or a non-APWine PT. The attacker can spend a single wei in order to perform the attack, and they can do so cheaply for every market that has APWine set, by buying one wei of PTs on the open market for each market before the roll, and sending the tokens after the roll.

One method to unblock things would be to buy the right FYTs on the open market, and send the right number back to the contract. However, a well-funded attacker could prevent this by buying up all available supply and have standing market orders for any new supply. One of the reasons for the Illuminate project is to concentrate liquidity since liquidity for these instruments is sparse, so cornering the market is well within the realm of possibility, and after the roll most other users not stuck in the contract will have redeemed their futures, so there will be little to no supply left, and the tokens will be stuck forever.

While the Illuminate project does have an emergency `withdraw()` function that would allow an admin to rescue the funds and manually distribute them, this would not be trustless and defeats the purpose of having a smart contract.

Code Snippet

The `Redeemer` fetches the total `Lender` balance of PTs, and asks to redeem the whole amount (rolled amount + attacker ammount):

```
// File: src/Redeemer.sol : Redeemer.redeem()   #1

263            // Get the amount to be redeemed
264 @>         uint256 amount = IERC20(principal).balanceOf(cachedLender);
265
266            // Receive the principal token from the lender contract
267            Safe.transferFrom(
268                IERC20(principal),
269                cachedLender,
270                address(this),
271                amount
272            );
...
314            } else if (p == uint8(MarketPlace.Principals.Apwine)) {
315:@>             apwineWithdraw(principal, u, amount);
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L263-L315

Inside `apwineWithdraw()`, `amount` from above becomes `a`, and that number is used in the call to `transferFYTs()`, which will fail due to the 'attacker amount' portion:

```
// File: src/Redeemer.sol : Redeemer.apwineWithdraw()   #2

551        function apwineWithdraw(
552            address p,
553            address u,
554 @>         uint256 a
555        ) internal {
556            // Retrieve the vault which executes the redemption in APWine
557            address futureVault = IAPWineToken(p).futureVault();
558
559            // Retrieve the controller that will execute the withdrawal
560            address controller = IAPWineFutureVault(futureVault)
561                .getControllerAddress();
562
563            // Retrieve the next period index
564            uint256 index =
 ↪  IAPWineFutureVault(futureVault).getCurrentPeriodIndex();
```

```
565
566             // Get the FYT address for the current period
567             address fyt =
  ↪ IAPWineFutureVault(futureVault).getFYTofPeriod(index);
568
569             // Trigger claim to FYTs by executing transfer
570             // Safe.transferFrom(IERC20(fyt), address(lender), address(this),
  ↪ a);
571 @>          ILender(lender).transferFYTs(fyt, a);
572
573             // Redeem the underlying token from APWine to Illuminate
574:            IAPWineController(controller).withdraw(futureVault, a);
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L551-L574

Redemptions of Illuminate PTs for underlyings is based on shares of each Illuminate PT's `totalSupply()` of the *available* underlying, not the expect underlying total: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L422 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L464 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L517

Tool used

Manual Review

Recommendation

Do an explicit check for the number of FYTs received during the roll, and only transfer that amount. After the transfer, only `withdraw()` the minimum of `a` and the current FYT balance (to use up any FYTs manually transferred to the `Redeemer` if the attacker tries to undo what they did)

Discussion

sourabhmarathe

After APWine rolls over to a new period, the Lender contract will be receive an equivalent amount of FYTs for each APWine PT it holds. As a result, we do not expect this to be an issue for APWine's redemption process.

IIIIIIIOOO

@sourabhmarathe the *Vulnerability Detail* section mentions that the issue occurs when the attacker transfers new PTs *after* the roll, so the Lender won't get FYTs for those new PTs

JTraversa

Yup! Moved over to confirmed, and the suggested solution is pretty elegant!

Issue M-1: Incorrect parameters

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/233

Found by

IIIIIII, HonorLt

Summary

Some functions and integrations receive the wrong parameters.

Vulnerability Detail

Here, this does not work:

```
} else if (p == uint8(Principals.Notional)) {
      // Principal token must be approved for Notional's lend
      ILender(lender).approve(address(0), address(0), address(0), a);
```

because it basically translates to:

```
} else if (p == uint8(Principals.Notional)) {
 if (a != address(0)) {
      Safe.approve(IERC20(address(0)), a, type(uint256).max);
}
```

It tries to approve a non-existing token. It should approve the underlying token and Notional's token contract.

Another issue is with Tempus here:

```
// Swap on the Tempus Router using the provided market and params
ITempus(controller).depositAndFix(x, lent, true, r, d);

// Calculate the amount of Tempus principal tokens received after the deposit
uint256 received = IERC20(principal).balanceOf(address(this)) - start;

// Verify that a minimum number of principal tokens were received
if (received < r) {
    revert Exception(11, received, r, address(0), address(0));
}
```

It passes `r` as a slippage parameter and later checks that `received>=r`. However, in Tempus this parameter is not exactly the minimum amount to receive, it is the ratio which is calculated as follows:

```
  /// @param minTYSRate Minimum exchange rate of TYS (denominated in TPS) to
↪  receive in exchange for TPS
    function depositAndFix(
        ITempusAMM tempusAMM,
        uint256 tokenAmount,
        bool isBackingToken,
        uint256 minTYSRate,
        uint256 deadline
    ) external payable nonReentrant {
...
  uint256 minReturn = swapAmount.mulfV(minTYSRate, targetPool.backingTokenONE());
```

Impact

Inaccurate parameter values may lead to protocol misfunction down the road, e.g. insufficient approval or unpredicted slippage.

Code Snippet

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L236-L239

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L189-L199

Tool used

Manual Review

Recommendation

Review all the integrations and function invocations, and make sure the appropriate parameters are passed.

Issue M-2: Converter cannot be changed in Redeemer

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/223

Found by

hyh, Ruhum

Summary

Redeemer's setConverter() can be used to switch the converter contract, for example when new type of interest bearing token is introduced as Converter employs hard coded logic to deal with various types of IBTs. However new converter cannot be functional as there is no way to introduce the approvals needed, it can be done only once.

Vulnerability Detail

Upgrading the converter contract is not fully implemented as switching the address without providing approvals isn't sufficient, while it is the only action that can be done now.

Impact

If there are some issues with converter or IBTs it covers there will not be possible to upgrade the contract.

Also, as currently the converter uses hard coded logic to cover Compound, Aave and Lido only, any new IBT cannot be introduced to the system as it requires new Converter to be rolled out for that.

Given that substantial part of Redeemer's logic is dependent on Converter's exchange from IBT to underlying that means the net impact can be up to massive fund freeze.

Code Snippet

setConverter() allows for changing the converter contract:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L145-L152

```
/// @notice sets the converter address
/// @param c address of the new converter
/// @return bool true if successful
function setConverter(address c) external authorized(admin) returns (bool) {
    converter = c;
```


SHERLOCK

```
    emit SetConverter(c);
    return true;
}
```

approve() can provide the approvals needed, but it's `marketPlace` only:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L201-L207

```
/// @notice approves the converter to spend the compounding asset
/// @param i an interest bearing token that must be approved for conversion
function approve(address i) external authorized(marketPlace) {
    if (i != address(0)) {
        Safe.approve(IERC20(i), address(converter), type(uint256).max);
    }
}
```

And there `approve` is run solely on the new market introduction, via createMarket():

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L120-L201

```
/// @notice creates a new market for the given underlying token and maturity
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param t principal token addresses for this market
/// @param n name for the Illuminate token
/// @param s symbol for the Illuminate token
/// @param e address of the Element vault that corresponds to this market
/// @param a address of the APWine router that corresponds to this market
/// @return bool true if successful
function createMarket(
    address u,
    uint256 m,
    address[8] calldata t,
    string calldata n,
    string calldata s,
    address e,
    address a
) external authorized(admin) returns (bool) {
    {
        // Get the Illuminate principal token for this market (if one exists)
        address illuminate = markets[u][m][
            (uint256(Principals.Illuminate))
        ];

        // If illuminate PT already exists, a new market cannot be created
        if (illuminate != address(0)) {
```

```
            revert Exception(9, 0, 0, illuminate, address(0));
        }
    }

    // Create an Illuminate principal token for the new market
    address illuminateToken = address(
        new ERC5095(
            ...
        )
    );

    {
        ...

        // Set the market
        markets[u][m] = market;

        // Have the lender contract approve the several contracts
        ILender(lender).approve(u, e, a, t[7]);

        // Have the redeemer contract approve the Pendle principal token
        if (t[3] != address(0)) {
            address underlyingYieldToken = IPendleToken(t[3])
                .underlyingYieldToken();
            IRedeemer(redeemer).approve(underlyingYieldToken);
        }

        if (t[6] != address(0)) {
            address futureVault = IAPWineToken(t[6]).futureVault();
            address interestBearingToken = IAPWineFutureVault(futureVault)
                .getIBTAddress();
            IRedeemer(redeemer).approve(interestBearingToken);
        }

        emit CreateMarket(u, m, market, e, a);
    }
    return true;
}
```

And via setPrincipal():

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L203-L243

```
/// @notice allows the admin to set an individual market
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
```

```
/// @param a address of the new principal token
/// @return bool true if the principal set, false otherwise
function setPrincipal(
    uint8 p,
    address u,
    uint256 m,
    address a
) external authorized(admin) returns (bool) {
    // Get the current principal token for the principal token being set
    address market = markets[u][m][p];

    // Verify that it has not already been set
    if (market != address(0)) {
        revert Exception(9, 0, 0, market, address(0));
    }

    // Set the principal token in the markets mapping
    markets[u][m][p] = a;

    if (p == uint8(Principals.Pendle)) {
        // Principal token must be approved for Pendle's redeem
        address underlyingYieldToken = IPendleToken(a)
            .underlyingYieldToken();
        IRedeemer(redeemer).approve(underlyingYieldToken);
    } else if (p == uint8(Principals.Apwine)) {
        address futureVault = IAPWineToken(a).futureVault();
        address interestBearingToken = IAPWineFutureVault(futureVault)
            .getIBTAddress();
        IRedeemer(redeemer).approve(interestBearingToken);
    } else if (p == uint8(Principals.Notional)) {
        // Principal token must be approved for Notional's lend
        ILender(lender).approve(address(0), address(0), address(0), a);
    }

    emit SetPrincipal(u, m, a, p);
    return true;
}
```

In both cases it's required that either Illuminate or `market` is `address(0)`, i.e. both functions cannot be run repeatedly.

I.e. it's impossible to run approve if the market exists, so there is no way to approve and use new Converter as without approval it will not be functional, the corresponding Redeemer functions will be reverting as it's expected that converter can pull funds out of Redeemer.

Currently Converter functionality is fixed to deal with 3 types of IBTs:

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Converter.sol#L21-L51

```
function convert(
    address c,
    address u,
    uint256 a
) external {
    // first receive the tokens from msg.sender
    Safe.transferFrom(IERC20(c), msg.sender, address(this), a);

    // get Aave pool
    try IAaveAToken(c).POOL() returns (address pool) {
        // Allow the pool to spend the funds
        Safe.approve(IERC20(u), pool, a);
        // withdraw from Aave
        IAaveLendingPool(pool).withdraw(u, a, msg.sender);
    } catch {
        // attempt to redeem compound tokens to the underlying asset
        try ICompoundToken(c).redeem(a) {
            // get the balance of underlying assets redeemed
            uint256 balance = IERC20(u).balanceOf(address(this));
            // transfer the underlying back to the user
            Safe.transfer(IERC20(u), msg.sender, balance);
        } catch {
            // get the current balance of wstETH
            uint256 balance = IERC20(c).balanceOf(address(this));
            // unwrap wrapped staked eth
            uint256 unwrapped = ILido(c).unwrap(balance);
            // Send the unwrapped staked ETH to the caller
            Safe.transfer(IERC20(u), msg.sender, unwrapped);
        }
    }
}
```

Tool used

Manual Review

Recommendation

Consider running the approvals setting on the introduction of the new Converter, i.e. run Marketplace's createMarket() approval logic as a part of Redeemer's setConverter(), also clearing the approvals for the old one.

Discussion

JTraversa

Though the issue is likely valid, along with Sherlock's scoring guide, this likely does not end up being accepted as a valid issue as no funds are at risk: https://docs.sherlock.xyz/audits/watsons/judging

While the intention of the method is to allow an admin to set a new converter, this is simply a convenience/upgradability method meaning no funds are at risk for any deployments.

IllIllI000

@JTraversa Wouldn't funds be at risk since the Redeemer would be unable to convert PTs to underlying, for users to redeem their IPTs? Looking at some of the other issue comments though, I believe this would will fall under the admin input validation category, and would thus be classified as Low

Issue M-3: Holders of worthless external PTs can stick other Illuminate PT holders with bad debts

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/119

Found by

IllIllI

Summary

Holders of worthless external PTs can stick other Illuminate PT holders with bad debts

Vulnerability Detail

Some of the supported external PTs can pause their activity. One such PT, Pendle, not only can pause activity, but can turn on emergency mode where the admin can transfer the underlying tokens to an arbitrary contract for safekeeping until they decide what to do with the funds. The Illuminate code does not handle such cases, and in fact, if the Pendle protocol is in emergency mode, will still allow users to convert their possibly worthless Pendle PTs to Illuminate ones.

While there is a mechanism for the Illuminate admin to pause a market, there's no guarantee that the Illuminate admin will notice the Pendle pause before other users, and even if they do, it's possible that users have automation set up to front-run such pauses for Pendle markets, so that they never are stuck with worthless tokens.

Impact

Direct theft of any user funds, whether at-rest or in-motion, other than unclaimed yield

Other users that deposited principal in the form of external PTs (e.g. by minting Illuminate PTs in order to be pool liquidity providers) that have actual value, will have their shares of available underlying diluted by Pendle PTs that cannot be redeemed. Illuminate PTs are on a per-share basis rather than a one-for-one basis, so the less underlying there is at redemption time, the less underlying every Illuminate PT holder gets.

Code Snippet

There are no checks that the protocol of the external PT is paused or has any value:

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

```solidity
    function mint(
        uint8 p,
        address u,
        uint256 m,
        uint256 a
    ) external unpaused(u, m, p) returns (bool) {
        // Fetch the desired principal token
        address principal = IMarketPlace(marketPlace).token(u, m, p);

        // Transfer the users principal tokens to the lender contract
        Safe.transferFrom(IERC20(principal), msg.sender, address(this),
 ↪  a);

        // Mint the tokens received from the user
        IERC5095(principalToken(u, m)).authMint(msg.sender, a);

        emit Mint(p, u, m, a);

        return true;
    }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L270-L288

Redemptions of Illuminate PTs for underlyings is based on shares of each Illuminate PT's `totalSupply()` of the *available* underlying, not the expect underlying total: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L422 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L464 https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L517

Tool used

Manual Review

Recommendation

Ensure that the protocol being used as principal is not paused before allowing minting

Discussion

sourabhmarathe

In the event of insolvency, we expect the `admin` to pause any principal token using the `unpaused` modifier to block minting.

IIIIIII000

@sourabhmarathe there is no guarantee that the admin will be aware of the insolvency and do the manual step of pausing, before automated tools notice and take advantage of the issue

JTraversa

This is generally the case with most integrations across most protocols, there is the chance of an atomic attack on multiple protocols preventing the pausing of markets after detection.

So there arent immediately extremely easy solutions, that said specifically we have already implemented the recommended auditor remediation,

> Ensure that the protocol being used as principal is not paused before allowing minting

We do ensure that the protocol being used as a principal does not flag the `unpaused` modifier before any minting.

IllIllI000

The recommendation is to check whether the protocol itself is paused, not to check whether Illuminate has its own paused flag set

Evert0x

Valid issue but downgrading to medium severity as the conditions are dependent on an external protocol their admin functions.

JTraversa

Understood although this presupposes the idea that all of them can even be paused.

Again, im unsure if this is a reasonable request, as you could submit the same exact report for every single sherlock audit and it would be equally valid for every single integration ever?

Further, if there is an attack, the attacker would simply just attack illuminate before the external protocol can be paused, completely bypassing any checks and just leaving normal users paying more gas.

It all just seems kind of unreasonable, especially as you add additional integrations to the stack (e.g. Illuminate -> Swivel -> Euler -> Lido, do we somehow check EACH of these before every transaction?)

Issue M-4: No markets can be created since Illuminate PTs are not ERC-4626 tokens

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/106

Found by

IllIllI

Summary

No markets can be created since Illuminate PTs are not ERC-4626 tokens, and will cause pool creation to fail

Vulnerability Detail

I checked with the sponsor and they confirmed that the plan was to use yieldspace-tv pools to swap Illuminate PTs for underlying, and that they planned to deploy the existing pool contract, rather than writing a new special module. The existing `Pool` contract relies on the ERC-4626 interface to accomplish some of its tasks (and tokens that do not comply with it need to create new modules in order to override those functions). One such task is the fetching of the price, which relies on `IERC4626.convertToAssets()` which does not exist in the EIP-5095 spec that the Illuminate PT follows. The fetching of the price is done in the pool constructor, and Illuminate PTs require the pool to already have been immutably set in the market before they're constructed, so therefore there is no way to create a market for any asset.

In addition to not being able to construct the pools, there are other functions such as `asset()`, and `deposit()` (note the flipped args), which do not exist in `ERC5095` but are relied on by the `Pool`, so even if the constructor issue is addressed, things will fail later.

Impact

Smart contract unable to operate due to lack of token funds

`MarketPlace.createMarket()` can't be called with a valid pool, so nobody can use any feature of the Illuminate project.

Code Snippet

Market creation unconditionally constructs Illuminate PTs:

```
// File: src/MarketPlace.sol : MarketPlace.createMarket()   #1

150            // Create an Illuminate principal token for the new market
```

```
            address illuminateToken = address(
@>              new ERC5095(
                    u,
                    m,
                    redeemer,
                    lender,
                    address(this),
                    n,
                    s,
                    IERC20(u).decimals()
                )
            );
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/MarketPlace.sol#L150-L162

Illuminate PTs are EIP-5095 contracts, not EIP-4626 ones, and do not implement the `convertToAssets()` function:

```
// File: src/tokens/ERC5095.sol    #2

@>  contract ERC5095 is ERC20Permit, IERC5095 {
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L13

The `immutable` pool is set in the constructor, and comes from the `MarketPlace`:

```
// File: src/tokens/ERC5095.sol : ERC5095.constructor()    #3

    constructor(
        address _underlying,
        uint256 _maturity,
        address _redeemer,
        address _lender,
        address _marketplace,
        string memory name_,
        string memory symbol_,
        uint8 decimals_
    ) ERC20Permit(name_, symbol_, decimals_) {
        underlying = _underlying;
        maturity = _maturity;
        redeemer = _redeemer;
        lender = _lender;
        marketplace = _marketplace;
@>      pool = IMarketPlace(marketplace).pools(underlying, maturity);
    }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/tokens/ERC5095.sol#L37-L53

Pools must be set ahead of time, and cannot change once set:

```
// File: src/MarketPlace.sol : MarketPlace.setPool()   #4

259        function setPool(
260            address u,
261            uint256 m,
262            address a
263        ) external authorized(admin) returns (bool) {
264            // Verify that the pool has not already been set
265            address pool = pools[u][m];
266
267            // Revert if the pool already exists
268 @>         if (pool != address(0)) {
269 @>             revert Exception(10, 0, 0, pool, address(0));
270 @>         }
271
272            // Set the pool
273            pools[u][m] = a;
274
275            emit SetPool(u, m, a);
276            return true;
277:       }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/MarketPlace.sol#L259-L277

Yieldspace-tv `Pool`s rely on the function that does not exist in the Illuminate PT:

```
    function _getCurrentSharePrice() internal view virtual returns (uint256) {
        uint256 scalar = 10**baseDecimals;
@>      return IERC4626(address(sharesToken)).convertToAssets(scalar);
    }


    /// Returns current price of 1 share in 64bit.
    /// Useful for external contracts that need to perform calculations related
↪   to pool.
    /// @return The current price (as determined by the token) scalled to 18
↪   digits and converted to 64.64.
    function getC() external view returns (int128) {
@>      return _getC();
    }
```

```
    /// Returns the c based on the current price
    function _getC() internal view returns (int128) {
@>      return (_getCurrentSharePrice() * scaleFactor).divu(1e18);
    }
```

https://github.com/yieldprotocol/yieldspace-tv/blob/8685abc2f57c2f3130165404a77620a3220fb182/src/Pool/Pool.sol#L1400-L1415

`getC()` is called by the constructor, so pools cannot be constructed with Illuminate PTs:

```
@>        if ((mu = _getC()) == 0) {
```

https://github.com/yieldprotocol/yieldspace-tv/blob/8685abc2f57c2f3130165404a77620a3220fb182/src/Pool/Pool.sol#L193

The existing fork tests mostly use the Yield USDC pool rather than creating an actual new pool.

Tool used

Manual Review

Recommendation

Implement a new yieldspace-tv module for EIP-5095 contracts

Issue M-5: The Pendle version of `lend()` uses the wrong function for swapping fee-on-transfer tokens

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/105

Found by

IllIllI

Summary

The Pendle version of `lend()` uses the wrong function for swapping fee-on-transfer tokens

Vulnerability Detail

The Pendle version of `lend()` is not able to handle fee-on-transfer tokens properly (USDT is a fee-on-transfer token which is supported) and pulls out the contract's fee balance (I've filed this issue separately). Once that is fixed there still is the fact that the Pendle version uses the wrong Sushiswap function (the Pendle router is a Sushiswap router). The function uses `swapExactTokensForTokens()` when it should use swapExactTokensForTokensSupportingFeeOnTransferTokens() instead.

Impact

Smart contract unable to operate due to lack of token funds

Users will be unable to use the Pendle version of `lend()` when the underlying is a fee-on-transfer token with the fee turned on (USDT currently has the fee turned off, but they can turn it on at any moment).

Code Snippet

The pulling in of the amount by `IPendle` will either take part of the Illuminate protocol fees, or will revert if there is not enough underlying after the fee is applied for the Sushiswap transfer (depending on which fee-on-transfer fix is applied for the other issue I filed):

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

                address[] memory path = new address[](2);
                path[0] = u;
                path[1] = principal;

                // Swap on the Pendle Router using the provided market and
 ↪  params
```

```
@>                returned = IPendle(pendleAddr).swapExactTokensForTokens(
@>                    a - fee,
@>                    r,
@>                    path,
@>                    address(this),
@>                    d
@>                )[1];
            }

            // Mint Illuminate zero coupons
            IERC5095(principalToken(u, m)).authMint(msg.sender, returned);

            emit Lend(p, u, m, returned, a, msg.sender);
            return returned;
        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L536-L560

Tool used

Manual Review

Recommendation

Use `swapExactTokensForTokensSupportingFeeOnTransferTokens()`

Issue M-6: ERC777 transfer hooks can be used to bypass fees for markets that support Swivel

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/104

Found by

IIIIIII

Summary

ERC777 transfer hooks can be used to bypass fees for markets that support Swivel

Vulnerability Detail

Most of the `lend()` functions calculate fees based on an amount that is directly transferred by the `Lender` contract. In the case of the Swivel version of `lend()`, it assumes that the Swivel orders provided are operating on the underlying, and only calculates fees based on those. After that, it allows the user to swap any excess underlying with `swivelLendPremium()`, and there are no checks that the 'premium' amount is a dust amount, and there are no fees charged on this amount.

If a user submits a Swivel order that *adds* one wei of Notional tokens (one of Swivel's supported tokens) to a Swivel position, which are ERC777 tokens, the user can use the pre-transfer hook to send a large amount of underlying to the `Lender` contract, so that when `swivelLendPremium()` is called, the large balance is swapped without fees. The one wei of Notional contributes zero to the fee, since the `feenominator` calculation is vulnerable to loss of precision.

A malicious user can automate this process by deploying a contract that does this automatically for novice users.

Impact

No protocol fees

Users can pay zero fees

Code Snippet

Fees are based on the order amounts:

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

                   // Lent represents the total amount of underlying to be lent
    @>             uint256 lent = swivelAmount(a);
```

```
385
386                 // Transfer underlying token from user to Illuminate
387                 Safe.transferFrom(IERC20(u), msg.sender, address(this), lent);
388
389                 // Get the underlying balance prior to calling initiate
390                 uint256 starting = IERC20(u).balanceOf(address(this));
391
392                 // Verify and collect the fee
393                 {
394                     // Calculate fee for the total amount to be lent
395:@>                  uint256 fee = lent / feenominator;
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L383-L395

No fees are charged on the premium:

```
// File: src/Lender.sol : Lender.lend()   #2

                uint256 received;
                {
                    // Get the starting amount of principal tokens
                    uint256 startingZcTokens = IERC20(
                        IMarketPlace(marketPlace).token(u, m, p)
                    ).balanceOf(address(this));

                    // Fill the given orders on Swivel
                    ISwivel(swivelAddr).initiate(o, a, s);

@>                  if (e) {
@>                      // Calculate the premium
@>                      uint256 premium = IERC20(u).balanceOf(address(this)) -
@>                          starting;
@>
@>                      // Swap the premium for Illuminate principal tokens
@>                      swivelLendPremium(u, m, y, premium, premiumSlippage);
@>                  }

                    // Compute how many principal tokens were received
                    received =
                        IERC20(IMarketPlace(marketPlace).token(u, m, p)).balanceOf(
                            address(this)
                        ) -
                        startingZcTokens;
                }

```


SHERLOCK

```
434:                // Mint Illuminate principal tokens to the user
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L407-L434

Notional tokens are proxies of ERC777 tokens

Tool used

Manual Review

Recommendation

Charge a fee based on the total underlying after the Swivel orders are executed

Issue M-7: There can only ever be one market with USDT as the underlying

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/99

Found by

IIIIIII, Ruhum

Summary

There can only ever be one market with USDT as the underlying

Vulnerability Detail

USDT, and other tokens that have approval race protections will revert when `approve()` is called if the current approval isn't currently zero. The `MarketPlace` contract always approves the underlying during market creation, and on the second market created, the creation will revert.

Impact

Smart contract unable to operate due to lack of token funds

No USDT markets except for the first one will be able to be created. An admin can work around this by passing `0x0` as every entry in the principal array, and later calling `setPrincipal()`, but this is error-prone, especially since once set, principals are immutable.

Code Snippet

`Marketplace.createMarket()` unconditionally calls `Lender.approve()`:

```
// File: src/MarketPlace.sol : MarketPlace.createMarket()   #1

            // Set the market
            markets[u][m] = market;

            // Have the lender contract approve the several contracts
@>          ILender(lender).approve(u, e, a, t[7]);
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L178-L182

`approve()` is called on the underlying if the `e`, `a`, or `t[7]` arguments are non-null:

```
// File: src/Lender.sol : Lender.approve()   #2

        function approve(
            address u,
            address a,
            address e,
            address n
        ) external authorized(marketPlace) {
            uint256 max = type(uint256).max;
            IERC20 uToken = IERC20(u);
            if (a != address(0)) {
                Safe.approve(uToken, a, max);
            }
            if (e != address(0)) {
                Safe.approve(uToken, e, max);
            }
            if (n != address(0)) {
                Safe.approve(uToken, n, max);
            }
            if (IERC20(u).allowance(address(this), swivelAddr) == 0) {
                Safe.approve(uToken, swivelAddr, max);
            }
        }
```

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L194-L214

If CTokens ever are upgraded to have the protection, a similar approval issue will occur for the PTs themselves.

Tool used

Manual Review

Recommendation

Modify `Safe.approve()` to always call `approve(0)` before doing the real approval

Discussion

sourabhmarathe

Duplicate of #167

Evert0x

Agree with medium severity

Issue M-8: User-supplied AMM pools and no input validation allows stealing of stEth protocol fees

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/47

Found by

IIIIIII, kenzo

Summary

Some of the protocols `lend` methods take as user input the *underlying asset* and the *pool to swap on*. They do not check that they match. **Therefore a user can supply to `Lender` DAI underlying, instruct `Lender` to swap stEth with 0 `minAmountOut`, and sandwich the transaction to 0, thereby stealing all of Lender's stEth fees.**

Vulnerability Detail

In Tempus, APWine, Sense, Illuminate and Swivel's `lend` methods, the underlying, the pool to swap on, and the minAmountOut, are all user inputs. **There is no check that they match**, and the external swap parameters do not contain the actual asset to swap - only the pool to swap in. Which is a user input. So an attacker can do the following, for example with APWine:

- Let's say `Lender` has accumulated 100 stEth in fees.
- The attacker will call APWine's `lend`, with `underlying=DAI`, `amount=100eth`, `minimumAmountOfTokensToBuy=0`, and AMM pool (`x`) that is actually for stEth (*tam tam tam!*).
- `lend` will pull 100 DAI from the attacker.
- `lend` will call APWine's router with the *stEth pool*, and 0 `minAmountOut`. (I show this in code snippet section below).
- The attacker will sandwich this whole `lend` call such that `Lender` will receive nearly 0 tokens. This is possible since the user-supplied `minAmountOut` is 0.
- `lend` will execute this swapping operation. It will receive nearly 0 APWine-stEth-PTs.
- Since the attacker sandwiched this transaction to 0, he will gain all the stEth that Lender tried to swap - all the stEth fees of the protocol.

Impact

Theft of stEth fees, as detailed above.

SHERLOCK

Code Snippet

Here is APWine's `lend` method. You can notice the following things. Specifically note the `swapExactAmountIn` operation.

- There is no check that user-supplied `pool` swaps token `u`
- `apwinePairPath()` and `apwineTokenPath()` do not contain actual asset addresses, but only relative `0` or `1`
- Therefore, `pool` can be totally unrelated to `u`
- The user supplies the slippage limit - `r` - so he can use `0`
- The swap will be executed for the same amount (minus fees) that has been pulled from the user; but user can supply DAI and swap for same amount of stEth, a Very Profitable Trading Strategy
- We call the real APWine router so `Lender` has already approved it

Because of these, the attack described above will succeed - the user can supply DAI as underlying, but actually make Lender swap stEth with 0 minAmountOut.

```
/// @notice lend method signature for APWine
/// @param p principal value according to the MarketPlace's Principals Enum
/// @param u address of an underlying asset
/// @param m maturity (timestamp) of the market
/// @param a amount of underlying tokens to lend
/// @param r slippage limit, minimum amount to PTs to buy
/// @param d deadline is a timestamp by which the swap must be executed
/// @param x APWine router that executes the swap
/// @param pool the AMM pool used by APWine to execute the swap
/// @return uint256 the amount of principal tokens lent out
function lend( uint8 p, address u, uint256 m, uint256 a, uint256 r, uint256 d,
↪   address x, address pool) external unpaused(u, m, p) returns (uint256) {
    address principal = IMarketPlace(marketPlace).token(u, m, p);

    // Transfer funds from user to Illuminate
    Safe.transferFrom(IERC20(u), msg.sender, address(this), a);

    uint256 lent;
    {
        // Add the accumulated fees to the total
        uint256 fee = a / feenominator;
        fees[u] = fees[u] + fee;

        // Calculate amount to be lent out
        lent = a - fee;
    }
```

```
    // Get the starting APWine token balance
    uint256 starting = IERC20(principal).balanceOf(address(this));

    // Swap on the APWine Pool using the provided market and params
    IAPWineRouter(x).swapExactAmountIn(
        pool,
        apwinePairPath(),
        apwineTokenPath(),
        lent,
        r,
        address(this),
        d,
        address(0)
    );

    // Calculate the amount of APWine principal tokens received after the swap
    uint256 received = IERC20(principal).balanceOf(address(this)) -
        starting;

    // Mint Illuminate zero coupons
    IERC5095(principalToken(u, m)).authMint(msg.sender, received);

    emit Lend(p, u, m, received, a, msg.sender);
    return received;
}

function apwineTokenPath() internal pure returns (uint256[] memory) {
    uint256[] memory tokenPath = new uint256[](2);
    tokenPath[0] = 1;
    tokenPath[1] = 0;
    return tokenPath;
}

/// @notice returns array pair path required for APWine's swap method
/// @return array of uint256[] as laid out in APWine's docs
function apwinePairPath() internal pure returns (uint256[] memory) {
    uint256[] memory pairPath = new uint256[](1);
    pairPath[0] = 0;
    return pairPath;
}
```

The situation is similar in:

- Tempus, where `x` is the pool to swap on.
- Sense, where `adapter` is user-supplied.
- Illuminate, where if the principal is Yield, the function is checking that the un-

derlying token matches the pool. But the user can supply the principal to be Illuminate, bypassing this check, and supplying the YieldPool `y` to be one that swaps stEth for fyEth.

- Swivel, where I believe that the user can supply an order to swap stEth instead of DAI.

Tool used

Manual Review

Recommendation

Check that the user-supplied pool/adapter/order's tokens match the underlying. This should ensure that the user only swaps assets he supplied.

Issue M-9: Lending on Swivel: protocol fees not taken when remainder of underlying is swapped in YieldPool

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/45

Found by

kenzo, cccz

Summary

The `lend` function for Swivel allows swapping the remainder underlying on Yield. But it does not take protocol fees on this amount.

Vulnerability Detail

When executing orders on Swivel, if the user has set `e==true` and there is remaining underlying, the lending function will swap these funds using YieldPool. But it does not take the protocol fees on that amount.

Impact

Some protocol fees will be lost. Users may even use this function to trade on the YieldPool without incurring protocol fees. While I think it can be rightfully said that at that point they can just straight away trade on the YieldPool without incurring fees, that can also be said about the general Illuminate/Yield `lend` function, which swaps on the YieldPool and does extract fees.

Code Snippet

In Swivel's `lend`, if the user has set `e` to true, the following block will be executed. Note that no fees are extracted from the raw balance.

```
if (e) {
    // Calculate the premium
    uint256 premium = IERC20(u).balanceOf(address(this)) - starting;
    // Swap the premium for Illuminate principal tokens
    swivelLendPremium(u, m, y, premium, premiumSlippage);
}
```

`swivelLendPremium` being:

```
// Lend remaining funds to Illuminate's Yield Space Pool
uint256 swapped = yield(u, y, p, address(this),
 ↪  IMarketPlace(marketPlace).token(u, m, 0), slippageTolerance);
```

```
// Mint the remaining tokens
IERC5095(principalToken(u, m)).authMint(msg.sender, swapped);
```

And `yield` doesn’t take protocol fees either. So the fees are lost from the premium.

Tool used

Manual Review

Recommendation

In the `if(e)` block of Swivel's `lend`, extract the protocol fee from `premium`.

Issue M-10: setPrincipal fails to approve Notional contract to spend lender's underlying tokens

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/41

Found by

rvierdiiev, bin2chen, neumo, Holmgren

Summary

If the **Notional** principal is not set at Marketplace creation, when trying to add it at a later time via **setPrincipal**, the call will not accomplish that the lender approves the notional contract to spend its underlying tokens, due to passing the zero address as underlying to the lender's approve function.

Vulnerability Detail

The vulnerability lies in line 238 of **Marketplace** contract: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L238 Function **approve** of **Lender** contract expects the address of the underlying contract as the first parameter: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L194-L214 As the underlying address passed in in the buggy line above is the zero address, **uToken** is also the zero adress and `Safe.approve(uToken,n,max);` just calls approve on the zero address, which does nothing (not even reverting because there's no contract deployed there).

Impact

If there was no way for the lender contract to approve the notional address, I would rate this issue as High, but since there is an admin function `functionapprove(address[]calldatau,address[]calldataa)`, the admin could fix this issue approving the notional contract over the underlying token, making the impact less severe. But in the meantime **Notional**'s lending would revert due to the lack of approval.

Code Snippet

The following test, that can be added in the **MarketPlace.t.sol** file, proves this vulnerability:

```
function testIssueSetPrincipalNotional() public {

    address notional = address(token7);

    address[8] memory contracts;
```

```
contracts[0] = address(token0); // Swivel
contracts[1] = address(token1); // Yield
contracts[2] = address(token2); // Element
contracts[3] = address(token3); // Pendle
contracts[4] = address(token4); // Tempus
contracts[5] = address(token5); // Sense
contracts[6] = address(token6); // APWine
contracts[7] = address(0); // Notional unset at market creation

mock_erc20.ERC20(underlying).decimalsReturns(10);
mock_erc20.ERC20 compounding = new mock_erc20.ERC20();
token6.futureVaultReturns(address(apwfv));
apwfv.getIBTAddressReturns(address(compounding));

token3.underlyingYieldTokenReturns(address(compounding));

mp.createMarket(
    address(underlying),
    maturity,
    contracts,
    'test-token',
    'tt',
    address(elementVault),
    address(apwineRouter)
);

// verify approvals
assertEq(r.approveCalled(), address(compounding));

// We verify that the notional address approved for address(0) is unset
(, , address approvedNotional) = l.approveCalled(address(0));
assertEq(approvedNotional, address(0));
// and that the approved notional for address(underlying) is unset
(, , approvedNotional) = l.approveCalled(address(underlying));
assertEq(approvedNotional, address(0));

// Then we call setPrincipal for the notional address
mp.setPrincipal(uint8(MarketPlace.Principals.Notional), address(underlying),
↪ maturity, notional);

// Now we verify that, after the call to setPrincipal, the notional address
// approved for address(0) is the Notional address provided in the call
(, , approvedNotional) = l.approveCalled(address(0));
assertEq(approvedNotional, notional);
// and that the approved notional for address(underlying) is still unset
(, , approvedNotional) = l.approveCalled(address(underlying));
assertEq(approvedNotional, address(0));
```

```
}
```

Tool used

Forge Tests and manual Review

Recommendation

Change this line in Marketplace.sol: https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L238 with this: `ILender(lender).approve(address(u),address(0),address(0),a);`

Issue M-11: Marketplace.setPrincipal do not approve needed allowance for Element vault and APWine router

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/40

Found by

pashov, rvierdiiev

Summary

`Marketplace.setPrincipal` do not approve needed allowance for `Elementvault` and `APWinerouter`

Vulnerability Detail

`Marketplace.setPrincipal` is used to provide principal token for the base token and maturity when it was not set yet. To set PT you also provide protocol that this token belongs to.

In case of `APWine` protocol there is special block of code to handle all needed allowance. But it is not enough.

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L231-L236

```
} else if (p == uint8(Principals.Apwine)) {
    address futureVault = IAPWineToken(a).futureVault();
    address interestBearingToken = IAPWineFutureVault(futureVault)
        .getIBTAddress();
    IRedeemer(redeemer).approve(interestBearingToken);
} else if (p == uint8(Principals.Notional)) {
```

In `Marketplace.createMarket` function 2 more params are used to provide allowance of Lender for Element vault and APWine router. https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Marketplace.sol#L182 `ILender(lender).approve(u,e,a,t[7]);`

But in `setPrincipal` we don't have such params and allowance is not set. So `Lender` will not be able to work with that tokens correctly.

Impact

Lender will not provide needed allowance and protocol integration will fail.

Code Snippet

Provided above.

Tool used

Manual Review

Recommendation

Add 2 more params as in `createMarket` and call `ILender(lender).approve(u,e,a,address(0));`

Discussion

sourabhmarathe

Suggested severity is Low on the grounds that we have an `admin` method that would allow us to handle these particular approvals. That being said, we will be implementing a fix based on this report.

Evert0x

Issue will stay medium severity, although Illuminate is able to fix it using admin powers.. it's still a broken codebase that can potentially impact user funds.

Issue M-12: Redeemer.setFee function will always revert

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/34

Found by

cryptphi, ak1, JohnSmith, rvierdiiev, bin2chen, hansfriese, John

Summary

`Redeemer.setFee` function will always revert and will not give ability to change `feenominator`.

Vulnerability Detail

`Redeemer.setFee` function is designed to give ability to change `feenominator` variable.

https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Redeemer.sol#L168-L187

```
function setFee(uint256 f) external authorized(admin) returns (bool) {
    uint256 feeTime = feeChange;
    if (feeTime == 0) {
        revert Exception(23, 0, 0, address(0), address(0));
    } else if (feeTime < block.timestamp) {
        revert Exception(
            24,
            block.timestamp,
            feeTime,
            address(0),
            address(0)
        );
    } else if (f < MIN_FEENOMINATOR) {
        revert Exception(25, 0, 0, address(0), address(0));
    }
    feenominator = f;
    delete feeChange;
    emit SetFee(f);
    return true;
}
```

As `feeChange` value is 0(it's not set anywhere), this function will always revert wtih `Exception(23,0,0,address(0),address(0))`. Also even if `feeChange` was not 0, the function will give ability to change fee only once, because in the end it calls `delete feeChange` which changes it to 0 again.

Impact

Fee can't be changed.

Code Snippet

Provided above.

Tool used

Manual Review

Recommendation

Add same functions as in `Lender`. https://github.com/sherlock-audit/2022-10-illuminate/blob/main/src/Lender.sol#L813-L829;

Issue M-13: ERC5095.mint function calculates slippage incorrectly

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/31

Found by

rvierdiiev, cccz, pashov, bin2chen, kenzo, hansfriese, hyh

Summary

ERC5095.mint function calculates slippage incorrectly. This leads to lost of funds for user.

Vulnerability Detail

`ERC5095.mint` function should take amount of shares that user wants to receive and then buy this amount. It uses hardcoded 1% slippage when trades base tokens for principal. But it takes 1% of calculated assets amount, not shares.

```
function mint(address r, uint256 s) external override returns (uint256) {
    if (block.timestamp > maturity) {
        revert Exception(
            21,
            block.timestamp,
            maturity,
            address(0),
            address(0)
        );
    }
    uint128 assets = Cast.u128(previewMint(s));
    Safe.transferFrom(
        IERC20(underlying),
        msg.sender,
        address(this),
        assets
    );
    // consider the hardcoded slippage limit, 4626 compliance requires no minimum
↪   param.
    uint128 returned = IMarketPlace(marketplace).sellUnderlying(
        underlying,
        maturity,
        assets,
        assets - (assets / 100)
    );
    _transfer(address(this), r, returned);
```

```
    return returned;
}
```

This is how slippage is provided

```
uint128 returned = IMarketPlace(marketplace).sellUnderlying(
            underlying,
            maturity,
            assets,
            assets - (assets / 100)
        );
```

But the problem is that assets it is amount of base tokens that user should pay for the shares he want to receive. Slippage should be calculated using shares amount user expect to get.

Example. User calls mint and provides amount 1000. That means that he wants to get 1000 principal tokens. While converting to assets, assets = 990. That means that user should pay 990 base tokens to get 1000 principal tokens. Then the `sellUnderlying` is send and slippage provided is `990*0.99=980.1`. So when something happens with price it's possible that user will receive 980.1 principal tokens instead of 1000 which is 2% lost.

To fix this you should provide `s-(s/100)` as slippage.

Impact

Lost of users funds.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Use this.

```
uint128 returned = IMarketPlace(marketplace).sellUnderlying(
            underlying,
            maturity,
            assets,
```

```
        s- (s / 100)
    );
```

Issue M-14: ERC5095.deposit doesn't check if received shares is less then provided amount

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/30

Found by

rvierdiiev

Summary

`ERC5095.deposit` doesn't check if received shares is less then provided amount. In some cases this leads to lost of funds.

Vulnerability Detail

The main thing with principal tokens is to buy them when the price is lower (you can buy 101 token while paying only 100 base tokens) as underlying price and then at maturity time to get interest(for example in one month you will get 1 base token in our case).

`ERC5095.deposit` function takes amount of base token that user wants to deposit and returns amount of shares that he received. To not have loses, the amount of shares should be at least bigger than amount of base tokens provided by user.

```
function deposit(address r, uint256 a) external override returns (uint256) {
    if (block.timestamp > maturity) {
        revert Exception(
            21,
            block.timestamp,
            maturity,
            address(0),
            address(0)
        );
    }
    uint128 shares = Cast.u128(previewDeposit(a));
    Safe.transferFrom(IERC20(underlying), msg.sender, address(this), a);
    // consider the hardcoded slippage limit, 4626 compliance requires no minimum
↪   param.
    uint128 returned = IMarketPlace(marketplace).sellUnderlying(
        underlying,
        maturity,
        Cast.u128(a),
        shares - (shares / 100)
    );
    _transfer(address(this), r, returned);
```

```
    return returned;
}
```

While calling market place, you can see that slippage of 1 percent is provided.

```
uint128 returned = IMarketPlace(marketplace).sellUnderlying(
            underlying,
            maturity,
            Cast.u128(a),
            shares - (shares / 100)
        );
```

But this is not enough in some cases.

For example we have `ERC5095` token with short maturity which provides `0.5%` of interests. userA calls `deposit` function with 1000 as base amount. He wants to get back 1005 share tokens. And after maturity time earn 5 tokens on this trade.

But because of slippage set to `1%`, it's possible that the price will change and user will receive 995 share tokens instead of 1005, which means that user has lost 5 base tokens.

I propose to add one more mechanism except of slippage. We need to check if returned shares amount is bigger then provided assets amount.

Impact

Lost of funds.

Code Snippet

Provided above.

Tool used

Manual Review

Recommendation

Add this check at the end `require(returned>a,"receivedlessthanprovided")`

Discussion

Evert0x

@sourabhmarathe which severity would you give this? And why?

sourabhmarathe

Low severity on the basis that ultimately, user funds are not at risk in this case. However, it is still worth noting that this issue should be addressed using the recommended changes provided in this report.

Evert0x

Will keep it medium severity as the protocol agrees to fix the issues which can lead to a loss of funds according to the description.

Issue M-15: Extra minting after `yield()` function causes iPT supply inflation and skewed accounting

Source: https://github.com/sherlock-audit/2022-10-illuminate-judging/issues/27

Found by

kenzo

Summary

In Swivel and Illuminate's `lend` functions, `yield()` is being called, which swaps PTs for iPTs. After that call, additional iPTs are minted and sent to the user. This means that Lender ends up holding extra iPTs which will skew the accounting.

Vulnerability Detail

Described above and below.

Impact

Redemption accounting is off. If iPT supply is inflated and Lender holds iPTs, then upon redemption, every user will get less underlying than deserved. The underlying can still be rescued by Illuminate team if they withdraw the iPT from Lender, redeem it themselves, and distribute it rightfully to all the users. But I think that's probably not something that should happen nor that Illuminate wants to have to do. As this functionality is legit use of the protocol, it means the funds will have to be rescued and distributed manually to all the users every time.

Code Snippet

When a user calls `lends` for Illuminate principal, the function will call `yield()` and then mint iPTs to the user.

```
uint256 returned = yield(u, y, a - a / feenominator, address(this), principal,
 ↪  minimum);
IERC5095(principalToken(u, m)).authMint(msg.sender, returned);
```

The same thing happens in `swivelLendPremium`.

```
uint256 swapped = yield(u, y, p, address(this),
 ↪  IMarketPlace(marketPlace).token(u, m, 0), slippageTolerance);
IERC5095(principalToken(u, m)).authMint(msg.sender, swapped);
```

But `yield()` function already swaps PTs for iPTs, which end up in `Lender` itself (3rd parameter above, `address(this)`) - so there is no need to mint additional ones.

Therefore, `Lender` has bought iPTs from the pool for the user, and then proceeds to mint additional ones and send them to the user, leaving the swapped ones in Lender's possession. This leads to inflated supply, and as Redeemer redeems user's iPTs as per iPT's total supply, this leads to the discrepancy detailed above.

Tool used

Manual Review

Recommendation

If `yield()` has bought from the YieldPool iPTs for the user, send them to him, instead of minting extra new ones.