Illuminate contest
Findings & Analysis Report

2022-08-23

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Illuminate smart contract system written in Solidity. The audit contest took place between June 21—June 26 2022.

Wardens

99 Wardens contributed reports to the Illuminate contest:

1. kenzo
2. datapunk
3. cccz
4. kirk-baird
5. 0x52
6. csanuragjain
7. hyh
8. WatchPug (jtp and ming)
9. Alex the Entreprenerd
10. unforgiven
11. Lambda
12. IIIIIII
13. Kumpa
14. Metatron

15. pashov
16. defsec
17. bardamu
18. ladboy233
19. shenwilly
20. Chom
21. BowTiedWardens (BowTiedHeron, BowTiedPickle,m4rio_eth, Dravee, and BowTiedFirefox)
22. 0x29A (0x4non and rotcivegaf)
23. cryptphi
24. auditor0517
25. sashik_eth
26. hansfriese
27. Picodes
28. dipp
29. joestakey
30. itsmeSTYJ
31. Kulk0
32. zer0dot
33. 0x1f8b
34. GimelSec (rayn and sces60107)
35. StErMi
36. robee
37. 0xkowloon
38. oyc_109
39. poirots (DavideSilva, resende and naps62)
40. TomJ
41. simon135
42. 0xkatana

43. Soosh
44. Waze
45. _Adam
46. catchup
47. grGred
48. Bnke0x0
49. z3s
50. fatherOfBlocks
51. ElKu
52. MadWookie
53. delfin454000
54. JC
55. slywaters
56. 0xf15ers (remora and twojoy)
57. 0xNazgul
58. asutorufos
59. Funen
60. hake
61. kebabsec (okkothejawa and FlameHorizon)
62. rfa
63. 0xDjango
64. saian
65. 0xKitsune
66. ajtra
67. Tomio
68. jah
69. 0xmint
70. 0xNineDec
71. ak1

72. aysha
73. Kenshin
74. Limbooo
75. zeesaw
76. Yiko
77. Kaiziron
78. Noah3o6
79. UnusualTurtle
80. 0v3rf10w
81. c3phas
82. Fitraldys
83. ignacio
84. Nyamcil
85. RoiEvenHaim
86. sach1r0
87. samruna
88. tintin

This contest was judged by gzeon.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 29 unique vulnerabilities. Of these vulnerabilities, 16 received a risk rating in the category of HIGH severity and 13 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 62 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 56 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Illuminate contest repository, and is composed of 3 smart contracts written in the Solidity programming language and includes 1,297 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (16)

[H-01] The lend function for tempus uses the wrong return value of depositAndFix

Submitted by cccz, also found by 0x52 and datapunk

The depositAndFix function of the TempusController contract returns two uint256 data, the first is the number of shares exchanged for the underlying token, the second is the number of principalToken exchanged for the shares, the second return value should be used in the lend function for tempus.

This will cause the contract to mint an incorrect number of illuminateTokens to the user.

Proof of Concept

https://github.com/code-423n4/2022-06-illuminate/blob/92cbb0724e594ce025d6b6ed050d3548a38c264b/lender/Lender.sol#L452-L453

https://github.com/tempus-finance/tempus-protocol/blob/master/contracts/TempusController.sol#L52-L76

Recommended Mitigation Steps

interfaces.sol

```
interface ITempus {
    function maturityTime() external view returns (uint256);

    function yieldBearingToken() external view returns (IERC20Me

    function depositAndFix(
        Any,
        Any,
        uint256,
        bool,
        uint256,
        uint256
    ) external returns (uint256, uint256);
}
```

Lender.sol

```
        (,uint256 returned) = ITempus(tempusAddr).depositAndFix(
        returned -= illuminateToken.balanceOf(address(this));
```

sourabhmarathe (Illuminate) confirmed

[H-02] Division Before Multiplication Can Lead To Zero Rounding Of Return Amount

Submitted by kirk-baird, also found by csanuragjain, datapunk, and ladboy233

There is a division before multiplication bug that exists in `lend()` for the Swivel case.

If `order.premium` is less than `order.principal` then `returned` will round to zero due to the integer rounding.

When this occurs the user's funds are essentially lost. That is because they transfer in the underlying tokens but the amount sent to `yield(u, y, returned, address(this))` will be zero.

Proof of Concept

```
        function lend(
            uint8 p,
            address u,
            uint256 m,
            uint256[] calldata a,
            address y,
            Swivel.Order[] calldata o,
            Swivel.Components[] calldata s
        ) public unpaused(p) returns (uint256) {

            // lent represents the number of underlying tokens lent
            uint256 lent;
            // returned represents the number of underlying tokens t
            uint256 returned;

            {
                uint256 totalFee;
                // iterate through each order a calculate the total
                for (uint256 i = 0; i < o.length; ) {
                    Swivel.Order memory order = o[i];
                    // Require the Swivel order provided matches the
                    if (order.underlying != u) {
                        revert NotEqual('underlying');
                    } else if (order.maturity > m) {
```

```
                    revert NotEqual('maturity');
                }
                // Determine the fee
                uint256 fee = calculateFee(a[i]);
                // Track accumulated fees
                totalFee += fee;
                // Sum the total amount lent to Swivel (amount c
                lent += a[i] - fee;
                // Sum the total amount of premium paid from Swi
                returned += (a[i] - fee) * (order.premium / orde

                unchecked {
                    i++;
                }
            }
            // Track accumulated fee
            fees[u] += totalFee;

            // transfer underlying tokens from user to illuminat
            Safe.transferFrom(IERC20(u), msg.sender, address(thi
            // fill the orders on swivel protocol
            ISwivel(swivelAddr).initiate(o, a, s);

            yield(u, y, returned, address(this));
        }

        emit Lend(p, u, m, lent);
        return lent;
    }
```

Specifically the function `returned += (a[i] - fee) * (order.premium / order.principal);`

Recommended Mitigation Steps

The multiplication should occur before division, that is `((a[i] - fee) * order.premium) / order.principal);`.

JTraversa (Illuminate) confirmed

Alex the Entreprenerd (warden) commented:

> Also see how Swivel Calculates it

[H-03] Pendle Uses Wrong Return Value For `swapExactTokensForTokens()`

Submitted by kirk-baird, also found by 0x52, cccz, csanuragjain, kenzo, and WatchPug

The function `swapExactTokensForTokens()` will return and array with the 0 index being the input amount follow by each output amount. The 0 index is incorrectly used in Pendle `lend()` function as the output amount. As a result the value of `returned` will be the invalid (i.e. the input rather than the output).

Since this impacts how many PTs will be minted to the `msg.sender`, the value will very likely be significantly over or under stated depending on the exchange rate. Hence the `msg.sender` will receive an invalid number of PT tokens.

Proof of Concept

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

returned = IPendle(pendleAddr).swapExactTokensForTok
```

Recommended Mitigation Steps

The amount of `principal` returned should be index 1 of the array returned by `swapExactTokensForTokens()`.

sourabhmarathe (Illuminate) confirmed

[H-04] Allowance check always true in ERC5095 redeem

*Submitted by Lambda, also found by 0x29A, Chom, cryptphi, itsmeSTYJ, kenzo, kirk-baird, and sashik*eth_

In `redeem`, it is checked that the allowance is larger than `underlyingAmount`, which is the return parameter (i.e., equal to 0 at that point). Therefore, this check is always true and there is no actual allowance check, allowing anyone to redeem for another user.

Recommended Mitigation Steps

Change the `underlyingAmount` to `principalAmount`, which is the intended parameter.

sourabhmarathe (Illuminate) disputed and commented:

> While we did not actually intend to audit the 5095 implementation, as 5095 itself is not yet final, we did describe its purpose in our codebase in the initial readme, and didn't specify that it was not in scope. (we wanted wardens to understand its role in our infra)

> With that context, we will leave it up to the judges whether or not to accept issues related to the ERC5095 token.

gzeoneth (judge) commented:

> I think it is fair to accept issues related to the ERC5095 token.

[H-05] Redeemer.redeem() for Element withdraws PT to wrong address.

Submitted by auditor0517, also found by 0x52, cccz, datapunk, kenzo, and pashov

Redeemer.redeem() for Element withdraws PT to wrong address.

This might cause a result of loss of PT.

Proof of Concept

According to the ReadMe.md, Redeemer should transfer external principal tokens from Lender.sol to Redeemer.sol.

But it transfers to the “marketPlace” and it would lose the PT.

Recommended Mitigation Steps

Modify IElementToken(principal).withdrawPrincipal(amount, marketPlace); like this.

```
IElementToken(principal).withdrawPrincipal(amount, address(this)
```

sourabhmarathe (Illuminate) confirmed

[H-06] Tempus lend method wrongly calculates amount of iPT tokens to mint

Submitted by kenzo, also found by cccz, Metatron, unforgiven, and WatchPug

The Tempus `lend` method calculates the amount of tokens to mint as `amountReturnedFromTempus - lenderBalanceOfMetaPrincipalToken`. This seems wrong as there’s no connection between the two items. Tempus has no relation to the iPT token.

Impact

Wrong amount of iPT will be minted to the user. If the Lender contract has iPT balance, the function will revert, otherwise, user will get minted 0 iPT tokes.

Proof of Concept

This is how the `lend` method calculates the amount of iPT tokens to mint:

```
uint256 returned = ITempus(tempusAddr).depositAndFix(Any
    illuminateToken.balanceOf(address(this));
illuminateToken.mint(msg.sender, returned);
```

The Tempus `depositAndFix` method does not return anything. Therefore this calculation will revert if `illuminateToken.balanceOf(address(this)) > 0`, or will return 0 if the balance is 0.

[Note: there's another issue here where the depositAndFix sends wrong parameters - I will submit it in another issue.]

Recommended Mitigation Steps

I believe that what you intended to do is to check how many Tempus principal tokens the contract received.

So you need to check Lender's `x.tempusPool().principalShare()` before and after the swap, and the delta is the amount received.

sourabhmarathe (Illuminate) confirmed

[H-07] Redeem Sense can be bricked

Submitted by kenzo

Sense's `redeem` can be totally DOSd due to user supplied input.

Impact

Using this attack, Sense market can not be redeemed.

Proof of Concept

This is how Sense market is being redeemed:

```
IERC20 token = IERC20(IMarketPlace(marketPlace).markets(
uint256 amount = token.balanceOf(lender);
Safe.transferFrom(token, lender, address(this), amount);
ISense(d).redeem(o, m, amount);
```

The problem is that `d` is user supplied input and the function only tries to redeem the amount that was transferred from Lender.

A user can supply malicious `d` contract which does nothing on `redeem(o, m, amount)`. The user will then call Redeemer's `redeem` with his malicious contract. Redeemer will transfer all the prinicipal from Lender to itself, will call `d` (noop), and finish. Sense market has not been redeemed.

Now if somebody tries to call Sense market's `redeem` again, the `amount` variable will be 0, and Redeemer will try to redeem 0 from Sense.

All the original principal is locked and lost in the contract, like tears in rain.

Recommended Mitigation Steps

I think you should either use a whitelisted Sense address, or send to `ISense(d).redeem` Redeemer's whole principal balance.

sourabhmarathe (Illuminate) confirmed

[H-08] ERC5095 redeem/withdraw does not update allowances

*Submitted by kenzo, also found by 0x29A, cccz, csanuragjain, GimelSec, kirk-baird, Lambda, sashik*eth, shenwilly, and StErMi_

ERC5095's `redeem` / `withdraw` allows an ERC20-approved account to redeem user's tokens, but does not update the allowance after burning.

Impact

User Mal can burn more tokens than Alice allowed him to. He can set himself to be the receiver of the underlying, therefore Alice will lose funds.

Proof of Concept

`withdraw` and `redeem` functions check that the msg.sender has enough approvals to redeem the tokens:

```
require(_allowance[holder][msg.sender] >= underlying
```

But they do not update the allowances. They then call `authRedeem`, which also does not update the allowances. Therefore, an approved user could "re-use his approval" again and again and redeem whole of approver's funds to himself.

Recommended Mitigation Steps

Update the allowances upon spending.

sourabhmarathe (Illuminate) confirmed and commented:

> While we did not actually intend to audit the 5095 itself, as 5095 itself is not yet final, we did describe its purpose in our codebase in the initial readme, and didn't specify that it was not in scope.

> With that context, we will leave it up to the judges whether or not to accept issues related to the ERC5095 token.

[H-09] Lender: no check for paused market on mint

Submitted by kenzo, also found by bardamu, csanuragjain, and lllllll

Lender's `mint` function **does not check** whether the supplied market is paused.

Impact

Even if a market is paused due to insolvency/bugs, an attacker can issue iPTs.

This renders the whole pause and insolvency protection mechanism ineffective. See POC.

Proof of Concept

Let's say market P has become insolvent, and Illuminate pauses that market, as it doesn't want to create further bad debt.

Let's say P's principal tokens's value has declined severely in the market because of the insolvency.

An attacker can buy many worthless P principal tokens for cheap, then call Lender and mint from them iPT.

The attacker is now owed underlying which belongs to the legitimate users. There won't be enough funds to repay everybody.

Recommended Mitigation Steps

Check in `mint` that the market is not paused.

sourabhmarathe (Illuminate) confirmed

[H-10] Able to mint any amount of PT

*Submitted by dipp, also found by 0x1f8b, bardamu, Chom, datapunk, Alex the Entreprenerd, GimelSec, hyh, jah, kenzo, kirk-baird, Kumpa, ladboy233, Metatron, oyc*109, shenwilly, simon135, unforgiven, and zer0dot_

Lender.sol#L192-L235

Lender.sol#L486-L534

Lender.sol#L545-L589

Impact

Some of the `lend` functions do not validate addresses sent as input which could lead to a malicous user being able to mint more PT tokens than they should.

Functions affect:

- Illuminate and Yield `lend` function.
- Sense `lend` function.
- APWine `lend` function.

Proof of Concept

In the Illuminate and Yield `lend` function:

1. Let the Yieldspace pool `y` be a malicious contract that implements the `IYield` interface.
2. The `base` and `maturity` functions for `y` may return any value so the conditions on lines 208 and 210 are easily passed.
3. The caller of `lend` sends any amount `a` for the desired underlying `u`.
4. If principal token `p` corresponds to the Yield principal, then the `yield` function is called which has a **return value controlled by the malicious contract** `y`.
5. The `mint` function is then called for the principal token with an underlying `u` and a maturity `m` which will then mint the `returned` amount of principal tokens to the malicious user.

In the Sense `lend` function:

1. Let the amm `x` input variable be a malicous contract that implements the `ISense` interface.
2. The malicious user sends any amount of underlying to `Lender.sol`.
3. Since the amm isn't validated, the `swapUnderlyingForPTs` function can return any amount for `returned` that is used to mint the Illuminate tokens.
4. The malicious user gains a disproportionate amount of PT.

In the APWine `lend` function:

1. Let the APWine `pool` input variable be a malicous contract that implements the `IAPWineRouter` interface.
2. The malicious user sends any amount of underlying to `Lender.sol`.
3. The `swapExactAmountIn` function of the malicious `pool` contract returns any amount for `returned`.
4. The `mint` function is called for the PT with underlying `u` and maturity `m` with the attacker controlled `returned` amount.

Recommmended Mitigation Steps

Consider validating the input addresses of `y`, `x` and `pool` through a whitelisting procedure if possible or validating that the `returned` amounts correspond with the amount of PT gained from the protocols by checking the balance before and after the PTs are gained and checking the difference is equal to `returned`.

sourabhmarathe (Illuminate) confirmed

[H-11] Not minting iPTs for lenders in several lend functions

Submitted by Metatron, also found by 0x52, auditor0517, cccz, datapunk, hansfriese, hyh, kenzo, kirk-baird, shenwilly, unforgiven, and WatchPug

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L247-L305

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L317-L367

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L192-L235

Impact

Using any of the `lend` function mentioned, will result in loss of funds to the lender - as the funds are transferred from them but no iPTs are sent back to them!

Basically making lending via these external PTs unusable.

Proof of Concept

There is no minting of iPTs to the lender (or at all) in the 2 `lend` functions below:
https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L247-L305

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L317-L367

Corresponding to lending of (respectively):
swivel
element

Furthermore, in:
https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L227-L234
Comment says “Purchase illuminate PTs directly to msg.sender”, but this is not happening. sending yield PTs at best.

Recommended Mitigation Steps

Mint the appropriate amount of iPTs to the lender - like in the rest of the lend functions.

sourabhmarathe (Illuminate) confirmed

[H-12] Funds may be stuck when `redeeming` for Illuminate

Submitted by Picodes, also found by auditor0517, Chom, cryptphi, csanuragjain, hansfriese, hyh, kenzo, kirk-baird, Lambda, pashov, unforgiven, and zer0dot

Funds may be stuck when `redeeming` for Illuminate.

Proof of Concept

Assuming the goal of calling `redeem` for Illuminate here is to redeem the Illuminate principal held by the lender or the redeemer, then there is an issue because the wrong balance is checked. So if no `msg.sender` has a positive balance funds will be lost.

Now assuming the goal of calling `redeem` for Illuminate here is for users to redeem their Illuminate principal and receive the underlying as suggested by this comment,

then the underlying is not sent back to users because `Safe.transferFrom(IERC20(u), lender, address(this), amount);` send the funds to the redeemer, not the user.

Recommended Mitigation Steps

Clarify the purpose of this function and fix the corresponding bug.

sourabhmarathe (Illuminate) confirmed

[H-13] Illuminate PT redeeming allows for burning from other accounts

Submitted by hyh, also found by 0x1f8b, 0x29A, cccz, Chom, csanuragjain, hansfriese, itsmeSTYJ, kenzo, pashov, shenwilly, Soosh, and unforgiven

Illuminate PT burns shares from a user supplied address account instead of user's account. With such a discrepancy a malicious user can burn all other's user shares by having the necessary shares on her balance, while burning them from everyone else.

Setting the severity to be high as this allows for system-wide stealing of user's funds.

Proof of Concept

Redeemer's Illuminate redeem() checks the balance of msg.sender, but burns from the balance of user supplied `o` address:

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/redeemer/Redeemer.sol#L114-L128

L120:

```
    uint256 amount = token.balanceOf(msg.sender);
```

L126:

```
token.burn(o, amount);
```

```
        address principal = IMarketPlace(marketPlace).markets(u,

        if (p == uint8(MarketPlace.Principals.Illuminate)) {
            // Get Illuminate's principal token
            IERC5095 token = IERC5095(principal);
            // Get the amount of tokens to be redeemed from the
            uint256 amount = token.balanceOf(msg.sender);
            // Make sure the market has matured
            if (block.timestamp < token.maturity()) {
                revert Invalid('not matured');
            }
            // Burn the prinicipal token from Illuminate
            token.burn(o, amount);
            // Transfer the original underlying token back to th
            Safe.transferFrom(IERC20(u), lender, address(this),
```

- address isn't validated and used as provided.

Burning proceeds as usual, Illuminate PT burns second argument `a` from the first argument `f`, i.e. `f`'s balance to be reduced by `a`:

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/marketplace/ERC5095.sol#L121-L127

```
    /// @param f Address to burn from
    /// @param a Amount to burn
    /// @return bool true if successful
    function burn(address f, uint256 a) external onlyAdmin(redee
        _burn(f, a);
        return true;
    }
```

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/marketplace/ERC5095.sol#L7

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

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/marketplace/ERC20.sol#L187-L196

```
    function _burn(address src, uint wad) internal virtual retur
        unchecked {
            require(_balanceOf[src] >= wad, "ERC20: Insufficient
            _balanceOf[src] = _balanceOf[src] - wad;
            _totalSupply = _totalSupply - wad;
            emit Transfer(src, address(0), wad);
        }

        return true;
    }
```

This way a malicious user owning some Illuminate PT can burn the same amount of PT as she owns from any another account, that is essentially from all other accounts, obtaining all the underlying tokens from the system. The behavior is somewhat similar to the public burn case.

Recommended Mitigation Steps

- address looks to be not needed in Illuminate PT case.

Consider burning the shares from `msg.sender`, for example:

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/redeemer/Redeemer.sol#L125-L126

```
            // Burn the prinicipal token from Illuminate
-           token.burn(o, amount);
+           token.burn(msg.sender, amount);
```

sourabhmarathe (Illuminate) confirmed

[H-14] `Redeemer.sol#redeem()` can be called by anyone before maturity, which may lead to loss of user funds

Submitted by WatchPug, also found by csanuragjain, datapunk, and Lambda

```
    function redeem(
        uint8 p,
        address u,
        uint256 m
    ) public returns (bool) {
        // Get the principal token that is being redeemed by the use
        address principal = IMarketPlace(marketPlace).markets(u, m,

        // Make sure we have the correct principal
        if (
            p != uint8(MarketPlace.Principals.Swivel) &&
            p != uint8(MarketPlace.Principals.Element) &&
            p != uint8(MarketPlace.Principals.Yield) &&
            p != uint8(MarketPlace.Principals.Notional)
        ) {
            revert Invalid('principal');
        }

        // The amount redeemed should be the balance of the principa
        uint256 amount = IERC20(principal).balanceOf(lender);

        // Transfer the principal token from the lender contract to
        Safe.transferFrom(IERC20(principal), lender, address(this),

        if (p == uint8(MarketPlace.Principals.Swivel)) {
            // Redeems zc tokens to the sender's address
            ISwivel(swivelAddr).redeemZcToken(u, m, amount);
        } else if (p == uint8(MarketPlace.Principals.Element)) {
            // Redeems principal tokens from element
            IElementToken(principal).withdrawPrincipal(amount, marke
        } else if (p == uint8(MarketPlace.Principals.Yield)) {
            // Redeems prinicipal tokens from yield
            IYieldToken(principal).redeem(address(this), address(thi
        } else if (p == uint8(MarketPlace.Principals.Notional)) {
            // Redeems the principal token from notional
            amount = INotional(principal).maxRedeem(address(this));
        }
```

```
        emit Redeem(p, u, m, amount);
        return true;
    }
```

There are some protocols (eg Notional) that allows redeem before maturity, when doing so, they will actually make a market sell, usually means a discounted sale.

Since `redeem()` is a public function, anyone can call it before maturity, and force the whole protocol to sell it's holdings at a discounted price, causing fund loss to the stake holders.

https://github.com/notional-finance/wrapped-fcash/blob/8f76be58dda648ea58eef863432c14c940e13900/contracts/wfCashERC4626.sol#L155-L169

```
    function previewRedeem(uint256 shares) public view override retu
        if (hasMatured()) {
            assets = convertToAssets(shares);
        } else {
            // If withdrawing non-matured assets, we sell them on th
            (uint16 currencyId, uint40 maturity) = getDecodedID();
            (assets, /* */, /* */, /* */) = NotionalV2.getPrincipalF
                currencyId,
                shares,
                maturity,
                0,
                block.timestamp
            );
        }
    }
```

Recommendation

Consider only allow unauthenticated call after maturity.

JTraversa (Illuminate) confirmed

[H-15] Incorrect implementation of APWine and Tempus `redeem`

Submitted by shenwilly, also found by cccz, Chom, datapunk, kenzo, Picodes, and unforgiven

Redeeming APWine and Tempus PT will always fail, causing a portion of iPT to not be able to be redeemed for the underlying token.

The issue is caused by the incorrect implementation of `redeem`:

```
uint256 amount = IERC20(principal).balanceOf(lender);
Safe.transferFrom(IERC20(u), lender, address(this), amount);
```

The first line correctly calculates the balance of PT token available in `Lender`. However, the second line tries to transfer the underlying token `u` instead of `principal` from Lender to `Redeemer`. Therefore, the redeeming process will always fail as both `APWine.withdraw` and `ITempus.redeemToBacking` will try to redeem non-existent PT.

Recommended Mitigation Steps

Fix the transfer line:

```
Safe.transferFrom(IERC20(principal), lender, address(this), amou
```

sourabhmarathe (Illuminate) confirmed

kenzo (warden) commented:

> (Referring all dups here, severity should be upgraded as user funds at risk)

gzeoneth (judge) increased severity to High and commented:

> Agree.

[H-16] Unable to redeem from Notional

Submitted by dipp, also found by cccz, cryptphi, datapunk, hyh, kenzo, Lambda, and WatchPug

The `maxRedeem` function is a view function which only returns the balance of the `Redeemer.sol` contract. After this value is obtained, the PT is not redeemed from Notional. The user will be unable to redeem PT from Notional through `Redeemer.sol`.

Proof of Concept

Notional code:

```
        function maxRedeem(address owner) public view override retur
            return balanceOf(owner);
        }
```

Recommmended Mitigation Steps

Call `redeem` from Notional using the `amount` from `maxRedeem` as the `shares` input after the call to `maxRedeem`.

kenzo (warden) commented:

> Should be high severity as affects user funds.

gzeoneth (judge) increased severity to High

sourabhmarathe (Illuminate) confirmed and commented

> This is confirmed as a high-risk issue.

Medium Risk Findings (13)

[M-01] sellPrincipalToken, buyPrincipalToken, sellUnderlying, buyUnderlying uses pool funds but pays msg.sender

Submitted by 0x52, also found by datapunk, kenzo, kirk-baird, and pashov

Fund loss from marketplace.

Proof of Concept

sellPrincipalToken, buyPrincipalToken, sellUnderlying, buyUnderlying are all unpermissioned and use marketplace funds to complete the action but send the resulting tokens to msg.sender. This means that any address can call these functions and steal the resulting funds.

Recommended Mitigation Steps

All functions should use safetransfer to get funds from msg.sender not from marketplace.

JTraversa (Illuminate) confirmed, but disagreed with severity and commented:

> 🤷‍♀️ Unsure if reasonable to disagree with severity, but in this case none of these methods would work at all lol, so I suppose no value is at risk?

gzeoneth (judge) decreased severity to Medium and commented:

> Unclear how fund would end up in the contract and doesn't seem to be possible during normal operation. Judging as Med Risk.

[M-02] Marketplace calls unimplemented function

Submitted by 0x52

https://github.com/code-423n4/2022-06-illuminate/blob/3ca41a9f529980b17fdc67baf8cbee5a8035afab/marketplace/MarketPlace.sol#L203-L204

https://github.com/code-423n4/2022-06-illuminate/blob/3ca41a9f529980b17fdc67baf8cbee5a8035afab/marketplace/Mar

ketPlace.sol#L220

Vulnerability details

While safe.sol isn't explicitly listed in scope it is listed as a library for all in scope contracts so I believe it should be in scope

Impact

mint and mintWithUnderlying won't function.

Proof of Concept

`safe.sol` never implements a transferFrom function meaning it will revert whenever a user calls either function.

Recommended Mitigation Steps

Create an implementation for transferFrom.

JTraversa (Illuminate) disagreed with severity and commented:

> I don't believe this should be considered high risk (an incorrectly copy/pasted safeTransfer lib) based on principal hah, but beyond that I don't think it should be high risk because without the expected safeTransfer lib, the contracts would not properly compile / no value would be at risk.

gzeoneth (judge) decreased severity to Medium and commented:

> Judging as Med Risk as no value at risk and function of the protocol would be impacted.

[M-03] Calls To `Swivel.initiate()` Do Not Verify `o.exit` or `o.vault` Allowing An Attacker To Manipulate Accounting In Their Favour

Submitted by kirk-baird

Swivel `lend()` does not validate the `o.exit` and `o.vault` for each order before making the external call to Swivel. These values determine which internal functions is called in Swivel.

The intended code path is `initiateZcTokenFillingVaultInitiate()` which takes the underlying tokens and mints zcTokens to the `Lender`. If one of the other functions is called the accounting in `lend()`. Swivel may transfer more tokens from `Lender` to `Swivel` than paid for by the caller of `lend()`.

The impact is that underlying tokens may be stolen from `Lender`.

Proof of Concept

Consider the example where initiateZcTokenFillingZcTokenExit() is called. This will transfer `a - premiumFilled + fee` from `Lender` to `Swivel` rather than the expected `a + fee`.

Recommended Mitigation Steps

In `lend()` restrict the values of `o.exit` and `o.vault` so only one case can be triggered in `Swivel.initiate()`.

sourabhmarathe (Illuminate) commented:

> While it is true that a user could get better execution by submitting certain orders, we don't think this is a problem. Invalid orders would be rejected by Swivel, and users should be free to execute the best possible orders.

JTraversa (Illuminate) confirmed, but disagreed wtih severity and commented:

> So reviewing this, there is an issue though it may not be high-risk.
>
> The user *can* manipulate the method by sending it an order that is not the correct type to go down the intended order path.
>
> That said, the result on line 297 is still that the calculated lent value is sent to the contract.

> So the result is that the user inputting this manipulation actually still pays for their iPTs, and their underlying just sits in lender.sol until maturity with no personal benefit. The attack would none-the-less leak value and with that in mind I'd probably just drop it down to medium?

gzeoneth (judge) decreased severity to Medium and commented:

> Judging as Med Risk as no fund is lost (after maturity).

[M-04] Checking yieldBearingToken against u instead of backingToken

Submitted by datapunk

The lend function for tempus will fail with the right market.

Proof of Concept

Checks `if (ITempus(principal).yieldBearingToken() != IERC20Metadata(u))` , while it should check `ITempus(principal).backingToken()`

Recommendation

Do this instead:

```
        if (ITempus(principal).backingToken() != IERC20Metadata(u))
```

sourabhmarathe (Illuminate) confirmed, but disagreed with severity

kenzo (warden) commented:

> I think should probably be medium severity as user funds not at risk.

gzeoneth (judge) decreased severity to Medium

[M-05] Centralisation Risk: Admin Can Change Important Variables To Steal Funds

Submitted by kirk-baird, also found by 0xDjango, Alex the Entreprenerd, kenzo, Kumpa, pashov, shenwilly, tintin, and unforgiven

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L78

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L107

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L137

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L145

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L156

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L708

Impact

There are numerous methods that the admin could apply to rug pull the protocol and take all user funds.

- `Lender.approve()`
    - Both the functions on lines #78 and #107.

    - Admin can approve any token for an arbitrary address and transfer tokens out.
- `Lender.setFee()`
    - Does not have an lower limit.
    - `feeNominator = 1` implies 100% of amount is taken as fees.
- `Lender.withdraw()`
    - Allows withdrawing any arbitrary ERC20 token
    - 3 Days is insufficient time for users to withdraw funds in the case of a rugpull.
- `MarketPlace.setPrincipal()`
    - Use (u, m, 0) -> to be an existing Illuminate PT from another market
    - Then set (u, m, 1) -> to be some malcious admin created ERC20 token to which they have infinite supply
    - Then call `Lender.mint()` for (u, m, 1) and later redeem these tokens on the original market

Recommended Mitigation Steps

Without significant redesign it is not possible to avoid the admin being able to rug pull the protocol.

As a result the recommendation is to set all admin functions behind either a timelocked DAO or at least a timelocked multisig contract.

sourabhmarathe (Illuminate) marked as duplicate:

> Duplicate of **#390**.

gzeoneth (judge) commented:

> Input sanitization and centralization is out-of-scope in this contest, however, the arbitrary approval violated.

> The admin must not be able to withdraw more fees than what he is entitled to and fee calculation is correct.

sourabhmarathe (Illuminate) disputed and commented:

> This was not considered part of our threat model. As the remediation suggested, the `admin` address will be DAO-locked behind a multisig. As a result, we do not consider this to be an issue.

JTraversa (Illuminate) commented:

> As an additional comment, we understand that multisigs are not a solution for decentralization, but feel strongly that certain centralization is necessary for nascent lending protocols when their integration platform is so large (8 integrations, each of them integrating 3-4 money markets).

[M-06] Principal types in Illuminate and Yield lending are mixed up

Submitted by hyh

Lender's Illuminate and Yield lend() mistreats the principal type `p` requested by a user, producing another type each time: when `p == uint8(MarketPlace.Principals.Yield` the Illuminate PT is minted to the user, and vice versa, when `p == uint8(MarketPlace.Principals.Illuminate` the Yield PT is minted to the user.

Setting the severity to be high as that's a violation of system's logic, that can lead to various errors on user side, and particularly in downstream systems.

For example, any additional income can be missed as a downstream system expects Yield PT to appear and tries to stake it further somewhere, while it will be in fact not available and no staking will be done. Say with `if (amount/balance > 0) {...}` type of logic, which is correct and common, there will be no revert in such a case. Generally such types of mistakes are hard to notice in production and a silent loss of an additional yield is quite probable here.

Proof of Concept

Currently when type is Principals.Yield the Yield PT is minted to Lender contract and Illuminate PT is minted to the user, while when type is Principals.Illuminate the Yield PT is minted to the user:

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L214-L234

```
        // transfer from user to illuminate
        Safe.transferFrom(IERC20(u), msg.sender, address(this),

        if (p == uint8(MarketPlace.Principals.Yield)) {
            // Purchase yield PTs to lender.sol (address(this))
            uint256 returned = yield(u, y, a - calculateFee(a),
            // Mint and distribute equivalent illuminate PTs
            IERC5095(principalToken(u, m)).mint(msg.sender, retu

            emit Lend(p, u, m, returned);

            return returned;
        }
        else {
            // Purchase illuminate PTs directly to msg.sender
            uint256 returned = yield(u, y, a - calculateFee(a),

            emit Lend(p, u, m, returned);

            return returned;
        }
```

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L199-L202

```
        // check the principal is illuminate or yield
        if (p != uint8(MarketPlace.Principals.Illuminate) && p !
            revert Invalid('principal');
        }
```

Recommended Mitigation Steps

When `p == uint8(MarketPlace.Principals.Yield` the Yield PT to be minted to the user, while when `p == uint8(MarketPlace.Principals.Illuminate` the Yield PT to be minted to Lender contract and Illuminate PT to be minted to the user, i.e. the logic should be switched.

For example:

```
            // transfer from user to illuminate
            Safe.transferFrom(IERC20(u), msg.sender, address(this),

-           if (p == uint8(MarketPlace.Principals.Yield)) {
+           if (p == uint8(MarketPlace.Principals.Illuminate)) {
                // Purchase yield PTs to lender.sol (address(this))
                uint256 returned = yield(u, y, a - calculateFee(a),
                // Mint and distribute equivalent illuminate PTs
                IERC5095(principalToken(u, m)).mint(msg.sender, retu

                emit Lend(p, u, m, returned);

                return returned;
            }
            else {
                // Purchase illuminate PTs directly to msg.sender
                uint256 returned = yield(u, y, a - calculateFee(a),

                emit Lend(p, u, m, returned);

                return returned;
            }
```

sourabhmarathe (Illuminate) disputed and commented:

> This suggested mitigation is not accurate. Although there is a problem with Illuminate's `lend` method, the suggested mitigation does not resolve the problem. Furthermore, the suggested mitigation would result in Yield's `lend` not minting tokens to the user. We will leave it up to the judges to decide what to do with this issue.

gzeoneth (judge) commented:

> Given it is not very clear what `p` means in this function / what this function is doing and is not exploitable for profit, this seems to be an input validation / comment issue which is out-of-scope in this context. I will judge this as Low / QA because of the unclear comment.

hyh (warden) commented:

> Well, p is a type of principal and I see little ambiquity in this. Furthermore, the issue is of logic type, not comments and quite far from input validation. @gzeoneth could you please comment on the decision? That's logic violation/incompleteness, which isn't QA.

gzeoneth (judge) decreased severity to Medium and commented:

> Reviewed and adjusting to Med Risk.

[M-07] Swivel lend method doesn't pull protocol fee from user

Submitted by kenzo, also found by 0x52, 0xkowloon, cccz, Alex the Entreprenerd, hansfriese, kirk-baird, Metatron, and WatchPug

The Swivel `lend` method adds to `fees[u]` the order fee, but does not pull that fee from the user. It only pulls the order-post-fee amount.

Impact

`withdrawFee` will fail, as it tries to transfer more tokens than are in the contract.

Proof of Concept

The Swivel `lend` method sums up the fees to `totalFee`, and the amount to send to Swivel in `lent`:

```
                totalFee += fee;
                // Amount lent for this order
                uint256 amountLent = amount - fee;
                // Sum the total amount lent to Swivel (amou
```

```
lent += amountLent;
```

It then [increments](#) `fees[u]` by `totalFee`, but only pulls from the user `lent`:

```
fees[u] += totalFee;
// transfer underlying tokens from user to illuminat
Safe.transferFrom(IERC20(u), msg.sender, address(thi
```

Therefore, `totalFee` has not been pulled from the user. The `fees` variable now includes tokens which are not in the contract, and `withdrawFee` will fail as [it tries to transfer](#) `fees[u]`.

Recommended Mitigation Steps

Pull `lent + totalFee` from the user.

[sourabhmarathe (Illuminate) confirmed and commented](#):

> There were several issues that marked this ticket as a high-severity issue. Because the current code does not put user funds at risk (that otherwise would not be spent), we believe this issue is a `Med Risk` severity issue.

[M-08] Lend method signature for illuminate does not track the accumulated fee

Submitted by Kumpa, also found by cccz, cryptphi, hansfriese, jah, kenzo, kirk-baird, Metatron, pashov, and poirots

Normally the amount of fees after `calculateFee` should be added into `fees[u]` so that the admin could withdraw it through `withdrawFee`. However, illuminate ledning does not track `fees[u]`. Therefore, the only way to get fees back is through `withdraw` which admin needs to wait at least 3 days before receiving the fees.

Recommended Mitigation Steps

Add the amount of fee after each transaction into `fees[u]` like other lending method.
for example: `fees[u] += calculateFee(a);`

sourabhmarathe (Illuminate) confirmed

[M-09] `Lender.mint()` May Take The Illuminate PT As Input Which Will Transfer And Mint More Illuminate PT Cause an Infinite Supply

Submitted by kirk-baird, also found by cccz, kenzo, and unforgiven

Lender.mint() may use `p = 0` which will mean `principal` is the same as `principalToken(u, m)` i.e. the Illuminate PT. The impact is we will transfer some `principal` to the `Lender` contract and it will mint us an equivalent amount of `principal` tokens.

This can be repeated indefinitely thereby minting infinite tokens. The extra balance will be store in `Lender`.

This is rated high as although the attacker does not receive the extra tokens stored within the `Lender` they may be consumed via any contract which has been approved via the `approve()` functions (e.g. `Redeemer`).

Proof of Concept

Lender.mint()

```
    function mint(
        uint8 p,
        address u,
        uint256 m,
        uint256 a
    ) public returns (bool) {
        //use market interface to fetch the market for the giver
        address principal = IMarketPlace(marketPlace).markets(u,
        //use safe transfer lib and ERC interface...
        Safe.transferFrom(IERC20(principal), msg.sender, address
```

```
            //use ERC5095 interface...
            IERC5095(principalToken(u, m)).mint(msg.sender, a);


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


            return true;
        }
```

Steps:

- `Lender.lend()` with `p = 0` to get some Illuminate principal tokens
- `token.approve()` gives `Lender` allowance to spend these tokens
- loop:
    - `Lender.mint()` with `p = 0` minting more principal tokens

Recommended Mitigation Steps

In `Lender.mint()` ensure `p != uint8(MarketPlace.Principals.Illuminate))` .

gzeoneth (judge) decreased severity to Medium and commented:

> The 1:1 circular minting will cause a few issues as highlighted in the duplicate that impact the functionality of the protocol, but no direct fund loss.

sourabhmarathe (Illuminate) disagreed with severity and commented:

> As noted above, this will not cause major loss of funds. As a result, this should be considered a lower level issue. However, given that it could lead to problems down the line, we should address this issue.

[M-10] Easily bypassing admins 'pause' for swivel

Submitted by Metatron, also found by cccz, kenzo, and Kulk0

Assuming admin decides to pause an external principle when it's dangerous, malicious or unprofitable.

Bypassing the admins decision can result in loss of funds for the project.

Proof of Concept

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L247-L305

- The principals enum `p` is only used for `unpaused(p)` modifier, and to emit an event.
- Attacker can bypass the `unpaused(p)` modifier check by simply passing an enum of another principle that is not paused.
- The function will just continue as normal, without any other side-effect, as if the `pause` is simple ignored.

Recommended Mitigation Steps

Add this check at the beginning of the function (just like in similar functions of this solution) `if (p != uint8(MarketPlace.Principals.Swivel)) { revert Invalid('principal'); }`

sourabhmarathe (Illuminate) confirmed

[M-11] `withdraw` eToken before `withdrawFee` of eToken could render `withdrawFee` of eToken unfunctioning

Submitted by Kumpa

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L705-L720

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.s

ol#L659-L675

Vulnerability Details

`withdrawFee` of eToken requires the amount of eToken in `Lender.sol` >= `fees[eToken]` so `Safe.transfer` will not revert. However if the admin `withdraw(eToken)` first, the balance of eToken in `Lender.sol` will equal to zero while `fees[eToken]` remains the same and `withdrawFee(eToken)` will become unfunctioning since eToken in the contract does not match `fees[eToken]`. The admin will need to rely on `withdraw`, which takes 3 days before transfering, to get the future fees of eToken.

Recommended Mitigation Steps

Add `fees[eToken] = 0;` after `withdrawals[e] = 0;` in `withdraw`.

sourabhmarathe (Illuminate) confirmed and commented:

> Appears to be similar to #115 but not exactly the same.

JTraversa (Illuminate) commented:

> Yeah I'd say definitely a separate ticket / reasonable recommended remediation.

> Turns out this one should've been disputed (the only case where the emergency withdraw is called is when we are migrating contracts due to an emergency, meaning the old state actually doesnt matter at all).

JTraversa (Illuminate) commented:

> As a quick comment in our final review, the inability to use methods on an aborted contract would only matter should you want to reinitialize the same contracts.

> Given we would only abort our contracts due to vulnerabilities, there would never be such a scenario.

[M-12] Sandwich attacks are possible as there is no slippage control option in Marketplace and in Lender yield swaps

Submitted by hyh, also found by datapunk, Alex the Entreprenerd, and unforgiven

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/marketplace/MarketPlace.sol#L131-L189

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L634-L657

Vulnerability Details

Swapping function in Marketplace and Lender's yield() can be sandwiched as there is no slippage control option. Trades can happen at a manipulated price and end up receiving fewer tokens than current market price dictates.

Placing severity to be medium as those are system core operations, while funds there can be substantial, so sandwich attacks are often enough economically viable and thus probable, while they result in a partial fund loss.

Proof of Concept

All four swapping functions of Marketplace do not allow for slippage control:

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/marketplace/MarketPlace.sol#L131-L189

```
    /// @notice sells the PT for the PT via the pool
    /// @param u address of the underlying asset
    /// @param m maturity (timestamp) of the market
    /// @param a amount of PT to swap
    /// @return uint128 amount of PT bought
    function sellPrincipalToken(
        address u,
        uint256 m,
        uint128 a
    ) external returns (uint128) {
```

```
        IPool pool = IPool(pools[u][m]);
        Safe.transfer(IERC20(address(pool.fyToken())), address(p
        return pool.sellFYToken(msg.sender, pool.sellFYTokenPrev
    }

    /// @notice buys the underlying for the PT via the pool
    /// @param u address of the underlying asset
    /// @param m maturity (timestamp) of the market
    /// @param a amount of underlying tokens to sell
    /// @return uint128 amount of PT received
    function buyPrincipalToken(
        address u,
        uint256 m,
        uint128 a
    ) external returns (uint128) {
        IPool pool = IPool(pools[u][m]);
        Safe.transfer(IERC20(address(pool.base())), address(pool
        return pool.buyFYToken(msg.sender, pool.buyFYTokenPrevie
    }

    /// @notice sells the underlying for the PT via the pool
    /// @param u address of the underlying asset
    /// @param m maturity (timestamp) of the market
    /// @param a amount of underlying to swap
    /// @return uint128 amount of underlying sold
    function sellUnderlying(
        address u,
        uint256 m,
        uint128 a
    ) external returns (uint128) {
        IPool pool = IPool(pools[u][m]);
        Safe.transfer(IERC20(address(pool.base())), address(pool
        return pool.sellBase(msg.sender, pool.sellBasePreview(a)
    }

    /// @notice buys the underlying for the PT via the pool
    /// @param u address of the underlying asset
    /// @param m maturity (timestamp) of the market
    /// @param a amount of PT to swap
    /// @return uint128 amount of underlying bought
    function buyUnderlying(
        address u,
        uint256 m,
        uint128 a
    ) external returns (uint128) {
        IPool pool = IPool(pools[u][m]);
```

```
        Safe.transfer(IERC20(address(pool.fyToken())), address(p
        return pool.buyBase(msg.sender, pool.buyBasePreview(a),
    }
```

Similarly, Lender's yield does the swapping without the ability to control the slippage:

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L634-L657

```
    /// @notice transfers excess funds to yield pool after princ
    /// @dev this method is only used by the yield, illuminate a
    /// @param u address of an underlying asset
    /// @param y the yield pool to lend to
    /// @param a the amount of underlying tokens to lend
    /// @param r the receiving address for PTs
    /// @return uint256 the amount of tokens sent to the yield p
    function yield(
        address u,
        address y,
        uint256 a,
        address r
    ) internal returns (uint256) {
        // preview exact swap slippage on yield
        uint128 returned = IYield(y).sellBasePreview(Cast.u128(a

        // send the remaing amount to the given yield pool
        Safe.transfer(IERC20(u), y, a);

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

        return returned;
    }
```

Recommended Mitigation Steps

Consider adding minimum accepted return argument to the five mentioned functions and condition execution success on it so the caller can control for the realized slippage and sustain the sandwich attacks to an extent.

sourabhmarathe (Illuminate) confirmed

[M-13] Leak of Value in `yield` function, slippage check is not effective

Submitted by Alex the Entreprenerd

The function `yield` is using the input from `sellBasePreview` and then using it.

https://github.com/code-423n4/2022-06-illuminate/blob/912be2a90ded4a557f121fe565d12ec48d0c4684/lender/Lender.sol#L641-L654

```
function yield(
    address u,
    address y,
    uint256 a,
    address r
) internal returns (uint256) {
    // preview exact swap slippage on yield
    uint128 returned = IYield(y).sellBasePreview(Cast.u128(a

    // send the remaing amount to the given yield pool
    Safe.transfer(IERC20(u), y, a);

    // lend out the remaining tokens in the yield pool
    IYield(y).sellBase(r, returned);
```

The output of `sellBasePreview` is meant to be used off-chain to avoid front-running and price changes, additionally no validation is performed on this value (is it zero, is it less than 95% of amount) meaning the check is equivalent to setting `returned = 0`

I'd recommend to add checks, or ideally have a trusted keeper bulk `sellBase` with an additional slippage check as the function parameter

sourabhmarathe (Illuminate) confirmed

Low Risk and Non-Critical Issues

For this contest, 62 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **defsec** received the top score from the judge.

The following wardens also submitted reports: lllllll, hyh, Alex the Entreprenerd, robee, joestakey, 0x29A, Lambda, oyc_109, saian, shenwilly, StErMi, TomJ, zer0dot, BowTiedWardens, 0x1f8b, datapunk, ElKu, fatherOfBlocks, hansfriese, MadWookie, Picodes, poirots, sashik_eth, simon135, Waze, kenzo, _Adam, 0xDjango, 0xf15ers, 0xkowloon, 0xmint, 0xNazgul, 0xNineDec, ak1, asutorufos, aysha, bardamu, catchup, Chom, delfin454000, dipp, Funen, GimelSec, grGred, hake, kebabsec, Kenshin, kirk-baird, Kulk0, Limbooo, pashov, rfa, Soosh, WatchPug, zeesaw, BnkeOx0, JC, Metatron, slywaters, Yiko, *and* z3s.

Issue List

- 1: Missing events for only functions that change critical parameters
- 2: Critical changes should use two-step procedure
- 3 Pragma Version
- 4: Missing zero-address check in the setter functions and initiliazers
- 5: Low level calls with solidity version 0.8.14 can result in optimiser bug.
- 6: The Contract Should safeApprove(0) first.
- 7: Use of Block.timestamp
- 8: Incompatibility With Rebasing/Deflationary/Inflationary tokens
- 9: Lack of setter function for the apwineAddr

[1] Missing events for only functions that change critical parameters

The functions that change critical parameters should emit events. Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services.

The alternative of directly querying on-chain contract state for such changes is not considered practical for most users/usages.

Missing events and timelocks do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in liquidity which could negatively impact protocol TVL and reputation.

Proof of Concept

Navigate to the following contracts:

https://github.com/code-423n4/2022-06-illuminate/blob/main/marketplace/MarketPlace.sol#L109

https://github.com/code-423n4/2022-06-illuminate/blob/main/marketplace/MarketPlace.sol#L98

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L129

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L137

See similar High-severity H03 finding OpenZeppelin's Audit of Audius (https://blog.openzeppelin.com/audius-contracts-audit/#high) and Medium-severity M01 finding OpenZeppelin's Audit of UMA Phase 4 (https://blog.openzeppelin.com/uma-audit-phase-4/)

Recommended Mitigation Steps

Add events to all functions that change critical parameters.

[2] Critical changes should use two-step procedure

The critical procedures should be two step process.

Proof of Concept

1. Navigate to the following contracts:

https://github.com/code-423n4/2022-06-illuminate/blob/main/marketplace/MarketPlace.sol#L109

https://github.com/code-423n4/2022-06-illuminate/blob/main/marketplace/MarketPlace.sol#L98

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L129

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L137

Recommended Mitigation Steps

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two step procedure on the critical functions.

[3] Pragma Version

In the contracts, floating pragmas should not be used. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Proof of Concept

https://swcregistry.io/docs/SWC-103

All Contracts

Recommended Mitigation Steps

Lock the pragma version: delete pragma solidity 0.8.15 in favor of pragma solidity 0.8.15.

[4] Missing zero-address check in the setter functions and initiliazers

Missing checks for zero-addresses may lead to infunctional protocol, if the variable addresses are updated incorrectly.

Proof of Concept

Navigate to the following contracts.

https://github.com/code-423n4/2022-06-illuminate/blob/main/marketplace/MarketPlace.sol#L109

https://github.com/code-423n4/2022-06-illuminate/blob/main/marketplace/MarketPlace.sol#L98

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L129

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L137

Recommended Mitigation Steps

Consider adding zero-address checks in the discussed constructors: require(newAddr != address(0));.

[5] Low level calls with solidity version 0.8.14 can result in optimiser bug.

The protocol is using low level calls with solidity version 0.8.14 which can result in optimizer bug.

https://medium.com/certora/overly-optimistic-optimizer-certora-bug-disclosure-2101e3f7994d

Recommended Mitigation Steps

Consider upgrading to solidity 0.8.15.

[6] The Contract Should safeApprove(0) first

Some tokens (like USDT L199) do not work when changing the allowance from an existing non-zero allowance value. They must first be approved by zero and then the actual allowance must be approved.

```
IERC20(token).safeApprove(address(operator), 0);
IERC20(token).safeApprove(address(operator), amount);
```

When trying to re-approve an already approved token, all transactions revert and the protocol cannot be used.

Proof of Concept

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L93

Recommended Mitigation Steps

Approve with a zero amount first before setting the actual amount.

[7] Use of Block.timestamp

Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion for further details), locking funds for periods of time, and various state-changing conditional statements that are time-dependent. Miners have the ability to adjust timestamps slightly, which can prove to be dangerous if block timestamps are used incorrectly in smart contracts.

Proof of Concept

Navigate to the following contract.

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L688

Recommended Mitigation Steps

Block timestamps should not be used for entropy or generating random numbers—i.e., they should not be the deciding factor (either directly or through some derivation) for winning a game or changing an important state.

Time-sensitive logic is sometimes required; e.g., for unlocking contracts (time-locking), completing an ICO after a few weeks, or enforcing expiry dates. It is sometimes recommended to use block.number and an average block time to estimate times; with a 10 second block time, 1 week equates to approximately, 60480 blocks. Thus, specifying a block number at which to change a contract state can be more secure, as miners are unable to easily manipulate the block number.

[8] Incompatibility With Rebasing/Deflationary/Inflationary tokens

PrePo protocol do not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest.

Example Test

During the lending, If the inflationary/deflationary tokens are used excepted amount will be lower than deposit.

Proof of Concept

Navigate to the following contract.

https://github.com/code-423n4/2022-06-illuminate/blob/main/lender/Lender.sol#L215

Recommended Mitigation Steps

- Ensure that to check previous balance/after balance equals to amount for any rebasing/inflation/deflation
- Add support in contracts for such tokens before accepting user-supplied tokens
- Consider supporting deflationary / rebasing / etc tokens by extra checking the balances before/after or strictly inform your users not to use such tokens if they don't want to lose them.

[9] Lack of setter function for the apwineAddr

On the Redemeer contract, there is not setter function on the apwineAddr address. This can cause misfunctionality on the redeemer contract.

Proof of Concept

1. Navigate to contract: https://github.com/code-423n4/2022-06-illuminate/blob/main/redeemer/Redeemer.sol#L33
2. apwineAddr address is set on the constructor.
3. Setter function is missing on the contract. Misdeployed contract can cause failure of apwineAddr integration.

Recommended Mitigation Steps

Consider adding setter function for apwineAddr.

Gas Optimizations

For this contest, 56 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **BowTiedWardens** received the top score from the judge.

The following wardens also submitted reports: lllllll, defsec, joestakey, 0xkatana, 0xKitsune, _Adam, ajtra, Bnke0x0, catchup, Alex the Entreprenerd, grGred, ladboy233, pashov, robee, sashik_eth, Tomio, TomJ, Waze, z3s, 0x1f8b, delfin454000, fatherOfBlocks, hansfriese, JC, Kaiziron, Noah3o6, oyc_109, slywaters, UnusualTurtle, 0v3rf10w, 0x29A, 0xf15ers, 0xkowloon, 0xNazgul, asutorufos, bardamu, c3phas, datapunk, ElKu, Fitraldys, Funen, hake, hyh, ignacio, kebabsec, Lambda, MadWookie, Nyamcil, poirots, rfa, RoiEvenHaim, sach1r0, samruna, simon135, *and* zer0dot.

Overview

Risk Rating	Number of issues	
Gas Issues	9	

Table of Contents

- G-01. Unchecking arithmetics operations that can't underflow/overflow
- G-02. Caching storage values in memory
- G-03. Cheap Contract Deployment Through Clones
- G-04. Use `calldata` instead of `memory`
- G-05. `<array>.length` should not be looked up in every loop of a `for-loop`
- G-06. `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)
- G-07. It costs more gas to initialize variables with their default value than letting the default value be applied]
- G-08. Some variables should be immutable
- G-09. Use Custom Errors instead of Revert Strings to save Gas

[G-01] Unchecking arithmetics operations that can't underflow/overflow

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an `unchecked` block:
https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

By keeping in mind the following function (calculateFee) which makes it so that `a - calculateFee(a) > 0`:

```
File: Lender.sol
661:     function calculateFee(uint256 a) internal view returns
662:         return feenominator > 0 ? a / feenominator : 0;
663:     }
```

Consider wrapping with an `unchecked` block here:

- File: Lender.sol

```
- 219:              uint256 returned = yield(u, y, a - calculateF
+ 219:              unchecked { uint256 returned = yield(u, y, a
```

```
- 229:             uint256 returned = yield(u, y, a - calculateFe
+ 229:             unchecked { uint256 returned = yield(u, y, a -
```

```
- 400:         uint256 returned = IPendle(pendleAddr).swapExactTo
+ 400:       unchecked { uint256 returned = IPendle(pendleAddr).
```

```
- 502:         uint256 lent = a - fee;
+ 502:         unchecked { uint256 lent = a - fee; }
```

```
- 557:         uint256 lent = a - fee;
+ 557:         unchecked { uint256 lent = a - fee; }
```

```
- 605:         uint256 returned = token.deposit(a - fee, address(
+ 605:         unchecked { uint256 returned = token.deposit(a - f
```

[G-02] Caching storage values in memory

The code can be optimized by minimizing the number of SLOADs.

SLOADs are expensive (100 gas after the 1st one) compared to MLOADs/MSTOREs (3 gas each). Storage values read multiple times should instead be cached in memory the first time (costing 1 SLOAD) and then read from this cache to avoid multiple SLOADs.

- `feenominator` : https://github.com/code-423n4/2022-06-illuminate/blob/92cbb0724e594ce025d6b6ed050d3548a38c264b/lender/Lender.sol#L662

```
File: Lender.sol
661:     function calculateFee(uint256 a) internal view returns
```

```
662:         return feenominator > 0 ? a / feenominator : 0;
663:     }
```

- `lender` : https://github.com/code-423n4/2022-06-illuminate/blob/92cbb0724e594ce025d6b6ed050d3548a38c264b/redeemer/Redeemer.sol#L134-L136

```
File: Redeemer.sol
134:             uint256 amount = IERC20(principal).balanceOf(le
135:             // Transfer the principal token from the lender
136:             Safe.transferFrom(IERC20(u), lender, address(th
```

- `lender` : https://github.com/code-423n4/2022-06-illuminate/blob/92cbb0724e594ce025d6b6ed050d3548a38c264b/redeemer/Redeemer.sol#L177-L180

```
File: Redeemer.sol
177:         uint256 amount = IERC20(principal).balanceOf(lender
178:
179:         // Transfer the principal token from the lender cor
180:         Safe.transferFrom(IERC20(principal), lender, addres
```

- `marketPlace` : https://github.com/code-423n4/2022-06-illuminate/blob/92cbb0724e594ce025d6b6ed050d3548a38c264b/redeemer/Redeemer.sol#L164-L187

```
File: Redeemer.sol
164:         address principal = IMarketPlace(marketPlace).marke
...
187:             IElementToken(principal).withdrawPrincipal(amou
```

- `lender` : https://github.com/code-423n4/2022-06-illuminate/blob/92cbb0724e594ce025d6b6ed050d3548a38c264b/redeemer/Redeemer.sol#L221-L224

```
File: Redeemer.sol
221:         uint256 amount = token.balanceOf(lender);
222:
223:         // Transfer the user's tokens to the redeem contrac
224:         Safe.transferFrom(token, lender, address(this), amo
```

- `lender` : https://github.com/code-423n4/2022-06-illuminate/blob/92cbb0724e594ce025d6b6ed050d3548a38c264b/redeemer/Redeemer.sol#L256-L259

```
File: Redeemer.sol
256:         uint256 amount = token.balanceOf(lender);
257:
258:         // Transfer the user's tokens to the redeem contrac
259:         Safe.transferFrom(token, lender, address(this), amo
```

[G-03] Cheap Contract Deployment Through Clones

```
marketplace/MarketPlace.sol:80:         address iToken = address(
```

There's a way to save a significant amount of gas on deployment using Clones: https://www.youtube.com/watch?v=3Mw-pMmJ7TA .

This is a solution that was adopted, as an example, by Porter Finance. They realized that deploying using clones was 10x cheaper:

- https://github.com/porter-finance/v1-core/issues/15#issuecomment-1035639516
- https://github.com/porter-finance/v1-core/pull/34

Consider applying a similar pattern.

[G-04] Use `calldata` instead of `memory`

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. Each iteration of this for-loop costs at least 60 gas (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Structs have the same overhead as an array of length one

When arguments are read-only on external functions, the data location should be `calldata`:

```
marketplace/MarketPlace.sol:70:        address[8] memory t,
```

[G-05] `<array>.length` should not be looked up in every loop of a `for-loop`

Reading array length at each iteration of the loop consumes more gas than necessary.

In the best case scenario (length read on a memory variable), caching the array length in the stack saves around 3 gas per iteration. In the worst case scenario (external calls at each iteration), the amount of gas wasted can be massive.

Here, Consider storing the array's length in a variable before the for-loop, and use this new variable instead:

```
lender/Lender.sol:265:              for (uint256 i = 0; i < o.leng
```

[G-06] `++i` costs less gas compared to `i++` or `i += 1` (same for `--i` vs `i--` or `i -= 1`)

Pre-increments and pre-decrements are cheaper.

For a `uint256 i` variable, the following is true with the Optimizer enabled at 10k:

Increment:

- `i += 1` is the most expensive form
- `i++` costs 6 gas less than `i += 1`
- `++i` costs 5 gas less than `i++` (11 gas less than `i += 1`)

Decrement:

- `i -= 1` is the most expensive form
- `i--` costs 11 gas less than `i -= 1`
- `--i` costs 5 gas less than `i--` (16 gas less than `i -= 1`)

Note that post-increments (or post-decrements) return the old value before incrementing or decrementing, hence the name *post-increment*:

```
uint i = 1;
uint j = 2;
require(j == i++, "This will be false as i is incremented after
```

However, pre-increments (or pre-decrements) return the new value:

```
uint i = 1;
uint j = 2;
require(j == ++i, "This will be true as i is incremented before
```

In the pre-increment case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`.

Affected code:

```
lender/Lender.sol:96:                    i++;
lender/Lender.sol:120:                    i++;
lender/Lender.sol:283:                        i++;
```

Consider using pre-increments and pre-decrements where they are relevant (meaning: not where post-increments/decrements logic are relevant).

[G-07] It costs more gas to initialize variables with their default value than letting the default value be applied

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Affected code:

```
lender/Lender.sol:265:             for (uint256 i = 0; i < o.leng
```

Consider removing explicit initializations for default values.

[G-08] Some variables should be immutable

These variables are only set in the constructor and are never edited after that:

```
lender/Lender.sol:26:    address public admin;
lender/Lender.sol:28:    address public marketPlace;
lender/Lender.sol:33:    address public swivelAddr;
marketplace/MarketPlace.sol:41:    address public admin;
redeemer/Redeemer.sol:19:    address public admin;
redeemer/Redeemer.sol:21:    address public marketPlace;
redeemer/Redeemer.sol:23:    address public lender;
redeemer/Redeemer.sol:27:    address public swivelAddr;
redeemer/Redeemer.sol:33:    address public apwineAddr;
```

Consider marking them as immutable, as it would avoid the expensive storage-writing operation (around 20 000 gas)

[G-09] Use Custom Errors instead of Revert Strings to save Gas

As this is the case in almost the whole solution, consider also using custom errors here:

```
lender/Lender.sol:691:          require (when != 0, 'no withdrawal
lender/Lender.sol:693:          require (block.timestamp >= when,
```

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top