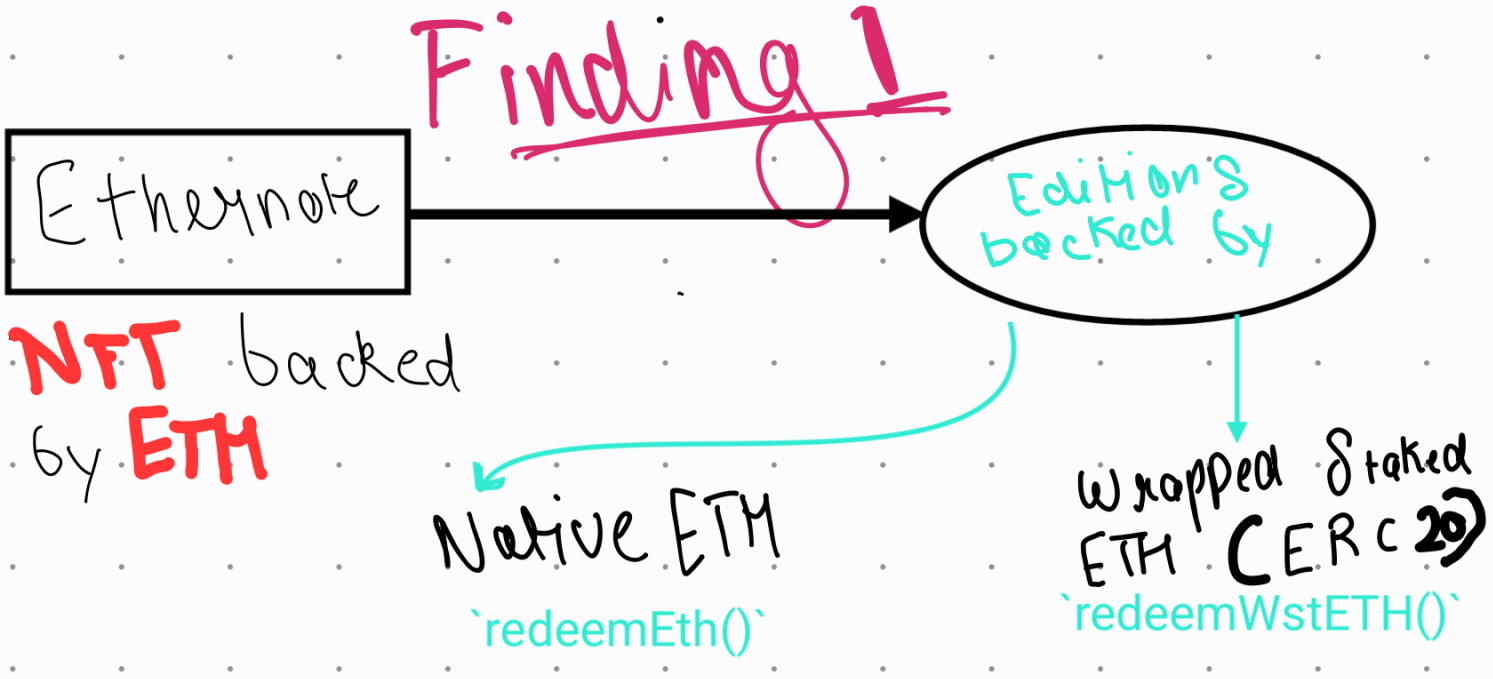
Description: Important findings from gaurdian Audits




```
182
183     function _redeemEth(uint256 _tokenId) internal {
184         safeTransferFrom(msg.sender, address(this), _tokenId);
185         _burn(_tokenId);
186         circulatingNotes--;
187         uint256 redeemFee = (metadata[_tokenId].balance * notes[metadata[_tokenId].note].redeemFee) / 1e4;
188         uint256 amount =  metadata[_tokenId].balance - redeemFee;
189         fees += redeemFee;
190         metadata[_tokenId].balance = 0;
191         (bool success, ) = msg.sender.call{value: amount}(" ");
192         require(success, " ");
193         emit NoteRedeemed(_tokenId, msg.sender, amount);
194     }
195
196     function _redeemWstEth(uint256 _tokenId) internal {
197         safeTransferFrom(msg.sender, address(this), _tokenId);
198         uint256 redeemFee = metadata[_tokenId].balance * notes[metadata[_tokenId].note].redeemFee / 1e4;
199         uint256 amount = metadata[_tokenId].balance - redeemFee;
200         fees += redeemFee;
201         metadata[_tokenId].balance = 0;
202         _burn(_tokenId);
203         circulatingNotes--;
204         IwstETH(wstETH).transfer(msg.sender, amount);
205         emit NoteRedeemed(_tokenId, msg.sender, amount);
206     }
207
208     function redeem(uint256 _tokenId) nonReentrant external {
209         require(_isApprovedOrOwner(msg.sender, _tokenId), "Not authorized");
210         if(editions[metadata[_tokenId].edition].ethereum) {
211             _redeemEth(_tokenId);
212         } else {
213             _redeemWstEth(_tokenId);
214         }
215     }
```

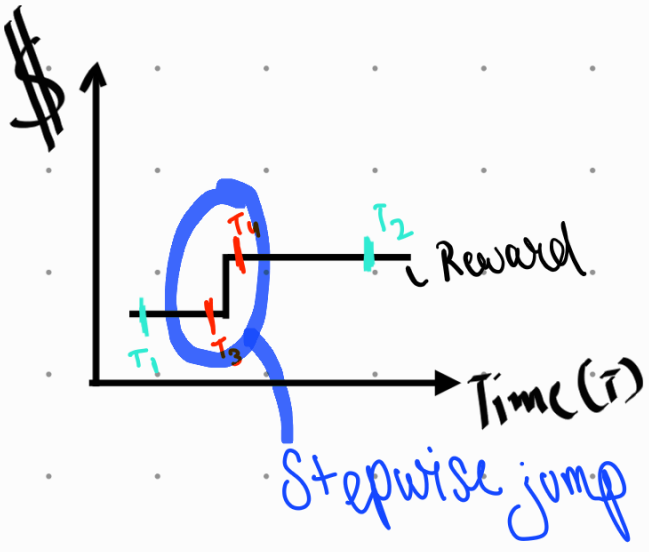

This contract does not account for difference in value of ETH balance Vs WstETH balance & Fees is only charged in ETH So If this Contract runs out of ETH `redeemWstETH()`

will revert and funds will be stuck.

Finding 2

Key Finance

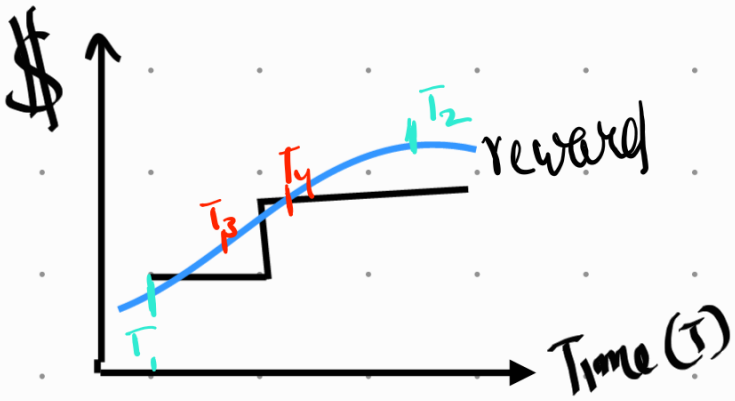
Sandwichable Rewards



- $T_2 - T_1 > T_4 - T_3$

But reward is the same due to sudden movement in price

- Reward for staking is calculated in steps which makes sandwich possible
- Attacker can enter at T_3 with huge stake and exit at T_4 stealing profits



- Continuos reward tracking ensures that such MEV attacks are mitigated.

Finding 3

```
function distributeDividends(uint amount) public payable lock{
    require(amount == msg.value,"don't cheat");
    uint length = users.length;
    amount = amount.mul(magnitude);
    for (uint i; i < length; ++i) {
        if (users[i] != address(0)){
            UserInfo storage user = userInfo[users[i]];
            user.rewards += (amount.mul(IERC20(address(this)).balanceOf(users[i])).div(totalSupply.sub(MINIMUM_LIQUIDITY)));
        }
    }
}
```

Unbound array

For Loop on unbound array can lead to **OOG DoS**.

The logic for adding new users to users array makes it possible to cheaply fill array to have large amount of addresses making it possible to **DoS** this function.

> Any kind of loop on Unbounded array will almost always lead to D.O.S.

Mitigation.

- Points per Share implementation.
  - Instead of updating rewards for each user, update points per token

Eg: 1 Token = 10 points

investor with 10 tokens get 100 pts.

then 1 token = 25 points

investor with 10 tokens get

(10 * (25 - 10)) = 150 pts.

Balance * (div - position div per)

Finding 4

```
function swapMargin(address _assetIn, address _assetOut, uint256 _amountIn, uint256 _amountOut)
    external
    nonReentrant
    onlyMarginUser
{
    //require asset in and asset out is supported
    if (RiskEngine.isSupportedAsset(_assetIn) == false) {
        revert AssetNotSupported(_assetIn);
    }
    if (RiskEngine.isSupportedAsset(_assetOut) == false) {
        revert AssetNotSupported(_assetOut);
    }

    require(_amountIn <= userMarginBalance(_assetIn), "cannot swap more than the balance");

    ERC20(_assetIn).safeApprove(address(Exchange), _amountIn);
    uint256 amountOut = Exchange.swapOnUniswap(_assetIn, _assetOut, _amountIn, _amountOut, address(this));

    emit MarginSwapped(msg.sender, _assetIn, _assetOut, _amountIn, amountOut);
}
```

User Controlled

No validation on wether user can liquidate themselves

EXPLOIT :

→ make a position with 0 `_amountOut` uniswap pool will now have some price difference.

→ Frontrun yourself & extract that value

→ repeat until approximately all collateral has been extracted.

Mitigation :

add a check ... to the ... liquidate themselves.

Finding 5

GMX

Perpetuals Protocol

Order execution

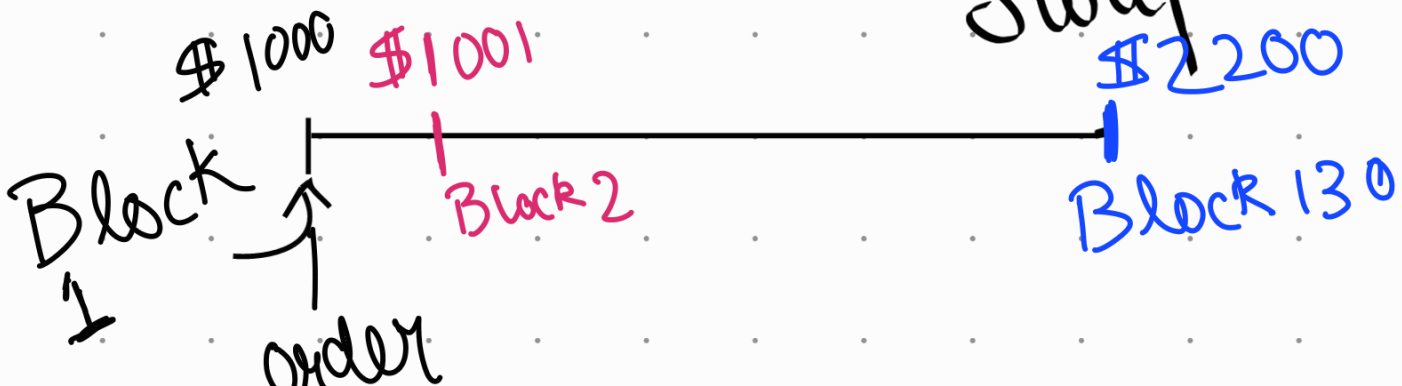
Whenever an order is registered with the keeper it is **executed with whatever asset price was at that block.**

SwapPath

the amount of swaps to carry out

[ETH-DAI, DAI-USDC, USDC-WETH, ...]

In GMX there was **no upper limit** in amount of swaps.



(increase position)

- At block1 attacker creates order for a long position at $1000.
- Order will always revert as it was created with 'swapPath' so long such that tx will fall outside block gas limit just
- At block 130 when prices are high and attacker will make a good Profit, attacker uses controlled callback contract to expend less gas such that Tx falls back in block gas limit and executes.

> Huge risk free profits for attacker losses for LPs.

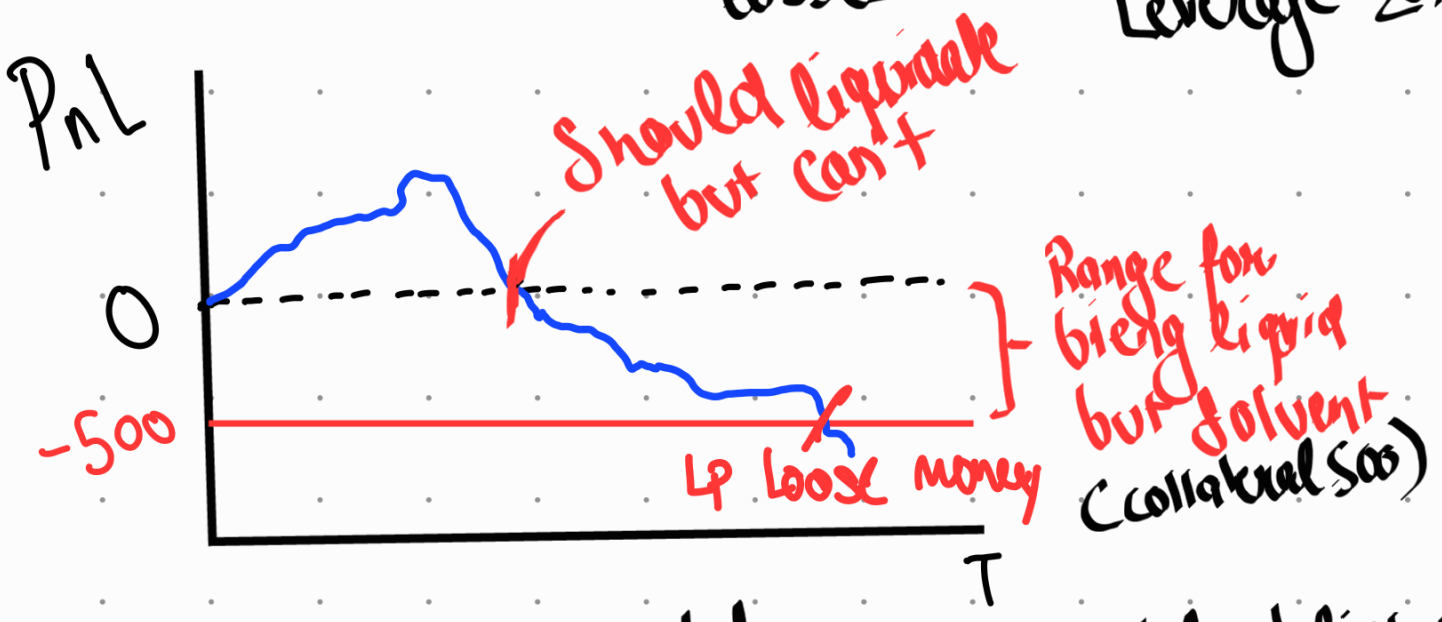
Finding 6

> BlackListed Addresses.

liquidation Case.

Collateral = 500
Size : 10k
Leverage 20x

Solvent = can I cover losses



If a trader address is blacklisted by the token contract then it won't liquidate even when should and when position collateral is exhausted LPs suffer Losses.

Finding 7