Alixa Protocol Security Review

Reviewed by: MrPotatoMagic

August 8th, 2024

Contents

1 About MrPotatoMagic

MrPotatoMagic is an independent smart contract security researcher. Having disclosed over 100 security vulnerabilities across numerous protocol types, he consistently aims to elevate the security of the blockchain ecosystem by offering top quality smart contract security reviews.

2 Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100 percent security after the review or even if the review will find any problems with your smart contracts. Subsequent testing, security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3 Introduction

Alixa protocol has forged a strategic alliance with Ether.fi/EigenLayer by leveraging their eETH token to generate a consistent revenue stream. With immutablity, zero fees and permissionless staking as some of the many features embedded on the core level, it ensures a fair and equitable experience by avoiding insider allocations and preferential treatment. It also plays a pivotal and catalytic role in fuelling up the Xenify ecosystem to enable sustained growth of both protocols.

4 Risk Classification

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

4.1 Impact

- High - Leads to significant material loss of assets in the protocol or significant harm to majority of users.
- Medium - Loss/Leakage of small asset amounts or core functionality of the protocol is affected.
- Low - Any kind of unexpected behaviour that is non-critical.

4.2 Likelihood

- High - Almost certain to happen, easy to perform, or not easy but highly incentivized.
- Medium - Only conditionally possible or incentivized, but still relatively likely.
- Low - Involves too many or unlikely assumptions with little-to-no incentive.

4.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

5 Executive Summary

Over the course of the security review, Alixa engaged with MrPotatoMagic to review the alixa-contracts protocol. In this period of time, a total of 35 issues were found.

Summary

Project Name	Alixa
Repository	alixa-contracts
Commit	2e0736f...de6a
Protocol Type	Yield, DeFi
Review Timeline	July 3rd 2024 - July 21st 2024
One week fix period	July 21st 2024

Scope

- ALXStake.sol
- Core.sol
- Getter.sol
- Reserve.sol
- Stake.sol
- LibALXStake.sol
- LibClaimAndSnipe.sol
- LibCommon.sol
- LibCore.sol
- LibReserve.sol
- LibStake.sol
- LibStorage.sol
- ALX.sol
- ETHX.sol
- POL.sol
- SwapHandler.sol

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	**3**	**3**	**0**
High Risk	**5**	**5**	**0**
Medium Risk	**5**	**5**	**0**
Low Risk	**10**	**7**	**3**
Non-Critical	**3**	**3**	**0**
Gas Optimizations	**9**	**9**	**0**
Total	**35**	**32**	**3**

6 Findings

6.1 Critical Risk

6.1.1 Function initialize() can be re-initialized in SwapHandler.sol

Context: SwapHandler.sol#L124
Impact: High
Likelihood: High

Description: Once the function initialize() is called during deployment, it can be called again by a malicious actor. This occurs since the function is external and does not act like a constructor by implementing a one-time initialization check.

Due to this, an attacker could change the state variables to affect the core functionality of the protocol. For example, the attacker could change the swapRouter variable to a malicious contract, which would use up the approval that was made to it for the swap. Due to this, all calls routing from functions burn() and burnPrivilege() in ETHx.sol to SwapHandler.sol would lead to all user tokens to be stolen.

```
function initialize(
      uint24 _fee,
      address _eETH,
      address _weETH,
      address _WETH,
      address _swapRouter
  ) external {
      fee = _fee;
      swapRouter = ISwapRouter02(_swapRouter);
      eETH = _eETH;
      WETH = _WETH;
      weETH = _weETH;
  }
```

Recommendation: Implement a one-time initialization check to ensure the function initialize() can be invoked at most once.

6.1.2 Underflow in function utilisePositiveRebasing() due to ETHx over-minting

Context: ETHx.sol#L409
Impact: High
Likelihood: High

Description: The ETHx token is backed by EtherFi's positive rebasing eETH token. This is facilitated through the mint() function that takes in users' ETH, deposits it to EtherFi's LiquidityPool.sol contract to receive eETH in return and finally mints the ETHx tokens to the user.

```
function mint(
    uint256 _amount,
    address _token
)
    external
    payable
    returns (uint256)
{
    IERC20 _EETH = EETH;
    IEETHZapper _eethZapper = eethZapper;
    if (_token != address(_EETH)) {
        _amount = _eethZapper.deposit{value: msg.value}(team);
        _amount = _eethZapper.amountForShare(_amount);
    } else {...};
    }
    _mint(msg.sender, _amount);
    return _amount;
}
```

The issue occurs in the if conditional block where the _amount variable tracked for ETHx minting is higher than the eETH received from the LiquidityPool contract. This is due to how the liquidity pool's deposit() logic works.

Let's say a user calls the mint() function with 1e18 as _amount parameter and address(0) as the _token parameter (i.e. ETH). Due to this, we enter the if block and the deposit() function is called on the Liquidity Pool contract. If we follow the execution path, we arrive on the _sharesForDepositAmount() function which performs the calculations to mint eETH to the msg.sender (i.e. the ETHx contract).

```
function _sharesForDepositAmount(uint256 _depositAmount) internal
    view returns (uint256) {
    uint256 totalPooledEther = getTotalPooledEther() - _depositAmount;
    if (totalPooledEther == 0) {
        return _depositAmount;
    }
    return (_depositAmount * eETH.totalShares()) / totalPooledEther;
}
```

Based on data from the eETH and LiquidityPool onchain contracts, the calculations occur as follows:

$$\frac{10^{18} \times 1748272584969505593923365}{1823891445137707579031072} \approx 958539824083394061.5 \text{ wei} \approx 0.958 \text{ ether} \quad (1)$$

This is the eETH amount received by the contract on deposit of 1e18 tokens. Following this, the amountForShare() function is called on the Liquidity Pool contract from the ETHx.sol contract. This uses the 0.958 ether as input to calculate the ETHx to mint to the user. Here are the calculations:

$$\frac{958539824083394061.5 \times (1823891445137707579031072 + 1e18)}{(1748272584969505593923365 + 958539824083394061.5)} \approx 1e18 \text{ wei} \approx 1 \text{ ether} \quad (2)$$

Through this, we can see how the contract receives 0.958 eETH but the ETHx minted was 1 ETHx. Due to this, the user receives more ETHx, which during burning through function burn(), would tap into eETH deposited by other users. Another impact of ETHx over-minting is that the call from ETHxReserve() in Reserve.sol to utilize positive rebase will fail due to which the ETHx allocated to the ETHxReserveShare will remain locked in the contract. This occurs since the eETH balance of the contract is lesser than the ETHxSupply, which causes the underflow.

Recommendation: In the mint() function, consider checking the balance before and after the deposit() function call occurs. Use the difference to mint ETHx tokens to the user.

6.1.3 buybackByTeam() only allows decreasing liquidity and collecting fees from ETHx-vXNF pool and not ALX-vXNF

Context: Reserve.sol#L725
Impact: High
Likelihood: High

Description: The POL contract mints ETHx-vXNF and ALX-vXNF position NFTs to the Core contract. The liquidity from these positions can be decreased by the team through the function buybackByTeam() in the Reserve.sol contract. The liquidity collected is converted into ALX to either reward ALX stakers or to burn it as part of the ALX buyback and burn mechanism.

The issue is that currently the function only supports decreasing liquidity from the ETHx-vXNF position. Due to this, the tokens locked in the ALX-vXNF position remain permanently inaccessible to the team to either reward ALX stakers or to perform buyback and burn. As per the conditions in POL.sol, since the amount of tokens locked in the ALX-vXNF position would be a minimum of 25000 ALX and 25000 vXNF, the severity is critical.

Recommendation: Consider implementing support for managing the ALX-vXNF position either through a separate function or in the buybackByTeam() function itself.

6.2 High Risk

6.2.1 ALX sniper with deactivated bond can continue sniping ALX stakers risk-free

Context: LibALXStake.sol#L175-L204
Impact: Medium
Likelihood: High

Description: There is a specific edge case in LibALXStake.sol which allows deactivated ALX bond users to continue sniping.

If we observe the code block here in function _claimLockedStakingRewards(), we can see that it checks if bond exists and whether the bond is an ALX bond or not to ensure the sniper is valid. In the if block here, we can see that if it has been 7 days since user stake ended, the sniper can claim rewards and user's ALX is burnt. Now if the sniper is deactivated, there is no way they can profit from this if block since ETHxMaxPayout - claimedETHx would be 0, so no rewards are transferred and it would simply be a waste of gas.

But if we check the else block here, which is callable when it has not been 7 days since staker's lock end, the rewards are transferred to the msg.sender i.e. sniper even though the sniper's ALX bond is deactivated. Due to this, the sniper can snipe risk free in the protocol without any active bond at stake. This is a high-severity issue since the likelihood of entering the else block is high (i.e. in most cases block.timestamp won't be 7 days since staker's lock end) and the impact is medium due to the snipers only requiring one bond ever. Due to this, users are not incentivized to buy any more ALX bonds to snipe, thus affecting the core functionality and mechanics of the protocol.

A related bug with a different root cause that allows a deactivated sniper to snipe is that this check here should use <= instead of <. This is important since if the sniper is deactivated i.e. their claimedETHx was set to ETHxMaxPayout last time out, the condition would evaluate as ETHxMaxPayout < claimedETHx = false since claimedETHx is ETHxMaxPayout and ETHxMaxPayout < ETHxMaxPayout = false. Thus, we would enter the else block always.

Recommendation: Check here if ALX bond of sniper is deactivated or not. This would solve both issues pointed above though it would be better to also set < to <=.

6.2.2 Missing claimedETHx update allows sniper to exceed ETHxMaxPayout

Context: LibALXStake.sol#L198-L204
Impact: Medium
Likelihood: High

Description: The else block here is entered most of the time i.e. when it has not been 7 days since staker's stake lock end. The issue is that the ETHx is transferred but not accounted in the claimedETHx of the sniper.

Due to this, a sniper's claimedETHx never increases, thus allowing to exceed the max limit allowed. Since there is no limit upto which the sniper can earn and the sniper only needs one ALX bond (the smallest one), the ALX bond will never be deactivated. Due to this, users are not incentivized to buy any more ALX bonds to snipe, thus affecting the core functionality and mechanics of the protocol.

```
            } else {
                if (amountALX != 0) {
                    s.ALX.transfer(recipient, amountALX);
                }
                s.ETHx.transfer(msg.sender, amountETHx / 5);
                s.ETHxReserveShare += amountETHx / 20;
            }
```

Recommendation: Update sniper's claimedETHx in the else block.

6.2.3 Users with 0.0199 ETHx or lesser will not be able burn their ETHx to receive their EETH or ETH back

Context: ETHx.sol#L318-L337
Impact: Medium
Likelihood: High

Description: Users provide ETH or eETH and receive ETHx tokens in return through the mint() function. The burn() function allows users to burn ETHx and receive an ETH bond + the remaining eETH or ETH amount.

The issue with the current implementation of the burn() function is that if the ETHx amount / 2 is less than the base cost of 1e16 or 0.01 ether, we revert. This is a problem since users with ETHx amount = 0.0199 or less will never be able to burn their tokens and receive ETH or eETH back.

Based on the code here in the if-else blocks, 0.0199 / 2 < 0.01 = true, we revert.

```
        if (_desiredOutToken != address(_EETH)) {
            ...
            if (_amount >> 1 < baseCost) {
                revert LessThenMinBondCount();
            }
            ...
        } else {
            if (_amount >> 1 < baseCost) {
                revert LessThenMinBondCount();
            }
```

To evaluate how much 0.0199 ETHx would be in terms of USD, let's so some basic math. We know ETHx is backed by EETH, so the USD value of ETHx increases as EETH positively rebases. We also know that EETH is backed by ETH and when we provide let's say 1 ETH, we currently receive 0.95 shares of EETH approximately, meaning that EETH is valued more than ETH in terms of USD.

Using current ETH price as a baseline i.e. $3390 per ETH, 0.0199 ETH would be 68 USD approximately. Through this we can deduce that users trying to burn 0.0199 ETHx, which is worth 70-80 USD minimum or more (due to EETH being valued more), will not be able to receive that much amount back (atleast through the protocol's ETHx.sol contract).

Recommendation: When _amount / 2 is not greater than or equal to the base cost, consider just sending back the whole EETH or ETH amount.

6.2.4 Early stakeStart and duration updates in extendLockedStake() leads to incorrect loyalty points calculation

Context: ALXStake.sol#L275-L288
Impact: Medium
Likelihood: High

Description: In the extendLockedStake() function, the code block over here is executed when the lock has expired past the duration and is being extended. The issue is that on Line 284, when we make the conditional check **lockedStaker.stakeStart + lockedStaker.duration + 7 days < block.timestamp**, the stakeStart and duration variables used here are not the old values of the lock but the new ones which are updated on Lines 279-280.

```
    if (lockedStaker.stakeStart + lockedStaker.duration <= block.timestamp) {
            ...
            lockedStaker.stakeStart = block.timestamp;
            lockedStaker.duration = _duration;
            ...
            if (lockedStaker.stakeStart + lockedStaker.duration + 7 days
               < block.timestamp) {
                lockedStaker.accLoyaltyPointsBonus = 0;
            } else {
                s.totalLoyaltyPoints += lockedStaker.accLoyaltyPointsBonus;
            }
```

Due to this, the condition would always evaluate to false since **block.timestamp + 30 days (as duration example) + 7 days < block.timestamp** is false. Due to this, we always account the accumulated loyalty points bonus of the user in the totalLoyaltyPoints, which causes ETHx rewards dilution in LibALXStake.sol.

Recommendation: Update the stakeStart and duration state variables after the conditional check in order to use the old values and not the new updated ones.

6.2.5 Potential double claiming of ETHx rewards due to missing state changes in ETHxReserve()

Context: Reserve.sol#L257-L263
Impact: Medium
Likelihood: High

Description: When ETHxReserve() is called, we claim the users pending ETHx and the 10% incentive. When ETHx is claimed, these are the only state changes made for the user.

```
        } else {
            s.userInfo[msg.sender][_bondID].claimedETHx += pendingTotal;
            s.ETHx.transfer(msg.sender, pendingTotal);
            if (maxPayout == claimedPayout) {
                s.userInfo[msg.sender][_bondID].isDeactivated = true;
                s.totalEarningPower -= (_user.earningPower + (_user.earningPower
                * (s.bonusPerPower - _user.currentPowerBonus)) / 1e18);
            }
```

If we view the _claimETHx() function here in LibClaimAndSnipe.sol, we can compare to see that some state changes are missing.

```
        if (userInfo.ETHxMaxPayout < userInfo.claimedETHx + pending) {
            ...
            userInfo.earningPower += (userInfo.earningPower * (s.bonusPerPower
             - userInfo.currentPowerBonus)) / 1e18;
            ...
        } else {
            ...
            s.totalEarningPower += (userInfo.earningPower * (s.bonusPerPower
             - userInfo.currentPowerBonus)) / 1e18;
            userInfo.earningPower += (userInfo.earningPower * (s.bonusPerPower
             - userInfo.currentPowerBonus)) / 1e18;
            userInfo.currentPowerBonus = s.bonusPerPower;
            userInfo.currentETHxPerPower = s.ETHxPerPower;
        }
```

The impact of this issue is that the user could call the claimETHxRewards() function on the Core contract and possibly double claim ETHx rewards. This is mainly because the user's currentETHxPerPower is not updated. Overall, this results in inconsistent calculations for the user.

Recommendation: It is recommended to keep the ETHxReserve() simple by not claiming user's pending ETHx and only claim the 10% incentive. But if ETHx claiming is being accommodated, make sure to add the state changes to avoid the issue.

6.3 Medium Risk

6.3.1 Consider retrieving ETHDollarCentValue from a price oracle instead of hardcoding it

Context: Core.sol#L1035-L1040
Impact: Medium
Likelihood: Medium

Description: In function getTokenAmounts(), which is used when ALX bonds are being purchased, if the ETHxPOL address is not set, we default to hardcoded cent values for ETH and ALX i.e. 300000 and 25. This is fine with ALX since the price of the token in USD cents would not be available during the first hour but for the ETHDollarCentValue, it should use an oracle to retrieve the price and convert it to cents or at the very least the value should be configurable by the team so that it is updated to represent better accurate values.

Due to the use of the hardcoded cent value, if the price of ETH in USD is higher than $3000, the ETH bonds being purchased through the protocol would be cheaper for the users while if the price of ETH in USD is lower than $3000, the ETH bonds would be costlier.

```
        if (s.ETHxPOL == address(0)) {
            uint256 ETHDollarCentValue = 300000;
            uint256 ALXDollarCentValue = 25;
            uint256 ALXRequired = tokenRequired_ * ETHDollarCentValue /
            ALXDollarCentValue;
            return (ALXRequired);
        }
```

Recommendation: Either use an oracle and convert the output to cents or let the value be configurable so that it is updated every 5-10 minutes possibly by a off-chain relayer/script set by the team. This is only required uptil the first hour after which the ETHxPOL should be set.

6.3.2 Incorrect condition in LibALXStake.sol sniping mechanism

Context: LibALXStake.sol#L186-L190
Impact: Low
Likelihood: High

Description: If we look at the code here, we can see that in the if condition, we check **_sniper.ETHxMaxPayout < claimedPayout**. This is incorrect and it should instead check **sniper.ETHxMaxPayout < claimedPayout + pendingETHxRewards**. This is

because if we inspect the logic in the if block, **pendingETHxRewards + claimedPayout** are used in calculating the extra deficit to provide to the reserve.

```
            uint256 pending = amountETHx / 2;
            uint256 pendingETHxRewards = LibClaimAndSnipe._pendingETHx(
            msg.sender, _bondID, _indexInArray);
            uint256 claimedPayout = _sniper.claimedETHx + pending;
            if (_sniper.ETHxMaxPayout < claimedPayout) {
                s.ETHx.transfer(msg.sender, _sniper.ETHxMaxPayout -
                _sniper.claimedETHx);
                _sniper.claimedETHx = _sniper.ETHxMaxPayout;
                _sniper.isDeactivated = true;
                s.ETHxReserveShare += pendingETHxRewards + claimedPayout
                - _sniper.ETHxMaxPayout;
                s.totalEarningPower -= (_sniper.earningPower +
                (_sniper.earningPower * (s.bonusPerPower -
                _sniper.currentPowerBonus)) / 1e18);
            } else {
                s.ETHx.transfer(msg.sender, pending);
                _sniper.claimedETHx = claimedPayout;
            }
```

Due to this, it takes longer than expected to hit the ETHxMaxPayout since pendingETHxRewards are not taken into consideration.

Recommendation: Currently including pendingETHxRewards does not seem to serve any purpose other than providing the extra difference to the reserve. Other than the solution already provided above, there is also the option remove the pendingETHxRewards from the equation completely.

6.3.3 Incorrect placing of rewards claiming can prevent users from unstaking ETH tokens in Stake.sol

Context: Stake.sol#L165
Impact: Medium
Likelihood: Medium

Description: In the function unstake(), we place _claimStakingRewards() before the 10 block delay check. The issue is that a user who wants to fully or partially unstake their tokens but wants to restake their rewards is not able to do so. This is because the restake would cause staker.stakeBlock to updated to the current block. Due to this, the user would encounter the 10 block delay check and revert.

```
        LibStake._claimStakingRewards(msg.sender, 0, 0, _restake);
        ...
        ...
        ...
        if (staker.stakeBlock + 10 > block.number) {
            revert TooEarlyToUnstake();
        }
```

Recommendation: Consider placing the _claimStakingRewards() call right after the 10 block delay check to ensure correct behaviour.

6.3.4 Permitting anyone to claim airdrop for any account opens up different DOS attacks

Context: ALX.sol#L198
Impact: Medium
Likelihood: Medium

Description: Currently the claim() function in ALX.sol allows anyone to claim airdrop tokens for any account. This requires a single merkle proof, which can be used to claim until the claimed amount equals the _airdroppedAmount variable. The ALX.sol contract also allows users to use their airdropped ALX to purchase an ALX bond or stake them using functions purchaseBondWithAirdroppedALX() and stakeWithAirdroppedALX() respectively.

Due to this open claiming feature through the claim() function though, anybody can launch a DOS attack on a specific user in different ways.

```
    function claim(
        bytes32[] calldata _proof,
        address _account,
        uint256 _airdroppedAmount,
        uint256 _amountToClaim
    ) external {
```

For example, let's say a user has 100 tokens to claim from the airdrop.

Scenario 1: If user does not want to claim but purchase bonds or stake their 100 tokens, they call the respective functions to do so with 100 tokens as amount. But since claiming allows anyone to claim for the user, the attacker can make a 1 wei claim by frontrunning the user's transaction. This would cause the user's transaction to revert since now only 100 tokens - 1 wei are available.

Scenario 2: If user wants to claim full amount of 100 tokens, the user passes in 100 as parameter. The attacker performs the same frontrunning attack of 1 wei, thus causing the user's transaction to revert.

Scenario 3: Let's say user decides to claim 50 tokens and use the remaining 50 to claim later on or purchase and stake. The attacker can perform the same attack to cause the revert for the remaining 50 tokens.

The question is how does the attacker retrieve the user's proof since that is the only requirement along with some gas fees to launch this attack? There are currently three ways in which this could be possible: 1) It is possible if the tree is made public 2) It is simply possible to copy the user's transaction data from the mempool and use it in the attack 3) If the user had made a partial claim previously (like the one in scenario 3), the attacker can just use the onchain proof data available in the user's previous transaction for the attack on the remaining tokens. The proof is the same since the merkle tree is immutable once the contract is deployed with the merkle root.

Recommendation: Similar to the purchaseBondWithAirdroppedALX() and stakeWithAirdroppedALX() functions, consider using msg.sender directly and remove the _account parameter in function claim().

6.3.5 User cannot transfer a bond before reaching max payout or bond maturity

Context: Core.sol#L532
Impact: Low
Likelihood: High

Description: A user cannot transfer a bond if it has not reached max payout and the bond has not matured. This occurs since during the transfer of a bond through the transferBond() function, we claim the user's ETHx rewards. During this claiming process, it only allows the user to transfer the bond if the max payout has been hit and the bond has matured. If netiher conditions are met, the call reverts with the error TooEarlyToClaim().

```
        if (userInfo.ETHxMaxPayout < userInfo.claimedETHx + pending) {
            ...
        } else {
            if (userInfo.startTimeStamp + userInfo.daysCount * 24 hours >
            block.timestamp) {
                revert TooEarlyToClaim();
            }
```

Bonds should be allowed to be transferred even if these conditions are not met since in most cases it would only make sense for a user to transfer a bond to their other address or someone else while it is active. Since the protocol makes users aware that ETHx rewards are only available once bond matures or ETHxMaxPayout has been reached, the user would be aware of losing possible claimETHx rewards by transferring to another address.

Recommendation: In the function transferBond(), check if the user has met the max payout and user's bond has not matured. If the user has not hit the max payout and the bond has not matured, we skip claiming and continue the bond transfer to avoid the revert.

6.4 Low Risk

6.4.1 Implement msg.value check in mint() function of ETHx.sol contract

Context: ETHx.sol#L260

Description: The mint() function in ETHx.sol allows taking in either ETH or EETH in order to mint ETHx tokens. Since the function is payable, for the case of the else block i.e. when input token is EETH, it does not ensure msg.value is not sent. Since it is a user facing function, consider reverting when input token is EETH and msg.value is also sent along with the call.

Due to this being a user mistake, the severity is low but protection should be put in place to avoid such a situation.

Recommendation: Add the msg.value check in the else block.

```
        else {
            if (msg.value > 0) {
                revert();
            }
            EETH.transferFrom(
                msg.sender,
                address(this),
                _amount
            );
        }
```

6.4.2 Consider marking user-facing functions as non-payable

Context: ALXStake.sol#L181

Description: Function stakeLocked() in ALXStake.sol takes in ALX tokens and puts them at stake. This function does not deal with msg.value but it is still marked payable. Although marking functions as payable saves gas (see issue 6.6.3 in this report), it should not be done in user-facing functions to ensure msg.value sent by mistake is not locked in the contract. Another instance of this is present in function purchaseALXBond() in Core.sol.

Recommendation: Remove the payable keyword from function stakeLocked() and purchaseALXBond().

6.4.3 Possible to convert ALX bonds to ETH bonds during reactivation

Context: Core.sol#L412

Description: Currently the reactivate() function only checks if a bondID exists and is deactivated. If it exists and is deactivated, it allows reactivating ETH bonds. The issue

is that ALX bonds can be reactivated and potentially converted into ETH bonds. This is possible since ALX bonds are pushed to the userToIds array as well and can be deactivated if the sniper reached their max payout. Since they've reached their max payout and cannot mint their ALX back, they could potentially convert their originally ALX bonds to ETH bonds by calling the reactivate() function with their ALX bondId.

Recommendation: This could potentially be seen as a feature more than a bug since there seem to be no issues arising through the conversion i.e. it is simply a creation of an ETH bond. The ALX bond ceases to exist and ETH bond starts to exist. I would recommend adding an ETH bond check and disallowing this conversion though.

6.4.4 Consider using old restakeEnabled value when reactivating a bond

Context: Core.sol#L412

Description: Over here, we set _restakeEnabled to false when we reactivate a bond for a user. It would be better to use the user's old configuration i.e. restakeEnabled being true or false. There is a small likelihood that the msg.sender (which could be a contract) is not able to handle ETHx rewards due to which they restake the rewards or if someone snipes them as well. Due to this reactivation, if ETHx is transferred to the contract (due to restake being false), the tokens could be locked in there.

Recommendation: Instead of hardcoding restakeEnabled to false, use user's old restakeEnabled value to carry forward user's previous configuration.

6.4.5 Ensure recipient is not address(0) when transferring a bond

Context: Core.sol#L532

Description: In transferBond(), if users forget to pass the recipient address, it defaults to address(0), which can cause the bond to be lost and any future ETHx rewards to be locked or ALX waiting to be re-minted back to user (for ALX bond if ETHxMaxPayout was not reached). This is also a protective measure to ensure an attacker cannot claim rewards and dump a bond to address(0), whose assigned ETHx rewards are locked in the Core contract.

Some related instances where recipient should be checked to not be address(0):
1. Link
2. Link

Recommendation: Ensure that the recipient parameter passed is not address(0).

6.4.6 Consider disallowing self bond transfers to avoid loss of bonds

Context: Core.sol#L550

Description: When a bond is transferred using transferBond(), it does not ensure recipient is not the same as the owner of the bond i.e. msg.sender. Due to this, what could happen

here is that the bond would be set to the same user and be deleted for the same user in the next step. This effectively deletes the bond from existence and causes loss of future ETHx rewards (for ETH bond) and ALX waiting to be re-minted back to user (for ALX bond if ETHxMaxPayout was not reached). Normally this would be considered as a user mistake but the protocol should enforce such protection since it could be used by attackers to phish users into self transfers.

Recommendation: Check if the msg.sender is the recipient. If true, revert.

6.4.7 Consider sending rewards and incentive to user's reward recipient in ETHxReserve() function

Context: Reserve.sol#L259

Description: In ETHxReserve(), when the pending ETHx of the user is claimed + the 10% incentive, the rewards are sent to msg.sender instead of reward recipient. If msg.sender is a contract which cannot handle or withdraw the rewards from their contract, the tokens would be locked.

Recommendation: Consider using the user's reward recipient.

6.4.8 Sniper could possibly delay and prevent users from unstaking in certain conditions

Context: Stake.sol#L344

Description: If a user has restakeEnabled set to true, a sniper can call the function claimStakingRewards() in Stake.sol to frontrun the user's unstake call. This would claim the tokens for the user and restake them, thus causing stakeBlock to be updated to block.number. When the user's call goes through, it reverts due to the 10 block delay i.e. 2 minutes. If during these 2 minutes, more rewards accumulate (during periods of high participation), the sniper can perform the same attack.

Recommendation: A 5 to 10 minute delay could be introduced for a user once someone snipes them. During this period, no one can snipe the user, which allows a grace period to exit the system. If the user does not exit the system, sufficient rewards would possibly be accumulated and ready to snipe.

Alixa comments: User's can prevent sniper's from staking by disabling the restake option.

6.4.9 buybackAndBurn() does not check the current cycle

Context: Reserve.sol#L256

Description: In function buybackAndBurn(), we check the previous cycle to be less than 2 ether here. This is done to check if the protocol has enough participation or not. The issue is that it should also check it for the current daily cycle. This is because let's say previous

cycle participation was 1.8 ether and currently we are 1 day into the current week where the cycle participation is 5 ether. This means the protocol is not experiencing low participation. But the function still remains callable.

The same instance of this issue is present in ETHxReserve(). The recommendation provided below should also be applied for it except that we use weekly cycles and the value is 7 ether.

Recommendation: Add the condition as shown below:

```
if (s.ETHInCycle[dailyCycle - 1] >= 2 ether && s.ETHInCycle[dailyCycle]
>= 2 ether) {
    revert CanBeTriggeredWhenThereIsNotEnoughtParticipation();
}
```

Alixa comments: We are putting the suggested check in the off-chain component. This will allow us to have more customisation options in future depending on how the protocol performs.

6.4.10 ETH unbonding would not be possible due to mint() max cap revert

Context: Core.sol#L669

Description: The ALX mint() function can revert if we hit the max cap enforced in it.

In function unbondingETHBond(), users who decide to unbond will not be able to do so through the function since we revert if the max cap condition is met. Due to this, the call will revert causing s.userInfo for the user to never be deleted and the bondID to not be removed from userToIds array.

Another instance of this is in unbondingALXBond() function and the batch ETH/ALX unbonding functions.

Recommendation: Since ALX cannot be minted, user's have to wait till the ALX supply decreases. In such cases, the buybackByTeam() function can come in handy to burn some ALX from the protocol's position.

Alixa comments: In this case the user will have to wait to reach their maxPayout or to claim their ALX share proportional to the remaining payout, they will have to wait for protocol to burn ALX tokens.

6.5 Non-Critical

6.5.1 lastClaim variable not updated for user in Reserve.sol, and LibClaimAndSnipe.sol

Context: LibClaimAndSnipe.sol#L483, Reserve.sol#L259

Description: In Reserve.sol, when a user calls ETHxReserve, their pending ETHx is also claimed along with the extra 10% incentive. But the lastClaim variable for the user's ETH bondID is not updated to block.timestamp.

In LibClaimAndSnipe.sol, it is not updated in function _snipe() when ALX bond snipers snipe to earn ETHx rewards.

Recommendation: For both instances, set lastClaim = block.timestamp

6.5.2 Rename function setRewardsForStaking() to setRestakingSetting()

Context: Stake.sol#L322

Description: The function setRewardsForStaking() provides users with the ability to enable and disable restaking. But the function name does not match it's functionality.

Recommendation: Rename the function to setRestakingSetting().

6.5.3 Rename parameter in _checkTokensAmountsAndConvertToETHx() function

Context: LibCore.sol#L171

Description: The parameter should be renamed to _withoutTransfer instead of _sellvXNF to better represent the actual value passed to the function when purchasing ETH bonds.

Recommendation: Rename the parameter to _withoutTransfer.

6.6 Gas Optimizations

6.6.1 Avoid caching address(this) to memory

Context: POL.sol#L275, POL.sol#L298, POL.sol#L320

Description: When address(this) is accessed, it uses the ADDRESS opcode which costs 2 gas only while caching to memory causes 3 gas for MSTORE opcode and 3 gas for MLOAD everytime it is accessed.

Recommendation: Consider using address(this) everywhere instead of caching it to memory

6.6.2 Avoid copying immutable variable to memory

Context: POL.sol#L390, SwapHandler.sol#L165

Description: When an immutable variable is created and assigned during construction, it is replaced in all places where it is accessed i.e. it becomes hardcoded in the bytecode of the

contract. It is much cheaper to read these variables directly than to copy them to memory (MSTORE - 3 gas) and access them through memory (MLOAD - 3 gas).

Recommendation: Use immutable variables directly wherever required.

6.6.3 Consider marking stakeWithAirdroppedALX() as payable to save gas

Context: ALXStake.sol#L146

Description: We know for a fact that function stakeWithAirdroppedALX() in ALXStake.sol will never be able to receive msg.value due to the call originating from ALX.sol not allowing passing msg.value. Due to this, we can mark stakeWithAirdroppedALX() as payable since the solidity compiler would not need to check if msg.value was passed or not. The solidity compiler adds extra opcodes when functions are marked non-payable to ensure msg.value cannot be sent to those functions. When functions are marked payable, this set of extra opcodes i.e. the check for msg.value is not added. Since we can guarantee that stakeWithAirdroppedALX() will not receive msg.value, it can thus be marked payable.

Recommendation: Add the payable keyword to function stakeWithAirdroppedALX().

6.6.4 Consider using do-while loops instead of for-loops

Context: ALXStake.sol#L330, ALXStake.sol#L358, Core.sol#L546, Core.sol#L653, Core.sol#L767, Core.sol#L794, Core.sol#L827, Core.sol#L850, Core.sol#L1025

Description: Solidity do-while loops are much gas-efficient than for-loops even if you add an extra if-condition check for the case where the do-while loop doesn't execute at all.

Recommendation: Replace all for loops with do-while loops.

6.6.5 Earning power is attempted to be decreased for ALX bonds

Context: LibALXStake.sol#L193

Description: ALX bonds i.e. the sniper bonds have no earning power in the system, which is visible when the bond is created through the Core contract. Over here though we attempt to decrease snipers' earning power when there isn't one for ALX bond snipers.

Recommendation: Consider removing the statement to save gas.

6.6.6 Cache earning power calculations in LibClaimAndSnipe.sol to save gas

Context: LibClaimAndSnipe.sol#L173

Description: The two state updates below have similar calculations. Caching them to a memory variable and using them would save more gas since we avoid repeated use of math related opcodes such as MUL, DIV, SUB etc.

```
File: LibClaimAndSnipe.sol
172:         } else {
173:             ...
176:
177:             s.totalEarningPower += (userInfo.earningPower *
                 (s.bonusPerPower - userInfo.currentPowerBonus)) / 1e18;
178:             userInfo.earningPower += (userInfo.earningPower *
                 (s.bonusPerPower - userInfo.currentPowerBonus)) / 1e18;
```

Recommendation: Calculate the equation once and access it in both places using a memory variable.

6.6.7 Consider using named returns to save gas

Context: LibCommon.sol#L69

Description: In LibCommon.sol, in function _calculateCycle(), consider using named return variables. The solidity compiler outputs more efficient code when the variable is declared in the return statement.

Recommendation: Using named returns should also be applied to other instances in the codebase and tested once the test suite has been implemented to verify.

6.6.8 Consider dividing by 1e15 to save gas on ETH bond purchases in LibCore.sol

Context: LibCore.sol#L242

Description: Instead of **tokenRequiredCalculated * 10 / 0.01 ether;** use **tokenRequiredCalculated / 1e15;** to save 5 gas (i.e. the MUL opcode) per function call by users.

```
        if (s.totalALXLockedStakes != 0) {
            ...
            _loyaltyPoints = tokenRequiredCalculated * 10 / 0.01 ether;
            ...
        }
```

Recommendation: Divide by 1e15 as stated above.

6.6.9 Optimize distribute() function in LibCore.sol

Context: LibCore.sol#L363

Description: There are two optimizations that could be applied to the distribute() function in LibCore.sol.

1. Instead of _amount * 5 / 100; , just use _amount / 20; to save 5 gas
2. For totalALXRewards, instead of performing same operation, just use buybackAmount. DIV and MUL opcodes cost 10 gas in total, while using an existing memory variable just requires using 3 gas for SLOAD, thus net saving 7 gas per call.

```
            buybackAmount = _amount * 5 / 100;
                LibReserve.deposit(
                    buybackAmount,
                    0,
                    _amount / 100,
                    _amount * 4 / 100
                );

                s.totalALXRewards += (_amount * 5 / 100);
            }
```

Recommendation: As stated above in point (1) and (2), apply the respective optimizations.