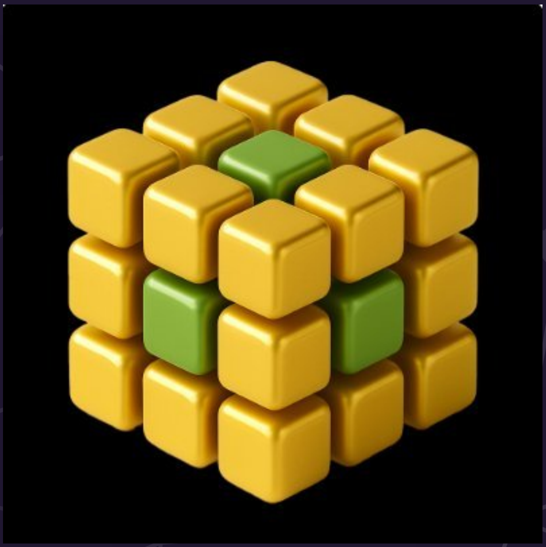
BuilDefi

Security Review



Nov 6, 2025

Contents

1. About SBSecurity

SBSecurity is a team of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol's functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

- **High** - almost **certain** to happen, easy to perform, or highly incentivized.
- **Medium** - only **conditionally possible**, but still relatively likely.
- **Low** - requires specific state or **little-to-no incentive**.

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

A time-boxed security review of the **De-centraX/buildefi-contracts** repository was conducted by **SBSecurity**. The review was performed by a team of 2 security researchers, who identified **35 issues** in total.

Overview

Project	BuilDefi
Commit Hash	54c18d16f6055f9857f3fb64e03a319f35f65ae1
Resolution	a6b18074216531e725a2533883873077721b1a12
Timeline	Audit: September 11 - October 8, 2025 Mitigation: October 8, 2025 - November 5, 2025

Scope

AaveStaking.sol, SwapActions.sol, ChainlinkConsumer.sol, ArrayLib.sol, BuyAndBurnLifeLib.sol, LeaderboardLib.sol, MemetokenInfoLibrary.sol, SwapActionsLib.sol, V3OracleLibrary.sol, V4OracleLibrary.sol, XLaunchConfigLib.sol, XPositionManagerLib.sol, XUniswapLib.sol, Memetoken.sol, TruncatedOracle.sol, TruncGeoOracle.sol, FairLaunchInfo.sol, LaunchOptions.sol, LeaderboardItem.sol, Common.sol, Constants.sol, Math.sol, XLiquidityManagementHelper.sol, XPaymentsUpgradeable.sol, Whitelisting.sol, XBoost.sol, XBuyAndBurn.sol, XCloner.sol, XFairLaunch.sol, XLaunch.sol, XLaunchConfig.sol, XLeaderboard.sol, XLiquidityManager.sol, XNFT.sol, XPoints.sol, XPositionManager.sol, XStaking.sol, XTaxDistributor.sol, XV3LiquidityManager.sol

Post Audit Condition

The codebase underwent significant refactoring and architectural changes compared to its initial state, followed by the resolution of a substantial number of issues.

SBSecurity does not approve this codebase as deployment ready. Additional audits are recommended before considering deployment.

5. Findings

Issues Found

Critical Risk	5
High Risk	5
Medium Risk	14
Low/Info Risk	11

ID	Title	Severity	Resolution
[C-01]	Inflation attack in XStaking, allows stealing all the memetoken stakes	Critical	**Fixed**
[C-02]	tokenToPurchaseStatus is not cleared allowing all rejections/ refunds allows users to steal residue ETH from the Whitelisting contract	Critical	**Fixed**
[C-03]	TaxDistributor taxes can be distributed maliciously	Critical	**Fixed**
[C-04]	Anyone can sent all ETH to `platformGenesis` via `triggerCTO`	Critical	**Fixed**
[C-05]	tokenToPoolBalance is not decreased upon triggerCTO	Critical	**Fixed**
[H-01]	`MAX_FREE_LAUNCHES_PER_ADDRESS` is infinite	High	**Fixed**
[H-02]	`PointsPackages` not ordered ascending when new one is added	High	**Fixed**
[H-03]	tokens leaderboard isn't sorted and wrong tokens will be removed	High	**Fixed**
[H-04]	changing contracts in `xLaunch` will break functionality	High	Acknowledged
[H-05]	triggering CTO can be prevented with a dust swap	High	Acknowledged
[M-01]	AavePool address should have setter	Medium	**Fixed**

[M-02]	AaveStaking deposit will fail when supply cap is reached	Medium	**Fixed**
[M-03]	Aave incentive rewards will be lost	Medium	**Fixed**
[M-04]	Affiliate can reject whitelisted purchase approval	Medium	**Fixed**
[M-05]	Points holder can avoid burning his tokens on refund	Medium	**Fixed**
[M-06]	LP allocation fees not claimed before increasing position in `LiquidityManager::addLP`	Medium	**Fixed**
[M-07]	When swap fails with anon error all the exec reverts, halting the hook swaps	Medium	**Fixed**
[M-08]	`_getTwapAmountV4Sequential` & `_calculateAmountOutMinimum` account the fee but fee is being taken 60 mins after pool deployment	Medium	Acknowledged
[M-09]	hardcoded 15% slippage will cause dos of the swap functions	Medium	**Fixed**
[M-10]	hardcoded twap lookback	Medium	**Fixed**
[M-11]	`_loopProcessExternalToken` distributes in LIFO order, incentivising latest queue receivers	Medium	Acknowledged
[M-12]	XSS attack on the unsanitized string inputs	Medium	**Fixed**
[M-13]	`_beforeSwap` can run out of gas	Medium	**Fixed**
[M-14]	`deltaForFeeCurrency` uses already taxed amount for exactOutput swaps	Medium	Acknowledged
[L-01]	Missmatches between documentation and code	Low	**Fixed**
[L-02]	onlyXLaunch modifier is enough in several places	Low	**Fixed**
[L-03]	Longer boosting tiers are not incetivizing	Low	**Fixed**
[L-04]	Weird ERC20 tokens not supported	Low	**Fixed**

[L-05]	Protocol assumes all the stablecoins are pegged	Low	Acknowledged
[L-06]	Sequencer downtime can go unnoticed	Low	**Fixed**
[L-07]	Simplification of launch flow	Low	**Fixed**
[L-08]	`_applyForRank` does redundant operation	Low	Acknowledged
[L-09]	duplicate tax allocation targets can be passed	Low	**Fixed**
[L-10]	Anyone can stop memetoken from graduating	Low	**Fixed**

Critical severity

[C-01] Inflation attack in XStaking, allows stealing all the memetoken stakes

Severity: Critical Risk

Description: There's an inflation attack vulnerability in the `XStaking` contract, that allows all the initial deposits to be stolen by attacker.

1. attacker enters with 1 wei and receives 1 wei share

2. victim enters with 10e18 memetoken, attacker frontrun with 10e18 direct donation to the `XStaking` contract.

3. victim receives 0 shares, but his 10e18 eth deposit is taken

4. attacker withdraws his 1 wei share and receives back all the total assets of the vault, which is 10e18 from victim + 10e18 + 1 wei from his own deposit and donation

```
function enter(uint256 _amount) external nonReentrant {
    uint256 totalTokens = memetoken.balanceOf(address(this));
    uint256 totalShares = totalSupply();
    uint256 shares;
    if (totalShares == 0 || totalTokens == 0) {
        shares = _amount;
    } else {
        shares = _amount * (totalShares) / (totalTokens);
    }

    _mint(msg.sender, shares);
    IERC20(memetoken).safeTransferFrom(msg.sender, address(this), _amount);

    emit Stake(msg.sender, _amount, shares, address(memetoken));
}

/// @inheritdoc IXInternalStaking
function leave(uint256 _shares) external nonReentrant {
    uint256 totalShares = totalSupply();
    uint256 amount = _shares * (memetoken.balanceOf(address(this))) / (totalShares);

    _burn(msg.sender, _shares);
    IERC20(memetoken).safeTransfer(msg.sender, amount);

    emit Unstake(msg.sender, amount, _shares, address(memetoken));
}
```

Recommendation: You can mint "dead shares" on the first deposit, so the donation needed for inflating the share price rises up exponentially or use the OZ `_decimalsOffset` implementation.

Resolution: Fixed

[C-02] tokenToPurchaseStatus is not cleared allowing all rejections/refunds allows users to steal residue ETH from the Whitelisting contract

Severity: Critical Risk

Description: Rejected whitelisting requests use pull pattern and assign the payment amount to a mapping, which allows the buyer to claim later. There's an issue which allows refund claims to be replayed, because the check ensuring it is based on the purchase deadline.

Check only prevents not expired purchases from being refunded:

```
function refundPurchasedWhitelist(address _tokenAddress) external {
    require(
        tokenToPurchaseStatus[_tokenAddress].purchaseDeadline < block.timestamp,
        Whitelisting__TokenPurchaseWhitelistApprovalNOTPassed()
    );
    _refundPurchaseWhitelist(_tokenAddress);
}
```

```
function _refundPurchaseWhitelist(address _tokenAddress) internal {
    PurchaseWhitelistStatus memory purchaseStatus = tokenToPurchaseStatus[_tokenAddress];
    purchaseStatus.purchaseDeadline = 0;

    userToPaymentTokenToBalance[purchaseStatus.buyer][purchaseStatus.paymentToken] +=
purchaseStatus.paymentAmount;

    tokenToPurchaseStatus[_tokenAddress] = purchaseStatus;
    emit WhitelistPurchaseRefundCredit(
        purchaseStatus.buyer, purchaseStatus.paymentToken, purchaseStatus.paymentAmount
    );
}
```

But once refund is done, `purchaseDeadline` is set to 0, allowing the users to call the function infinite number of times (`0 < block.timestamp`), each one increasing his `userToPaymentTokenToBalance`.

After that, they only have to wait until there's enough `paymentToken` balance in the `Whitelisting` contract and drain the whole contract assets.

```
function pullTokens(address _user, address _paymentToken) external {
    uint256 amount = userToPaymentTokenToBalance[_user][_paymentToken];
    userToPaymentTokenToBalance[_user][_paymentToken] = 0;

    emit WhitelistPurchaseRefundDebit(_user, _paymentToken, amount);

    sendTokens(_paymentToken, amount, _user);
}
```

Recommendation: Reject refunds when `tokenToPurchaseStatus[_tokenAddress].purchaseDeadline == 0`.

Resolution: Fixed

[C-03] TaxDistributor taxes can be distributed maliciously

Severity: Critical Risk

Description: `TaxDistributor::processTaxes` relies on externally passing the `destinations` in a permissionless manner. This allows the token creator to syphon all the taxes collected, not matter the initial distribution configuration from `setTaxes` doesn't allocate 100% to him.

```
function processTaxes(address memetoken, BuySellTaxAllocation[] calldata destinations) external
nonReentrant {
    require(xLaunch.memetokenToTokenId(memetoken) > 0, XTaxDistributor__MemetokenNotValid());

    uint256 totalTaxes = IERC20(memetoken).balanceOf(address(this));
    require(totalTaxes > 0, XTaxDistributor__NoTaxesToProcess());
    for (uint256 i = 0; i < destinations.length; i++) {
        BuySellTaxAllocation destination = destinations[i];
        _processTax(memetoken, destination, totalTaxes);
    }
}
```

As we see, there's no mechanism preventing distributions from happening multiple times to the same destination.

Here are at least 2 scenarios possible currently:

1. duplicated destinations in a single call - `[TOKEN_CREATOR, TOKEN_CREATOR, TOKEN_CREATOR]` each loop will distribute

2. multiple separate calls - each time invoking with `TOKEN_CREATOR` as a single destination multiple times

Recommendation:

1. Disable the possibility users to pass the destinations as arguments.

2. Track the destinations and enforce complete distribution on each invocation.

Resolution: Fixed

[C-04] Anyone can sent all ETH to `platformGenesis` via `triggerCTO`

Severity: Critical Risk

Description: The purpose of `triggerCTO` is to list the ownership of the token for sale and it should send 50% of the leaderboard pool of this token to the `platformGenesis` and the rest 50% for the Platform Leaderboard Pool. The function stores whether the token is listed for CTO in tokenToIsListedForCTO but does not check it and thus this function can be called infinitely, draining all different leaderboard pools as they are all with ETH. 50% of the `platformGenesisAmount` will be the same value over and over again, this can be done in a loop until the contract ETH balance is over.

```
function triggerCTO(address _token) external {
    XLaunchConfig xLaunchConfig = XLaunchConfig(xLaunch.getContract(XLaunchConfigName));
    uint256 timeSinceLastFeesPassed = block.timestamp
        -
IXPositionManager(payable(xLaunch.getContract(XPositionManagerName))).memetokenToLastTradedTimestamp(_token
);
    uint256 CTO_INTERVAL = xLaunchConfig.uint256ConfigToValue(XLaunchConfigLib.CTO_INTERVAL);
    FairLaunchStatus status =
XFairLaunch(xLaunch.getContract(XFairLaunchName)).getFairLaunchInfo(_token).status;
    require(
        timeSinceLastFeesPassed >= CTO_INTERVAL && status == FairLaunchStatus.CLOSED,
        XLeaderboard__InvalidTokenCTOTrigger()
    );

    uint256 platformLeaderboardPoolAmount = tokenToPoolBalance[_token] * 50_00 / BPS;
    uint256 platformGenesisAmount = tokenToPoolBalance[_token] - platformLeaderboardPoolAmount;

    tokenToPoolBalance[PLATFORM_LEADERBOARD] =
        tokenToPoolBalance[PLATFORM_LEADERBOARD] + platformLeaderboardPoolAmount;

    _safeETHTransfer(xLaunchConfig.platformGenesis(), platformGenesisAmount); // The platform genesis is
trusted not to revert

    tokenToIsListedForCTO[_token] = true; <-----------------------
    emit ListForCTO(_token);
}
```

Recommendation: Check if `tokenToIsListedForCTO[_token] = true`, at the beginning.

Resolution: Fixed

[C-05] tokenToPoolBalance is not decreased upon triggerCTO

Severity: Critical Risk

Description: When `triggerCTO()` is called 50% of the leaderboard pool of that token are distributed to `PLATFORM_LEADERBOARD` and the rest to `platformGenesis`. But `tokenToPoolBalance[_token]` is never lowered/cleared and therefore`payout` can be called, which will use other pools tokens and mess all the calculations.

```
function triggerCTO(address _token) external {
    XLaunchConfig xLaunchConfig = XLaunchConfig(xLaunch.getContract(XLaunchConfigName));
    uint256 timeSinceLastFeesPassed = block.timestamp
        -
IXPositionManager(payable(xLaunch.getContract(XPositionManagerName))).memetokenToLastTradedTimestamp(_token
);
    uint256 CTO_INTERVAL = xLaunchConfig.uint256ConfigToValue(XLaunchConfigLib.CTO_INTERVAL);
    FairLaunchStatus status =
XFairLaunch(xLaunch.getContract(XFairLaunchName)).getFairLaunchInfo(_token).status;
    require(
        timeSinceLastFeesPassed >= CTO_INTERVAL && status == FairLaunchStatus.CLOSED,
        XLeaderboard__InvalidTokenCTOTrigger()
    );

    uint256 platformLeaderboardPoolAmount = tokenToPoolBalance[_token] * 50_00 / BPS;
    uint256 platformGenesisAmount = tokenToPoolBalance[_token] - platformLeaderboardPoolAmount;

    tokenToPoolBalance[PLATFORM_LEADERBOARD] =
        tokenToPoolBalance[PLATFORM_LEADERBOARD] + platformLeaderboardPoolAmount; <-----------

    _safeETHTransfer(xLaunchConfig.platformGenesis(), platformGenesisAmount); // The platform genesis is
trusted not to revert

    tokenToIsListedForCTO[_token] = true;
    emit ListForCTO(_token);
}
```

`payout()` will distribute 20% of the old 100% of `tokenToPoolBalance[_token]` or if left uncalled the newly accumulated will be used also for bigger drain of the other pools, while all this should be impossible since upon CTO 100% are distributed.

Recommendation: Set `tokenToPoolBalance[_token] = 0`.

Resolution: Fixed

High severity

[H-01] MAX_FREE_LAUNCHES_PER_ADDRESS is infinite

Severity: High Risk

Description: MAX_FREE_LAUNCHES_PER_ADDRESS is mistakenly hardcoded to uint.max, which allows anyone to launch as many tokens as they wish, with all possible add-ons available for free. This will harm the protocol, since there won't be launch fees flowing in.

```
uint256ConfigToValue[XLaunchConfigLib.MAX_FREE_LAUNCHES_PER_ADDRESS] = type(uint256).max;
```

```
function launch(LaunchOptions memory _opts)
    external
    payable
    nonReentrant
    returns (address memetoken, uint256 tokenId)
{
    // Validate and get the memetoken info
    XLaunchConfig xLaunchConfig = XLaunchConfig(contractRegistry[XLaunchConfigName]);
    XNFT xNFT = XNFT(contractRegistry[XNFTName]);

    MemetokenInfo memory info = xLaunchConfig.validateLaunchOptions(_opts, xNFT.nextTokenId());
    uint256 requiredEth;
    if (info.launchOptions.launchFree) {
        xLaunchConfig.verifyFreeLaunch(userToFreeLaunches[msg.sender]);
        userToFreeLaunches[msg.sender]++;
    }
```

Furthermore, since the Points can be traded, they'll have some monetary value. By leaving users with infinite free launches, they'll be launching all sorts of tokens, farming points, and directly profiting from them.

Recommendation: In XLaunchConfig, implement a setter function where you can give free launches to each address individually.

Resolution: Fixed

[H-02] PointsPackages not ordered ascending when new one is added

Severity: High Risk

Description: When adding new points packages, if the `_pointsAmount < biggest existing one`, `pointsPackages` array becomes unsorted. After that, users buying multiple points packages won't get the optimal what their entitled to, resulting in a opportunity loss for them.

```
function addPointsPackage(uint256 _packageCost, uint256 _pointsAmount, uint256 _pointsBonus)
    external
    onlyXLaunchOwner
{
    require(_packageCost != 0, Common__ZeroAmount());
    require(_pointsAmount != 0, Common__ZeroAmount());

    pointsPackages.push(
        PointsPackage({packageCost: _packageCost, pointsAmount: _pointsAmount, pointsBonus: _pointsBonus})
    );

    emit PointsPackageAdded(_packageCost, _pointsAmount, _pointsBonus);
}
```

In fact, users can pay for the most expensive package, expecting the biggest bonus, but end up with a minimal or no bonus if the newly added is the smallest from the array.

```
function calculatePoints(uint256 _usdValue) public view returns (uint256 totalPoints, uint256 totalBonus) {
    uint256 remainingUsd = _usdValue;
    uint256 packages = pointsPackages.length;

    uint256 BUY_POINTS_PER_DOLLAR =
XLaunchConfig(xLaunch.getContract(XLaunchConfigName)).uint256ConfigToValue(
        XLaunchConfigLib.BUY_POINTS_PER_DOLLAR
    );

    // Continue buying packages while there's enough ETH
    while (remainingUsd > 0) {
        bool foundPackage = false;

        // Find the largest package that can be afforded
        for (uint256 i = packages; i > 0; i--) {
            uint256 packageIndex = i - 1;
            if (pointsPackages[packageIndex].packageCost <= remainingUsd) {
                // Add this package to calculation
                PointsPackage storage package = pointsPackages[packageIndex];

                // Deduct the cost from remaining USD
                totalPoints = totalPoints + package.pointsAmount;
                totalBonus = totalBonus + package.pointsBonus;
                remainingUsd -= package.packageCost;
                foundPackage = true;
                break;
            }
        }

        // If no package can be afforded, break the loop and calculate regular points
        if (!foundPackage) {
            uint256 totalRegularPoints = remainingUsd * BUY_POINTS_PER_DOLLAR / PRECISION;
            totalPoints = totalPoints + totalRegularPoints;
            remainingUsd = 0;
            break;
        }
    }
}
```

Recommendation: Reorder the `pointsPackages` when adding new package.

Resolution: Fixed

[H-03] tokens leaderboard isn't sorted and wrong tokens will be removed

Severity: High Risk

Description: When a token applies for a rank in the leaderboard and the count is `LEADERBOARD_SIZE`, the last, lowest-ranked token will be removed. This is wrong, because tokens aren't being sorted with the latest, up-to-date volumes, and there's a possibility of removing a token that is not the one with the lowest volume, per the `PositionManager::getVolumesForCurrentEpoch`.

Leaderboard array is sorted only on payout, which in the best-case scenario happens every 2 weeks:

```
function payout(address _token) external {//OK
    XLaunchConfig config = XLaunchConfig(xLaunch.getContract(XLaunchConfigName));
    uint256 payoutInterval = config.uint256ConfigToValue(XLaunchConfigLib.LEADERBOARD_PAYOUT_INTERVAL);
    uint256 minPayoutThreshold =
config.uint256ConfigToValue(XLaunchConfigLib.LEADERBOARD_PAYOUT_THRESHOLD);

    uint64 lastPayoutAgo = uint64(block.timestamp) - tokenToLastPayout[_token];
    require(
        lastPayoutAgo >= payoutInterval,
        XLeaderboard__PayoutIntervalNotPassed(
            _token, payoutInterval, lastPayoutAgo > payoutInterval ? 0 : payoutInterval - lastPayoutAgo
        )
    );
    tokenToLastPayout[_token] = uint64(block.timestamp);

    uint256 feesDeposited = tokenToFeesDeposited[_token];

    LeaderboardItem[] storage leaderboardItems = tokenToRankings[_token];
    _updateLeaderboardVolumes(leaderboardItems, _token);
    _sortLeaderboardItems(leaderboardItems);//<---- only time when leaderboard is sorted
```

And since volume is being modified through swapping in the `memetoken/baseToken` pair, there's 0 guarantee the volume order will be preserved after payout and before we add a new token to the leaderboard.

Also, if `_candidateToken` is already in the leaderboard, it will loop to search for it and order it based on its previous ones, but if the tokens after it are with more volume, the order will be wrong.

Recommendation: Invoke `_updateLeaderboardVolumes` and `_sortLeaderboardItems(leaderboardItems)` at the beginning of `_applyForRank`

Resolution: Fixed

[H-04] changing contracts in xLaunch will break functionality

Severity: High Risk

Description: There's no restriction the owner to alter any of the contracts after the protocols start operating:

```
function setContract(bytes32 _name, address _contract) external nonZeroAddress(_contract) onlyOwner {
    emit XContractChange(_name, contractRegistry[_name], _contract);
    contractRegistry[_name] = _contract;
}
```

If that's done, the storages will be messed up, values passed to functions also. For example, changing the XNFT contract will make it possible owners of the old NFTs to burn any of the new ones. They only need to match token ids.

Recommendation: Enforce restrictions, disallowing the `setContract` from being invoked after protocol's initialization.

Resolution: Acknowledged

[H-05] triggering CTO can be prevented with a dust swap

Severity: High Risk

Description: To trigger CTO, there must be inactivity in the main `baseToken/memetoken` pool for at least `CTO_INTERVAL` time. The problem with that is the fact that anyone can do swaps with minimal amount and prevent listing the memetoken as CTO.

```
function triggerCTO(address _token) external {
    XLaunchConfig xLaunchConfig = XLaunchConfig(xLaunch.getContract(XLaunchConfigName));
    uint256 timeSinceLastFeesPassed = block.timestamp
        -
IXPositionManager(payable(xLaunch.getContract(XPositionManagerName))).memetokenToLastTradedTimestamp(_token
);
    uint256 CTO_INTERVAL = xLaunchConfig.uint256ConfigToValue(XLaunchConfigLib.CTO_INTERVAL);
    FairLaunchStatus status =
XFairLaunch(xLaunch.getContract(XFairLaunchName)).getFairLaunchInfo(_token).status;
    require(
        timeSinceLastFeesPassed >= CTO_INTERVAL && status == FairLaunchStatus.CLOSED,
        XLeaderboard__InvalidTokenCTOTrigger()
    );
```

`memetokenToLastTradedTimestamp` is updated in the `_beforeSwap` function of the `PositionManager` hook:

```
function _beforeSwap(address, PoolKey calldata key, SwapParams calldata params, bytes calldata)
    internal
    override
    returns (bytes4, BeforeSwapDelta, uint24)
{
    // _beforeSwap for TruncGeoOracle
    _beforeSwap(key);

    IXLiquidityManager xLiquidityManager =
IXLiquidityManager(payable(xLaunch.getContract(XLiquidityManagerName)));
    XLaunchConfig xLaunchConfig = XLaunchConfig(xLaunch.getContract(XLaunchConfigName));
    IXLiquidityManager.CorePoolInfo memory poolInfo = xLiquidityManager.getCorePoolInfo(key.toId());
    memetokenToLastTradedTimestamp[poolInfo.memetoken] = uint64(block.timestamp);
```

Recommendation: Implement a more resilient requirement for triggering CTO, which cannot be blocked with dust swaps.

Resolution: Acknowledged

Medium severity

[M-01] AavePool address should have setter

Severity: Medium Risk

Description: `AavePool` address must be able to be set dynamically, since it gets changed with the new releases of the protocol and the liquidity is being migrated.

> See recent upgrade (AAVE v3 Governance proposal 252).

```
/// @notice Aave V3 Pool
IAavePool public pool;
```

If the current pool stops being supported, the deposits won't get any yield and even further, `deposit` can start reverting (due to setting supply cap to 0 or pausing the pool) it'll block the main BuilDefi operation - `_distributeMintAllocations`.

```
function _distributeMintAllocations(address _memetoken) internal {
    FairLaunchETHAllocations memory allocations =
        MemetokenInfoLibrary.calculateAllocation(allocationsBPS[_memetoken],
fairLaunchProgress[_memetoken].revenue);
...MORE CODE
    if (allocations.stakeInOtherPlatformAllocation > 0) {//OK 3
        IXExternalStaking(payable(memetokenToExternalStakingAdapter[_memetoken])).deposit{
            value: allocations.stakeInOtherPlatformAllocation
        }();
    }
```

Recommendation: Use Aave's `PoolDataProvider` fetch the pool. Broadly, anticipate AAVE changes that can impact the Farm wrapper. Monitoring of the governance forum along with on-chain monitoring is advised. The `CoreControlled` functionality of these Farms does assist in migrations if needed in the future. Don't forget to clear the approvals to old pool and give to the new one.

Resolution: Fixed

[M-02] AaveStaking deposit will fail when supply cap is reached

Severity: Medium Risk

Description: Staking into Aave can block the other operations in case the supply cap of the pool is reached. Since ETH will be supplied, supply cap can be reached relatively easy (currently it sits at around 80%).

If that happens and the launch has toggled that option, finalizing and distributing the ETH, collected from mints, won't be possible until Aave deposit passes through, either by other suppliers withdrawing or governance increasing the cap.

```
function _distributeMintAllocations(address _memetoken) internal {
    FairLaunchETHAllocations memory allocations =
        MemetokenInfoLibrary.calculateAllocation(allocationsBPS[_memetoken],
fairLaunchProgress[_memetoken].revenue);
...MORE CODE
    if (allocations.stakeInOtherPlatformAllocation > 0) {//OK 3
        IXExternalStaking(payable(memetokenToExternalStakingAdapter[_memetoken])).deposit{
            value: allocations.stakeInOtherPlatformAllocation
        }();
    }
```

Recommendation: Ensure there's enough room for the `allocations.stakeInOtherPlatformAllocation` in the respective Aave pool, otherwise forward the ETH, in order to being deposited on a latter stage.

Resolution: Fixed

[M-03] Aave incentive rewards will be lost

Severity: Medium Risk

Description: The `AaveStaking` contract supplies **`ETH`** into the Aave pool to earn yield. However, it does not implement any function to claim incentive rewards from **Aave's `RewardsController`**. As a result, any LP incentives accrued from Aave remain unclaimed and locked in the Aave pool, never being utilized.

This contradicts the intended design stated in the project documentation, which indicates that funds (including rewards) should be transferred to the `stakeYieldReceiver` address. Without an explicit claim mechanism, BuilDefi loses access to yield opportunities provided by Aave’s incentive system.

Recommendation: Introduce a function that allows claiming rewards from Aave's `RewardsController`.

Resolution: Fixed

[M-04] Affiliate can reject whitelisted purchase approval

Severity: Medium Risk

Description: Whitelisting can be done with Affiliates and the Affiliate creator gets a % of the whitelist price, but the payment to the Affiliate creator is done in `approvePurchasedWhitelist` and this opens up an issue where they can update their contract to revert to this call.

```
function approvePurchasedWhitelist(address _tokenAddress) external onlyXLaunchAuthorized {
    PurchaseWhitelistStatus memory purchaseStatus = tokenToPurchaseStatus[_tokenAddress];
    require(
        purchaseStatus.purchaseDeadline >= block.timestamp,
Whitelisting__TokenPurchaseWhitelistApprovalPassed()
    );
    _addToWhitelist(_tokenAddress, WhitelistType.PURCHASED);

    Affiliate storage affiliate = affiliates[purchaseStatus.affiliateCodeHash];
    address affiliateWallet = affiliate.wallet;

    uint256 affiliateAmount = 0;
    bool hasAffiliate = purchaseStatus.affiliateCodeHash != bytes32(0) && affiliate.active;
    if (hasAffiliate) {
        affiliateAmount = purchaseStatus.paymentAmount * affiliateIncentiveBPS / BPS;
    }

    if (affiliateAmount > 0) {
        affiliate.salesCount++;
        sendTokens(purchaseStatus.paymentToken, affiliateAmount, affiliateWallet); <------------------
        emit AffiliatePaid(affiliateWallet, affiliateAmount, purchaseStatus.paymentToken);
    }

    uint256 platformTokenAmount = purchaseStatus.paymentAmount - affiliateAmount;
    address platformGenesis = sendTokens(purchaseStatus.paymentToken, platformTokenAmount,
ZERO_ADDRESS); // ZERO_ADDRESS for platform genesis
    emit PlatformGenesisPaid(platformGenesis, platformTokenAmount, purchaseStatus.paymentToken);
}
```

This opens 2 issues:

- The creator of the affiliate program can abuse the buyers who used his code, even if he gets a % of the money, he can do it and thus the buyer has to recover his tokens and then start a new one, which will waste time.

- The affiliate creator can steal the money of all users who used his code. This will be done in combination with C-02

1. The Affiliate creator will create their userToPaymentTokenToBalance entry in the same way.

2. He will force buyers to refund the amount by make approvePurchasedWhitelist reverting, which will keep their tokens in the contract.

3. He will call the `refundPurchasedWhitelist` several times and then claim the users' money.

Recommendation: Make the affiliate creator to use a "pull over push" model in terms of fee transfer.

Resolution: Fixed

[M-05] Points holder can avoid burning his tokens on refund

Severity: Medium Risk

Description: When a fair launch is VOIDED, the ETH used for the launch can be refunded via `XFairLaunch.refundTokens()`, which the owner of the memecoin (creator) will call. The function also burns the Points that were minted upon creation. However, the points can be transferred to another address and `burnPointsOnRefund` will burn 0. This way, the creator will keep the Points.

```
function refundTokens(address _memetoken, address _receiver) external nonReentrant {
    ...

    // Burn points upon refund
    XPoints(payable(xLaunch.getContract(XPointsName))).burnPointsOnRefund(msg.sender,
ethRefund); <---------

    ...
}
```

```
function burnPointsOnRefund(address _user, uint256 _eth) external onlyAuthorized {
    uint256 points = _eth
        *
XLaunchConfig(xLaunch.getContract(XLaunchConfigName)).uint256ConfigToValue(XLaunchConfigLib.TOKEN_MINT_POIN
TS)
        / PRECISION;
    uint256 balance = balanceOf(_user);

    if (balance < points) {
        points = balance; <------- // 0
    }

    _burnPoints(_user, points); <------- burn 0
}
```

Recommendation: Add require these points to be burned or ask for a fee if he keeps them.

Resolution: Fixed

[M-06] LP allocation fees not claimed before increasing position in `LiquidityManager::addLP`

Severity: Medium Risk

Description: When adding to the LP, fees collected aren't claimed and won't be used in the `ADD_LIQUIDITY` operation.

```
function addLP(PairArgs calldata _args) external {
...MORE CODE
        xPosm.collectAndDistributeLPFees(corePoolInfo[state.poolKey.toId()], state.poolKey, false);
        _liquidityAction(
            xLiquidityManagementHelper.getAddLPLiquidityArgs(
                _args, state, baseTokenAmount, memetokenAmount, baseSSLpositionInfo, currentTick
            ),
            _poolAccounting
        );
    } else {
        xPosm.collectAndDistributeLPFees(corePoolInfo[state.poolKey.toId()], state.poolKey, false);

        _createBaseTokenSSL(_args, _poolAccounting, state);
    }
}
```

Instead, they'll sit in the `XLiquidityManager` until collected with `_collectFeesFromPositionManager` and positions are rebalanced.

```
function rebalanceLP(PairArgs calldata _args) external {
    require(block.timestamp <= _args.deadline, XLiquidityManager__Expired());
    PoolActionState memory state = _getPoolActionState(_args);
    PoolAccounting storage _poolAccounting = poolAccounting[_args.baseToken][_args.memetoken];

    require(corePoolInfo[state.poolId].memetoken != ZERO_ADDRESS, XLiquidityManager__PoolNotInitialized());
    require(
        block.timestamp - lastRebalanceTimestamp[state.poolId]
            >= XLaunchConfig(xLaunch.getContract(XLaunchConfigName)).uint256ConfigToValue(
                XLaunchConfigLib.REBALANCE_INTERVAL
            ),
        XLiquidityManager__IntervalNotReached()
    );

    // If base token in ETH, try swapping ETH for base token
    if (_poolAccounting.ethBalance > 0) {
        _swapETHForBaseToken(_poolAccounting, _args.baseToken, _poolAccounting.ethBalance);
    }

    lastRebalanceTimestamp[state.poolId] = uint64(block.timestamp);

    // Collect all fees from the positions before rebalance
    IXPositionManager xPositionManager =
IXPositionManager(payable(xLaunch.getContract(XPositionManagerName)));
    xPositionManager.collectAndDistributeLPFees(corePoolInfo[state.poolKey.toId()], state.poolKey, false);

    // Collect the fees from Buy & Sell tax or fees from the custom LP fee structure
    _collectFeesFromPositionManager(_args, xPositionManager);
```

```
function _collectFeesFromPositionManager(PairArgs calldata _args, IXPositionManager _xPositionManager)
internal {
    FeeInfo memory feeInfo = xLaunch.getFeeInfo(_args.memetoken);
    if ((feeInfo.lpTokenFeeCustomDistribution.length > 0 && feeInfo.lpTokenFeeCustomDistribution[0] > 0)) {
        uint256 baseTokenBalanceBefore = _getBalance(_args.baseToken);
        uint256 memetokenBalanceBefore = _getBalance(_args.memetoken);
        _xPositionManager.collectLiquidityManagerFees(_args.memetoken, _args.baseToken);
        PoolAccounting storage _poolAccounting = poolAccounting[_args.baseToken][_args.memetoken];
        _poolAccounting.baseTokenBalance =
            _poolAccounting.baseTokenBalance + (_getBalance(_args.baseToken) - baseTokenBalanceBefore);
        _poolAccounting.memetokenBalance =
            _poolAccounting.memetokenBalance + (_getBalance(_args.memetoken) - memetokenBalanceBefore);
    }
}
```

This is problematic as it doesn't enforce the Liquidity Engine efficiently, leaving tokens idling in the manager for prolonged time.

Recommendation: Invoke `_collectFeesFromPositionManager` after `xPositionManager.collectAndDistributeLPFees` in `addLP`.

Resolution: Fixed

[M-07] When swap fails with anon error all the exec reverts, halting the hook swaps

Severity: Medium Risk

Description: If any of the function used throughout the swap keeps failing with an unhandled exception, it will stop the entire pool from functioning. One notable example is the processing of an external token:

```
function _beforeSwap(address, PoolKey calldata key, SwapParams calldata params, bytes calldata)
    internal
    override
    returns (bytes4, BeforeSwapDelta, uint24)
{
 ...MORE CODE
    if (poolInfo.isExternal) {
        XPositionManagerLib.processExternalToken(
            poolInfo.baseToken, xLaunch, platformGenesisFees, tokenToFeesQueue, addressToETH
        );
    }

    return (this.beforeSwap.selector, toBeforeSwapDelta(int128(specifiedDelta), int128(unspecifiedDelta)),
0);
}
```

```
function processExternalToken(
      address _externalToken,
      XLaunch _xLaunch,
      mapping(address token => uint256 fees) storage _platformGenesisFees,
      mapping(address token => IXPositionManager.FeesQueueItem[]) storage _tokenToFeesQueue,
      mapping(address user => uint256 fees) storage _addressToETH
  ) public {
      SwapActions swapActions = SwapActions(payable(_xLaunch.getContract(SwapActionsName)));
      SafeERC20.safeIncreaseAllowance(
          IERC20(_externalToken), address(swapActions), IERC20(_externalToken).balanceOf(address(this))
      );
      uint256 totalAvailable = _platformGenesisFees[_externalToken];
      if (totalAvailable != 0) {
          (uint256 amountReceived, uint256 unusedAmount) =
              swapActions.swapExternalTokenForETH(_externalToken, totalAvailable, uint32(block.timestamp));
          if (unusedAmount == totalAvailable) {
              // If the external token has fee on transfer this check won't catch it
              return; // The slippage was not met even with very small swap amount
          }
          _platformGenesisFees[_externalToken] -= (totalAvailable - unusedAmount);
          _platformGenesisFees[address(0)] = _platformGenesisFees[address(0)] + amountReceived;
          emit XPositionManager.PlatformGenesisFeesFulfilled(_externalToken, amountReceived);
      }

      uint256 queueETHCollected = _loopProcessExternalToken(
          InternalProcessExternalLoopArgs({
              externalToken: _externalToken,
              xLeaderboard: XLeaderboard(payable(_xLaunch.getContract(XLeaderboardName))),
              xLaunch: _xLaunch,
              swapActions: swapActions
          }),
          _addressToETH,
          _tokenToFeesQueue
      );

      emit XPositionManager.FeesFulfilled(_externalToken, queueETHCollected);
  }
```

Since `externalToken` should be swapped for `ETH`, if there's insufficient liquidity in the pool, the while loop will iterate multiple times, eventually exhausting the gas and reverting the swap.

Also a huge amount of iterations, combined with the heavy logic of the **`beforeSwap()`** hook, will DoS the hook contract.

Recommendation: Place a max bound in all the while loops to prevent multiple iterations without result. Consider adding even more tests, ensuring there's no way hooks functionality to revert and halt the swaps.

Resolution: Fixed

[M-08] `_getTwapAmountV4Sequential` & `_calculateAmountOutMinimum` account the fee but fee is being taken 60 mins after pool deployment

Severity: Medium Risk

Description: Both `_getTwapAmountV4Sequential` & `_calculateAmountOutMinimum` apply the fee to the `amountIn/amountOut` respectively, but in reality, the 1st 60 minutes after the oldest observation aren't taxed.

This will result in a wrong outputs from the function, unfairly taking the buy and sell taxes into account without the need to:

```
function _calculateAmountOutMinimum(SwapArgs memory _args, IXLiquidityManager _xLiquidityManager)
    internal
    view
    returns (uint256 amountOutMinimum)
{
    uint128 amountIn = _args.core.amountIn;
    uint256 amountInWithTax = _args.core.isTokenOutExternal
        ? amountIn
        : amountIn
            -
xLaunch.memetokenToTokenBuySellTaxBPS(_xLiquidityManager.getCorePoolInfo(_args.poolId).memetoken) *
amountIn / BPS;
    uint256 twapAmount =
        getTwapAmount(_args.core.tokenIn, _args.core.tokenOut, _args.poolKey, amountInWithTax,
_args.isUniV4);

    uint256 slippage = 0.15e18; // 15% base slippage, we cannot have config for each pool

    amountOutMinimum = wmul(twapAmount, WAD - slippage);
}
```

```
function _getTwapAmountV4Sequential(address _tokenIn, uint256 _amountIn, PathKey[] memory _path, bool
quote)
    internal
    view
    returns (uint256 amountOutMinimum)
{
    uint256 length = _path.length;
    amountOutMinimum = _amountIn;
    uint256 slippage = 0.15e18; // 15% base slippage

    for (uint256 i = 0; i < length;) {
        address nextToken = Currency.unwrap(_path[i].intermediateCurrency);
        if (xLaunch.memetokenToTokenId(nextToken) != 0) {
            uint16 taxBps = xLaunch.memetokenToTokenBuySellTaxBPS(nextToken);
            amountOutMinimum = amountOutMinimum - (amountOutMinimum * taxBps) / BPS;
        }

        PoolKey memory poolKey = XUniswapLib.getPoolKey(_tokenIn, nextToken, address(_path[i].hooks));
        uint128 hopAmount = getTwapAmountV4(_tokenIn, nextToken, poolKey, amountOutMinimum);
        amountOutMinimum = quote ? uint256(hopAmount) : wmul(uint256(hopAmount), WAD - slippage);
        _tokenIn = nextToken;
        unchecked {
            ++i;
        }
    }
}
```

Recommendation: Place the same oldest observation check in both functions

```
if (
    V4OracleLibrary.getOldestObservationSecondsAgo(this, key)
        < uint32(xLaunchConfig.uint8ConfigToValue(XLaunchConfigLib.TWAP_LOOKBACK_MINUTES)) * 60
) {
  //No fee
}
```

Resolution: Acknowledged

[M-09] hardcoded 15% slippage will cause dos of the swap functions

Severity: Medium Risk

Description: Using a hardcoded slippage param is generally unsafe mechanism as it can completely block any operation, involving `_getTwapAmountV4Sequential` and `_calculateAmountOutMinimum` functions.

The reason is that 15% base slippage is a lot for bluechip pairs and LPs with high liquidity, but not enough for pairs with low liquidity.

Currently, it opens MEV opportunities, as even the tick-based liquidity is prone to manipulation in unstable pools. Furthermore, it can even block legit swaps if they're causing high price impact, but the protocol's functionality relying on that to pass is blocked, until enough liquidity enters the pool, decreasing the impact.

```
function _calculateAmountOutMinimum(SwapArgs memory _args, IXLiquidityManager _xLiquidityManager)
    internal
    view
    returns (uint256 amountOutMinimum)
{
    uint128 amountIn = _args.core.amountIn;
    uint256 amountInWithTax = _args.core.isTokenOutExternal
        ? amountIn
        : amountIn
            -
xLaunch.memetokenToTokenBuySellTaxBPS(_xLiquidityManager.getCorePoolInfo(_args.poolId).memetoken) *
amountIn / BPS;
    uint256 twapAmount =
        getTwapAmount(_args.core.tokenIn, _args.core.tokenOut, _args.poolKey, amountInWithTax,
_args.isUniV4);

    uint256 slippage = 0.15e18; // 15% base slippage, we cannot have config for each pool

    amountOutMinimum = wmul(twapAmount, WAD - slippage);
}
```

```
function _getTwapAmountV4Sequential(address _tokenIn, uint256 _amountIn, PathKey[] memory _path, bool
quote)
    internal
    view
    returns (uint256 amountOutMinimum)
{
    uint256 length = _path.length;
    amountOutMinimum = _amountIn;
    uint256 slippage = 0.15e18; // 15% base slippage

    for (uint256 i = 0; i < length;) {
        address nextToken = Currency.unwrap(_path[i].intermediateCurrency);
        if (xLaunch.memetokenToTokenId(nextToken) != 0) {
            uint16 taxBps = xLaunch.memetokenToTokenBuySellTaxBPS(nextToken);
            amountOutMinimum = amountOutMinimum - (amountOutMinimum * taxBps) / BPS;
        }

        PoolKey memory poolKey = XUniswapLib.getPoolKey(_tokenIn, nextToken, address(_path[i].hooks));
        uint128 hopAmount = getTwapAmountV4(_tokenIn, nextToken, poolKey, amountOutMinimum);
        amountOutMinimum = quote ? uint256(hopAmount) : wmul(uint256(hopAmount), WAD - slippage);
        _tokenIn = nextToken;
        unchecked {
            ++i;
        }
    }
}
```

Recommendation: Consider having per whitelisted token and per memetoken adjustable slippageBPS parameters.

Resolution: Fixed

[M-10] hardcoded twap lookback

Severity: Medium Risk

Description: Using the same `15` minute lookback period on all the pairs is extremely inefficient and can lead to price manipulations in the pair.

The inefficiency is visible when we have a pair with high liquidity, this lookback is a lot more than needed and the TWAP price will be lagging behind the real spot price. This will produce wrong TWAP results, leading to unfair slippage reverts and wrong amounts estimations. The most notable issue will be the fact that `XPositionManager::_wrapHandleMemetokenBuy` will be using wrong TWAP price and can either benefit or harm the users who're executing memetoken buy orders.

```
function getTwapSqrtPriceX96V4(PoolKey memory _poolKey) public view returns (uint160 twapSqrtPriceX96) {
      uint32 secondsAgo = _getTwapLookback();

      uint32 oldestObservationSecondsAgo =
          V4OracleLibrary.getOldestObservationSecondsAgo(ITruncGeoOracle(address(_poolKey.hooks)),
_poolKey);

      // If this hits, we need to increase the observationCardinality of the pool
      require(oldestObservationSecondsAgo >= secondsAgo, SwapActions__TooShortPoolCardinality());

      // Get the arithmetic mean tick from the oracle
      (int24 arithmeticMeanTick,) =
          V4OracleLibrary.consult(ITruncGeoOracle(address(_poolKey.hooks)), _poolKey, secondsAgo);

      // Get sqrt price from tick
      twapSqrtPriceX96 = TickMath.getSqrtPriceAtTick(arithmeticMeanTick);
  }

function _getTwapLookback() internal view returns (uint32) {
  return uint32(
      XLaunchConfig(xLaunch.getContract(XLaunchConfigName)).uint8ConfigToValue(
          XLaunchConfigLib.TWAP_LOOKBACK_MINUTES
      )
  ) * 60;
}
```

```
function _beforeSwap(address, PoolKey calldata key, SwapParams calldata params, bytes calldata)
    internal
    override
    returns (bytes4, BeforeSwapDelta, uint24)
{
   ...MORE CODE
    if (
        (params.zeroForOne && Currency.unwrap(key.currency1) == poolInfo.memetoken)
            || (!params.zeroForOne && Currency.unwrap(key.currency0) == poolInfo.memetoken)
    ) {
        if (params.amountSpecified < 0) {
            uint160 twapSqrtPriceX96 =
                SwapActions(payable(xLaunch.getContract(SwapActionsName))).getTwapSqrtPriceX96V4(key);
            (specifiedDelta, unspecifiedDelta) = _wrapHandleMemetokenBuy(
                specifiedDelta, unspecifiedDelta, params.amountSpecified, twapSqrtPriceX96, poolInfo
            );
        }
    }
```

This potentially wrong `twapSqrtPriceX96` will produce inaccurate `amountOfMemetokenToBuy` as well as wrong base/meme conversion:

```
function _wrapHandleMemetokenBuy(
    int256 oldSpecifiedDelta,
    int256 oldUnspecifiedDelta,
    int256 amountSpecified,
    uint160 twapSqrtPriceX96,
    IXLiquidityManager.CorePoolInfo memory poolInfo
) internal returns (int256 specifiedDelta, int256 unspecifiedDelta) {
    specifiedDelta = oldSpecifiedDelta;
    unspecifiedDelta = oldUnspecifiedDelta;

    // We skip the buying logic if the swap is exact output
    (uint256 amountBaseToTake, uint256 amountFulfilled) = XPositionManagerLib.handleMemetokenBuy(
        V4OracleLibrary.getQuoteForSqrtRatioX96(
            twapSqrtPriceX96, uint256(-(amountSpecified + specifiedDelta)), poolInfo.baseToken,
poolInfo.memetoken
        ),
        twapSqrtPriceX96,
        poolInfo,
        xLaunch,
        platformGenesisFees,
        tokenToFeesQueue,
        addressToETH,
        poolManager
    );
```

```
function handleMemetokenBuy(
    uint256 amountOfMemetokenToBuy,
    uint160 twapSqrtPriceX96,
    IXLiquidityManager.CorePoolInfo calldata poolInfo,
    XLaunch _xLaunch,
    mapping(address token => uint256 fees) storage _platformGenesisFees,
    mapping(address token => IXPositionManager.FeesQueueItem[]) storage _tokenToFeesQueue,
    mapping(address user => uint256 fees) storage _addressToETH,
    IPoolManager _poolManager
) public returns (uint256 amountBaseToTake, uint256 amountFulfilled) {
    // Fulfill from platform genesis
    (amountFulfilled, amountBaseToTake) =
        _fulfillFromPlatformGenesis(amountOfMemetokenToBuy, twapSqrtPriceX96, poolInfo,
_platformGenesisFees);

function _fulfillFromPlatformGenesis(
      uint256 amount,
      uint160 twapSqrtPriceX96,
      IXLiquidityManager.CorePoolInfo calldata poolInfo,
      mapping(address token => uint256 fees) storage _platformGenesisFees
  ) internal returns (uint256 amountFulfilled, uint256 amountBaseToTake) {
      uint256 totalAvailable = _platformGenesisFees[poolInfo.memetoken];
      address baseToken = poolInfo.baseToken;
      address memetoken = poolInfo.memetoken;

      if (totalAvailable == 0) return (0, 0);

      amountFulfilled = totalAvailable > amount ? amount : totalAvailable;
      amountBaseToTake =
          V4OracleLibrary.getQuoteForSqrtRatioX96(twapSqrtPriceX96, amountFulfilled, memetoken, baseToken);
      _platformGenesisFees[memetoken] -= amountFulfilled;
      _platformGenesisFees[baseToken] = _platformGenesisFees[baseToken] + amountBaseToTake;

      emit XPositionManager.PlatformGenesisFeesFulfilled(memetoken, amountFulfilled);
  }
```

Recommendation: Have a fine-grained TWAP lookback and be able to modify it for each memetoken separately. One option is to have multiple categories: `safe/moderate/dangerous` and based on the liquidity of the pools change the lookback.

Resolution: Fixed

[M-11] `_loopProcessExternalToken` distributes in LIFO order, incentivising latest queue receivers

Severity: Medium Risk

Description: `_loopProcessExternalToken` loops through `_tokenToFeesQueue`, but when the first item in the queue is swapped entirely, it will pop the first element and the last element will become first which continue looping backwards, leaving the first ones with delayed fulfilment.

```
function _loopProcessExternalToken(
    InternalProcessExternalLoopArgs memory _args, // To avoid stack too deep
    mapping(address user => uint256 fees) storage _addressToETH,
    mapping(address token => IXPositionManager.FeesQueueItem[]) storage _tokenToFeesQueue
) internal returns (uint256 queueETHCollected) {
    IXPositionManager.FeesQueueItem[] storage queue = _tokenToFeesQueue[_args.externalToken];
    if (queue.length == 0) return 0;

    uint256 processedItems;
    uint256 queueProcessSize =
XLaunchConfig(_args.xLaunch.getContract(XLaunchConfigName)).uint256ConfigToValue(
        XLaunchConfigLib.QUEUE_PROCESS_SIZE
    );

    while (queue.length > 0 && processedItems < queueProcessSize) {
        IXPositionManager.FeesQueueItem storage item = queue[0];
        (uint256 amountReceived, uint256 unusedAmount) =
            _args.swapActions.swapExternalTokenForETH(_args.externalToken, item.amount,
uint32(block.timestamp));

        if (unusedAmount == item.amount) {
            // If the external token has fee on transfer this check won't catch it
            break; // The slippage was not met even with very small swap amount
        }

        if (item.isLeaderboard) {
            _args.xLeaderboard.accountDepositToLeaderboardPool(item.receiver, amountReceived);
        } else {
            _addressToETH[item.receiver] = _addressToETH[item.receiver] + amountReceived;
        }
        item.amount -= (item.amount - unusedAmount);
        queueETHCollected = queueETHCollected + amountReceived;

        if (item.amount == 0) {
            queue[0] = queue[queue.length - 1]; <----------------------
            queue.pop();
        }

        unchecked {
            ++processedItems;
        }
    }
```

Recommendation: Change the item remove logic and continue with the the next in the queue.

Resolution: Acknowledged

[M-12] XSS attack on the unsanitized string inputs

Severity: Medium Risk

Description: Memetoken and XStaking's string arguments are prone to XSS attack, because no input sanitization is being done. Basically, token creators can input something like: `<img src/ onerror=alert(1)>` and `<svg/onload=alert("xss")>`. This will produce a reflected XSS on all websites that load the malicious SVG image. The consequence can be harmful for the protocol's image but not entirely critical for the end user as it only shows an annoying pop up.

```
function initialize(string calldata _name, string calldata _symbol, string calldata _tokenUri, XLaunch
_xLaunch)
    public
    initializer
{
    __ERC20_init(_name, _symbol);
    __ERC20Burnable_init();
    __ERC20Permit_init(_name);

    xLaunch = _xLaunch;
    tokenURI = _tokenUri;
    emit TokenURIUpdated(_tokenUri);
}
```

If an attacker creates an asset with a symbol containing the malicious javascript payload above, he could get a stored XSS on all websites that render his malicious SVG image, which is legitimately shown in BuilDefi. This could allow the attacker for example, to run a keylogger script to collect all inputs typed by a user including his password or to create a fake Metamask pop up asking a user to sign a malicious transaction.

Recommendation: Add an input sanitization to the string arguments in `Memetoken` and `XStaking`

Resolution: Fixed

[M-13] `_beforeSwap` can run out of gas

Severity: Medium Risk

Description: There's a high risk that swaps in the `baseToken/memetoken` pairs to run out of gas, blocking most of the memetoken operations that involve the usage of the before/afterSwap hook functions.

Here's the overview of all the operations:

1. `TruncGeoOracle` observation

2. Collecting the LP fee of all the 3 positions.

3. Apply buy/sell tax

4. Fulfill memetoken buy order from platform genesis and token fees queue.

5. Process external token's queue

6. Actual swap

7. Update the total and user volume

8. Apply buy/sell tax on the other token

Some the the operations also involve extensive `while` and `for` loops.

Recommendation: Consider extensively testing the gas usage with all the possible functionality branches being invoked.

Resolution: Fixed

[M-14] deltaForFeeCurrency uses already taxed amount for exactOutput swaps

Severity: Medium Risk

Description: deltaForFeeCurrency in the PositionManager::applyBuySellTaxAfterSwap will use already taxed amounts when exact output swaps are don.

The reason is that the fee is taken from the output currency in the beforeSwap function:

```
function applyBuySellTaxBeforeSwap(
      SwapParams calldata _params,
      PoolKey calldata _key,
      XLaunch _xLaunch,
      IPoolManager _poolManager
  ) public returns (int256 specifiedDelta) {
      // Get the fee currency based on the swap direction and amount sign
      //exact output: zeroForOne = true, amountSpecified > 0 -> _key.currency1 = feeCurrencyAddress
      //swap X ETH for 10 FIST amountSpecified == 10 FIST, after tax it becomes 10.1 FIST
      address feeCurrencyAddress =
          Currency.unwrap(((_params.amountSpecified < 0) == (_params.zeroForOne)) ? _key.currency0 :
_key.currency1);
```

As a result, the amountToSwap (amountSpecified in the Hooks contract of UniV4) will be increased with the specified delta, resulting in higher exact output.

```
function beforeSwap(IHooks self, PoolKey memory key, SwapParams memory params, bytes calldata hookData)
    internal
    returns (int256 amountToSwap, BeforeSwapDelta hookReturn, uint24 lpFeeOverride)
{
    amountToSwap = params.amountSpecified;
    if (msg.sender == address(self)) return (amountToSwap, BeforeSwapDeltaLibrary.ZERO_DELTA,
lpFeeOverride);

    if (self.hasPermission(BEFORE_SWAP_FLAG)) {
        bytes memory result = callHook(self, abi.encodeCall(IHooks.beforeSwap, (msg.sender, key, params,
hookData)));

        // A length of 96 bytes is required to return a bytes4, a 32 byte delta, and an LP fee
        if (result.length != 96) InvalidHookResponse.selector.revertWith();

        // dynamic fee pools that want to override the cache fee, return a valid fee with the override
flag. If override flag
        // is set but an invalid fee is returned, the transaction will revert. Otherwise the current LP fee
will be used
        if (key.fee.isDynamicFee()) lpFeeOverride = result.parseFee();

        // skip this logic for the case where the hook return is 0
        if (self.hasPermission(BEFORE_SWAP_RETURNS_DELTA_FLAG)) {
            hookReturn = BeforeSwapDelta.wrap(result.parseReturnDelta());

            // any return in unspecified is passed to the afterSwap hook for handling
            int128 hookDeltaSpecified = hookReturn.getSpecifiedDelta();

            // Update the swap amount according to the hook's return, and check that the swap type doesn't
change (exact input/output)
            if (hookDeltaSpecified != 0) {
                bool exactInput = amountToSwap < 0;
                amountToSwap += hookDeltaSpecified;//<------ amtSpecified +ve + hookDeltaSpecified +ve
                if (exactInput ? amountToSwap > 0 : amountToSwap < 0) {
                    HookDeltaExceedsSwapAmount.selector.revertWith();
```

Later on, this `swapDelta` is passed to the `afterSwap` function:

```
function afterSwap(
      IHooks self,
      PoolKey memory key,
      SwapParams memory params,
      BalanceDelta swapDelta,
      bytes calldata hookData,
      BeforeSwapDelta beforeSwapHookReturn
  ) internal returns (BalanceDelta, BalanceDelta) {
      if (msg.sender == address(self)) return (swapDelta, BalanceDeltaLibrary.ZERO_DELTA);

      int128 hookDeltaSpecified = beforeSwapHookReturn.getSpecifiedDelta();
      int128 hookDeltaUnspecified = beforeSwapHookReturn.getUnspecifiedDelta();

      if (self.hasPermission(AFTER_SWAP_FLAG)) {
          hookDeltaUnspecified += self.callHookWithReturnDelta(
              abi.encodeCall(IHooks.afterSwap, (msg.sender, key, params, swapDelta, hookData)),
              self.hasPermission(AFTER_SWAP_RETURNS_DELTA_FLAG)
          ).toInt128();
      }—
```

But the problem is that the `swapDelta`'s `amount0` (input assets) reflects the tokens needed to satisfy `amountSpecified + deltaSpecified` (required + taxed amount) and that delta is used to calculate the unspecified fee delta:

```
function applyBuySellTaxAfterSwap(
    SwapParams calldata _params,
    PoolKey calldata _key,
    BalanceDelta _delta,
    XLaunch _xLaunch,
    IPoolManager _poolManager
) public returns (int128 hookUnspecifiedDelta) {
    // Get the fee currency based on the swap direction and amount sign
    address feeCurrencyAddress =
        Currency.unwrap(((_params.amountSpecified < 0) == (_params.zeroForOne)) ? _key.currency1 :
_key.currency0);

    uint16 taxBps = _xLaunch.memetokenToTokenBuySellTaxBPS(feeCurrencyAddress);
    if (taxBps == 0) return hookUnspecifiedDelta;

    // After-swap tax applies to the output token amount. Use _delta to get the actual output amount.
    int128 deltaForFeeCurrency =
        feeCurrencyAddress == Currency.unwrap(_key.currency0) ? _delta.amount0() : _delta.amount1();
    uint256 swapAmount = deltaForFeeCurrency < 0 ? uint128(-deltaForFeeCurrency) :
uint128(deltaForFeeCurrency);

    hookUnspecifiedDelta = int128(int256((swapAmount * taxBps) / 100_00));
```

Recommendation: take fee only from the unspecified amount in the afterSwap function.

Resolution: Acknowledged

Low severity

[L-01] Missmatches between documentation and code

Severity: Low Risk

Description: There are some mismatches between the official documentation and the actual implementation:

1. 28% instead of 8% can be taken from ETH refunds. Docs say 8% constant. Either fix the docs or use different parameter for refundFee BPS.

```
function refundTokens(address _memetoken, address _receiver) external nonReentrant {//OK
    FairLaunchInfo storage info = fairLaunchInfo[_memetoken];
    require(info.status == FairLaunchStatus.VOIDED, XFairLaunch__FairLaunchNotVoided());

    // Calculate the refund amoints
    uint256 userTokenBalance = Memetoken(_memetoken).balanceOf(msg.sender);
    uint256 ethRefund = userTokenBalance / info.mintPrice;
    uint256 platformLeaderboardPoolFee = (ethRefund * info.additionallyMintedTokensForLPBPS) / BPS;
    ethRefund = ethRefund - platformLeaderboardPoolFee;
```

```
function validateAdditionallyMintedTokensForLPBPS(LaunchOptions memory _opts) private view {//OK
    // Set the default allocation if not set
    if (_opts.additionallyMintedTokensForLPBPS == 0) {
        _opts.additionallyMintedTokensForLPBPS =
            configToBPS[XLaunchConfigLib.DEFAULT_ADDITIONALLY_MINTED_TOKENS_FOR_LP_BPS];
            //QA: return early here
    }

    // Check if the additionally minted LP tokens are in range
    uint16 minTokensForLP = configToBPS[XLaunchConfigLib.MIN_ADDITIONALLY_MINTED_TOKENS_FOR_LP_BPS];//8%
    uint16 maxTokensForLP = configToBPS[XLaunchConfigLib.MAX_ADDITIONALLY_MINTED_TOKENS_FOR_LP_BPS];//28%
    require(
        _opts.additionallyMintedTokensForLPBPS >= minTokensForLP
            && _opts.additionallyMintedTokensForLPBPS <= maxTokensForLP,
        XLaunchConfig__InvalidAdditionallyMintedTokensForLP(
            _opts.additionallyMintedTokensForLPBPS, minTokensForLP, maxTokensForLP
        )
    );
}
```

2. Ouroboros isn't supported as stake platform

Docs mention that Ouroboros is supported already, but in the code only Aave is available.

```
enum StakeYieldPlatform {
    NONE,
    // OROBOROUS,
    // PENDLE,
    // LIDO
    AAVE
}
```

3. Wrong hardcoded base token leaderboard fee

Docs ('FREE' Platform Created Token LP Allocations image) mention that the fee must be adjustable between 10%-30%, but it's 30% constant now.

```
function collectAndDistributeLPFees(
    IXLiquidityManager.CorePoolInfo memory _info,
    PoolKey memory _key,
    bool _unlocked
) public {
    ...MORE CODE
        uint256 baseTokenPlatformLeaderboard = 30_00 * (baseTokenFees - baseTokenFeesDistributed) / BPS;

        _addToFeesQueue(_info.baseToken, ZERO_ADDRESS, true, baseTokenPlatformLeaderboard, xLeaderboard);
        baseTokenFeesDistributed = baseTokenFeesDistributed + baseTokenPlatformLeaderboard;

        // Fees in Base for the BoT leaderboard
        _addToFeesQueue(
            _info.baseToken, feeInfo.tokenBuiltOn, true, baseTokenFees - baseTokenFeesDistributed,
xLeaderboard
        );

        // Fees in Memetoken for the memetoken leaderboard
        _addToFeesQueue(
            _info.memetoken, _info.memetoken, true, memetokenFees - memetokenFeesDistributed, xLeaderboard
        );
    }
}
```

4. Reflections are missing as allocation target.

Docs mention that `Reflections` are valid targets, but the `BuySellTaxAllocation` enum is missing the option.

```
enum BuySellTaxAllocation {
    TOKEN_CREATOR,
    ADDITIONAL_SUPPORT,
    STAKE_IN_ANOTHER_PLATFORM,
    BURN_TOKEN,
    ADD_TO_LP,
    STAKING_PAYOUT_POOL,
    TOKEN_LEADERBOARD_POOL
}
```

Recommendation: Revisit the documentation and the code, and fix all the inconsistencies.

Resolution:
1. Fixed
2. For now only Aave is supported
3. It is automatically adjusted based on the custom distribution, it is not a manual range.
4. Reflections are not implemented for now

[L-02] onlyXLaunch modifier is enough in several places

Severity: Low Risk

Description: In many places, `onlyXLaunch` modifier is sufficient instead of checking for the `XLeaderboard`, `XFairlaunch`, `PositionManager`, `owner`, etc.

These are few of the functions: XPoints.mintPointsForCreateToken(), XCloner.cloneMemetoken(), XCloner.cloneXStaking(), XCloner.cloneExternalStaking(), etc.

Recommendation: If the function is only called from one source, there is no need for multiple msg.sender checks.

Resolution: Fixed

[L-03] Longer boosting tiers are not incetivizing

Severity: Low Risk

Description: XBoost allows users to boost their token on the platform, but this does not incentivize them to buy a larger boost, as buying the cheapest one multiple times will give them the same duration for less money.

```
function initialize(address _xLaunch) public initializer {
    __XPayments_init(_xLaunch);

    _addTier(4 hours, 49);
    _addTier(8 hours, 124);
    _addTier(16 hours, 199);
    _addTier(24 hours, 449);
    _addTier(1 weeks, 1999);
}
```

Recommendation: Make the expensive ones to give more duration compared to multiple stacked cheap ones.

Resolution: Fixed

[L-04] Weird ERC20 tokens not supported

Severity: Low Risk

Description:

1. There's an incorrect assumptions across the entire codebase that tokens that take fees on transfer are supported, but that won't be possible because both UniswapV3 and V4 don't support such tokens.

2. Another instance where tokens that revert on 0 value approval will cause issues is the `XLiquidityManager::_swapDirect`:

```
function _swapDirect(PoolAccounting storage _poolAccounting, CorePoolInfo memory _corePoolInfo, uint256
_amount)
    internal
    returns (uint256 amountReceived, uint256 unusedAmount)
{
    SwapActions swapActions = SwapActions(payable(xLaunch.getContract(SwapActionsName)));
    bool isETH = _corePoolInfo.baseToken == ZERO_ADDRESS;
    if (!isETH) {
        SafeERC20.safeIncreaseAllowance(IERC20(_corePoolInfo.baseToken), address(swapActions), _amount);
    }

    (amountReceived, unusedAmount) = swapActions.swapDirect{value: isETH ? _amount : 0}(
        _corePoolInfo.baseToken, _corePoolInfo.memetoken, uint128(_amount)
    );
    _poolAccounting.baseTokenBalance -= (_amount - unusedAmount);
    _poolAccounting.memetokenBalance += amountReceived;
    if (!isETH) {
        SafeERC20.forceApprove(IERC20(_corePoolInfo.baseToken), address(swapActions), 0);
    }
}
```

One token, but not limited to, is BNB, which reverts on approval with 0. Such a token will revert all the operations involving `_swapDirect`.

Recommendation:

1. Don't whitelist such tokens.

2. Explicitly mention in the docs these tokens aren't supported.

3. Remove the logic from the codebase that accounts for fees.

Resolution: Fixed

[L-05] Protocol assumes all the stablecoins are pegged

Severity: Low Risk

Description: All the supported USD stablecoins are priced the same without fetching their real price from an oracle. But there's a risk of 1 token to depeg and deviate from the intended $1 = 1 token. If token falls below that, users using it will profit from it, resulting in a decreased revenue for the protocol.

```
function pay(address _paymentToken, uint256 _usdAmount, address _receiver, bool _refundExcess)
    internal
    returns (address receiver, uint256 tokenAmount)
{
...MORE CODE
    else {
        require(msg.value == 0, XPayments__MsgValueShouldNotBeProvided());
        uint256 tokenDecimals = ERC20(_paymentToken).decimals();
        tokenAmount = (_usdAmount * (10 ** tokenDecimals)) / 1e18;
        uint256 receiverBalanceBefore = IERC20(_paymentToken).balanceOf(receiver);
        IERC20(_paymentToken).safeTransferFrom(msg.sender, receiver, tokenAmount);
        tokenAmount = IERC20(_paymentToken).balanceOf(receiver) - receiverBalanceBefore;
    }

    emit PaymentReceived(receiver, isPlatformGenesis, _paymentToken, tokenAmount);
}
```

Recommendation: Consider introducing an oracle and pricing the stablecoins accordingly.

Resolution: Acknowledged

[L-06] Sequencer downtime can go unnoticed

Severity: Low Risk

Description: The `startedAt` variable returns 0 only on Arbitrum when the Sequencer Uptime contract is not yet initialized.

https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-consumer-contract

Therefore, the `timeSinceUp` will falsely pass if protocol is deployed on Arbitrum and sequencer isn't initialized yet.

```
function verifySequencerIsUp() private view {
    // If not on L2 skip the check
    if (address(sequencer) == ZERO_ADDRESS) {
        return;
    }

    (
        /* uint80 roundId */
        ,
        int256 answer,
        uint256 startedAt,
        /* uint256 updatedAt */
        ,
        /* uint80 answeredInRound */
    ) = sequencer.latestRoundData();

    bool isSequencerUp = answer == 0;
    require(isSequencerUp, SequencerDown());

    uint256 timeSinceUp = block.timestamp - startedAt;
    require(
        timeSinceUp
            >= XLaunchConfig(xLaunch.getContract(XLaunchConfigName)).uint256ConfigToValue(
                XLaunchConfigLib.MIN_SEQUENCER_UPTIME
            ),
        SequencerPending()
    );
}
```

Recommendation: If you intend to deploy on Arb, consider handling the `startedAt == 0` scenario.

Resolution: Fixed

[L-07] Simplification of launch flow

Severity: Low Risk

Description:

1. internal functions, invoked in `validateLaunchOptions` of `LaunchConfig` can return early in case default settings are chosen:

```
function validateFairLaunchDuration(LaunchOptions memory _opts) private view {
    // Set the default duration if not set
    if (_opts.fairLaunchDuration == 0) {
        _opts.fairLaunchDuration =
uint64(uint256ConfigToValue[XLaunchConfigLib.DEFAULT_FAIR_LAUNCH_DURATION]);
        //QA: return early here
    }

function validateAdditionallyMintedTokensForLPBPS(LaunchOptions memory _opts) private view {//OK
    // Set the default allocation if not set
    if (_opts.additionallyMintedTokensForLPBPS == 0) {
        _opts.additionallyMintedTokensForLPBPS =
            configToBPS[XLaunchConfigLib.DEFAULT_ADDITIONALLY_MINTED_TOKENS_FOR_LP_BPS];
            //QA: return early here
  }

function validateMintPrice(LaunchOptions memory _opts) private view {//OK
    // Set the default mint price if not set
    if (_opts.mintPrice == 0) {
        _opts.mintPrice = uint128(uint256ConfigToValue[XLaunchConfigLib.DEFAULT_MINT_PRICE]);
        //QA: return early here
    }
```

2. Remove the double checks in `validateBuySellTax` allocations. `if` branch already enforces the allocations to belong to the right target.

```
function validateBuySellTax(LaunchOptions memory _opts) private view {//OK
    if (_opts.buySellTaxBPS != 0) {
        require(_opts.buySellTaxAllocations.length > 0, XLaunchConfig__InvalidBuySellTaxAllocations());
        require(
            _opts.allocationsBuySellTaxBPS.length == _opts.buySellTaxAllocations.length,
            XLaunchConfig__InvalidBuySellTaxAllocations()
        );
    } else {
        require(_opts.buySellTaxAllocations.length == 0, XLaunchConfig__InvalidBuySellTaxAllocations());
        require(_opts.allocationsBuySellTaxBPS.length == 0, XLaunchConfig__InvalidBuySellTaxAllocations());
        return;
    }

    bool additionalSupportAllocatinInvalid = false;
    bool stakeInAnotherPlatformInvalid = false;
    bool stakingPoolAllocationInvalid = false;

    for (uint256 i = 0; i < _opts.buySellTaxAllocations.length;) {
        BuySellTaxAllocation buySellTaxAllocation = _opts.buySellTaxAllocations[i];
        //QA: simplify this, by removing the double checks
        if (buySellTaxAllocation == BuySellTaxAllocation.ADDITIONAL_SUPPORT) {
            additionalSupportAllocatinInvalid = buySellTaxAllocation ==
BuySellTaxAllocation.ADDITIONAL_SUPPORT
                && _opts.additionalSupportAllocationBPS == 0;
        } else if (buySellTaxAllocation == BuySellTaxAllocation.STAKE_IN_ANOTHER_PLATFORM) {
            stakeInAnotherPlatformInvalid = buySellTaxAllocation ==
BuySellTaxAllocation.STAKE_IN_ANOTHER_PLATFORM
                && _opts.stakeInPlatformAllocationBPS == 0;
        } else if (buySellTaxAllocation == BuySellTaxAllocation.STAKING_PAYOUT_POOL) {
            stakingPoolAllocationInvalid = buySellTaxAllocation == BuySellTaxAllocation.STAKING_PAYOUT_POOL
                && !_opts.stakeTokenForRewardsInToken;
        }
        unchecked {
            ++i;
        }
    }
```

Recommendation: Consider applying the changes. This will reduce the code size and decrease gas cost.

Resolution: Fixed

[L-08] _applyForRank does redundant operation

Severity: Low Risk

Description: _applyForRank manually pushes new token to the end of the tokenRankings array, instead of invoking _pushToEnd:

```
function _applyForRank(address _candidateToken) internal {//OK
...MORE CODE
    } else if (totalVolume > tokenRankings[length - 1].totalVolume) {
        // Replace the lowest-ranked token if the candidate qualifies
        leaderboardTokenToIsInLeaderboard[_tokenLeaderboard][tokenRankings[length - 1].token] = false;
        tokenRankings[length - 1] =//QA: use pushToEnd
            LeaderboardItem({token: _candidateToken, baseVolume: baseVolume, totalVolume: totalVolume});
        leaderboardTokenToIsInLeaderboard[_tokenLeaderboard][_candidateToken] = true;
        _bubbleUp(tokenRankings, length - 1);
    }

    require(getTokenIsInLeaderboard(_tokenLeaderboard, _candidateToken), XLeaderboard__LeaderboardFull());
}
```

Recommendation: Remove the manual push and replace it with _pushToEnd.

Resolution: Acknowledged

[L-09] duplicate tax allocation targets can be passed

Severity: Low Risk

Description: In `TaxDistributor::setTaxes` there's a missing check whether the destinations array contains duplicates. As a result, token creator can assign single target many times.

```
function setTaxes(address memetoken, BuySellTaxAllocation[] calldata destinations, uint16[] calldata bpss)
    external
{
    //QA: duplicate destination check missing
    require(msg.sender == address(xLaunch), XTaxDistributor__CallerNotXLaunchOwner());

    for (uint256 i = 0; i < destinations.length;) {
        BuySellTaxAllocation destination = destinations[i];
        uint16 bps = bpss[i];
        memetokenTaxBreakdown[memetoken][destination] = bps;
        unchecked {
            ++i;
        }
        emit TaxesSet(memetoken, destination, bps);
    }
}
```

If that's happens intentionally or not only the last occurrence will be assigned.

Recommendation: Disable the ability to assign same duplicate, as it will be overriding the mapping continuously, resulting in inability to distribute 100% of the taxes later on.

Resolution: Fixed

[L-10] Anyone can stop memetoken from graduating

Severity: Low Risk

Description: `_distributeMintFee` doesn't revert on failure, but malicious recipient can still consume all the available gas and halt the transaction execution, causing an OOG exception.

```
function _distributeMintFee(uint256 _amount, address _receiver) internal returns (uint256 leftover) {
    leftover = _amount;
    if (_amount > 0) {
        // wake-disable-next-line
        (bool success,) = payable(_receiver).call{value: _amount}(""); //QA: anyone can consume the GAS
        if (success) {
            leftover = 0;
        }
    }
}
```

Recommendation: Use pattern which reserves the mint fees for their recipients.

Resolution: Fixed

Notes

Severity: Informational Risk

Description:

1. `_getBalance` in SwapActions isn't used across the entire contract. The idea is to abstract the external functions.

2. in `handleMemetokenBuy`, if `amountOfMemetokenToBuy` is filled after platform genesis, directly take from the `poolManager`.

3. In the BNB's `burn` functions, move the incentive distribution last.

4. Add check in the fee collection to be possible only if there's enough amount accumulated to be worth it, otherwise skip collecting in the `_beforeSwap` functions, because for 5 swaps there won't be enough fee.

5. There are unused imports across the codebase:

- `console2` in `TaxDistributor`

- `UUPSUpgradeable` (this should be inherited, not removed), `Common` and `XLeaderboard` in `XBoost`

Resolution: Fixed