SALUS SECURITY

AUG 2025

CODE SECURITY ASSESSMENT

APEX

Overview

Project Summary

- Name: APEX
- Platform: The BSC Blockchain
- Language: Solidity
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	APEX
Version	v2
Type	Solidity
Dates	Aug 10 2025
Logs	Aug 08 2025; Aug 10 2025

Vulnerability Summary

Total High-Severity issues	2
Total Medium-Severity issues	5
Total Low-Severity issues	5
Total informational issues	3
Total	15

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	AutoSwapBurn mechanism is flawed	High	Business logic	Resolved
2	Unstake function is broken	High	Business logic	Resolved
3	DoS attack against specific users	Medium	Denial of Service	Resolved
4	Flashloan protection can be bypassed	Medium	Business logic	Resolved
5	ToggleAllAutoRenew has logic error	Medium	Business logic	Resolved
6	EndTime miscalculation in toggleAutoRenew	Medium	Business logic	Resolved
7	The autorenew user can unstake	Medium	Business logic	Resolved
8	Incorrect calculation due to overridden totalsupply	Low	Business logic	Resolved
9	GetFuturePrice() ignores current pricing configuration	Low	Business logic	Resolved
10	GetSwapBurnInfo() misreports totalBurned	Low	Business logic	Resolved
11	ToggleAutoRenew lacks per-stake granularity	Low	Business logic	Resolved
12	Centralization risk	Low	Centralization	Resolved
13	Missing events for functions that change critical state	Informational	Logging	Resolved
14	Gas optimization suggestions	Informational	Gas Optimization	Resolved
15	Use of floating pragma	Informational	Configuration	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. AutoSwapBurn mechanism is flawed

Severity: High	Category: Business logic
Target: - APEX.sol	

Description

In PancakeSwap, when users swap, flashloan, add, or remove liquidity, the system uses a callback mechanism to check the pool's balance at the end of the transaction, , as shown in the code below.

pancakeswap/v3-core/contracts/PancakeV3Pool.sol:L803-L808

```
if (zeroForOne) {
   if (amount1 < 0) TransferHelper.safeTransfer(token1, recipient, uint256(-amount1));
   uint256 balance0Before = balance0();
   IPancakeV3SwapCallback(msg.sender).pancakeV3SwapCallback(amount0, amount1, data);
   require(balance0Before.add(uint256(amount0)) <= balance0(), 'IIA');
```

The purpose of `_detectAndHandleSwapBurn` is to detect and handle token swap transactions occurring in the exchange pool, which automatically triggers the smart burn mechanism. But this mechanism leads to the checks not passing.

APEX.sol:L459-L468

```
function _detectAndHandleSwapBurn(address from, address to, uint256 value) internal {
    if (swapPools[to] && _isValidSwapAmount(value)) {
        _executeSmartBurn(to, value, from, to, true);
    }
    else if (swapPools[from] && _isValidSwapAmount(value)) {
        _executeSmartBurn(from, value, from, to, false);     }
}
function _executeSmartBurn(address swapPool, uint256 amount, address from, address /* to
*/, bool isSellOrder) internal {
    ...
    if (burnAmount > poolBalance) {
        burnAmount = poolBalance;
    }

    if (burnAmount > 0) {
        _burn(swapPool, burnAmount);
        totalAutoBurnedAmount += burnAmount;

        emit AutoSwapBurnTriggered(from, swapPool, amount, burnAmount, isSellOrder);
        emit TokensBurned(swapPool, burnAmount);
    }
```

Attach Scenario:
For a simple swap, the call stack is:
swapRouter.exactInputSingle -> pool.swap() -> pancakeV3SwapCallback(swapRouter) -> apex.transferFrom(msg.sender, pool) -> apex.update -> apex._detectAndHandleSwapBurn -> burn(pool, 50% of swap amount) -> pool's balance check failed

Proof of Concept:

```
function test_swap() public {
        apex.toggleAutoSwapBurn(true);
        apex.setSwapPool(poolAddress, true);
        apex.setSwapThresholds(1e18, 1000e18);
        apex.approve(address(swapRouter), 1000e18);
        ISwapRouter.ExactInputSingleParams memory params =
ISwapRouter.ExactInputSingleParams({
            tokenIn: address(apex),
            tokenOut: address(usdt),
            fee: 500,
            recipient: address(this),
            deadline: block.timestamp + 1000,
            amountIn: 10e18,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        });
        vm.expectRevert(bytes("IIA"));
        swapRouter.exactInputSingle(params);

    }
```

Recommendation

Refactor the `AutoSwapBurn` design to accommodate Pancake's callback mechanism.

Status

This issue has been resolved by the team.

2. Unstake function is broken

Severity: High	Category: Business logic
Target: - APEXStaking.sol.sol	

Description

The `unstake()` function enforces users must wait until the current staking round is complete before they can unstake. This is designed to prevent premature withdrawals from negatively affecting the staking mechanism's economic model.

APEXStaking.sol.sol:L337-L386

```
function unstake(uint256 stakeIndex) external nonReentrant {
    ...
    if (targetStake.stakeType == StakeType.SEVEN_DAYS) {
        uint256 currentRound = (stakeDuration / ROUND_DURATION) + 1;
        uint256 currentRoundEndTime = targetStake.startTime + (currentRound *
ROUND_DURATION);

        require(block.timestamp >= currentRoundEndTime, "Cannot unstake before current
round ends");
    } else {
        uint256 currentRound = (stakeDuration / YEAR_DURATION) + 1;
        uint256 currentRoundEndTime = targetStake.startTime + (currentRound *
YEAR_DURATION);

        require(block.timestamp >= currentRoundEndTime, "Cannot unstake before current
round ends");
    }
    ...
    );
}
```

The code calculates the current round using the formula `(stakeDuration / ROUND_DURATION) + 1`, which contains a logical error.

Specifically, when stakeDuration is an exact multiple of `ROUND_DURATION`, the addition of +1 causes an incorrect round number. For example, if the staking duration is 7 days and `ROUND_DURATION` is also 7 days, the formula calculates the current round as 2, when it should correctly be 1.

As a result, users are consistently unable to unstake their assets..

Recommendation

Fixed the `currentRoundEndTime` check like remove `+1`.

Status

This issue has been resolved by the team.

3. DoS attack against specific users

Severity: Medium	Category: Denial of Service
Target: - APEX.sol	

Description

The `_checkFlashLoanProtection` function serves to prevent flash loan attacks. It does this by limiting the number of transactions and the transfer amount within each block to ensure the contract's security.

APEX.sol:L431-L510

```
function _update(
    address from,
    address to,
    uint256 value
) internal virtual override {
    require(!paused(), "ERC20Pausable: token transfer while paused");

    if (flashLoanProtectionEnabled && from != address(0) && to != address(0)) {
        if (!stakingContracts[from] && !stakingContracts[to]) {
            _checkFlashLoanProtection(from, value);
        }
    }
    ...
}
function _checkFlashLoanProtection(address user, uint256 amount) internal {
    if (stakingContracts[user]) {
        return;
    }

    uint256 currentBlock = block.number;

    if (_lastTransactionBlock[user] != currentBlock) {
        _transactionCountPerBlock[user] = 1;
        _cumulativeTransferAmount[user] = amount;
        _lastTransactionBlock[user] = currentBlock;
    } else {
        _transactionCountPerBlock[user]++;
        _cumulativeTransferAmount[user] += amount;
    }
    if (_transactionCountPerBlock[user] > maxTransactionsPerBlock) {
        emit RateLimitTriggered(user, _transactionCountPerBlock[user], currentBlock);
        revert("Rate limit: Too many transactions per block");
    }
    if (_cumulativeTransferAmount[user] > maxTransferAmountPerBlock) {
        emit FlashLoanSuspected(user, amount, currentBlock);
        revert("Flash loan suspected: Transfer amount exceeds limit");
    }
}
```

The OpenZeppelin ERC20 standard allows zero-amount transferFrom. Therefore, it's possible to forge transferFrom transactions to cause a specific user's _transactionCountPerBlock to reach its limit, thereby achieving the goal of a DOS attack. The attack has a relatively high cost; however, it has the potential to block key components of the protocol, including the foundation.

OpenZeppelin/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol#L42-147

```
function transferFrom(address from, address to, uint256 value) public virtual returns
(bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, value);
    _transfer(from, to, value);
    return true;
}
```

Recommendation

Fix the `update` function to prevent zero-value transfers.

Status

This issue has been resolved by the team.

4. Flashloan protection can be bypassed

Severity: Medium	Category: Business logic
Target: - APEX.sol	

Description

The `_checkFlashLoanProtection()` enforces `maxTransactionsPerBlock` and `maxTransferAmountPerBlock` only on the sender (from) side. Because the contract does not aggregate totals for the receiver (to) side, an attacker may request multiple flash-loans from different pools that deliver funds directly to a swarm of fresh addresses, letting each address stay below the per-block limits while the combined value far exceeds them.

APEX.sol:L418-L440

```
function _update(address from, address to, uint256 value) internal virtual override {
    require(!paused(), "ERC20Pausable: token transfer while paused");
    if (flashLoanProtectionEnabled && from != address(0) && to != address(0)) {
        if (!stakingContracts[from] && !stakingContracts[to]) {
            _checkFlashLoanProtection(from, value);
        }
    }
    ...
}
```

APEX.sol:L463-L496

```
function _checkFlashLoanProtection(address user, uint256 amount) internal {
    ...
    if (_transactionCountPerBlock[user] > maxTransactionsPerBlock) {
        emit RateLimitTriggered(user, _transactionCountPerBlock[user], currentBlock);
        revert("Rate limit: Too many transactions per block");
    }
    if (_cumulativeTransferAmount[user] > maxTransferAmountPerBlock) {
        emit FlashLoanSuspected(user, amount, currentBlock);
        revert("Flash loan suspected: Transfer amount exceeds limit");
    }
}
```

Attach Scenario:
In a single transaction the attacker can, for example, orchestrate ten 100,000 APEX flash-loans to ten newly deployed receivers, manipulate downstream protocols with an effective 1000,000 APEX position, and repay within the same transaction—all without triggering the existing checks. The scheme also defeats the per-block transaction-count limit because each receiver performs only one transfer.

The block-level caps fail to limit the aggregate volume a single EOA can mobilize within one block or one transaction. Price-manipulation that relies on large, transient token balances remain feasible despite the protection logic.

Recommendation

Track per-block cumulative amounts for both from and to addresses so that high-volume inflows are also rate-limited. Or enforce a single-transaction cumulative cap keyed to tx.origin to stop multi-address split operations inside the same call graph.

Status

This issue has been resolved by the team.

5. ToggleAllAutoRenew has logic error

Severity: Medium	Category: Business logic
Target: - APEXStaking.sol	

Description

The `toggleAllAutoRenew()` function is designed to toggle the auto-renewal status for all staking types (`SEVEN_DAYS`, `ONE_YEAR`, and `NODE`) in a single operation.

APEXStaking.sol.sol:L838-L891

```
function toggleAutoRenew(StakeType stakeType) external nonReentrant {...}
function toggleAllAutoRenew() external nonReentrant {
    this.toggleAutoRenew(StakeType.SEVEN_DAYS);
    this.toggleAutoRenew(StakeType.ONE_YEAR);
    this.toggleAutoRenew(StakeType.NODE);
}
```

Proof of Concept:

1. It can't call another `nonReentrant` function which will cause a revert.
2. According to the `this` keyword, the `msg.sender` will be the `apexStaking` contract, which can't change the actual user's config.

Recommendation

```
function toggleAutoRenew(StakeType stakeType) public nonReentrant {...}
function toggleAllAutoRenew() external {
    toggleAutoRenew(StakeType.SEVEN_DAYS);
    toggleAutoRenew(StakeType.ONE_YEAR);
    toggleAutoRenew(StakeType.NODE);
}
```

Status

This issue has been resolved by the team.

6. EndTime miscalculation in toggleAutoRenew

Severity: Medium	Category: Business logic

Target:
- APEX.sol

Description

When a user calls `toggleAutoRenew()` to turn off auto-renew, the contract recomputes `record.endTime` so the position becomes a fixed-term stake that should mature at the end of the current round (7 days / 360 days).

APEX.sol:L787-L826

```
function toggleAutoRenew(StakeType stakeType) external nonReentrant {
    ...
    for (uint256 i = 0; i < userStaking.stakeHistory.length; i++) {
        StakeRecord storage record = userStaking.stakeHistory[i];
        if (!record.isCompleted && record.stakeType == stakeType) {
            if (newAutoRenew) {
                record.endTime = 0;
            } else {
                uint256 elapsedTime = block.timestamp - record.startTime + 32;

                if (stakeType == StakeType.SEVEN_DAYS) {
                    uint256 currentRound = (elapsedTime / ROUND_DURATION);
                    record.endTime = record.startTime + (currentRound * ROUND_DURATION);
                } else {
                    uint256 currentRound = (elapsedTime / YEAR_DURATION);
                    record.endTime = record.startTime + currentRound * YEAR_DURATION;
                }
            }
        }
    }
```

APEX.sol:L631-L643

```
function _isStakeExpired(StakeRecord memory record) internal view returns (bool) {
    if (record.endTime == 0) {
        return false;
    }
    return record.endTime > 0 && block.timestamp >= record.endTime;
}
```

Because the `currentRound` omits + 1, the new `endTime` is equal to—or earlier than—the current `block.timestamp`. The stake is therefore considered already matured, `_isStakeExpired()` check will return true, whenever user want to claim reward, the flow:

`claimRewards/claimRewardsWithAD → _claimRewardsWithFee → calculatePendingRewards →_calculateCompletedStakeRewards → _calculateCrossRoundRewards(record, record.startTime, effectiveEndTime = 0)` will suspend user to claim the reward, cause the `_calculateCrossRoundRewards(record, record.startTime, effectiveEndTime = 0)` will return 0.

APEX.sol:L565-L602

```
function _calculateCrossRoundRewards(StakeRecord memory record, uint256 startTime,
uint256 endTime)
    internal
    view
    returns (uint256)
{
    if (startTime >= endTime) return 0;
    ...
}
```

Recommendation

Fix the round calculation by adding one extra period.

Status

This issue has been resolved by the team.

7. The autorenew user can unstake

Severity: Medium	Category: Business logic
Target: - APEXStaking.sol	

Description

Based on the `toggleAutoRenew()` function, users with auto-renewal enabled are not permitted to `unstake` at any time.

But the `unstake` function fails to block this behavior.

Proof of Concept:

```
function test_unstake() public {
    apex.approve(address(apexStaking), 1000e18);
    apexStaking.stake(100e18, APEXStaking.StakeType.SEVEN_DAYS); //New user's autorenew
is open.
    vm.warp(block.timestamp + 1 days);
    apexStaking.unstake(0);
    require(apexStaking.getUserStakeHistory(address(this))[0].isCompleted == true);
}
```

Recommendation

Add `isAutoRenew` and `endTime` check to the `unstake` function.

Status

This issue has been resolved by the team.

8. Incorrect calculation due to overridden totalsupply

Severity: Low	Category: Business logic
Target: - AD.sol	

Description

`getStats()` computes `totalBurned = totalMinted - totalSupply()`, but `totalSupply()` is overridden to return `totalMinted`. As a result, `totalBurned` is always 0 and `_circulatingSupply` is misreported as `totalMinted` instead of the base ERC20 supply after burns.

AD.sol:L431-L444

```
function getStats() external view returns (
    uint256 _totalMinted,
    uint256 _totalFlashSwapped,
    uint256 _totalBurned,
    uint256 _circulatingSupply
) {
    uint256 totalBurned = totalMinted - totalSupply();
    return (
        totalMinted,
        totalFlashSwapped,
        totalBurned,
        totalSupply()
    );
}
```

AD.sol:L133-L138

```
function totalSupply() public view override returns (uint256) {
    return totalMinted;
}
```

Recommendation

Use the base ERC20 implementation in the `getStats()` function.

Status

This issue has been resolved by the team.

9. GetFuturePrice() ignores current pricing configuration

Severity: Low	Category: Business logic
Target: - AD.sol	

Description

`getFuturePrice()` always calculates `INITIAL_PRICE_DEFAULT + (totalDays * DAILY_INCREMENT_DEFAULT),` so it ignores `basePrice / dailyIncrement` that may have been changed by `configurePricing()`, and `manualPriceMode`, where the price should remain fixed.

As a result, dashboards or off-chain callers receive misleading projections whenever the pricing parameters have been updated or manual mode is enabled.

AD.sol:431-L435

```
function getFuturePrice(uint256 futureDays) external view returns (uint256) {
   uint256 daysSinceDeploy = (block.timestamp - deployTime) / 1 days;
   uint256 totalDays = daysSinceDeploy + futureDays;
   return INITIAL_PRICE_DEFAULT + (totalDays * DAILY_INCREMENT_DEFAULT);
}
```

Recommendation

Use the active configuration.

Status

This issue has been resolved by the team.

10. GetSwapBurnInfo() misreports totalBurned

Severity: Low	Category: Business logic
Target: - APEX.sol	

Description

The `getSwapBurnInfo()` returns `burned = MAX_SUPPLY - totalSupply();` as `totalBurned`. This value merely shows the gap to `MAX_SUPPLY`, not the real amount destroyed. If tokens are never minted up to the cap, `burned` can overstate burns. The accurate figure should aggregate on-chain burn counters:

APEX.sol:L606-L621

```
function getSwapBurnInfo() external view returns (
        uint256 sellBurnRate,
        uint256 buyBurnRate_,
        uint256 totalBurned,
        bool autoSwapBurnEnabled_,
        uint256 minAmount,
        uint256 maxAmount
    )
{
    uint256 burned = MAX_SUPPLY >= totalSupply() ? MAX_SUPPLY - totalSupply() : 0;

    return (SWAP_BURN_RATE, buyBurnRate, burned, autoSwapBurnEnabled, minSwapAmount,
maxSwapAmount);
}
```

The same problem occurs in `getBurnStats().totalBurned`:

APEX.sol:L660-L674

```
function getBurnStats() external view returns (
        uint256 totalUserBurned,
        uint256 totalAutoBurned,
        uint256 totalBurned,
        uint256 autoSellBurns,
        uint256 autoBuyBurns
    )
{
    uint256 totalSystemBurned = MAX_SUPPLY >= totalSupply() ? MAX_SUPPLY - totalSupply()
: 0;

    return (totalUserBurnedAmount, totalAutoBurnedAmount, totalSystemBurned,
totalAutoSellBurns, totalAutoBuyBurns);
}
```

Recommendation

Replace the current calculation to reflect actual tokens burned.

Status

This issue has been resolved by the team.

11. ToggleAutoRenew lacks per-stake granularity

Severity: Low	Category: Business logic
Target: - APEXStaking.sol	

Description

The `toggleAutoRenew(StakeType)` flips the auto-renew flag for all active stakes of the given `stakeType`. Users with multiple concurrent positions (different sizes, rounds, or purposes) cannot enable/disable auto-renew on a single stake by index.

For example, a user has an auto-renew position and a normal position, both of which have a `stakeType` of `StakeType.SEVEN_DAYS`. If the user wants to end the auto-renew position, the `stakeType` of another normal position will change to auto-renew, which is not the user's intention.

APEXStaking.sol:L835-L878

```
function toggleAutoRenew(StakeType stakeType) external nonReentrant {
    UserStaking storage userStaking = userStakings[msg.sender];
    require(userStaking.stakeHistory.length > 0, "No staking history");
    bool newAutoRenew;
    if (stakeType == StakeType.SEVEN_DAYS) {
        newAutoRenew = !isAutoRenewSevenDays[msg.sender];
        isAutoRenewSevenDays[msg.sender] = newAutoRenew;
    } else if (stakeType == StakeType.ONE_YEAR) {
        newAutoRenew = !isAutoRenewOneYear[msg.sender];
        isAutoRenewOneYear[msg.sender] = newAutoRenew;
    } else if (stakeType == StakeType.NODE) {
        newAutoRenew = !isAutoRenewNode[msg.sender];
        isAutoRenewNode[msg.sender] = newAutoRenew;
    }
    emit AutoRenewToggled(msg.sender, stakeType, newAutoRenew);
}
```

Recommendation

Add `stakeIndex` as an input parameter, allowing users to toggle auto-renew for an individual position, then add `stakeIndex` in the relative event.

Status

This issue has been resolved by the team.

12. Centralization risk

Severity: Low	Category: Centralization

Target:
- APEXStaking.sol
- APEXStakingGovernance.sol
- AD.sol
- APEX.sol
- SimpleFeeProcessor.sol

Description

APEXStakingGovernance is intended to route all sensitive actions through a multi-sig when `multisigRequired == true`, enforced by the modifier:

APEXStakingGovernance.sol:

```
modifier onlyMultisigOrOwner() {
    if (multisigRequired) {
        require(msg.sender == multisigGovernance, "Only multisig governance allowed");
    } else {
        require(msg.sender == owner(), "Only owner allowed");
    }
}
```

The code comment claims: “when multi-sig is required, key operations must be executed by the multi-sig address.” However, two design flaws break that guarantee:

1. Owner permissions are strictly broader than multi-sig. Many downstream contracts (AD, APEX, APEXStaking, SimpleFeeProcessor) still rely on their own `onlyOwner` modifiers.
   Example: `SimpleFeeProcessor.setFeeRate()` is protected by `onlyOwner`, so the Governance owner may raise fees or drain funds even when `multisigRequired == true`. Thus a single compromised or malicious owner key can unilaterally change fee rates or fee collectors; mint or withdraw tokens in AD/APEX; disable flash-loan protection, alter burn rates, rearrange allocations, etc.

2. Owner supersedes the multi-sig. The single owner can always call `toggleMultisigRequirement(false)` or `setMultisigGovernance(...)`, then immediately execute any privileged function—effectively bypassing the multi-sig gate.

Recommendation

Make the multi-sig the sole admin of downstream contracts. Transfer every downstream `owner()` to the multi-sig (or a timelock) and revoke the Governance contract’s owner-level privileges.

Status

This issue has been resolved by the team.

2.3 Informational Findings

13. Missing events for functions that change critical state

Severity: Information	Category: Logging
Target: - AD.sol - APEXStaking.sol	

Description

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes that allow users to evaluate them. Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in protocol users.

In the AD contract, events are lacking in the privileged setter functions (e.g. `` `depositUSDT()` ``, `` `setUSDTToken()` ``, `` `emergencyWithdraw()` ``).

In the APEXStaking contract, events are lacking in the privileged setter functions (e.g. `` `setRoundDuration()` ``, `` `setYearDuration()` ``, `` `setApexTokenMinter()` ``,`` `setGovernanceContract()` ``, `` `setFeeRecipient()` ``, `` `setFeeRates()` ``).

Recommendation

It is recommended to emit events for critical state changes.

Status

This issue has been resolved by the team.

14. Gas optimization suggestions

Severity: Informational	Category: Gas Optimization

Target:
- APEXStaking.sol
- AD.sol
- APEX.sol

Description

Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective. Cache array length outside of the loop can save gas.

APEXStaking.sol:L313

```
for (uint256 i = 0; i < beneficiaries.length; i++) {
```

APEXStaking.sol:L575,L743,L765,L812,L856

```
for (uint256 i = 0; i < userStaking.stakeHistory.length; i++) {
```

AD.sol:L288

```
for (uint256 i = 0; i < amounts.length; i++) {
```

AD.sol:L294

```
for (uint256 i = 0; i < recipients.length; i++) {
```

APEX.sol:L294

```
for (uint256 i = 0; i < pools.length; i++) {
```

Recommendation

Consider using the above suggestions to save gas.

Status

This issue has been resolved by the team.

15. Use of floating pragma

Severity: Informational	Category: Configuration
Target: - All	

Description

```
pragma solidity ^0.8.0;
pragma solidity ^0.8.19;
```

The contract uses floating compiler versions ^0.8.0 and ^0.8.19.

Using floating pragma ^0.8.0 and ^0.8.19 statements is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

This issue has been resolved by the team.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
AD.sol	de9b9116ce843a0b6faa00883498e835f338783a
APEX.sol	bad7b882474aed446953cadc409c1524a95e3ca5
APEXStaking.sol	7274960491425a355e6763d81c1a65bbba38d184
APEXStakingGovernance.sol	9a03d0fe976a4a40f2f5a9cb3e02077f9289512d
SimpleFeeProcessor.sol	55f9b62ba79facc8ccf33bdfc9d2f68897f5c885

And we audited the fixed version of the project:

File	SHA-1 hash
AD.sol	cd84e62b8501d103bcdcef25db91fdb7d76c27cd
APEX.sol	da0a5feafd8b720fe7cf68c99d6dd6fdaa868b7b
APEXStaking.sol	11a628378e82fbd03ceb103993bc12d595d5c3d1
APEXStakingGovernance.sol	a58a44ca0d1697fa8107612320d26306ad3f6888
SimpleFeeProcessor.sol	a886fd9cf25a910f5d1a5aac5c5521ba95ba6076
APEXWithWhitelist.sol	f38f73ce039b3b78c53b521d4e980b856d8fb573