33AUDITS & CO.

SwitchX Audit Report

SwitchX Audit Report

Introduction

A security review of the SwitchX protocol was done, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Radev and Sam as Security Researchers.

Disclaimer

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

About 33 Audits & Company

33Audits is an independent smart contract security researcher company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @33audits.

About SwitchX

SwitchX is a decentralized protocol featuring swap functionality, a factory pattern, pool-based architecture, and a plugin architecture.

Repository: BuildTheTech/SwitchX-Contracts

Commit: https://github.com/BuildTheTech/SwitchX-Contracts/tree/c2d4de179c32d6f592664b581e96f6d5dae6ce0c

Severity Definitions

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

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[H-01]	Plugin fee misrouting and fee-flush DoS after `setPlugin()`	High	Acknowledged
[H-02]	Community fees + `communityVault = 0` can DoS pool or burn protocol fees	High	Fixed
[H-03]	Griefing Attack In V4 Factory Pool / Deployer	High	Acknowledged
[L-01]	`ExcessTokens` event emits underflowed values when only one token has excess	Low	Fixed
[L-02]	`createVaultForPool` allows multiple vaults per pool and silently re-points future fees	Low	Fixed
[L-03]	`V4CommunityVault` withdrawals can be DoS'ed by misordered fee configuration	Low	Fixed
[L-04]	Switching plugin with nonzero `pluginFeePending{0,1}` sends old plugin's fees to the new plugin	Low	Fixed
[L-05]	Use of Assert Instead of Require Will Consume All Gas	Low	Fixed
[L-06]	Potential For Users To Have Their Funds Sniped In Multicall	Low	Fixed
[L-07]	Incorrect Event Emission For Fees Distributed When Fees Are Clamped	Low	Fixed
[L-08]	Missing Slippage Protection For Rebalance Swaps	Low	Acknowledged
[L-09]	Missing Events For CommunityFee and PluginFee Transfers	Low	Fixed
[L-10]	Flash-donation trick lets malicious user repeatedly reset the fee timer and block plugin fees	Low	Fixed
[I-01]	Inconsistent dynamic-fee guard allows admin to set invalid static fees	Informational	Fixed
[I-02]	Using `V4VaultFactoryStub` in production breaks per-pool fee accounting	Informational	Fixed
[I-03]	`V4Factory.sol#createCustomPool()`: "deployer" parameter is inconsistent with implementation	Informational	Fixed

High

[H-01] Plugin fee misrouting and fee-flush DoS after `setPlugin()`

Description

Plugin fees accumulate in the pool as `pluginFeePending0/1`. When the pool flushes those fees (time threshold reached or pending overflows), payout recipient is always the current value of `plugin`, not the plugin that generated those fees:

```
assembly { feeRecipientSlot := plugin.slot }
(accrue and transfer)
IV4Plugin(plugin).handlePluginFee(...)
```

Two consequences:

1. If governance updates `plugin` while there is pending plugin fee, the new plugin receives the old plugin's fees.
2. If governance sets `plugin = address(0)` while pending exists, the next fee flush path tries to transfer fees to `address(0)` and then call `handlePluginFee` on `address(0)` → this reverts. After that, any swap/burn that triggers `_changeReserves()` and crosses the flush condition will revert, effectively bricking the pool until a valid plugin is restored.

Impact

- Plugin revenue loss / misrouting: Historical revenue ends up paid to the wrong plugin after a config change. This violates correct accounting and can't be undone once flushed.
- Protocol-wide DoS: If `plugin` is set to zero with pending fees, the next flush will revert. Normal swaps/burns that hit `_changeReserves()` will keep failing until governance fixes the plugin and retries.

Economic Impact:

- Incorrect revenue attribution to plugins, violating accounting principles
- Complete pool denial of service until governance intervention
- Loss of user trust and protocol functionality during DoS period

Recommendations

- When changing `plugin`, flush pluginFeePending0/1 to the old plugin first, then zero the pending counters.
- Do not allow setting `plugin = address(0)` while pending exists.

Status

Acknowledged - **empm0ney**: We've updated "V4Pool.sol" "setPlugin(address)" function to include a call to a new "ReservesManager.sol" "_flushPluginFees(address)" when the previous plugin has pending fees.

[H-02] Community fees + `communityVault = 0` can DoS pool or burn protocol fees

Description

Community fees accumulate in `communityFeePending0` / `communityFeePending1`. These pending amounts are flushed inside `_changeReserves()` whenever:

- The time threshold (`FEE_TRANSFER_FREQUENCY`) is reached, or
- Pending values overflow the uint104 range.

The flush code always uses the current value of `communityVault` as the payout recipient:

```
assembly {
    feePendingSlot := communityFeePending0.slot
    feeRecipientSlot := communityVault.slot
}
(feePending0, feePending1, feeSent0, feeSent1) =
_accrueAndTransferFees(...);
```

However, `setCommunityVault()` does not flush pending fees before updating the vault. It allows:

```
if (newCommunityVault == address(0) && globalState.communityFee != 0)
    _setCommunityFee(0);
_setCommunityFeeVault(newCommunityVault);
```

This leaves previously accrued `communityFeePending*` untouched even when `communityVault` becomes `address(0)`.

As soon as `_changeReserves()` executes after `FEE_TRANSFER_FREQUENCY` has passed or pending values grow, the pool tries to flush these old fees:

- Recipient = address(0)
- `_transfer(token, address(0), pendingAmount)` is executed
- Many ERC20s revert on transfers to the zero address.

If the transfer passes, the next step calls:

```
// in _accrueAndTransferFees
(recipient := sload(receiverSlot)) // = address(0)
_transferFees(..., recipient);
```

So fees are either burned (if token allows it) or the pool reverts (if token rejects zero-address transfers).

Both outcomes are incorrect.

Impact

1. Pool-wide DoS

- For tokens that revert on `transfer(address(0), amount)`, every operation that calls `_changeReserves()` (swap, burn, mint, flash, etc.) will revert once the flush condition is hit.
- The pool becomes unusable until governance assigns a valid `communityVault` and makes a successful flush.

2. Silent burning of protocol revenue

- If the token allows transfers to `address(0)`, all accumulated community fees are permanently burned instead of being sent to the old vault or the new vault.
- This permanently destroys protocol revenue and breaks accounting.
- This violates every expected lifecycle invariant around fee accounting and makes community fee logic unsafe around vault updates.

Economic Impact:

- Complete loss of accumulated community fees if tokens allow zero-address transfers
- Pool-wide denial of service affecting all users if tokens reject zero-address transfers
- Permanent destruction of protocol revenue and broken accounting
- Potential reputation damage from bricked pools

Recommendations

Before updating `communityVault`, enforce correct handling of previously accrued fees:

1. Flush pending fees to the *old* vault before updating

```
   function setCommunityVault(address newVault) external {
       // flush pending to old vault first
       if (communityVault != address(0)) {
           _flushCommunityFeesTo(communityVault);
       } else if (communityFeePending0 > 0 || communityFeePending1 > 0) {
           revert("Pending community fees must be cleared before setting
vault to zero");
       }
       _setCommunityFeeVault(newVault);
   }
```

2. Disallow setting `communityVault = address(0)` while pending fees exist.

Status

Fixed - The development team has addressed this finding. Please refer to the specific finding details above for the implementation status.

The fix ensures that:

- The vulnerability has been mitigated
- Protocol security has been improved
- User safety has been enhanced

[H-03] Griefing Attack In V4 Factory Pool / Deployer

Description

The V4PoolDeployer contract is responsible for deploying a new pool for a specific token pair, using CREATE2 for deterministic address generation.

```
 function deploy(address plugin, address token0, address token1, address
deployer) external override returns (address pool) {
    require(msg.sender == factory);
    _writeToCache(plugin, token0, token1);
    bytes memory _encodedParams;
    // @audit this is for standard pools, can be front run and deploy
first , stopping legit deployments for this tokens pool
    // @audit not using nonce or msg.sender within salt
    if (deployer == address(0)) {
      _encodedParams = abi.encode(token0, token1);
    } else {
      _encodedParams = abi.encode(deployer, token0, token1);
    }
    pool = address(new V4Pool{salt: keccak256(_encodedParams)}());
    (cache0, cache1) = (bytes32(0), bytes32(0));
  }
```

The deployment mechanism uses CREATE2 with a salt calculated as follows. The other parameters used in the CREATE2 are publicly available and can be seen and used by an attacker.

```
// Standard pools (deployer == address(0))
salt = keccak256(abi.encode(token0, token1))

// Custom pools
salt = keccak256(abi.encode(deployer, token0, token1))
```

The salt calculation does not include:

- msg.sender (V4Factory address)

- V4PoolDeployer address
- Any nonce or randomness
- The creator's address

There are some current protections that use msg.sender, but it is only used in the internal cache system and isn't used for the actual salt for address determination. Also, there is an internal mapping check for the tokens to ensure the address hasn't been used to prevent re-using an occupied address, but this is just an internal check and wouldn't prevent someone from actually determining and consuming the address:

1. V4Factory mapping check: Prevents duplicate pools in the factory's internal mapping, but does not prevent front-running the CREATE2 address
2. CREATE2 native protection: If address is occupied, CREATE2 fails, preventing malicious pool deployment

Why msg.sender Doesn't Help

While `V4Pool` uses `msg.sender` in its constructor to retrieve parameters from the deployer:

```
// V4PoolBase.sol:170-171
function _getDeployParameters() internal virtual returns (address,
address, address, address) {
  return IV4PoolDeployer(msg.sender).getDeployParameters();
}
```

This is only used for parameter passing via the cache mechanism, not for the CREATE2 salt calculation. The salt remains predictable and front-runnable.

To compute the exact CREATE2 address, a front-runner only needs three publicly accessible pieces of information:

1. `poolDeployer` address - Public immutable in `V4Factory.poolDeployer()`
2. `token0` and `token1` addresses - Observable from the pending `createPool()` transaction in mempool
3. `POOL_INIT_CODE_HASH` - Public constant in `V4Factory.POOL_INIT_CODE_HASH()` All of these are publicly accessible, making the address completely predictable.

Impact

- An attacker can grief the system by deploying to the address first and permanently block pool creation for token pairs.
- The attacker will consume the address and cause legitimate deployments to fail because CREATE2 requires the address to be unused.

Economic Impact:

- Permanent denial of service for affected token pairs, preventing pool creation
- Potential protocol disruption if critical token pairs cannot be deployed
- Loss of functionality and user trust due to blocked deployments

Recommendations

The recommendation is to add specific information for the legitimate deployment that would make an attacker attempting to front-run and execute this attack unable to determine and generate the address:

Nonce Protection example:

```
// In V4Factory
mapping(address => mapping(address => uint256)) public poolCreationNonce;

function _createPool(...) private returns (address pool) {
  // ...
  uint256 nonce = poolCreationNonce[token0][token1]++;
  pool = IV4PoolDeployer(poolDeployer).deploy(plugin, token0, token1,
deployer, nonce);
  // ...
}
```

Or you can use the specific creator in the Salt:

Include the `creator` address in the salt:

```
bytes memory _encodedParams = abi.encode(deployer, creator, token0,
token1);
```

Status

Acknowledged - The development team has reviewed this finding and determined that the current implementation is acceptable given the permissionless design of pool creation. The team has documented the behavior and considers it expected functionality rather than a vulnerability.

Low

[L-01] `ExcessTokens` event emits underflowed values when only one token has excess

Description

Inside `_updateReserves()`, the pool emits:

```
emit ExcessTokens(balance0 - _reserve0, balance1 - _reserve1);
```

When only one token has excess, the other balance is lower than its reserve. Because this block runs inside `unchecked`, the subtraction on the non-excess side underflows and wraps to a huge `uint256`.

Example:

- `balance0 > reserve0` → OK
- `balance1 < reserve1` → underflow emits a massive number

Internally nothing breaks, because the event is never used for state, but all off-chain systems reading this event see garbage data.

Recommendations

Emit zero for non-excess sides:

```
uint256 excess0 = hasExcessToken0 ? balance0 - _reserve0 : 0;
uint256 excess1 = hasExcessToken1 ? balance1 - _reserve1 : 0;

emit ExcessTokens(excess0, excess1);
```

This prevents underflow, avoids wrapped values, and keeps external analytics consistent.

Status

Fixed - The development team has implemented the recommended fix to emit zero for non-excess sides, preventing underflow and ensuring accurate off-chain data.

The fix ensures that:

- No underflow occurs in event emissions
- Off-chain systems receive accurate data
- External analytics remain consistent

[L-02] `createVaultForPool` allows multiple vaults per pool and silently re-points future fees

Description

`V4VaultFactory.createVaultForPool()` can be called multiple times for the same pool, and every call:

1. Deploys a new `V4CommunityVault`.
2. Silently overwrites the previous entry in `vaultForPool[pool]`.

```
mapping(address => address) private vaultForPool;

function createVaultForPool(address pool, address, address, address,
```

```
  address)
    external
    override
    returns (address communityFeeVault)
  {
      require(msg.sender == factory);

      address vault = address(new V4CommunityVault(factory, owner()));
      vaultForPool[pool] = vault;
      return vault;
  }
```

There is no guard that prevents re-creation of a vault for an already-initialized pool, and no event or warning that a vault was rotated. This breaks the implicit “one pool → one vault” invariant.

Result:

- Old vault (vaultA) continues to hold all previously accumulated community fees.
- New vault (vaultB) receives all future fees.
- Nothing alerts protocol operators that a replacement occurred.
- Off-chain accounting and monitoring will attribute all fees to vaultB, ignoring funds still sitting in vaultA.

This splits protocol revenue across multiple vault addresses and increases the chance of permanently stranded community fees.

Recommendations

Add an initialization guard:

```
require(vaultForPool[pool] == address(0), "Vault already exists");
```

Status

Fixed - The development team has implemented an initialization guard to prevent multiple vaults per pool, ensuring the one-pool-to-one-vault invariant is maintained.

The fix ensures that:

- Only one vault can be created per pool
- Protocol revenue is not split across multiple addresses
- Clear error messages prevent accidental vault replacement

[L-03] `V4CommunityVault` withdrawals can be DoS'ed by misordered fee configuration

Description

`V4CommunityVault.sol#withdraw()` and `V4CommunityVault.sol#withdrawTokens()` functions rely on `_readAndVerifyWithdrawSettings()`:

```
if (_switchxFee != 0) require(_switchxFeeReceiver != address(0), 'invalid
V4 fee receiver');
require(_communityFeeReceiver != address(0), 'invalid receiver');
```

The problem is that nothing in the fee-proposal flow ensures these receivers are set before switching `switchxFee` from `0` to a non-zero value.

Example:

1. System operates with `switchxFee = 0`, `switchxFeeReceiver = 0`, `communityFeeReceiver ≠ 0`.
2. Admin enables V4 fee: `proposeV4FeeChange(x)` → `acceptV4FeeChangeProposal(x)`.
3. Now: `switchxFee = x`, `switchxFeeReceiver = 0`.

From this point on every `withdraw` call reverts with `"invalid V4 fee receiver"` until governance manually fixes the receiver address.

Identical DoS happens if `communityFeeReceiver` is set to zero.

Recommendations

Add safety guards before accepting a new fee:

```
    require(switchxFeeReceiver != address(0), "missing V4 fee receiver");
    require(communityFeeReceiver != address(0), "missing community fee
receiver");
```

Status

Fixed - The development team has added safety guards to `acceptV4FeeChangeProposal` to ensure fee receivers are properly configured before accepting fee changes.

The fix ensures that:

- Fee receivers must be set before enabling fees
- Withdrawal functionality cannot be DoS'd by misconfiguration
- Clear error messages guide proper configuration

[L-04] Switching plugin with nonzero `pluginFeePending{0,1}` sends old plugin's fees to the new plugin

Description

When the plugin is changed via `ReservesManager.sol#setPlugin()`, any accumulated plugin fees stored in:

```
pluginFeePending0
pluginFeePending1
```

are not reset and not flushed to the old plugin.

```
function setPlugin(address newPluginAddress) external override
onlyUnlocked {
    _checkIfAdministrator();
    _setPluginConfig(0);
    _setPlugin(newPluginAddress);
}
```

Later, the next `ReservesManager.sol#_changeReserves()` call triggers fee payout through:

```
assembly {
    feeRecipientSlot := plugin.slot
}
recipient := sload(feeRecipientSlot); // <- current plugin
```

So the contract pays all previously accrued plugin fees to the newly configured plugin, not the plugin that actually generated those fees.

Example:

1. While plugin A is connected and charges plugin fees, `_changeReserves` accumulates them in `pluginFeePending0/1`.
2. If conditions aren't met to flush yet (time / thresholds), those fees remain sitting in `pluginFeePending{0,1}`.
3. Admin calls `setPlugin(newPluginAddress)`:
    - `plugin` is now plugin B.
    - `pluginFeePending0/1` still contain the fees accrued for plugin A.
4. Later, the first `_changeReserves(...)` call that triggers `_accrueAndTransferFees` flush:
    - `recipient := sload(receiverSlot)` -> current `plugin` i.e. plugin B.
    - All old pending plugin fees are transferred to plugin B.

- Then `IV4Plugin(plugin).handlePluginFee(...)` is called on plugin B.

Result: plugin B receives plugin A's accumulated fees. Plugin A can never claim them.

Impact

- The new plugin receives fees earned by the old plugin.
- Permanent loss for old plugin.
- Incorrect revenue attribution violates accounting principles.

Recommendations

Any of:

1. In `setPlugin()`, if `plugin != address(0)` and `pluginFeePending0|pluginFeePending1 != 0`, call `_accrueAndTransferFees` with `receiverSlot` pointing at the current plugin, before updating `plugin`.
2. If you accept that old plugin forfeits whatever is pending when admin switches, zero out state so redirecting never happens silently:

```
    pluginFeePending0 = 0;
    pluginFeePending1 = 0;
```

Status

Fixed - The development team has implemented the recommended fix to flush plugin fees to the current plugin before switching to a new plugin, ensuring correct fee attribution.

The fix ensures that:

- Old plugin receives its earned fees before being replaced
- Correct revenue attribution is maintained
- No fees are lost during plugin transitions

[L-05] Use of Assert Instead of Require Will Consume All Gas

Description

In `V4Pool.mint()`, the contract uses `assert()` statements for validation checks that should use `require()` instead. The `assert()` statements are not recommended and will result in the consumption of all gas if the transaction reverts, resulting in not refunding the gas

```
unchecked {
  // return leftovers
  if (amount0 > 0) {
    if (receivedAmount0 > amount0) _transfer(token0, leftoversRecipient,
```

```
receivedAmount0 - amount0);
    else assert(receivedAmount0 == amount0);
  }
  if (amount1 > 0) {
    if (receivedAmount1 > amount1) _transfer(token1, leftoversRecipient,
receivedAmount1 - amount1);
    else assert(receivedAmount1 == amount1);
  }
}
```

Impact

- If the transaction fails for any reason, all of the gas is consumed and the user is not refunded for the transaction, losing their gas as an expense

Recommendations

Use `require` instead of `assert`, both will revert if the condition is not met, but it is more user friendly and the user will be refunded the gas

Status

Fixed - The development team has replaced `assert` statements with `require` statements, ensuring users receive gas refunds on transaction failures.

The fix ensures that:

- Users receive gas refunds when transactions fail
- More user-friendly error handling
- Consistent error handling patterns throughout the codebase

[L-06] Potential For Users To Have Their Funds Sniped In Multicall

Description

In `NonfungiblePositionManager.collect()`, there are scenarios (primarily in multicall operations) where the `recipient` parameter is set to the `NonfungiblePositionManager` contract address instead of the user's address or a router. This follows Uniswap's design pattern, where tokens are temporarily custodied by the contract and users are expected to call `sweepToken()` to retrieve them.

```
// In collect()
(amount0, amount1) = pool.collect(recipient, tickLower, tickUpper,
amount0Collect, amount1Collect);
```

The `sweepToken()` function has no access control by design, as the contract is never expected to hold tokens permanently:

```
/// @inheritdoc IPeripheryPayments
function sweepToken(address token, uint256 amountMinimum, address
recipient) public payable override {
    uint256 balanceToken = _balanceOfToken(token);
    require(balanceToken >= amountMinimum, 'Insufficient token');

    if (balanceToken > 0) {
        TransferHelper.safeTransfer(token, recipient, balanceToken);
    }
}
```

The risk is that there is no enforcement mechanism that requires a `sweepToken()` call to be included in the same transaction when `recipient` is the `NonfungiblePositionManager` contract. If a user:

1. Calls `collect()` with `recipient = address(NonfungiblePositionManager)` in a multicall but forgets to include `sweepToken()`, OR
2. Intentionally calls `collect()` and `sweepToken()` in separate transactions

Then a front-runner can monitor the mempool, see the `collect()` transaction that leaves tokens in the contract, and front-run the user's `sweepToken()` call to steal the tokens.

Impact

1. User Fund Loss: If users forget to include `sweepToken()` in their multicall or attempt to sweep in a separate transaction, front-runners can steal the collected tokens before the user retrieves them.
2. User Error Vulnerability: This is primarily a user error/ignorance issue rather than a protocol vulnerability, but the lack of enforcement makes it easy for users to make this mistake.

This is not a threat to the protocol, but the lack of enforcement exposes user risk which would result in user loss of funds.

Recommendations

If it is decided that there shouldn't be any enforcement at the contract level, there should be significant and clear documentation of the risks and instructions on how to avoid this attack:

- Thoroughly document this behavior in the protocol documentation, including:
- When `recipient` should be set to the `NonfungiblePositionManager` address
- The requirement to always include `sweepToken()` calls in the same multicall
- The front-running risk if tokens are swept in a separate transaction

Or, for a more secure enforcement there can be a flag or check that enforces `sweepToken()` inclusion when `recipient` is the `NonfungiblePositionManager`:

Status

Fixed - The development team has added a new `collectAndSweep` function to `NonfungiblePositionManager` that combines collection and sweeping in a single atomic operation,

preventing the front-running vulnerability.

The fix ensures that:

- Users can collect and sweep tokens in a single transaction
- Front-running attacks are prevented
- Clear documentation and interfaces clarify the safe usage patterns

---

[L-07] Incorrect Event Emission For Fees Distributed When Fees Are Clamped

Description

In the ALMVault deposit flow, fees are accrued and distributed for base and limit positions before each deposit

```
 function _cleanPositions(bool withEvent) internal returns (uint256 fees0,
uint256 fees1) {
    (uint128 _basePositionId, uint128 _limitPositionId) = (basePositionId,
limitPositionId);

    fees0 = 0;
    fees1 = 0;
    if (_basePositionId != 0) {
 ->     (fees0, fees1) = _collectRewardsAndAccrueFees(_basePositionId, 0,
0);
    }
    if (_limitPositionId != 0) {
 ->     (fees0, fees1) = _collectRewardsAndAccrueFees(_limitPositionId,
fees0, fees1);
    }
    if (fees0 > 0 || fees1 > 0) {
  ->    _distributeFees(fees0, fees1);
      if (withEvent) {
   ->     emit CollectFees(msg.sender, fees0, fees1);
      }
```

The event always returns the original fees0 and fees1 , but in `_distributeFees` function, there is clamping logic which will not send that amount and clamp it to the amount of balance the contract has (which is less than the fees) if that scenario happens :

```
  function _distributeFees(uint256 fees0, uint256 fees1) internal {
    uint256 ammFee = IALMVaultFactory(almVaultFactory).ammFee();
    uint256 baseFee = IALMVaultFactory(almVaultFactory).baseFee();

    // Make sure there are always enough fees to distribute
->    fees0 = min(fees0, IERC20(token0).balanceOf(address(this)));
 ->   fees1 = min(fees1, IERC20(token1).balanceOf(address(this)));
```

Impact

- In these scenarios, when the clamping logic is executed, the event emission `CollectFees` will be wrong and misleading, as the actual result of fees will be less than that amount.

Recommendations

Use the actual `fees` that are sent in `_distributeFees` to emit the event, therefore if the clamping logic is executed the event will emit the clamped amount of fees, guaranteeing the correct event emission

Status

Fixed - The development team has fixed the event emission to use the actual fees sent in `_distributeFees`, ensuring that the `CollectFees` event reflects the clamped amount when clamping logic is executed.

The fix ensures that:

- Event emissions accurately reflect actual fee distributions
- Off-chain systems receive correct fee data
- No misleading information in event logs

[L-08] Missing Slippage Protection For Rebalance Swaps

Description

In ALMVault there is a lack of slippage protection; there is no slippage enforced on-chain. This is different from all other areas of the protocol where there are either user input slippage amounts or via the swapRouter where it checks the output of the swap against the minimum amounts that are acceptable.

In the V4Pool (core) -> where the swap is handled :

- swap itself doesn't know about user slippage preferences; it just enforces:
- limitSqrtPrice bounds (you can set tight or wide). -> Slippage is meant to be enforced at the periphery (router) by the min-out / max-in checks.

In the `SwapRouter` :

- exactInputSingle / exactInput:
- Take amountOutMinimum in params.
- After the swap, they check:
- So if price moves against the user, the tx reverts.
- exactOutput / exactOutputSingle:
- Use amountInMaximum (via cached amount in and bounds).
- If more input is required than allowed, the tx fails. All router swaps also pass a limitSqrtPrice into pool.swap, so you can bound the execution price as well.

But here in the ALMVault during a rebalance swap, there is no slippage and this is the exception: it calls `IV4Pool(pool).swap` with:

- Exact input size (swapQuantity).
- Extreme limitSqrtPrice (effectively no price limit).
- No minOut or post-swap check inside rebalance, so all slippage is borne by the vault.

```
    // swap tokens if required
     if (swapQuantity != 0) {
       IV4Pool(pool).swap(
         address(this),
         swapQuantity > 0,
         swapQuantity > 0 ? swapQuantity : -swapQuantity,
         swapQuantity > 0 ? UV3Math.MIN_SQRT_RATIO + 1 :
UV3Math.MAX_SQRT_RATIO - 1,
         abi.encode(address(this))
       );
     }
```

Impact

- While the protocol's normal user swaps via SwapRouter are slippage-protected; the ALM vault's internal rebalancing swap is intentionally unbounded and relies on trusted operator behavior. The result is the lack of slippage protection for the `rebalance` swaps.

Recommendations

As the `rebalance` function is an admin called function, allow the admin to be able to input min amounts for slippage protection, to enforce that the protocol is not forced to accept a loss under unexpected scenarios.

Status

Acknowledged - The development team has reviewed this finding and determined that the current implementation is acceptable given the permissionless design of pool creation. The team has documented the behavior and considers it expected functionality rather than a vulnerability.

[L-09] Missing Events For CommunityFee and PluginFee Transfers

Description

The protocol emits events when fees are accrued during swaps and burns, but does NOT emit custom events when fees are actually transferred to the plugin or community vault. Only generic ERC20 `Transfer` events are emitted, making it difficult to track fee distributions off-chain. This is specifically for the fees that are distributed through the `_changeReserves` flow which is called internally.

Fees are transferred through the following call chain:

1. `_changeReserves()` is called with fee amounts

2. `_accrueAndTransferFees()` is called to accumulate and potentially transfer fees
3. `_transferFees()` is called when fees need to be sent
4. `_transfer()` is called to actually send tokens

Step 1: _changeReserves() - No Events Emitted

```
function _changeReserves(
    int256 deltaR0,
    int256 deltaR1,
    uint256 communityFee0,
    uint256 communityFee1,
    uint256 pluginFee0,
    uint256 pluginFee1
) internal {
    if (communityFee0 > 0 || communityFee1 > 0 || pluginFee0 > 0 ||
pluginFee1 > 0) {
        // ... community fee handling ...
        (uint104 feePending0, uint104 feePending1, uint256 feeSent0,
uint256 feeSent1) = _accrueAndTransferFees(
            communityFee0,
            communityFee1,
            lastTimestamp,
            feeRecipientSlot,
            feePendingSlot
        );
        if (feeSent0 | feeSent1 != 0) {
            // sent fees so decrease deltas
            (deltaR0, deltaR1) = (deltaR0 - feeSent0.toInt256(), deltaR1 -
feeSent1.toInt256());
            feeSent = true;
            //  NO EVENT EMITTED HERE for community fee transfer
        }

        // ... plugin fee handling ...
        (feePending0, feePending1, feeSent0, feeSent1) =
_accrueAndTransferFees(
            pluginFee0,
            pluginFee1,
            lastTimestamp,
            feeRecipientSlot,
            feePendingSlot
        );
        if (feeSent0 | feeSent1 != 0) {
            // sent fees so decrease deltas
            (deltaR0, deltaR1) = (deltaR0 - feeSent0.toInt256(), deltaR1 -
feeSent1.toInt256());
            feeSent = true;

            // notify plugin about sent fees
            IV4Plugin(plugin).handlePluginFee(feeSent0,
feeSent1).shouldReturn(IV4Plugin.handlePluginFee.selector);
            //  NO EVENT EMITTED HERE for plugin fee transfer
```

```
            }
        }
    }
```

Likewise, the next functions are the executed `_accrueAndTransferFees` and `_transferFees` do not emit any events either. The events that are emitted `SwapFee` and `BurnFee` events emit the fee percentage (pluginFee), not the actual amounts. The actual amounts (fees.pluginFeeAmount or deltaPluginFeePending0/1) are calculated and passed to _changeReserves(). When those amounts are transferred, no event is emitted.

Impact

1. Off-chain systems cannot easily distinguish fee transfers from regular token transfers without parsing all `Transfer` events and cross-referencing with pool state.
2. There's no clear audit trail showing when fees were actually distributed vs. when they were accrued.
3. Real-time monitoring of fee distributions requires complex filtering of generic `Transfer` events.
4. Users and integrators cannot easily track fee payments to plugins and community vaults.

Recommendations

The recommendation is to add separate events in `_changeReserves()`, emitting an event for each fee (`communityFee` or `pluginFee`) that was sent, with the exact amount that was sent to each destination. Given the way the logic is configured, this is straightforward and provides perfect timing for these fee event emissions, as they will only emit if actual fees were sent.

```
// In _changeReserves(), after community fees are sent:
if (feeSent0 | feeSent1 != 0) {
    emit CommunityFeeTransferred(communityVault, feeSent0, feeSent1,
block.timestamp);
}

// After plugin fees are sent:
if (feeSent0 | feeSent1 != 0) {
    emit PluginFeeTransferred(plugin, feeSent0, feeSent1,
block.timestamp);
}
```

Status

Fixed - The development team has added a new `FeeTransferred` event to the `_transferFees` function within `ReserveManager`. This event is emitted whenever fees are transferred, providing clear on-chain tracking of fee distributions.

The fix ensures that:

- Fee transfers are now trackable via dedicated events

- Off-chain systems can easily monitor fee distributions
- Complete audit trail for all fee transfers

[L-10] Flash-donation trick lets malicious user repeatedly reset the fee timer and block plugin fees

Description

`ReservesManager._accrueAndTransferFees()` uses a single shared timestamp (`lastFeeTransferTimestamp`) for both fee streams:

- communityFeePending0/1 -> communityVault
- pluginFeePending0/1 -> plugin

A fee send in either stream sets:

```
lastFeeTransferTimestamp = block.timestamp;
```

This becomes exploitable because the function also flushes fees early if pending > uint104.max:

```
if (pending0 > uint104.max || pending1 > uint104.max) {
    // send fees, clear pending, reset timestamp
}
```

An attacker with a large-supply token (supported up to `uint128.max`) can intentionally donate a huge amount of tokens via `flash()` repayment or `burn()` paths, causing communityFeePending to exceed `uint104.max`. That triggers an immediate community payout even if the 8h frequency window has not elapsed and resets `lastFeeTransferTimestamp`.

Meanwhile, pluginFeePending remains below the overflow threshold and therefore does not get flushed, because the time window is constantly reset.

Repeating this just before every frequency window prevents plugin fees from ever reaching the timed payout condition. Plugin fees keep accumulating but never flush until they eventually overflow `uint104` - which may require massive volume.

Impact

- Plugin fees can be starved indefinitely by repeatedly forcing an early community-fee flush.
- Plugin payments become delayed by many cycles, potentially days or weeks, depending on attacker persistence.

Recommendations

Implement independent timestamps for the two fee streams:

- `lastCommunityFeeTransferTimestamp`
- `lastPluginFeeTransferTimestamp`

and update each only when its own fees are flushed.

Alternatively, keep a single timestamp but do not reset it when overflow-triggered payouts occur; only reset it when a time-based flush happens.

Status

Fixed - The development team has implemented distinct timestamps for community and plugin fee streams, ensuring each stream's timing is independent and cannot be manipulated by the other.

The fix ensures that:

- Community and plugin fee streams have independent timestamps
- Plugin fees cannot be starved by community fee manipulation
- Each fee stream flushes based on its own timing conditions

Informational

[I-01] Inconsistent dynamic-fee guard allows admin to set invalid static fees

Description

The pool enforces the dynamic-fee disabled guard correctly inside `V4PoolBase.sol#_callBeforeSwap()`, preventing plugins from injecting `overrideFee` or `pluginFee` unless the `DYNAMIC_FEE` flag is enabled.

However, `V4Pool.sol#setFee(uint16 newFee)` has no validation at all on the static fee value being assigned. Governance can set a value close to or above the protocol's effective upper bound (1e6), and while this won't break state immediately, it can later cause swaps to revert inside `SwapCalculation.sol#_calculateSwap()` when the combined fee path fails the `< 1e6` check.

This creates configuration traps where trades fail at runtime instead of failing when the admin sets the fee.

Recommendations

Add simple input validation to `setFee`:

```
if (newFee >= 1e6) revert invalidFee();
```

Status

Fixed - The development team has added input validation to the `setFee` function to prevent invalid static fee configurations.

The fix ensures that:

- Invalid fees are rejected at configuration time
- Runtime failures are prevented
- Better user experience with clear error messages

[I-02] Using `V4VaultFactoryStub` in production breaks per-pool fee accounting

Description

`V4VaultFactoryStub` ignores the `pool` argument entirely:

```
contract V4VaultFactoryStub is IV4VaultFactory {
  address public immutable defaultV4CommunityVault;

  constructor(address _v4CommunityVault) {
    require(_v4CommunityVault != address(0));
    defaultV4CommunityVault = _v4CommunityVault;
  }

  function getVaultForPool(address) external view override returns
(address) {
    return defaultV4CommunityVault;
  }

  function createVaultForPool(address, address, address, address, address)
    external
    view
    override
    returns (address)
  {
    return defaultV4CommunityVault;
  }
}
```

This means every pool receives the same exact vault address, regardless of how many pools exist or what their tokens are.

The stub never creates a new vault, never stores per-pool state, and does not maintain any mapping.

In other words, all pools share a single community vault.

This is correct for a mock/stub, but becomes a real operational issue if the stub is accidentally wired as the live `IV4VaultFactory` for a production deployment.

Recommendations

Make it explicit that:

- `V4VaultFactoryStub` must not be used for production deployments.
- Only the real `V4VaultFactory` should be passed to the V4PoolFactory when per-pool revenue isolation is required.

Status

Fixed - The development team has confirmed that `V4VaultFactoryStub` will not be used in production deployments, and only the real `V4VaultFactory` will be used.

The fix ensures that:

- Production deployments use the correct vault factory
- Per-pool revenue isolation is maintained
- No risk of accidental stub usage

[I-03] `V4Factory.sol#createCustomPool()`: "deployer" parameter is inconsistent with implementation

Description

The `IV4Factory` interface says:

> @param deployer The address of plugin deployer

... but the implementation does not treat `deployer` as the plugin deployer.

In `V4Factory.sol#createCustomPool`:

- The plugin hooks are called on `msg.sender`, not on `deployer`.
- `deployer` is only used as:
    - A namespacing salt for deterministic pool address (`computeCustomPoolAddress(deployer, token0, token1)`).
    - A metadata parameter forwarded to the plugin factory.

Example:

- Contract A has `CUSTOM_POOL_DEPLOYER` role.
- Call `createCustomPool(deployer = B)` from A.
- Hooks run on A, not B.

This contradicts the interface/NatSpec.

Recommendations

Pick a consistent semantic and enforce it:

- Option 1: Treat `deployer` as the real plugin factory and call hooks on `IV4PluginFactory(deployer)` and validate that the address implements the interface.

- Option 2: Update NatSpec + docs to clarify that: `msg.sender` is the plugin factory, `deployer` is only an address-space salt. And the two are unrelated.

Status

Fixed - The development team has clarified all comments across relevant interfaces to accurately reflect that `msg.sender` is the plugin factory and `deployer` is only an address-space salt.

The fix ensures that:

- Documentation accurately reflects implementation behavior
- Interface and implementation are consistent
- Developers understand the parameter's actual purpose

Conclusion

This security audit of the SwitchX protocol identified 16 vulnerabilities across 3 High, 10 Low, and 3 Informational severity levels. Most issues (13 fixed, 3 acknowledged) have been addressed by the development team.

The development team's commitment to addressing these findings demonstrates a strong security-focused approach to smart contract development. The protocol has undergone significant improvements based on the audit findings, with most critical and high-severity issues resolved. Continued monitoring and periodic security reviews are recommended as the protocol evolves.

This report was prepared by 33Audits & Co and represents our independent security assessment of the SwitchX protocol.