Strategy Builder Plugin Audit Report

Prepared by Cyfrin
Version 2.0

Lead Auditors

Farouk

Giovanni Di Siena

September 5, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

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

4 Protocol Summary

The `StrategyBuilderPlugin` by Pecunity is an ERC-6900 modular smart contract account plugin designed to automate and execute advanced DeFi strategies.

5 Audit Scope

The following files were included in the scope of this audit:

```
contracts/condition/examples/CoinOrERC20BalanceCondition.sol
contracts/condition/examples/TimeCondition.sol
contracts/condition/BaseCondition.sol
contracts/FeeController.sol
contracts/FeeHandler.sol
contracts/PriceOracle.sol
contracts/StrategyBuilderPlugin.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Strategy Builder Plugin smart contracts provided by Pecunity. In this period, a total of 16 issues were found.

1 High and 5 Medium severity issues were identified, along with several Low, Informational, and Gas Optimizations.

- 1 High stemmed from missing access control on critical `FeeController` setter functions which could have allowed an attacker to drain revenue and grief smart accounts leveraging this plugin.
- 3 Mediums related to the DoS of automation/bypass of volume-based fee while the other 2 arose due to incorrect handling of token decimals and price oracle validation.

- 3 Lows involve a variety of edge cases with low probability and impact.

While the Low and Informational issues are not likely to have material impact on the codebase in its current form, it is strongly recommended that these issues are addressed to prevent potential escalation with future development and other lower-severity edge cases.

Summary

Project Name	Strategy Builder Plugin
Repository	strategy-builder-plugin
Commit	3ef3b53355c3. . .
Audit Timeline	Jun 23rd - Jun 27th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	5
Low Risk	3
Informational	4
Gas Optimizations	3
Total Issues	16

Summary of Findings

[H-1] Missing access control on critical `FeeController` setters	Resolved
[M-1] Fee calculations break when token decimals are different from 18	Resolved
[M-2] Missing staleness check in `PriceOracle::getTokenPrice`	Resolved
[M-3] Automation DoS via blacklisted or reverting fee recipients	Resolved
[M-4] DoS of strategy execution due to array indices out of bounds	Resolved
[M-5] Volume-based fee can be bypassed with a wrapper contract	Resolved
[L-1] Accumulation of dust amounts due to rounding in `_tokenDistribution()`	Resolved
[L-2] `automationsToIndex` storage is not correctly reset when deleting automations	Resolved
[L-3] DoS of automation due to potential zero value transfer reverts	Resolved
[I-1] Zero price validation could be included in the price oracle	Resolved
[I-2] Positive Pyth oracle exponents should be explicitly handled	Resolved
[I-3] Empty strategies can be created due to missing zero length check	Resolved

[I-4] Condition addresses can re-enter `StrategyBuilderPlugin`	Acknowledged
[G-1] Unused cached value of `getStorageId(msg.sender, id)`	Resolved
[G-2] Use named return variables	Resolved
[G-3] Array length used in multiple loop iterations can be cached	Resolved

7 Findings

7.1 High Risk

7.1.1 Missing access control on critical `FeeController` setters

Description: `FeeController.setFunctionFeeConfig()`, `setTokenGetter()`, and `setGlobalTokenGetter()` are declared `external` but have **no modifier** restricting the caller. Any account can freely (re)configure fee percentages, token-getter addresses, or even swap in malicious contracts.

```
/// @inheritdoc IFeeController
function setFunctionFeeConfig(bytes4 _selector, FeeType _feeType, uint256 _feePercentage) external {
    if (_feePercentage > maxFeeLimits[_feeType]) {
        revert FeePercentageExceedLimit();
    }

    functionFeeConfigs[_selector] = FeeConfig(_feeType, _feePercentage);

    emit FeeConfigSet(_selector, _feeType, _feePercentage);
}

/// @inheritdoc IFeeController
function setTokenGetter(bytes4 _selector, address _tokenGetter, address _target) external {
    if (_target == address(0) || _tokenGetter == address(0)) {
        revert ZeroAddressNotValid();
    }

    tokenGetters[_target][_selector] = _tokenGetter;
    emit TokenGetterSet(_target, _selector, _tokenGetter);
}

/// @inheritdoc IFeeController
function setGlobalTokenGetter(bytes4 _selector, address _tokenGetter) external {
    if (_tokenGetter == address(0)) {
        revert ZeroAddressNotValid();
    }

    globalTokenGetters[_selector] = _tokenGetter;
    emit GlobalTokenGetterSet(_selector, _tokenGetter);
}
```

Impact: Attackers can:

- Set the fee percentage to 0 and drain executors' revenue, or maximize it to grief smart accounts using the plugin.
- Cause DoS by setting the `tokenGetters` and `globalTokenGetters` to a contract that reverts on all calls

Recommended Mitigation: Add `onlyOwner` modifier to all state-changing admin setters inside `FeeController`.

OctoDeFi: Fixed in PR #12.

Cyfrin: Verified. The `onlyOwner` modifier has been applied to the `setFunctionFeeConfig()`, `setTokenGetter()`, and `setGlobalTokenGetter()` functions.

7.2 Medium Risk

7.2.1 Fee calculations break when token decimals are different from 18

Description: `FeeController.calculateFee()` and `calculateTokenAmount()` implicitly treat every ERC-20 as if it had **18 decimals**, but many popular assets (USDC/USDT = 6, etc.) do not. Inside `calculateFee` the line

```
uint256 _feeInUSD = _feeAmount * _tokenPrice / 10**18;
```

divides by `1e18` to cancel the **oracle's** 18-dec scaling, but completely ignores the token's own decimals. When the token has fewer decimals the resulting "USD value" is shrunk by `10**(18-decimals)`; when it has more decimals it is bloated.

Because `_executeStep()` **adds** the individual action fees returned by `_executeAction()`, any step that mixes tokens of different precision sums numbers that are not expressed in the same unit:

Token	Decimals	Fee for $100 volume at 1 % (expected $1 = 1e18)	Fee actually returned
USDT	6	1×10^{1}	1×10 (1 e12 × too low)
DAI	18	1×10^{1}	1×10^{1} (correct)

When a strategy step executes an action in USDT followed by an action in DAI the total fee becomes:

```
total = 1e6  (USDT)  + 1e18 (DAI)  = 1000000000001000000  wei-USD
```

but the **semantically correct** amount should be 2×10^{1}. Down-stream effects:

Impact: * `StrategyBuilderPlugin.executeAutomation()` compares the aggregated fee against the user-supplied `maxFeeInUSD`. A user can set `maxFeeInUSD = 1.1e18` and still execute both actions (paying the DAI part only) because the mis-scaled USDT fee barely moves the total.

- The executor misses on revenue on every 6/8/12-dec token; conversely users of exotic high-decimal tokens may be over-charged and see their automations revert with `FeeExceedMaxFee()`.

Recommended Mitigation:

1. **Use token.decimals() instead of 18.**

```
uint8 decimals = IERC20Metadata(token).decimals();    // OZ ERC20Metadata
uint256 scale = 10 ** decimals;                        // token's native unit

// feeAmount is already in token-units
uint256 _feeInUSD = _feeAmount * _tokenPrice / scale;   // always 18-dec USD
```

2. **Likewise in** `calculateTokenAmount()`

```
uint8 decimals = IERC20Metadata(token).decimals();
uint256 scale = 10 ** decimals;
return feeInUSD * scale / tokenPrice;
```

OctoDeFi: Fixed in PR #13.

Cyfrin: To normalize the fee in USD to the oracle decimals (18), the token decimals should be divided out in both instances. Also consider using a `staticcall` to query the token decimals and fall back to 18 if it fails.

OctoDeFi: Fixed in PR #27.

Cyfrin: Verified. The fee amount is first normalized to 18 decimals and then divided by the oracle decimals. While this is a little more convoluted than necessary given that the oracle price is in 18 already decimals, given that the fee calculation could have simply been divided by the token decimals instead of oracle decimals without first normalizing the fee to 18 decimals, it is now correct.

7.2.2 Missing staleness check in `PriceOracle::getTokenPrice`

Description: `PriceOracle.getTokenPrice()` relays `pythOracle.getPriceUnsafe()` but never verifies the price **publish time** or confidence interval. A price that is hours old (or intentionally frozen) is accepted as fresh.

```
function getTokenPrice(address _token) external view returns (uint256) {
    bytes32 _oracleID = oracleIDs[_token];

    if (_oracleID == bytes32(0)) {
        revert OracleNotExist(_token);
    }

    PythStructs.Price memory price = pythOracle.getPriceUnsafe(_oracleID);

    return _scalePythPrice(price.price, price.expo);
}
```

Impact: * Automation fees may be computed from obsolete data, letting attackers over- or under-pay.

Recommended Mitigation: Consider using the getPriceNoOlderThan function instead of getPriceUnsafe.

OctoDeFi: Fixed in PR #24.

Cyfrin: Verified. When the oracle reports prices older than 120 seconds, `PriceOracle` will return 0 such that the minimum fee will be used. This is a good solution, although a 0 price will cause panic revert due to division by zero in `calculateTokenAmount()`.

OctoDeFi: Fixed in PR #27.

Cyfrin: Verified. The 0 price case is now explicitly handled to avoid reverting and the threshold has been reduced to 60 seconds. It is possible that this threshold could be too restrictive for specific feeds, and it is recommended to be configured on a per-feed basis.

OctoDeFi: Regarding the threshold, we also believe that 60 seconds is quite restrictive, especially considering this is a fee calculation and not a vault with collateral at risk.

Most Pyth oracles update roughly every 1–2 seconds for major tokens. However, it's important to note that these are pull-based oracles, so smaller or less liquid tokens could have much slower update rates.

It might actually make sense to set the threshold much higher to really detect when an oracle is no longer functioning, rather than simply outdated. In any case, we can't fully prevent price manipulation through a single threshold alone.

Cyfrin: Acknowledged.

7.2.3 Automation DoS via blacklisted or reverting fee recipients

Description: `FeeHandler.handleFee()` uses `safeTransferFrom()` to forward ERC-20 tokens to `beneficiary`, `creator`, `vault`, and `burnerAddress`. If any of those addresses are **black-listed** by USDT/USDC (transfer returns false) the call reverts. `handleFeeETH()` uses `transfer()` which has a limitation of 2300 gas units; if the destination contract's `receive()` reverts, for example a smart contract wallet that executes more logic than could be covered by the gas limit, the whole automation fails.

```
IERC20(token).safeTransferFrom(msg.sender, beneficiary, beneficiaryAmount);

if (creator != address(0)) {
    IERC20(token).safeTransferFrom(msg.sender, creator, creatorAmount);
    IERC20(token).safeTransferFrom(msg.sender, vault, vaultAmount);
} else {
    IERC20(token).safeTransferFrom(msg.sender, vault, vaultAmount + creatorAmount);
}

if (burnAmount > 0) {
    IERC20(token).safeTransferFrom(msg.sender, burnerAddress, burnAmount);
}
```

```
payable(beneficiary).transfer(beneficiaryAmount);

if (creator != address(0)) {
    payable(creator).transfer(creatorAmount);
    payable(vault).transfer(vaultAmount);
} else {
    payable(vault).transfer(vaultAmount + creatorAmount);
}

if (burnAmount > 0) {
    payable(burnerAddress).transfer(burnAmount);
}
```

Impact: Denial of service on all future `executeAutomation()` calls for the strategies with a blacklisted or reverting `creator`.

Recommended Mitigation: Use a **pull** pattern where each entity could claim fees on its own instead of a push funds model, or `try/catch` around transfers so a single failing destination cannot block execution. Additionally avoid making native token transfers using `transfer()` but rather leverage some library that implements safe native token transfers.

OctoDeFi: Fixed in PR #14.

Cyfrin: Verified. A withdrawal method has been implemented to allow users to claim their accumulated fee balances. Note that native token transfers still rely on the `transfer()` method which should also be updated. Application of the `nonReentrant()` modifier is also not necessary.

OctoDeFi: Fixed in commit 7c48784.

Cyfrin: Verified. The Solady `SafeTransferLib` is now used for all token transfers.

7.2.4 DoS of strategy execution due to array indices out of bounds

Description: `StrategyBuilderPlugin._validateStep()` intends to enforce that the condition results reference a new index that does not exceed the maximum step index:

```
function _validateStep(StrategyStep memory step, uint256 maxStepIndex) internal pure {
    if (step.condition.result0 > maxStepIndex || step.condition.result1 > maxStepIndex) {
        revert InvalidNextStepIndex();
    }
}
```

However, given `maxStepIndex` is passed as the array length, this fails to account for zero indexing and as a it is possible to add a strategy that reverts during execution due to array index out of bounds if an off-by-one index is specified.

Impact: Strategies can be created that will cause DoS of execution.

Proof of Concept: The following test can be added to `StrategyBuilderPlugin.t.sol`:

```
function test_executeStrategy_OOB() external {
    uint256 numSteps = 2;
    IStrategyBuilderPlugin.StrategyStep[] memory steps = _createStrategySteps(numSteps);
    steps[0].condition.result0 = 2;
    steps[0].condition.result1 = 2;
    uint32 strategyID = 222;

    deal(address(account1), 100 ether);

    //Mocks
    vm.mockCall(
        feeController,
        abi.encodeWithSelector(IFeeController.getTokenForAction.selector),
        abi.encode(address(0), false)
    );

    vm.mockCall(
        feeController,
        abi.encodeWithSelector(IFeeController.functionFeeConfig.selector),
        abi.encode(IFeeController.FeeConfig({feeType: IFeeController.FeeType.Deposit, feePercentage:
        ↪  0}))
    );
    vm.mockCall(feeController, abi.encodeWithSelector(IFeeController.minFeeInUSD.selector),
    ↪  abi.encode(0));

    //Act
    vm.startPrank(address(account1));
    strategyBuilderPlugin.createStrategy(strategyID, creator, steps);

    strategyBuilderPlugin.executeStrategy(strategyID);
    vm.stopPrank();

    //Assert
    assertEq(tokenReceiver.balance, numSteps * 2 * TOKEN_SEND_AMOUNT);
}
```

Recommended Mitigation: should be >= to account for zero indexing

```
    function _validateStep(StrategyStep memory step, uint256 maxStepIndex) internal pure {
--      if (step.condition.result0 > maxStepIndex || step.condition.result1 > maxStepIndex) {
++      if (step.condition.result0 >= maxStepIndex || step.condition.result1 >= maxStepIndex) {
            revert InvalidNextStepIndex();
        }
    }
```

OctoDeFi: Fixed in PR #15.

Cyfrin: Verified. Validation now uses the correct operator.

7.2.5 Volume-based fee can be bypassed with a wrapper contract

Description: `_executeAction()` levies a percentage fee only when `FeeController.getTokenForAction()` can map the `(target, selector)` pair to a tracked token. If the mapping is absent it falls back to `minFeeInUSD`. An attacker can therefore wrap any high-value call inside a helper contract that the `FeeController` does not know about. *Example – Aave deposit:*

```
User → StrategyBuilder → AAVEHandler.supplyFor(100 000 USDC) → Aave Pool.supply()
```

`AAVEHandler` receives the user's 100 000 USDC, approves the Aave pool, and supplies on the user's behalf, returning aUSDC. Because `(AAVEHandler, supplyFor)` is not registered, `getTokenForAction()` returns `(address(0), false)`, so the strategy pays only the minimum fee instead of ~1 000 USDC (1 %). The same trick works for withdrawals, swaps, or any volume-based selector.

```
function _executeAction(address _wallet, Action memory _action) internal returns (uint256 feeInUSD) {
    (address tokenToTrack, bool exist) =
        feeController.getTokenForAction(_action.target, _action.selector, _action.parameter);
    // If the volume token exist track the volume before and after the execution, else get the min fee

    uint256 preExecBalance = exist ? IERC20(tokenToTrack).balanceOf(_wallet) : 0;

    _execute(_wallet, _action);

    IFeeController.FeeType feeType = feeController.functionFeeConfig(_action.selector).feeType;
    if (exist) {
        uint256 postExecBalance = IERC20(tokenToTrack).balanceOf(_wallet);
        uint256 volume = feeType == IFeeController.FeeType.Deposit
            ? preExecBalance - postExecBalance
            : postExecBalance - preExecBalance;

        feeInUSD = feeController.calculateFee(tokenToTrack, _action.selector, volume);
    } else {
        feeInUSD = feeController.minFeeInUSD(feeType);
    }

    emit ActionExecuted(_wallet, _action);
}
```

Impact: Large transactions can be executed while paying the protocol's minimum flat fee, severely reducing or eliminating expected revenue for executors.

Recommended Mitigation: Because this stems from design choices rather than a simple coding bug, solving it on-chain is non-trivial. It is to document the behavior and discuss any design adjustment that can remediate its risk.

OctoDeFi: Fixed in PR #25.

Cyfrin: Verified. The `ActionRegistry` contract has been added to validate action contracts allowed to be integrated into `StrategyBuilderPlugin`.

7.3 Low Risk

7.3.1 Accumulation of dust amounts due to rounding in `_tokenDistribution()`

Description: `_tokenDistribution()` computes each share independently with integer division:

```
function _tokenDistribution(uint256 amount) internal view returns (uint256, uint256, uint256) {
    uint256 beneficiaryAmount = (amount * beneficiaryPercentage) / PERCENTAGE_DIVISOR;
    uint256 creatorAmount = (amount * creatorPercentage) / PERCENTAGE_DIVISOR;
    uint256 vaultAmount = (amount * vaultPercentage) / PERCENTAGE_DIVISOR;
    return (beneficiaryAmount, creatorAmount, vaultAmount);
}
```

If `beneficiaryPercentage + creatorPercentage + vaultPercentage == 10000` but any division truncates, `beneficiaryAmount + creatorAmount + vaultAmount < amount`, leaving 1–2 wei of "dust".

Impact: Dust gradually accumulating lost and never reaching the intended recipients.

Recommended Mitigation: Calculate `beneficiaryAmount` and `creatorAmount` as above, then set

```
vaultAmount = amount - beneficiaryAmount - creatorAmount;
```

to guarantee full distribution without dust.

OctoDeFi: Fixed in PR #16.

Cyfrin: Verified. The `vaultAmount` is now calculated as the remainder after deducting the beneficiary and creator amounts.

7.3.2 `automationsToIndex` storage is not correctly reset when deleting automations

Description: When deleting an automation, the logic pops the final element from the `_usedInAutomations` array but fails to correctly reset the `automationsToIndex` mapping storage. Regardless of whether the if statement executes, `automationsToIndex[automationSID]` should be reset to zero whenever there is a deletion:

```
    function _deleteAutomation(address wallet, uint32 id) internal {
        bytes32 automationSID = getStorageId(wallet, id);
        Automation memory _automation = automations[automationSID];

        uint32[] storage _usedInAutomations = strategiesUsed[getStorageId(wallet,
        ↪   _automation.strategyId)];

@>      uint32 _actualAutomationIndex = automationsToIndex[automationSID];
        uint256 _lastAutomationIndex = _usedInAutomations.length - 1;
        if (_actualAutomationIndex != _lastAutomationIndex) {
            uint32 _lastAutomation = _usedInAutomations[_lastAutomationIndex];
            _usedInAutomations[_actualAutomationIndex] = _lastAutomation;
@>          automationsToIndex[getStorageId(wallet, _lastAutomation)] = _actualAutomationIndex;
        }
        _usedInAutomations.pop();

        _changeAutomationInCondition(
            wallet, _automation.condition.conditionAddress, _automation.condition.id, id, false
        );

        delete automations[automationSID];

        emit AutomationDeleted(wallet, id);
    }
```

Impact: Impact is limited as it seems the mapping will simply be overwritten if the automation is ever added again at the same id, and given `_usedInAutomations` is correctly popped it does not seem that the entry will be erroneously referenced.

Recommended Mitigation: Reset the mapping value for the given automation storage id when deleting the corresponding automation.

OctoDeFi: Fixed in PR #17.

Cyfrin: Verified. The relevant `automationsToIndex` mapping key is now cleared alongside `automations`.

7.3.3 DoS of automation due to potential zero value transfer reverts

Description: Token burn percentages specified by `primaryTokenBurn` and `tokenBurn` are simply restricted to the range `(0, 100]`. If the burn percentage is configured to 100%, this can result in scenarios where the burn amount in `FeeHandler.handleFee()` returned by `_feeCalculation()` is non-zero while the total fee to be distributed between the beneficiary/creator/vault is zero. While the standard OpenZeppelin ERC-20 implementation does not revert on zero value transfers, and it is unlikely that the burn percentages will be set that high anyway, it is possible and so is advisable to skip transfers in this case if any of the distribution amounts is zero to avoid DoS for token implementations that do revert on zero value transfers.

Impact: Automation can revert when attempting to make payment if the burn percentage is configured as 100%.

Recommended Mitigation: Similar to the validation on `burnAmount`, only attempt to perform token transfers if the respective beneficiary/creator/vault amounts are non-zero.

OctoDeFi: Fixed by PR #14. Due to the introduced pull-based method from M-3, this issue has also been resolved, as no tokens are transferred directly anymore.

Cyfrin: Verified. The push transfer pattern has been updated to a pull pattern that resolves this issue.

7.4 Informational

7.4.1 Zero price validation could be included in the price oracle

Description: `FeeController.calculateTokenAmount()` validates that the price returned by the oracle is non-zero:

```
    function calculateTokenAmount(address token, uint256 feeInUSD) external view returns (uint256) {
        bytes32 oracleID = oracle.oracleID(token);

        if (oracleID == bytes32(0)) {
            revert NoOracleExist();
        }

        uint256 tokenPrice = oracle.getTokenPrice(token);

@>      if (tokenPrice == 0) {
            revert InvalidTokenWithPriceOfZero();
        }

        return feeInUSD * 10 ** 18 / tokenPrice;
    }
```

Instead, this could be included in `PriceOracle._scalePythPrice()` which already validates the price is non-negative:

```
    function _scalePythPrice(int256 _price, int32 _expo) internal pure returns (uint256) {
@>      if (_price < 0) {
            revert NegativePriceNotAllowed();
        }
        ...
    }
```

Recommended Mitigation:

```
    function _scalePythPrice(int256 _price, int32 _expo) internal pure returns (uint256) {
--      if (_price < 0) {
++      if (_price <= 0) {
            revert NegativePriceNotAllowed();
        }
        ...
    }

    function calculateTokenAmount(address token, uint256 feeInUSD) external view returns (uint256) {
        ...
--      if (tokenPrice == 0) {
--          revert InvalidTokenWithPriceOfZero();
--      }

        return feeInUSD * 10 ** 18 / tokenPrice;
    }
```

OctoDeFi: Fixed in PR #18.

Cyfrin: Verified. The validation is now performed exclusively within the price oracle.

7.4.2 Positive Pyth oracle exponents should be explicitly handled

Description: `PriceOracle._scalePythPrice()` assumes that the exponent `_expo` will always be negative:

```
    function _scalePythPrice(int256 _price, int32 _expo) internal pure returns (uint256) {
        if (_price < 0) {
            revert NegativePriceNotAllowed();
        }

@>      uint256 _absExpo = uint32(-_expo);

        if (_expo <= -18) {
            return uint256(_price) * (10 ** (_absExpo - 18));
        }

        return uint256(_price) * 10 ** (18 - _absExpo);
    }
```

While this assumption is not likely to be violated, it is possible for the exponent to be configured as a positive value based on its signed type and usage in other libraries.

If the protocol were to ever rely on a Pyth oracle with a positive exponent then `uint32(-expo)` could silently underflow, resulting in a huge absolute value and causing execution to revert during the final scaling.

Recommended Mitigation: Consider explicitly handling the case where the exponent is positive.

OctoDeFi: Fixed in PR #19.

Cyfrin: Verified. The positive exponent case is now explicitly handled.

7.4.3 Empty strategies can be created due to missing zero length check

Description: `StrategyBuilderPlugin.createStrategy()` does not currently revert if the `steps` array length is zero. This means strategies can be created that can never be deleted, per the `strategyExist` modifier logic that is applied to `StrategyBuilderPlugin.deleteStrategy()` checking the steps length (which is not enforced to be non-zero):

```
modifier strategyExist(address wallet, uint32 id) {
    if (strategies[getStorageId(wallet, id)].steps.length == 0) {
        revert StrategyDoesNotExist();
    }
    _;
}
```

Impact: Impact is limited as strategies cannot be created on behalf of other wallets; however this behavior is most likely undesirable.

Proof of Concept: The following test can be added to `StrategyBuilderPlugin.t.sol`:

```
function test_createStrategy_Empty() external {
    uint256 numSteps;
    IStrategyBuilderPlugin.StrategyStep[] memory steps = _createStrategySteps(numSteps);

    uint32 strategyID = 222;
    vm.prank(address(account1));
    strategyBuilderPlugin.createStrategy(strategyID, creator, steps);

    //Assert
    IStrategyBuilderPlugin.Strategy memory strategy = strategyBuilderPlugin.strategy(address(account1),
    ↪  strategyID);

    assertEq(strategy.creator, creator);
    assertEq(strategy.steps.length, numSteps);

    vm.prank(address(account1));
    vm.expectRevert(IStrategyBuilderPlugin.StrategyDoesNotExist.selector);
    strategyBuilderPlugin.deleteStrategy(strategyID);
}

function test_createStrategy_EmptyNonZeroLength() external {
    uint256 numSteps = 2;
    IStrategyBuilderPlugin.StrategyStep[] memory steps = new
    ↪  IStrategyBuilderPlugin.StrategyStep[](numSteps);

    uint32 strategyID = 222;
    vm.prank(address(account1));
    strategyBuilderPlugin.createStrategy(strategyID, creator, steps);

    //Assert
    IStrategyBuilderPlugin.Strategy memory strategy = strategyBuilderPlugin.strategy(address(account1),
    ↪  strategyID);

    assertEq(strategy.creator, creator);
    assertEq(strategy.steps.length, numSteps);

    vm.prank(address(account1));
    strategyBuilderPlugin.executeStrategy(strategyID);

    vm.prank(address(account1));
    strategyBuilderPlugin.deleteStrategy(strategyID);
}
```

Recommended Mitigation: Consider reverting if the `steps` length is zero. Note that it would still be possible to create what is effectively an empty strategy with non-zero array length by leveraging default condition address as the zero address as shown in the second PoC.

OctoDeFi: Fixed in PR #20.

Cyfrin: Verified. Additional validation has been implemented to prevent empty strategy steps without conditions or actions.

7.4.4 Condition addresses can re-enter `StrategyBuilderPlugin`

Description: Condition addresses can re-enter `StrategyBuilderPlugin` from the external call in `_changeStrategyInCondition()` and similarly in `_changeAutomationInCondition()`. In the latter case, it seems that the worst thing that can happen here is duplicating the array entry when pushing to `strategiesUsed` which corrupts `automationsToIndex` and prevents the duplicate from being removed (unless there is re-entrancy during the deletion as well). Impact is therefore limited, but it is important to be aware of this when making any future modifications.

OctoDeFi: Acknowledged. We have taken note of the reentry issue.

Cyfrin: Acknowledged.

7.5 Gas Optimization

7.5.1 Unused cached value of `getStorageId(msg.sender, id)`

Description: `createAutomation()` calls `getStorageId(msg.sender, id)` twice—once for `automationSID` and again when assigning `_newAutomation`. This duplicates the same keccak 256 computation and extra stack writes.

Recommended Mitigation: Store the first return value in a local variable and reuse it:

```
bytes32 automationSID = getStorageId(msg.sender, id);
Automation storage _newAutomation = automations[automationSID];
```

This saves one `STATICCALL/KECCAK256` operation and a few stack ops per invocation.

OctoDeFi: Fixed in PR #21.

Cyfrin: Verified. The cached value is now used.

7.5.2 Use named return variables

Description: There are a number of instances where named return variables could be used to avoid unnecessary stack variable assignments, for example in `FeeHandler._tokenDistribution()`. Consider modifying this and other relevant functions to save gas.

OctoDeFi: Fixed in PR #22.

Cyfrin: Verified. Name return variables are now used.

7.5.3 Array length used in multiple loop iterations can be cached

Description: When validating strategy steps, the length is retrieved for each loop iteration when it could instead be cached to save gas:

```
function _validateSteps(StrategyStep[] memory steps) internal pure {
    for (uint256 i = 0; i < steps.length; i++) {
        _validateStep(steps[i], steps.length);
    }
}
```

OctoDeFi: Fixed in PR #23.

Cyfrin: Verified. The length is now cached.