Auto Redemption Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Giovanni Di Siena

January 13, 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 Standard is an overcollateralized lending protocol and issuer of the `USDs` stablecoin. Smart Vault owners can borrow `USDs` against their deposited collateral, swap directly between accepted collateral tokens, and deposit their collateral into selected Gamma Vaults (aka Hypervisors) for additional yield-earning opportunities. Auto redemption is a new feature that swaps Smart Vault collateral directly to `USDs` to repay debt at a discount when the stablecoin value falls below the target threshold.

5 Audit Scope

Cyfrin conducted an audit of The Standard Smart Vault based on the code present in the repository commit hash bfec2ad.

The following were included in the scope of the audit:

- AutoRedemption.sol
- SmartVaultManagerV6.sol#L107-L114
- SmartVaultV4.sol#L299-L377
- SmartVaultV4Legacy.sol#L293-L321
- SmartVaultYieldManager.sol#L321-L332

6 Executive Summary

Over the course of 8 days, the Cyfrin team conducted an audit on the Auto Redemption smart contracts provided by The Standard DAO. In this period, a total of 21 issues were found.

This review of The Standard Smart Vault contracts, namely the diff containing the new Auto Redemption functionality, yielded two critical vulnerabilities that arose due to a misconfiguration and failure to consider the interactions of other existing functionality within the system. Specifically, collateral swaps intended to output `USDs` were incorrectly specified with paths to `USDC` which would have resulted in no debt being repaid and vaults becoming liquidatable as `USDC` is not a valid collateral token. The logic for redemption of Hypervisor collateral was particularly problematic as this relied on functionality already present within `SmartVaultYieldManager` without considering the assumption that it is the responsibility of the calling function(s) to handle slippage. This oversight would have also resulted in vaults becoming liquidatable due to auto redemption, likely leading to loss of user funds, and so should be addressed as a priority.

Five high severity issues were also identified, relating again to misconfiguration, insufficient access control, and assumptions about the behavior of external Chainlink integrations. While aspects of the mitigation for these are less clear, they too should be addressed to ensure the security and stability of the protocol. A number of additional findings were raised, largely related to missing validation which is also strongly recommended to be addressed.

The Foundry fork test suite was found to be sparse, covering only the core auto redemption functionality without considering any of the peripheral code involved in its integration. Both tests are currently failing and if written correctly then it is likely that a large number of the issues identified in the audit would have been caught by a more comprehensive test suite. It is recommended that the test suite be expanded to cover all new functionality, mocking the Chainlink integration where required and appropriate.

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. Due to the number of issues identified, the non-trivial changes required during mitigation, and the short turnaround time for reviewing the mitigation fixes, it is recommended that a competitive audit be undertaken prior to deploying significant monetary capital to production.

Summary

Project Name	Auto Redemption
Repository	smart-vault
Commit	bfec2ad17fb7...
Audit Timeline	Dec 9th - Dec 18th
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	4
Medium Risk	5
Low Risk	2
Informational	8
Gas Optimizations	0
Total Issues	21

Summary of Findings

[C-1] Hypervisor collateral redemption can cause vaults to become undercollateralized due to slippage	Resolved
[C-2] Vaults can be made erroneously liquidatable due to incorrect swap path	Resolved
[H-1] `AutoRedemption` mappings are not and can never be populated	Resolved
[H-2] Auto redemption logic can be abused by an attacker due to insufficient access control	Resolved
[H-3] `AutoRedemption::fulfillRequest` should never be allowed to revert	Resolved
[H-4] Auto redemption functionality broken by incorrect address passed to `SmartVaultV4::autoRedemption`	Resolved
[M-1] Chainlink Functions HTTP request is missing authentication	Resolved
[M-2] Automation and redemption could be artificially manipulated due to use of instantaneous `sqrtPriceX96`	Resolved
[M-3] `USDs` redemption calculation can be manipulated due to unsafe signed-unsigned cast	Resolved
[M-4] Incorrect collateral quote amounts due to use of incorrect swap path	Resolved
[M-5] Auto redemption will not function due to slippage misconfiguration	Resolved
[L-1] Concentrated liquidity tick logic is incorrect	Acknowledged
[L-2] Additional validation should be performed on the Chainlink Functions response	Resolved
[I-1] `AutoRedemption::calculateUSDsToTargetPrice` could be refactored to avoid repeated logic	Resolved
[I-2] Incorrectly named return variable `AutoRedemption::calculateUSDsToTargetPrice`	Resolved
[I-3] Unused response parameter should be removed	Resolved
[I-4] Unused function argument in `SmartVaultYieldManager::quickDeposit` should be removed	Resolved
[I-5] Incorrect `SmartVaultV4` function arguments should be renamed	Resolved
[I-6] The vault limit condition should not be checked for `address(0)`	Resolved
[I-7] Unused return value can be removed	Resolved
[I-8] Redemption of Hypervisor collateral can be suboptimal	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Hypervisor collateral redemption can cause vaults to become undercollateralized due to slippage

Description: When redeeming Hypervisor collateral, `SmartVaultV4::autoRedemption` performs a call to `SmartVaultYieldManager::quickWithdraw`:

```
if (_hypervisor != address(0)) {
    address _yieldManager = ISmartVaultManager(manager).yieldManager();
    IERC20(_hypervisor).safeIncreaseAllowance(_yieldManager, getAssetBalance(_hypervisor));
    _withdrawn = ISmartVaultYieldManager(_yieldManager).quickWithdraw(_hypervisor, _collateralToken);
    IERC20(_hypervisor).forceApprove(_yieldManager, 0);
}
```

This invokes `SmartVaultYieldManager::_withdrawOtherDeposit` to withdraw the underlying collateral without applying the additional protocol withdrawal fee:

```
function quickWithdraw(address _hypervisor, address _token) external returns (uint256 _withdrawn) {
    IERC20(_hypervisor).safeTransferFrom(msg.sender, address(this),
    ↪  IERC20(_hypervisor).balanceOf(msg.sender));
    _withdrawOtherDeposit(_hypervisor, _token);
    uint256 _withdrawn = _thisBalanceOf(_token);
    IERC20(_token).safeTransfer(msg.sender, _withdrawn);
}
```

After auto redemption is complete, the excess underlying collateral is redeposited to the Hypervisor through `SmartVaultV4::redeposit`:

```
function redeposit(uint256 _withdrawn, uint256 _collateralBalance, address _hypervisor, address
↪  _collateralToken)
    private
{
    uint256 _redeposit = _withdrawn > _collateralBalance ? _collateralBalance : _withdrawn;
    address _yieldManager = ISmartVaultManager(manager).yieldManager();
    IERC20(_collateralToken).safeIncreaseAllowance(_yieldManager, _redeposit);
    ISmartVaultYieldManager(_yieldManager).quickDeposit(_hypervisor, _collateralToken, _redeposit);
    IERC20(_collateralToken).forceApprove(_yieldManager, 0);
}
```

This performs a call to `SmartVaultYieldManager::quickDeposit` which similarly invokes `SmartVaultYieldManager::_otherDeposit_` to deposit the underlying collateral without applying the additional protocol deposit fee:

```
function quickDeposit(address _hypervisor, address _collateralToken, uint256 _deposit) external {
    IERC20(_collateralToken).safeTransferFrom(msg.sender, address(this), _deposit);
    HypervisorData memory _hypervisorData = hypervisorData[_collateralToken];
    _otherDeposit(_collateralToken, _hypervisorData);
}
```

The issue with this logic is that the `SmartVaultYieldManager` functions assume slippage is sufficiently handled by calling function:

```
// within _withdrawOtherDeposit()
IHypervisor(_hypervisor).withdraw(
    _thisBalanceOf(_hypervisor), address(this), address(this), [uint256(0), uint256(0), uint256(0),
    ↪  uint256(0)]
```

```
);

// within _swapToSingleAsset(), called from within _withdrawOtherDeposit()
// similar is present within _buy() and _sell() called from within _otherDeposit() -> _swapToRatio()
ISwapRouter(uniswapRouter).exactInputSingle(
    ISwapRouter.ExactInputSingleParams({
        tokenIn: _unwantedToken,
        tokenOut: _wantedToken,
        fee: _fee,
        recipient: address(this),
        deadline: block.timestamp + 60,
        amountIn: _balance,
        amountOutMinimum: 0,
        sqrtPriceLimitX96: 0
    })
);

// within _deposit(), called from within _otherDeposit()
IUniProxy(uniProxy).deposit(
    _thisBalanceOf(_token0),
    _thisBalanceOf(_token1),
    msg.sender,
    _hypervisor,
    [uint256(0), uint256(0), uint256(0), uint256(0)]
);
```

The current logic assumes vaults cannot become undercollateralized as a result of auto redemption and performs no such validation; however, an attacker can cause a vault to become undercollateralized by taking advantage of the lack of slippage handling.

A collateralization check should be performed immediately after the execution of redeposit logic due to empty slippage parameters.

Impact: Auto redemption of Hypervisor collateral can result in undercollateralized vaults.

Recommended Mitigation: Add additional validation to prevent vaults from becoming undercollateralized as a result of auto redemption.

The Standard DAO: Fixed by commit b71beb0.

Cyfrin: This prevents the new `SmartVaultV4` with Hypervisor deposits from becoming undercollateralized due to auto redemption, however this does not prevent a significant collateral drawdown to just above the liquidation threshold. As an aside, we should probably also validate that auto redemption is not trying to swap collateral for a vault that is already undercollateralized but not liquidated – prefer a significant drawdown check here instead.

The Standard DAO: Fixed by commits b92c98c, cd6bd7c, and 132a013.

Cyfrin: The collateral percentage runtime invariant check has been added; however, the case where vault debt is fully repaid needs to be handled explicitly to avoid division by zero in `SmartVaultV4::calculateCollaralPercentage`.

The Standard DAO: Fixed by commit aa4d5df.

Cyfrin: Verified. The collateral percentage validation will no longer execute if `minted` is zero.

7.1.2 Vaults can be made erroneously liquidatable due to incorrect swap path

Description: When auto redemption is executed, the `Quoter::quoteExactOutput` and `Quoter::quoteExactInput` functions of the Uniswap v3 view-only quoter are invoked for legacy and non-legacy vaults respectively. Given that these calls do not revert, execution will proceed to performing the swap; however, the swap path of `collateral -> USDC` is incorrect when attempting to swap to `USDs`:

```
_amountOut = ISwapRouter(_swapRouterAddress).exactInput(
    ISwapRouter.ExactInputParams({
        path: _swapPath,
        recipient: address(this),
        deadline: block.timestamp,
        amountIn: _collateralAmount,
        // minimum amount out should be at least usd value of collateral being swapped in
        amountOutMinimum: calculator.tokenToUSD(
            ITokenManager(ISmartVaultManager(manager).tokenManager()).getTokenIfExists(_collateralAddr),
            _collateralAmount
        )
    })
);
```

The swap will succeed, converting the collateral to `USDC`, but no debt will be redeemed as the `USDs` balance of the vault has not changed:

```
uint256 _usdsBalance = USDs.balanceOf(address(this));
minted -= _usdsBalance;
USDs.burn(address(this), _usdsBalance);
```

Given that `USDC` is not a valid collateral token, this will result in the vault becoming liquidatable if a sufficient amount of the collatetal token is swapped.

Impact: Vaults can become erroneously liquidatable as a result of auto redemption.

Recommended Mitigation: Ensure that the swap path is correct when performing auto redemption and add a check to ensure that the vault collateralization has not significantly decreased as a result of auto redemption.

The Standard DAO: Fixed by commits 3c5e136, 821bb26, and 132a013. Since we are storing both input and output swap paths, we use that to more efficiently calculate the required amount to swap.

Cyfrin: Verified. With the introduction of the `SwapPath` struct containing both input and output paths along with the relevant swap path and target amount variables renamed to correctly reference `USDs` instead of `USDC`, this now appears to be correct so long as paths are configured as communicated (e.g. input: `WETH -> USDC -> USDs`, output: `USDs -> USDC -> WETH`).

7.2 High Risk

7.2.1 `AutoRedemption` mappings are not and can never be populated

Description: The following mappings are declared within the `AutoRedemption` contract:

```
mapping(address => address) hypervisorCollaterals;
mapping(address => bytes) swapPaths;
```

However, they are never populated and there are no public functions capable of doing so either.

When a request is triggered, `lastRequestId` is assigned a non-zero identifier:

```
lastRequestId = _sendRequest(req.encodeCBOR(), subscriptionID, MAX_REQ_GAS, donID);
```

But the execution of `AutoRedemption::fulfillRequest` will revert due to empty swap paths, so the `lastRequestId` storage will not be reset:

```
bytes memory _collateralToUSDCPath = swapPaths[_token];
...
lastRequestId = bytes32(0);
```

Given the condition within `AutoRedemption::performUpkeep` that there must not be an existing unfulfilled request when creating a new one, this completely blocks all functionality and necessitates a complete redeployment.

Impact: Core auto redemption functionality will be broken for all vaults and cannot be fixed without redeployment.

Recommended Mitigation: Either:

- Pre-populate the mappings,
- Add access-controlled setter functions, or
- Query state from the `SmartVaultYieldManager` contract.

The Standard DAO: Fixed by commit d72cdce.

Cyfrin: Verified. The setter functions have been added.

7.2.2 Auto redemption logic can be abused by an attacker due to insufficient access control

Description: `AutoRedemption::performUpkeep` is exposed for use by the Chainlink Automation DON when upkeep is required:

```
function performUpkeep(bytes calldata performData) external {
    if (lastRequestId == bytes32(0)) {
        triggerRequest();
    }
}
```

However, there is an absence of access control that allows the function to be called by an address. Since `lastRequestId` is reset to `bytes32(0)` at the end of `AutoRedemption::fulfillRequest` execution, this means that upkeep can be repeatedly performed after the previous one has succeeded, regardless the trigger condition.

If `AutoRedemption::fulfillRequest` reverts, the `lastRequestId` state will not be reset which completely blocks all future auto redemptions due to the conditional in `AutoRedemption::performUpkeep` shown above. Combined with the use of `ERC20::balanceOf` within both `SmartVaultV4Legacy::autoRedemption` and `SmartVaultV4::autoRedemption` to determine the amount `USDs` repaid, an attacker can force this DoS condition by sending a small amount of `USDs` directly to the target vault:

```
uint256 _usdsBalance = USDs.balanceOf(address(this));
minted -= _usdsBalance;
```

This causes the vault balance to be inflated above the expected maximum `minted` amount and execution to revert due to underflow. Since the Chainlink Functions DON will not retry failed fulfilment, there will be no way to reset the state and recover core functionality without complete redeployment.

Impact: An attacker can repeatedly trigger auto redemption regardless of the trigger condition and without relying on price oracle manipulation. This could amount to a loss of funds to the protocol since the Chainlink subscription will be billed on every fraudulent fulfilment attempt. Alternatively, an attacker could completely block functionality of the auto redemption mechanism if fulfilment is made to revert.

Recommended Mitigation: * Re-check the trigger condition within `AutoRedemption::performUpkeep` and also consider adding access control.

- Calculate the amount of `USDs` repaid as the balance diff rather than using the vault balance directly.

The Standard DAO: Fixed by commit 5ec532e.

Cyfrin: The trigger condition is re-checked and a TWAP has been implemented, however:

- It is recommended to use a substantially large interval (at least 900 seconds, if not 1800 seconds) to protect against manipulation. Note: Uniswap V3 pool oracles are not multi-block MEV resistant.
- The `USDs` repayment amount calculation has not been modified to use balance diffs instead of direct `balanceOf()`.

The Standard DAO: Fixed by commit a8cdc77, using the amount out of the swap rather than balance checks.

Cyfrin: Verified. The TWAP interval has been increased and the redeemed amount has been modified to use the value output from the swap.

7.2.3 `AutoRedemption::fulfillRequest` should never be allowed to revert

Description: Given that the Chainlink Functions DON will not retry failed fulfilments, `AutoRedemption::fulfillRequest` should never be allowed to revert; otherwise, `lastRequestId` will not be reset to `bytes32(0)` which means `AutoRedemption::performUpkeep` will never be able to trigger new requests.

Currently, inline comments suggest that there is an intention to revert if the Chainlink Functions DON returns an error; however, this should be avoided for the reason explained above. Similarly, any potential malformed response or reverts caused by external calls should be handled gracefully and fall through to this line:

```
lastRequestId = bytes32(0);
```

Impact: Complete DoS of the auto redemption functionality.

Recommended Mitigation: * Do **not** revert if an error is reported.

- Validate the response against its expected length to ensure that the decoding does not revert.
- Handle reverts from all external calls using `try/catch` blocks.
- Short-circuit if `AutoRedemption::calculateUSDsToTargetPrice` returns `0` (since this will cause the swap to revert).
- Optionally add an access-controlled admin function to reset `lastRequestId`.

The Standard DAO: Fixed by commit 5235524.

Cyfrin: Verified. The response length is now validated against its expected length, which will also result in the logic being skipped if an error is reported. `AutoRedemption::runAutoRedemption` will only run if the target `USDs` amount is non-zero and other reverts from external calls are handled using `try/catch` blocks. An admin function to forcibly reset `lastRequestId` has not been added.

7.2.4 Auto redemption functionality broken by incorrect address passed to `SmartVaultV4::autoRedemption`

Description: `SmartVaultV4::autoRedemption` has the following signature:

```
function autoRedemption(
    address _swapRouterAddress,
    address _quoterAddress,
    address _collateralToken,
    bytes memory _swapPath,
    uint256 _USDCTargetAmount,
    address _hypervisor
)
```

However the invocation in `AutoRedemption::fulfillRequest` incorrectly passes the vault address in place of the swap router address:

```
IRedeemable(_smartVault).autoRedemption(
    _smartVault, quoter, _token, _collateralToUSDCPath, _USDsTargetAmount, _hypervisor
);
```

Impact: Auto redemption functionality will become completely broken when attempting to repay the debt of a non-legacy vault.

Recommended Mitigation: Pass the correct swap router address stored in the `SmartVaultManagerV6` contract.

The Standard DAO: Fixed by commit 4400e25.

Cyfrin: Verified. The correct `swapRouter` address is now passed.

7.3 Medium Risk

7.3.1 Chainlink Functions HTTP request is missing authentication

Description: The `source` constant defined within `AutoRedemption` is used to execute the corresponding JavaScript code within the Chainlink Functions DON; however, the target API endpoint is exposed without any form of authentication which allows any observer to send requests.

Impact: A coordinated DDoS attack on the API endpoint could result in the server going down. This will cause requests to fail, meaning the Chainlink subscription will be billed but the auto redemption peg mechanism will not function as intended.

Recommended Mitigation: At a minimum, implement rate limiting. Preferably add authentication to the request using Chainlink Functions secrets.

The Standard DAO: Partially fixed by adding rate limiting to the API. Will also consider later adding an encrypted secret to the request.

Cyfrin: Acknowledged. Use of encrypted secrets is recommended.

7.3.2 Automation and redemption could be artificially manipulated due to use of instantaneous `sqrtPriceX96`

Description: `AutoRedemption::checkUpkeep` is queried every block by the Chainlink Automation DON:

```
function checkUpkeep(bytes calldata checkData) external returns (bool upkeepNeeded, bytes memory
↪  performData) {
    (uint160 sqrtPriceX96,,,,,,) = pool.slot0();
    upkeepNeeded = sqrtPriceX96 <= triggerPrice;
}
```

However, use of `sqrtPriceX96` is problematic since it is based on the instantaneous pool reserves which can be manipulated to force upkeep into triggering even if it is not required. Additionally, this instantaneous price is used in calculation of the `USDs` that needs to be bought to reach the target price:

```
function calculateUSDsToTargetPrice() private view returns (uint256 _usdc) {
    int24 _spacing = pool.tickSpacing();
    (uint160 _sqrtPriceX96, int24 _tick,,,,,) = pool.slot0();
    int24 _upperTick = _tick / _spacing * _spacing;
    int24 _lowerTick = _upperTick - _spacing;
    uint128 _liquidity = pool.liquidity();
    ...
}
```

So despite the latency between checking and fulfilling upkeep, and additional latency in fulfilling the Functions request, the `USDs` price can be manipulated without relying on multi-block manipulation. Combined with an attacker providing just-in-time (JIT) liquidity, the calculation can be manipulated to redeem the entire debt of a vault as capped within `AutoRedemption::fulfillRequest`:

```
if (_USDsTargetAmount > _vaultData.status.minted) _USDsTargetAmount = _vaultData.status.minted;
```

Impact: A vault owner could manipulate the `USDs/USDC` pool reserves to force auto redemption even if it is not required. If their vault is the one returned by the off-chain service, they will have caused their debt to be repaid while circumventing fees. Note that other MEV attacks that have not been explored in this analysis may also be possible.

Recommended Mitigation: Re-check the upkeep condition within `AutoRedemption::performUpkeep` and `AutoRedemption::fulfilRequest` to minimize the impact of isolated price oracle manipulation; however, note that

repeated manipulation does not necessarily require multi-block manipulation. Therefore, additionally consider consuming a time-weighted average price or work with Chainlink Labs to create a decentralized USDs price oracle.

The Standard DAO: Fixed by commit 5ec532e.

Cyfrin: The trigger condition is re-checked and a TWAP has been implemented; however, it is recommended to use a substantially large interval (at least 900 seconds, if not 1800 seconds) to protect against manipulation. Note: Uniswap V3 pool oracles are not multi-block MEV resistant.

The Standard DAO: Fixed by commit a8cdc77.

Cyfrin: Verified. The TWAP interval has been increased.

7.3.3 USDs redemption calculation can be manipulated due to unsafe signed-unsigned cast

Description: The following logic is executed when calculating the USDs redemption amount, after the tick range has been advanced beyond that of the current tick:

```
} else {
    (, int128 _liquidityNet,,,,,) = pool.ticks(_lowerTick);
    _liquidity += uint128(_liquidityNet);
    (_amount0,) = LiquidityAmounts.getAmountsForLiquidity(
        _sqrtPriceX96,
        TickMath.getSqrtRatioAtTick(_lowerTick),
        TickMath.getSqrtRatioAtTick(_upperTick),
        _liquidity
    );
}
```

The _liquidityNet variable represents the net active liquidity delta when crossing between tick ranges, so this is considered along with the active liquidity of the original range; however, this value is cast from a signed integer to unsigned without actually checking the sign. Given that negative signed integers are represented using two's complement, net negative liquidities (i.e. less active liquidity in the subsequent range) will silently result in the incremented _liquidity being a very large number. This scenario could occur accidentally, or be forced by an attacker by the addition of just-in-time (JIT) liquidity, causing the target vault to be fully redeemed for the maximum USDs:

```
if (_USDsTargetAmount > _vaultData.status.minted) _USDsTargetAmount = _vaultData.status.minted;
```

Impact: In the best case, more debt could be repaid than intended. In the worst case, fulfilment could be made to revert which will permanently disable auto redemption functionality (as described in a separate issue).

Recommended Mitigation: The sign of _liquidityNet should be checked before incrementing _liquidity, subtracting the absolute value if it is negative. Consider using the LiquidityMath library.

The Standard DAO: Fixed by commit 49af007.

Cyfrin: Verified. The LiquidityMath library is now used.

7.3.4 Incorrect collateral quote amounts due to use of incorrect swap path

Description: Within AutoRedemption::legacyAutoRedemption, the amount of input collateral required is calculated as:

```
(uint256 _approxAmountInRequired,,,) =
IQuoter(quoter).quoteExactOutput(_collateralToUSDCPath, _USDsTargetAmount);
uint256 _amountIn = _approxAmountInRequired > _collateralBalance ? _collateralBalance :
↪  _approxAmountInRequired;
```

This quote is incorrect as it uses the `collateral -> USDC` swap path for calculation of a target `USDs` amount. Since `USDs` is expected to be below peg for this logic to be triggered, it will take more collateral to swap to `_USDsTargetAmount` amount of `USDC` compared to that of `USDs`. Furthermore, `Quoter::quoteExactOutput` expects the path to be reversed when calculating the amount of the input token required to be swapped for an exact amount of the output token.

Similarly, in `SmartVaultV4::calculateAmountIn`, the swap path is `collateral -> USDC` but the amount output is compared to a `USDs` target amount (which is incorrectly named as `_USDCTargetAmount`):

```
(uint256 _quoteAmountOut,,,) = IQuoter(_quoterAddress).quoteExactInput(_swapPath, _collateralBalance);
return _quoteAmountOut > _USDCTargetAmount
    ? _collateralBalance * _USDCTargetAmount / _quoteAmountOut
    : _collateralBalance;
```

Impact: The `_amountIn` variable will be inflated, causing more collateral to be swapped to `USDs` than needed to restore peg.

Recommended Mitigation: Use the reversed `collateral -> USDs` swap path to calculate the required amount in.

The Standard DAO: Fixed by commit 3c5e136.

Cyfrin: Verified. With the introduction of the `SwapPath` struct containing both input and output paths along with the relevant swap path and target amount variables renamed to correctly reference `USDs` instead of `USDC`, this now appears to be correct so long as paths are configured as communicated (e.g. input: `WETH -> USDC -> USDs`, output: `USDs -> USDC -> WETH`.

7.3.5 Auto redemption will not function due to slippage misconfiguration

Description: Both legacy and non-legacy vaults perform a call to `ISwapRouter::exactInput` when attempting to swap collateral to `USDs`, with the `amountOutMinimum` parameter being specified as the USD value of the collateral being swapped:

```
calculator.tokenToUSD(
    ITokenManager(ISmartVaultManager(manager).tokenManager()).getTokenIfExists(_collateralAddr),
    _collateralAmount
)
```

However, this fails to account for slippage incurred during the swap.

The further `USDs` is below peg, the more `USDs` will be redeemed for a given amount of collateral and so this will not likely present any issues. When swapping through high fee, lower liquidity pools, auto redemption will be more likely to fail, especially if `USDs` is closer to peg.

Therefore, auto redemption will likely succeed most of the time, although it may be possible that the slippage requirement of a `USDs` amount out equivalent to the USD value of the input collateral will occasionally cause auto redemption to fail if enough value is lost through price impact of the intermediate swaps and the difference in the effective `USDs` amount is negligible.

This means the impact on auto redemption as an effective peg mechanism is minimal, so long as redemption is not attempted on a vault that would need to route through low liquidity swap paths, since it is far more likely to succeed as the `USDs` price decreases.

Impact: Dependent on the fee tiers/liquidities and `USDs` price deviation, it may not be possible for auto redemption to function.

Recommended Mitigation: Consider passing a less restrictive slippage parameter to both calls and perform some variation on the `significantCollateralDrop()` validation at the end of execution.

The Standard DAO: Fixed by commit 88b2d9f. It's quite an arbitrary buffer, but there is no possibility of user input here. Again, ideally the amount of `USDs` bought should be more than the amount of collateral swapped in.

Cyfrin: Verified. The buffer has been applied to mitigate potential issues with low-liquidity/high-fee pools.

7.4 Low Risk

7.4.1 Concentrated liquidity tick logic is incorrect

Description: `AutoRedemption::calculateUSDsToTargetPrice` performs the following concentrated liquidity tick calculations to determine the current tick range corresponding to a given price:

```
int24 _spacing = pool.tickSpacing();
(uint160 _sqrtPriceX96, int24 _tick,,,,) = pool.slot0();
int24 _upperTick = _tick / _spacing * _spacing;
int24 _lowerTick = _upperTick - _spacing;
```

However, due to the behavior of signed integers in Solidity rounding *up* to zero rather than the next lowest integer, this logic is only correct for negative ticks and positive ticks that are an exact multiple of `_spacing`. Otherwise, if `tick` is a positive non-multiple of `_spacing`, the calculated `_upperTick` and `_lowerTick` will correspond to the range immediately below the true current range.

This error could cause the subsequent `while` loop to execute even if the current price is above the target price, or enter the incorrect conditional branch and consider net liquidity even when the current tick should be considered within range:

```
while (TickMath.getSqrtRatioAtTick(_lowerTick) < TARGET_PRICE) {
    uint256 _amount0;
    if (_tick > _lowerTick && _tick < _upperTick) {
        (_amount0,) = LiquidityAmounts.getAmountsForLiquidity(
            _sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(_lowerTick),
            TickMath.getSqrtRatioAtTick(_upperTick),
            _liquidity
        );
    } else {
        (, int128 _liquidityNet,,,,,) = pool.ticks(_lowerTick);
        _liquidity += uint128(_liquidityNet);
        (_amount0,) = LiquidityAmounts.getAmountsForLiquidity(
            _sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(_lowerTick),
            TickMath.getSqrtRatioAtTick(_upperTick),
            _liquidity
        );
    }
    _usdc += _amount0;
    _lowerTick += _spacing;
    _upperTick += _spacing;
}
```

Fortunately, this is highly unlikely to occur due to the difference in `USDs` and `USDC` decimals (`18` and `6` respectively). In calculation of `sqrtPriceX96`, which is a fixed point `Q64.96` number representing the square root of the ratio of the two pool assets (`token1/token0`), `USDs` is `token0` and `USDC` is `token1`. Given that the relative value of the two pool assets is expected to be reasonably stable, with the `TARGET_PRICE` defined as `79228162514264337593543` which corresponds to a ratio of `1e-12`, the square root price will realistically always be on the order 10^{-6}:

$$\text{price ratio (token1/token0)} = \frac{10^{\text{USDC decimals}}}{10^{\text{USDs decimals}}} = \frac{10^{6}}{10^{18}} = 10^{-12}$$

$$\sqrt{\text{price ratio}} = \sqrt{10^{-12}} = 10^{-6}$$

This means the pool tick should realistically always be negative considering that a tick represents the logarithmic index of the price ratio, and the logarithm is negative since the ratio 10^{-12} is much smaller than 1:

$$\text{tick}=\log_{ \sqrt{(1.0001)}}(\text{price ratio})$$

It would take a de-peg event of unrealistic magnitude to cause this to be an issue, although `USDC` is upgradeable and so this caveat should not be relied upon.

Impact: The `USDs` calculation could be affected by errors in the calculation of tick ranges.

Recommended Mitigation: Modify the logic to consider the sign of `_tick` when calculating tick ranges.

The Standard DAO: Acknowledged. `AutoRedemption` will be deployed with trigger price of at negative tick, target price is at negative tick. Upkeep is only required between trigger price tick (or lower) and target price tick. Ticks are therefore not going to be positive.

Cyfrin: Acknowledged, based on the assumption that Circle does not update the decimals of `USDC`.

7.4.2 Additional validation should be performed on the Chainlink Functions response

Description: When consuming a Chainlink Functions response, the following additional validation should be performed within `AutoRedemption::fulfillRequest`:

- Ensure the `_token` address is either a Hypervisor with valid data on the `SmartVaultYieldManager` contract (valid only for non-legacy vaults) or an accepted collateral token.
- Ensure the `_tokenID` has actually been minted such that `SmartVaultManagerV6::vaultData` will return a non-default `SmartVaultData` struct.

The Standard DAO Fixed by commit 8ee0921.

Cyfrin: Verified. `AutoRedemption::validData` has been added to verify that the vault address is non-zero and the token is a valid collateral token.

7.5 Informational

7.5.1 `AutoRedemption::calculateUSDsToTargetPrice` could be refactored to avoid repeated logic

Description: When calculating the target `USDs` amount, the call to `LiquidityAmounts::getAmountsForLiquidity` is common to both conditional branches, with the liquidity parameter being the only difference:

```
uint128 _liquidity = pool.liquidity();
while (TickMath.getSqrtRatioAtTick(_lowerTick) < TARGET_PRICE) {
    uint256 _amount0;
    if (_tick > _lowerTick && _tick < _upperTick) {
        (_amount0,) = LiquidityAmounts.getAmountsForLiquidity(
            _sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(_lowerTick),
            TickMath.getSqrtRatioAtTick(_upperTick),
            _liquidity
        );
    } else {
        (, int128 _liquidityNet,,,,,) = pool.ticks(_lowerTick);
        _liquidity += uint128(_liquidityNet);
        (_amount0,) = LiquidityAmounts.getAmountsForLiquidity(
            _sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(_lowerTick),
            TickMath.getSqrtRatioAtTick(_upperTick),
            _liquidity
        );
    }
    ...
}
```

This could be refactored to avoid repeated code such that it is only the liquidity calculations that remain in the conditionals.

The Standard DAO: Fixed by commit e48ccff.

Cyfrin: Verified. The logic has been correctly refactored to only consider the net liquidity delta when the current tick is below the range.

7.5.2 Incorrectly named return variable `AutoRedemption::calculateUSDsToTargetPrice`

Description: `AutoRedemption::calculateUSDsToTargetPrice` has the following signature:

```
function calculateUSDsToTargetPrice() private view returns (uint256 _usdc)
```

However, the named return variable is semantically incorrect and should instead be `_usds` to avoid confusion.

The Standard DAO: Fixed by commit a03f0d5.

Cyfrin: Verified. The return variable has been renamed.

7.5.3 Unused response parameter should be removed

Description: The `_estimatedCollateralValueUSD` parameter decoded from the Chainlink Functions response and passed as an argument to `AutoRedemption::legacyAutoRedemption` is not used and can be removed:

```
(uint256 _tokenID, address _token, uint256 _estimatedCollateralValueUSD) =
    abi.decode(response, (uint256, address, uint256));
...
legacyAutoRedemption(
    _smartVault, _token, _collateralToUSDCPath, _USDsTargetAmount, _estimatedCollateralValueUSD
```

```
);
```

The Standard DAO: Fixed by commit 59c4bd5.

Cyfrin: Verified. The parameter has been removed and the source/decoding/signatures have been updated accordingly.

7.5.4 Unused function argument in `SmartVaultYieldManager::quickDeposit` should be removed

Description: The `SmartVaultYieldManager::quickDeposit` function has the following signature:

```
function quickDeposit(address _hypervisor, address _collateralToken, uint256 _deposit)
```

However, the `_hypervisor` argument is never used and so can be removed.

The Standard DAO: Fixed by commit 6f943c5.

Cyfrin: Verified. The unused address has been removed.

7.5.5 Incorrect `SmartVaultV4` function arguments should be renamed

Description: The following functions in `SmartVaultV4` take an argument `_USDCTargetAmount`:

- `calculateAmountIn()`
- `swapCollateral()`
- `autoRedemption()`

However, this is semantically incorrect as it is intended to represent the target `USDs` redemption amount and so should be renamed to avoid confusion.

The Standard DAO: Fixed by commit a03f0d5.

Cyfrin: Verified. The function arguments have been renamed.

7.5.6 The vault limit condition should not be checked for `address(0)`

Description: The virtual function `ERC721Upgradeable::_update` has been overridden within `SmartVaultManagerV6` as follows:

```
function _update(address _to, uint256 _tokenID, address _auth) internal virtual override returns
↪  (address) {
    address _from = super._update(_to, _tokenID, _auth);
    require(vaultIDs(_to).length < userVaultLimit, "err-vault-limit");
    smartVaultIndex.transferTokenId(_from, _to, _tokenID);
    if (address(_from) != address(0))
    ↪  ISmartVault(smartVaultIndex.getVaultAddress(_tokenID)).setOwner(_to);
    emit VaultTransferred(_tokenID, _from, _to);
    return _from;
}
```

However, the `userVaultLimit` validation should not be performed for `address(0)` otherwise it will become impossible to burn more than this number of tokens. Fortunately, such functionality is not currently exposed and it is not possible to transfer tokens directly to `address(0)` due to do validation within the OpenZeppelin contract. Nevertheless, this edge case should be proactively avoided.

Similarly, the call to `SmartVaultIndex::removeTokenId` within `SmartVaultIndex::transferTokenId` should be skipped if the `_from` is `address(0)` and pushing to the `tokenIds` array should be skipped if `_to` is `address(0)`:

```
function transferTokenId(address _from, address _to, uint256 _tokenId) external onlyManager {
    removeTokenId(_from, _tokenId);
    tokenIds[_to].push(_tokenId);
}
```

The Standard DAO: Fixed by commit ff4ef5b.

Cyfrin: Verified. The validation will now pass for the zero address.

7.5.7 Unused return value can be removed

Description: The `IRedeemableLegacy::autoRedemption` function signature is as follows:

```
function autoRedemption(
    address _swapRouterAddress,
    address _collateralAddr,
    bytes memory _swapPath,
    uint256 _amountIn
) external returns (uint256 _redeemed);
```

Within `SmartVaultV4Legact::autoRedemption`, the `_amountOut` return value is bubbled-up by `SmartVaultManagerV6::vaultAutoRedemption`:

```
function vaultAutoRedemption(
    address _smartVault,
    address _collateralAddr,
    bytes memory _swapPath,
    uint256 _collateralAmount
) external onlyAutoRedemption returns (uint256 _amountOut) {
    return IRedeemableLegacy(_smartVault).autoRedemption(swapRouter, _collateralAddr, _swapPath,
    ↪  _collateralAmount);
}
```

However, it is never actually removed and so can be removed from both function signatures.

```
function legacyAutoRedemption(
    address _smartVault,
    address _token,
    bytes memory _collateralToUSDCPath,
    uint256 _USDsTargetAmount,
    uint256 _estimatedCollateralValueUSD
) private {
    ...
    ISmartVaultManager(smartVaultManager).vaultAutoRedemption(_smartVault, _token,
    ↪  _collateralToUSDCPath, _amountIn);
}
```

The Standard DAO: Fixed by commit 2c58fa5.

Cyfrin: Verified. The return value is now used to emit an event.

7.5.8 Redemption of Hypervisor collateral can be suboptimal

Description: Consider a vault with `20% WBTC`, `40% WETH`, and `40% WBTC Hypervisor` collateral in USD terms that is to be the target of auto redemption. Under the current logic, the `WBTC Hypervisor` address will be received from

the API response as the collateral token to be redeemed. The `_token` variable is then reassigned to whichever underlying token is stored in the `hypervisorCollaterals` mapping (either `WBTC` or `WETH`, but not both):

```
address _hypervisor;
if (hypervisorCollaterals[_token] != address(0)) {
    _hypervisor = _token;
    _token = hypervisorCollaterals[_hypervisor];
}
IRedeemable(_smartVault).autoRedemption(
    _smartVault, quoter, _token, _collateralToUSDCPath, _USDsTargetAmount, _hypervisor
);
```

If the mapping is configured such that `WBTC` is returned, it will not be possible to execute the most optimal redemption. This is a limitation the current design which could be improved by passing both a collateral token and an optionally non-zero Hypervisor token address in the API response.

The Standard DAO: Fixed by commit fd1fe84.

Cyfrin: The response has been modified to include both collateral token and Hypervisor token addresses; however, they should be validated to correctly correspond to one another before being used.

The Standard DAO: Fixed by commit 7346460.

Cyfrin: Verified. The additional validation has been added to `AutoRedemption::validData`.