Smart Vault Audit Report

Prepared by Cyfrin
Version 2.0

Lead Auditors

Giovanni Di Siena

Immeas

September 13, 2024

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.

5 Audit Scope

Cyfrin conducted an audit of The Standard Smart Vault based on the code present in the repository commit hash c6837d4.

The following were included in the scope of the audit:

- SmartVaultV4.sol#L79-L103
- SmartVaultV4.sol#L279-L338
- SmartVaultYieldManager.sol

6 Executive Summary

Over the course of 8 days, the Cyfrin team conducted an audit on the Smart Vault smart contracts provided by The Standard DAO. In this period, a total of 22 issues were found.

This review of The Standard Smart Vault contracts yielded two critical vulnerabilities that arose due to a failure to consider the interactions of all possible collateral tokens within the system. Specifically, Hypervisor tokens credited to Smart Vaults following collateral deposits to Gamma Vaults were correctly accounted for in the calculation of the Smart Vault's collateral value; however, these tokens were not considered during liquidation or removal of assets from the Smart Vault. This oversight would have resulted in an accrual of bad debt within the protocol, likely threatening the stability of the `USDs` stablecoin, and so should be addressed as a priority.

Two high severity issues were also identified, relating to assumptions about the pricing of stablecoins and the potential for `USDs` to be partially self-back due to interactions with the `USDs` Hypervisor. 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, including: insufficient validation of Chainlink data feeds, hardcoded pool fees, and suboptimal slippage protection. These issues are less severe but are still strongly recommended to be addressed.

The Hardhat test suite covers the main functionalities and interactions between `SmartVaultYieldManager` and `SmartVaultV4`, including basic testing of both happy and unhappy paths; however, aside from being heavily mocked, there are areas in which this could be performed more thoroughly by testing unusual token combinations, setup conditions, and sequences of function calls. Therefore, as part of this review, a Foundry test suite was developed to facilitate more comprehensive testing of all the Smart Vault contracts using various test fixtures and techniques such fork, differential and stateful property-based testing. Another peripheral deliverable is the reimplementation of `SmartVaultYieldManager::_swapToRatio` to use an algebraic approach in calculating the optimal swap amount, which is more efficient and less error-prone than the original implementation (tested against the original implementation using the differential test suite and also compiling without the need for IR codegen).

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	Smart Vault
Repository	smart-vault
Commit	c6837d4a296f...
Audit Timeline	Aug 26th - Sep 4th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	2
High Risk	2
Medium Risk	3
Low Risk	8
Informational	5
Gas Optimizations	2
Total Issues	22

Summary of Findings

[C-1] USDs stability can be compromised as collateral deposited to Gamma vaults is not considered during liquidation	Resolved

[C-2] USDs stability can be compromised as collateral can be stolen by removing Hypervisor tokens directly from a vault without repaying USDs debt	Resolved
[H-1] `USDs` self-backing breaks assumptions around economic peg-maintenance incentives	Resolved
[H-2] USD stablecoins are incorrectly assumed to always be at peg	Resolved
[M-1] Yield deposits are susceptible to losses of up to 10%	Resolved
[M-2] Hardcoded pool fees can result in increased slippage and failed swaps	Resolved
[M-3] Insufficient validation of Chainlink data feeds	Resolved
[L-1] Allowance reset for incorrect token in `SmartVaultYieldManager::_sellUSDC`	Resolved
[L-2] Insufficient deadline protection when adding/removing collateral from yield positions	Resolved
[L-3] Removal of Hypervisor data locks deposited Smart Vault collateral	Acknowledged
[L-4] Dust amounts of swapped collateral tokens remain in `SmartVaultYieldManager`	Resolved
[L-5] `WETH` collateral cannot be swapped in `SmartVaultV4`	Resolved
[L-6] Liquidations could be blocked by reverting ERC-20 transfers	Resolved
[L-7] Potentially incorrect encoding of swap paths	Acknowledged
[L-8] Collateral tokens with more than 18 decimals are not supported	Resolved
[I-1] Unnecessary typecast of `msg.sender` to `address`	Resolved
[I-2] Comment incorrectly refers to € when it should be	Resolved
[I-3] Inconsistent use of equivalent function parameter and immutable variable in `SmartVaultYieldManager::_withdrawUSDsDeposit` is confusing	Resolved
[I-4] `USDC` cannot be added as an accepted collateral token	Acknowledged
[I-5] Native asset cannot be removed using `SmartVaultV4::removeAsset`	Resolved
[G-1] Unnecessary call to `SmartVaultV4::usdCollateral` when depositing/withdrawing collateral to/from yield positions	Resolved
[G-2] Cached `_token0` not used	Resolved

7 Findings

7.1 Critical Risk

7.1.1 USDs stability can be compromised as collateral deposited to Gamma vaults is not considered during liquidation

Description: Users of The Standard can take out USDs stablecoin loans against their collateral deposited into an instance of SmartVaultV4. If the collateral value of a Smart Vault falls below 110% of the USDs debt value, it can be liquidated in full. Users can also move collateral tokens into Gamma Vaults (aka Hypervisors) that hold LP positions in Uniswap V3 to earn an additional yield on their deposited collateral.

Collateral held as yield positions in Gamma Vaults are represented by Hypervisor tokens transferred to and held by the SmartVaultV4 contract; however, these tokens are not affected by liquidation:

```
function liquidate() external onlyVaultManager {
    if (!undercollateralised()) revert NotUndercollateralised();
    liquidated = true;
    minted = 0;
    liquidateNative();
    ITokenManager.Token[] memory tokens =
    ↪  ITokenManager(ISmartVaultManagerV3(manager).tokenManager()).getAcceptedTokens();
    for (uint256 i = 0; i < tokens.length; i++) {
        if (tokens[i].symbol != NATIVE) liquidateERC20(IERC20(tokens[i].addr));
    }
}
```

Currently, Hypervisor tokens present in the SmartVaultV4::hypervisors array are not included in the array returned by TokenManager::getAcceptedTokens as this would require them to have a Chainlink data feed. Therefore, any collateral deposited as a yield position within a Gamma Vault will remain unaffected.

A user could reasonably have a Smart Vault with 100% of their collateral deposited to Gamma, with 100% of the maximum USDs minted. At this point, any small market fluctuation would leave the Smart Vault undercollateralised and susceptible to liquidation. Given that the minted state variable is reset to zero upon successful liquidation, the user is again able to access this collateral via SmartVaultV4::removeCollateral due to the validation in SmartVaultV4::canRemoveCollateral:

```
function canRemoveCollateral(ITokenManager.Token memory _token, uint256 _amount) private view returns
↪  (bool) {
    if (minted == 0) return true;
    /* snip: collateral calculations */
}
```

Consequently, the user can withdraw the collateral without repaying the original USDs loan. Given the liquidated state variable would now be set to true, any attacker would need to create a new Smart Vault before the collateral could be used again.

Impact: An attacker could repeatedly borrow against collateral deposited in a yield position after being liquidated, resulting in bad debt for the protocol and likely compromising the stability of USDs if executed on a large scale.

Proof of Concept: The following test can be added to SmartVault.js:

```
it('cant liquidate yield positions', async () => {
  const ethCollateral = ethers.utils.parseEther('0.1')
  await user.sendTransaction({ to: Vault.address, value: ethCollateral });

  let { collateral, totalCollateralValue } = await Vault.status();
  let preYieldCollateral = totalCollateralValue;
  expect(getCollateralOf('ETH', collateral).amount).to.equal(ethCollateral);
  depositYield = Vault.connect(user).depositYield(ETH, HUNDRED_PC.div(10));
  await expect(depositYield).not.to.be.reverted;
  await expect(depositYield).to.emit(YieldManager, 'Deposit').withArgs(Vault.address, MockWeth.address,
  ↪   ethCollateral, HUNDRED_PC.div(10));

  ({ collateral, totalCollateralValue } = await Vault.status());
  expect(getCollateralOf('ETH', collateral).amount).to.equal(0);
  expect(totalCollateralValue).to.equal(preYieldCollateral);

  const mintedValue = ethers.utils.parseEther('100');
  await Vault.connect(user).mint(user.address, mintedValue);

  await expect(VaultManager.connect(protocol).liquidateVault(1)).to.be.revertedWith('vault-not-undercol
  ↪   lateralised')

  // drop price, now vault is liquidatable
  await CL_WBTC_USD.setPrice(1000);

  await expect(VaultManager.connect(protocol).liquidateVault(1)).not.to.be.reverted;
  ({ minted, maxMintable, totalCollateralValue, collateral, liquidated } = await Vault.status());

  // hypervisor tokens (yield position) not liquidated
  await expect(MockWETHWBTCHypervisor.balanceOf(Vault.address)).to.not.equal(0);

  // since minted is zero, the vault owner still has access to all collateral
  expect(minted).to.equal(0);
  expect(maxMintable).to.not.equal(0);
  expect(totalCollateralValue).to.not.equal(0);
  collateral.forEach(asset => expect(asset.amount).to.equal(0));
  expect(liquidated).to.equal(true);

  // price returns
  await CL_WBTC_USD.setPrice(DEFAULT_ETH_USD_PRICE.mul(20));

  // user exits yield position
  await Vault.connect(user).withdrawYield(MockWETHWBTCHypervisor.address, ETH);
  await Vault.connect(user).withdrawYield(MockUSDsHypervisor.address, ETH);

  // and withdraws assets
  const userBefore = await ethers.provider.getBalance(user.address);
  await Vault.connect(user).removeCollateralNative(await ethers.provider.getBalance(Vault.address),
  ↪   user.address);
  const userAfter = await ethers.provider.getBalance(user.address);

  // user should have all collateral back minus protocol fee from yield withdrawal
  expect(userAfter.sub(userBefore)).to.be.closeTo(ethCollateral, ethers.utils.parseEther('0.01'));

  // and user also has the minted USDs
  const usds = await USDs.balanceOf(user.address);
  expect(usds).to.equal(mintedValue);
});
```

Recommended Mitigation: Ensure that collateral held in yield positions is also subject to liquidation.

The Standard DAO: Fixed by commit c6af5d2.

Cyfrin: Verified, `SmartVault4::liquidate` now also loops over the `SmartVaultV4::hypervisors` array. However, native liquidation should be performed last to mitigate re-entrancy risk. Similarly, revocation of roles in `SmartVaultManagerV6::liquidateVault` should occur before invoking `SmartVaultV4::liquidate`.

The Standard DAO: Fixed by commit 23c573c.

Cyfrin: Verified, the order of calls had been modified.

7.1.2 USDs stability can be compromised as collateral can be stolen by removing Hypervisor tokens directly from a vault without repaying USDs debt

Description: When the owner of a Smart Vault calls `SmartVaultV4::depositYield`, `SmartVaultYieldManager::deposit` is invoked to deposit the specified collateral tokens to a given Gamma Vault (aka Hypervisor) via the `IUniProxy` contract. Gamma Hypervisors work such that they hold a position in a Uniswap V3 pool that is made fungible to depositors who own shares in this position represented as an ERC-20 token. On completion of a deposit, the Hypervisor tokens are transferred back to the calling `SmartVaultV4` contract where they remain as backing for any minted `USDs` debt.

However, due to insufficient input validation, these Hypervisor collateral tokens can be removed from the Smart Vault by calling `SmartVaultV4::removeAsset`:

```
function removeAsset(address _tokenAddr, uint256 _amount, address _to) external onlyOwner {
    ITokenManager.Token memory token = getTokenManager().getTokenIfExists(_tokenAddr);
    if (token.addr == _tokenAddr && !canRemoveCollateral(token, _amount)) revert Undercollateralised();
    IERC20(_tokenAddr).safeTransfer(_to, _amount);
    emit AssetRemoved(_tokenAddr, _amount, _to);
}
```

Hypervisor tokens are present only in the `SmartVaultV4::hypervisors` array and are not handled by the `TokenManager` contract, so `token.addr` will equal `address(0)` and the collateralisation check will be bypassed. Thus, these tokens can be extracted from the contract, leaving the Smart Vault in an undercollateralised state and the protocol with bad debt.

Impact: An attacker can borrow the maximum mintable amount of `USDs` against their deposited yield collateral, then leave their Smart Vault undercollateralised by simply removing the collateral Hypervisor tokens. The attacker receives both the `USDs` and its backing collateral while the protocol is left with bad debt, likely compromising the stability of `USDs` if executed on a large scale or atomically with funds obtained from a flash loan.

Proof of Concept: The following test can be added to `SmartVault.js`:

```
it('can steal collateral hypervisor tokens', async () => {
  const ethCollateral = ethers.utils.parseEther('0.1')
  await user.sendTransaction({ to: Vault.address, value: ethCollateral });

  let { collateral, totalCollateralValue } = await Vault.status();
  let preYieldCollateral = totalCollateralValue;
  expect(getCollateralOf('ETH', collateral).amount).to.equal(ethCollateral);

  depositYield = Vault.connect(user).depositYield(ETH, HUNDRED_PC);
  await expect(depositYield).not.to.be.reverted;
  await expect(depositYield).to.emit(YieldManager, 'Deposit').withArgs(Vault.address, MockWeth.address,
  ↪  ethCollateral, HUNDRED_PC);

  ({ collateral, totalCollateralValue } = await Vault.status());
  expect(getCollateralOf('ETH', collateral).amount).to.equal(0);
  expect(totalCollateralValue).to.equal(preYieldCollateral);

  const mintedValue = ethers.utils.parseEther('100');
  await Vault.connect(user).mint(user.address, mintedValue);

  // Vault is fully collateralised after minting USDs
  expect(await Vault.undercollateralised()).to.be.equal(false);

  const hypervisorBalanceVault = await MockUSDsHypervisor.balanceOf(Vault.address);
  await Vault.connect(user).removeAsset(MockUSDsHypervisor.address, hypervisorBalanceVault ,
  ↪  user.address);
```

```
    // Vault has no collateral left and as such is undercollateralised
    expect(await MockUSDsHypervisor.balanceOf(Vault.address)).to.be.equal(0);
    expect(await Vault.undercollateralised()).to.be.equal(true);

    // User has both the minted USDs and Hypervisor collateral tokens
    expect(await MockUSDsHypervisor.balanceOf(user.address)).to.be.equal(hypervisorBalanceVault);
    expect(await USDs.balanceOf(user.address)).to.be.equal(mintedValue);
});
```

Recommended Mitigation: Validate that the asset removed is not a Hypervisor token present in the `hypervisors` array.

If considering adding the Hypervisor tokens as collateral in the `TokenManager`, ensure that they are excluded from this loop within `SmartVaultV4::usdCollateral` and the pricing calculation in `SmartVaultV4::getAssets` is also updated accordingly.

The Standard DAO: Fixed by commit `5862d8e`.

Cyfrin: Verified, Hypervisor tokens can no longer be removed without causing `SmartVault::removeAsset` to revert due to being undercollateralised. However, the use of the `remainCollateralised()` modifier in `SmartVaultV4::removeCollateralNative` has introduced a re-entrancy vulnerability whereby the protocol burn fee can be bypassed by the Smart Vault owner: deposit native collateral → mint USDs → remove native collateral → re-enter & self-liquidate. Here, the original validation should be used as this does not affect Hypervisor tokens.

The Standard DAO: Fixed by commit `d761d48`.

Cyfrin: Verified, `SmartVaultV4::removeCollateralNative` can no longer be used to re-enter in an undercollateralised state.

7.2 High Risk

7.2.1 USDs self-backing breaks assumptions around economic peg-maintenance incentives

Description: When `SmartVaultYieldManager::deposit` is called via `SmartVaultV4::depositYield`, at least 10% of the deposited collateral must be directed toward the `USDs` Hypervisor (which in turn holds an LP position in a protocol-managed `USDs/USDC` Ramses pool):

```
function deposit(address _collateralToken, uint256 _usdPercentage) external returns (address
↪  _hypervisor0, address _hypervisor1) {
    if (_usdPercentage < MIN_USDS_PERCENTAGE) revert StablePoolPercentageError();
    uint256 _balance = IERC20(_collateralToken).balanceOf(address(msg.sender));
    IERC20(_collateralToken).safeTransferFrom(msg.sender, address(this), _balance);
    HypervisorData memory _hypervisorData = hypervisorData[_collateralToken];
    if (_hypervisorData.hypervisor == address(0)) revert HypervisorDataError();
    _usdDeposit(_collateralToken, _usdPercentage, _hypervisorData.pathToUSDC);
    /* snip: other hypervisor deposit */
}
```

When the value of the Smart Vault's collateral is determined by `SmartVaultV4::yieldVaultCollateral`, ignoring the issue of hardcoding stablecoins to $1, the value of the tokens underlying each Hypervisor is used:

```
if (_token0 == address(USDs) || _token1 == address(USDs)) {
    // both USDs and its vault pair are € stablecoins, but can be equivalent to €1 in collateral
    _usds += _underlying0 * 10 ** (18 - ERC20(_token0).decimals());
    _usds += _underlying1 * 10 ** (18 - ERC20(_token1).decimals());
```

The issue for `USDs` Hypervisor deposits is that this underlying balance of `USDs` counts toward the total collateral value of the Smart Vault, and there is no restriction on the maximum amount of collateral that can be directed toward this Hypervisor. Hence, users can use `USDs` to collateralize their `USDs` loans with up to as much as 100% of the total collateral deposited to the `USDs/USDC` pool (50% in `USDs` if both stablecoin tokens are assumed to be at peg).

Impact: Once the peg is lost for endogenously collateralized stablecoins, such as those backed by themselves, it becomes increasingly difficult to return to recover as the value of both the stablecoin and its collateral decrease in tandem. This self-backing also breaks the assumptions surrounding the economic incentives of the protocol intended to contribute to peg-maintenance.

Recommended Mitigation: Consider disallowing the use of `USDs` Hypervisor tokens as backing collateral and implementing some other mechanism to ensure sufficient liquidity in the pool. Alternatively, the percentage of collateral allowed to be directed toward the `USDs` Hypervisor could be limited, but this would not completely mitigate the risk.

The Standard DAO: Fixed by commit `cc86606`.

Cyfrin: Verified, `USDs` no longer contributes to Smart Vault yield collateral.

7.2.2 USD stablecoins are incorrectly assumed to always be at peg

Description: `SmartVaultV4::yieldVaultCollateral` returns the value of collateral held in yield positions for a given Smart Vault. For all Gamma Vaults other than the `USDs` Hypervisor, the dollar value of the underlying amounts of collateral tokens fetched for each Gamma Vault in which collateral is deposited is calculated using prices reported by Chainlink.

```
function yieldVaultCollateral(ITokenManager.Token[] memory _acceptedTokens) private view returns
↪  (uint256 _usds) {
    for (uint256 i = 0; i < hypervisors.length; i++) {
        IHypervisor _Hypervisor = IHypervisor(hypervisors[i]);
        uint256 _balance = _Hypervisor.balanceOf(address(this));
        if (_balance > 0) {
            uint256 _totalSupply = _Hypervisor.totalSupply();
            (uint256 _underlyingTotal0, uint256 _underlyingTotal1) = _Hypervisor.getTotalAmounts();
            address _token0 = _Hypervisor.token0();
            address _token1 = _Hypervisor.token1();
            uint256 _underlying0 = _balance * _underlyingTotal0 / _totalSupply;
            uint256 _underlying1 = _balance * _underlyingTotal1 / _totalSupply;
            if (_token0 == address(USDs) || _token1 == address(USDs)) {
                // both USDs and its vault pair are  stablecoins, but can be equivalent to 1 in
                ↪  collateral
                _usds += _underlying0 * 10 ** (18 - ERC20(_token0).decimals());
                _usds += _underlying1 * 10 ** (18 - ERC20(_token1).decimals());
            } else {
                for (uint256 j = 0; j < _acceptedTokens.length; j++) {
                    ITokenManager.Token memory _token = _acceptedTokens[j];
                    if (_token.addr == _token0) _usds += calculator.tokenToUSD(_token, _underlying0);
                    if (_token.addr == _token1) _usds += calculator.tokenToUSD(_token, _underlying1);
                }
            }
        }
    }
}
```

On every collateral deposit to a Gamma Vault, a minimum amount is required to be directed to the `USDs/USDC` pair. Thus, there will always be a non-zero balance of `USDs` Hypervisor tokens if the balance of any other Hypervisor tokens is also non-zero for a given Gamma Vault.

As such, and because there is no Chainlink price feed for `USDs`, this Hypervisor is handled separately; however, this logic incorrectly assumes that the prices of `USDC` and `USDs` will always be equivalent at $1. This is not always true – there have been instances where USDC has experienced de-pegging events, significant in both magnitude and duration. Similar concerns are present for `USDs`, ignoring issues related to self-backing raised in a separate finding.

In the event of either stablecoin de-pegging, Smart Vault owners can borrow above their true collateral value. While strictly hypothetical, it may also be possible to bring about this scenario by direct manipulation of the`USDs/USDC` pool through the following actions:

- Flash loan `WETH` collateral & deposit to Smart Vault.
- Deposit 100% of collateral to the `USDs` Hypervisor.
- Mint a large amount of `USDs`.
- Sell into the `USDs/USDC` pool.
- Assuming `USDs` de-pegs such that more of the position underlying the `USDs` Hypervisor is in `USDs`, this "borrowing above collateral value" effect would be amplified and the Smart Vault yield collateral would increase.
- Borrow more `USDs` using the inflated yield vault collateral calculation, pay back the loan, and repeat.

Impact: If either `USDC` or `USDs` fall below their $1 peg, users can mint more `USDs` collateralized by a `USDs/USDC` Hypervisor deposit than should be possible. It may also be possible to directly influence the stability of `USDs` depending on conditions in the `USDs/USDC` pool when one or both stablecoins de-peg.

Additionally, ignoring the separate finding related to Hypervisor tokens not being affected by liquidations, a collateral deposit fully directed to the `USDs` Hypervisor can never be liquidated even if one or both stablecoins de-peg, causing the Smart Vault to become undercollateralised in reality.

Recommended Mitigation: Chainlink data feeds should be used to determine the price of `USDC`.

As a general recommendation, a manipulation-resistant alternative should be leveraged for pricing `USDs`; however, this finding underscores the issue with USDs self-backing as the intended economic incentives of the protocol will not apply in this scenario.

The Standard DAO: Fixed by commit `cc86606`.

Cyfrin: Verified, `USDC` price is now obtained from Chainlink and `USDs` is no longer included in yield vault collateral. Consider querying the Chainlink data feed decimals instead of hardcoding to `1e8`.

The Standard DAO: Fixed by commit `5febbc4`.

Cyfrin: Verified, decimals are now queried dynamically.

7.3 Medium Risk

7.3.1 Yield deposits are susceptible to losses of up to 10%

Description: To deal with the slippage incurred through multiple [1, 2, 3, 4, 5, 6, 7] intermediate DEX swaps and Gamma Vault interactions when `SmartVaultV4::depositYield` and `SmartVaultV4::withdrawYield` are called, there is a requirement that the total collateral value should not have decreased more than 90%:

```
    function significantCollateralDrop(uint256 _preCollateralValue, uint256 _postCollateralValue)
    ↪  private pure returns (bool) {
    return _postCollateralValue < 9 * _preCollateralValue / 10;
}
```

While this design will successfully protect users against complete and immediate loss, 10% is nevertheless a significant amount to lose on each deposit/withdrawal action.

Currently, due to the existence of a centralized sequencer, MEV on Arbitrum does not exist in the typical sense; however, it is still possible to execute latency-driven strategies for predictable events such as liquidations. As such, it may still be possible for MEV bots to cause collateral yield deposits/withdrawals to return 90% of the original collateral value, putting the Smart Vault unnecessarily close to liquidation.

Impact: Users could lose a significant portion of collateral when depositing into and withdrawing from Gamma Vaults.

Recommended Mitigation: While the existing validation can remain, consider allowing the user to pass a more restrictive collateral drop percentage and more fine-grained slippage parameters for the interactions linked above.

The Standard DAO: Fixed by commit `cc86606`.

Cyfrin: Verified, `SmartVaultV4::depositYield` and `SmartVaultV4::withdrawYield` now accept a user-supplied minimum collateral percentage parameter. Note that due to the re-ordering of validation in `SmartVaultV4::mint`, the `remainCollateralised()` modifier can be used for this function.

The Standard DAO: Fixed by commit e89daee.

Cyfrin: Verified, the modifier is now used for `mint()`.

7.3.2 Hardcoded pool fees can result in increased slippage and failed swaps

Description: The issue raised in the previous CodeHawks contest as report item M-03 remains present in `SmartVaultV4::swap` where the pool fee is hardcoded to `3000`:

```
ISwapRouter.ExactInputSingleParams memory params = ISwapRouter.ExactInputSingleParams({
        tokenIn: inToken,
        tokenOut: getTokenisedAddr(_outToken),
        fee: 3000, // @audit hardcoded pool fee
        recipient: address(this),
        deadline: block.timestamp + 60,
        amountIn: _amount - swapFee,
        amountOutMinimum: minimumAmountOut,
        sqrtPriceLimitX96: 0
    });
```

The same issue is present within `SmartVaultYieldManager::_usdDeposit` and `SmartVaultYieldManager::_withdrawDeposit`, where collateral tokens are swapped to/from `USDC` and `USDs` with a hardcoded pool fee of `500`:

```
function _usdDeposit(address _collateralToken, uint256 _usdPercentage, bytes memory _pathToUSDC)
↪  private {
    _swapToUSDC(_collateralToken, _usdPercentage, _pathToUSDC);
    _swapToRatio(USDC, usdsHypervisor, ramsesRouter, 500);
    _deposit(usdsHypervisor);
}
...
function _withdrawUSDsDeposit(address _hypervisor, address _token) private {
    IHypervisor(_hypervisor).withdraw(_thisBalanceOf(_hypervisor), address(this), address(this),
    ↪   [uint256(0),uint256(0),uint256(0),uint256(0)]);
    _swapToSingleAsset(usdsHypervisor, USDC, ramsesRouter, 500);
    _sellUSDC(_token);
}
```

Impact: As mentioned in M-03 of the CodeHawks contest, with the possible exception of the `USDs/USDC` pool created and maintained by the protocol, the pool with the highest liquidity will not necessarily always be equal to the hardcoded values, so trading in a pool with low liquidity will result in increased slippage or failed swaps. If the loss exceeds 10% of the collateral value, this results in a DoS of yield deposits/withdrawals due to validation in `SmartVaultV4::significantCollateralDrop`. For calls to `SmartVaultV4::swap`, there is no such validation to prevent the Smart Vault from being put unnecessarily close to liquidation – the minimum amount output from the swap is that required to remain collateralized within 1% of liquidation.

Recommended Mitigation: The same recommendation as in M-03 applies here – consider allowing the user to pass the pool fee as a parameter to the call(s).

The Standard DAO: Collateral swap pool fees fixed by commit `f9f7093`. Hypervisor swap pool fees acknowledged – not fixed as these swap routes will be managed by admins in `hypervisorData`.

Cyfrin: Verified, `SmartVaultV4::swap` now accepts a user-supplied pool fee parameter.

7.3.3 Insufficient validation of Chainlink data feeds

Description: `PriceCalculator` is a contract responsible for providing the Chainlink oracle prices for assets used by The Standard. Here, the price for an asset is queried and then normalized to 18 decimals before being returned to the caller:

```
function tokenToUSD(ITokenManager.Token memory _token, uint256 _tokenValue) external view returns
↪   (uint256) {
    Chainlink.AggregatorV3Interface tokenUsdClFeed = Chainlink.AggregatorV3Interface(_token.clAddr);
    uint256 scaledCollateral = _tokenValue * 10 ** getTokenScaleDiff(_token.symbol, _token.addr);
    (,int256 _tokenUsdPrice,,,) = tokenUsdClFeed.latestRoundData();
    return scaledCollateral * uint256(_tokenUsdPrice) / 10 ** _token.clDec;
}

function USDToToken(ITokenManager.Token memory _token, uint256 _usdValue) external view returns
↪   (uint256) {
    Chainlink.AggregatorV3Interface tokenUsdClFeed = Chainlink.AggregatorV3Interface(_token.clAddr);
    (, int256 tokenUsdPrice,,,) = tokenUsdClFeed.latestRoundData();
    return _usdValue * 10 ** _token.clDec / uint256(tokenUsdPrice) / 10 **
    ↪   getTokenScaleDiff(_token.symbol, _token.addr);
}
```

However, these calls to `AggregatorV3Interface::latestRoundData` lack the necessary validation for Chainlink data feeds to ensure that the protocol does not ingest stale or incorrect pricing data that could indicate a faulty feed.

Impact: Stale prices can result in unnecessary liquidations or the creation of insufficiently collateralised positions.

Recommended Mitigation: Implement the following validation:

```
-   (,int256 _tokenUsdPrice,,,) = tokenUsdClFeed.latestRoundData();
+   (uint80 _roundId, int256 _tokenUsdPrice, , uint256 _updatedAt, ) = tokenUsdClFeed.latestRoundData();
+   if(_roundId == 0) revert InvalidRoundId();
+   if(_tokenUsdPrice == 0) revert InvalidPrice();
+   if(_updatedAt == 0 || _updatedAt > block.timestamp) revert InvalidUpdate();
+   if(block.timestamp - _updatedAt > TIMEOUT) revert StalePrice();
```

Given the intention to deploy these contracts to Arbitrum, it is also recommended to check the sequencer uptime. The documentation for implementing this is here with a code example.

The Standard DAO: Fixed by commit `8e78f7c`.

Cyfrin: Verified, additional validation of Chainlink price feed data has been added; however, timeouts should be specified on a per-feed basis, and 24 hours is likely too long for most feeds. The sequencer uptime feed has also not been implemented, but this is an important addition. Note that the `hardhat/console.sol` import should be removed from `PriceCalculator.sol`.

The Standard DAO: Fixed by commit `7dfbff1`.

Cyfrin: Verified, additional timeout logic and the sequencer uptime check have been added.

7.4 Low Risk

7.4.1 Allowance reset for incorrect token in `SmartVaultYieldManager::_sellUSDC`

Description: When swapping `USDC` in `SmartVaultYieldManager::_sellUSDC`, there is an allowance given to the router:

```
IERC20(USDC).safeApprove(uniswapRouter, _balance);
ISwapRouter(uniswapRouter).exactInput(ISwapRouter.ExactInputParams({
    /* snip: swap */
}));
IERC20(USDs).safeApprove(uniswapRouter, 0);
```

Consistent with all other swaps performed in this contract, the allowance is reset after interaction with the router; however, in this instance, the allowance is incorrectly reset to `0` for `USDs` instead of `USDC`.

Impact: There can be small `USDC` dust allowances left on the router.

Recommended Mitigation: Replace `USDs` with `USDC`:

```
  IERC20(USDC).safeApprove(uniswapRouter, _balance);
  ISwapRouter(uniswapRouter).exactInput(ISwapRouter.ExactInputParams({
      /* snip: swap */
  }));
- IERC20(USDs).safeApprove(uniswapRouter, 0);
+ IERC20(USDC).safeApprove(uniswapRouter, 0);
```

The Standard DAO: Fixed by commit `217de3a`.

Cyfrin: Verified, approval is now reset for `USDC`.

7.4.2 Insufficient deadline protection when adding/removing collateral from yield positions

Description: When the owner of a Smart Vault transfers its collateral assets to/from one of the supported Gamma Vaults, several swaps are executed with a deadline of `block.timestamp + 60`. For example, in `SmartVaultYieldManager::_sellUSDC`:

```
ISwapRouter(uniswapRouter).exactInput(ISwapRouter.ExactInputParams({
    path: _pathFromUSDC,
    recipient: address(this),
    deadline: block.timestamp + 60,
    amountIn: _balance,
    amountOutMinimum: 0
}));
```

This deadline will always be valid whenever the transaction is included in a block, with the addition of 60 seconds from the current timestamp doing nothing, as the timestamp of execution will always be `block.timestamp`.

Impact: The lack of a proper deadline can result in swaps being executed in market conditions that differ significantly from those intended, possibly resulting in less favorable outcomes. This is somewhat mitigated by the `significantCollateralDrop()` protection in `SmartVaultV4`; however, this relies on Chainlink oracle values for calculation of the Smart Vault collateral that might have also changed since the transaction was submitted.

Recommended Mitigation: Consider allowing the user to specify a deadline for the swaps executed when adding/removing collateral from yield positions. Note that the deadline does not need to be passed directly to all swap invocations but can be checked once directly in the function bodies of `SmartVaultV4::depositYield` and `SmartVaultV4::withdrawYield`.

The Standard DAO: Fixed by commit `71bad0a`.

Cyfrin: Verified, `SmartVaultV4::depositYield`, `SmartVaultV4:withdrawtYield`, and `SmartVaultV4::swap` now accept a user-supplied deadline parameter.

7.4.3 Removal of Hypervisor data locks deposited Smart Vault collateral

Description: A Gamma Vault (aka Hypervisor) is an external contract that maintains and offers fungible shares in Uniswap V3 liquidity positions. The Standard leverages multiple Hypervisors to enable the collateral backing `USDs` to earn yield, configured by admin calls to `SmartVaultYieldManager::addHypervisorData`. When Smart Vault collateral is deposited into one of these Hypervisors, it is minted Hypervisor ERC-20 tokens to represent a share of the underlying position and internally calls `SmartVaultV4::addUniqueHypervisor` to maintain a list of Hypervisors in which it has collateral deposited.

If an admin call is made to `SmartVaultYieldManager::removeHypervisorData` to remove a Hypervisor in which Smart Vaults still have open positions, the underlying collateral will be locked. This is due to the following validation in `SmartVaultYieldManager::_withdrawOtherDeposit` that requires the Hypervisor data to be valid and configured:

```
function _withdrawOtherDeposit(address _hypervisor, address _token) private {
    HypervisorData memory _hypervisorData = hypervisorData[_token];
    if (_hypervisorData.hypervisor != _hypervisor) revert IncompatibleHypervisor();
    /* snip: withdraw and swap */
}
```

However, this collateral locked in the removed Hypervisor will still contribute to the collateral calculation of the Smart Vault due to looping over its independently maintained `SmartVaultV4::hypervisors` array (from which Hypervisors are only removed when collateral is withdrawn).

Impact: Hypervisor tokens and the corresponding Smart Vault collateral can be locked indefinitely unless the protocol admin re-adds the Hypervisor data, ignoring a separate finding detailing the malicious removal of Hypervisor tokens from Smart Vaults.

Proof of Concept: The following test can be added to `SmartVault.js`:

```
it('locks collateral when hypervisor is removed', async () => {
  const ethCollateral = ethers.utils.parseEther('0.1')
  await user.sendTransaction({ to: Vault.address, value: ethCollateral });

  let { collateral, totalCollateralValue } = await Vault.status();
  let preYieldCollateral = totalCollateralValue;
  expect(getCollateralOf('ETH', collateral).amount).to.equal(ethCollateral);

  depositYield = Vault.connect(user).depositYield(ETH, HUNDRED_PC.div(10));
  await expect(depositYield).not.to.be.reverted;
  await expect(depositYield).to.emit(YieldManager, 'Deposit').withArgs(Vault.address, MockWeth.address,
  ↪  ethCollateral, HUNDRED_PC.div(10));

  ({ collateral, totalCollateralValue } = await Vault.status());
  expect(getCollateralOf('ETH', collateral).amount).to.equal(0);
  expect(totalCollateralValue).to.equal(preYieldCollateral);

  await YieldManager.connect(admin).removeHypervisorData(MockWeth.address);

  // collateral is still counted
  ({ collateral, totalCollateralValue } = await Vault.status());
  expect(getCollateralOf('ETH', collateral).amount).to.equal(0);
  expect(totalCollateralValue).to.equal(preYieldCollateral);

  // user cannot remove collateral
  await expect(Vault.connect(user).withdrawYield(MockWETHWBTCHypervisor.address, ETH))
    .to.be.revertedWithCustomError(YieldManager, 'IncompatibleHypervisor');
});
```

Recommended Mitigation: If it is necessary to have the ability to remove Hypervisors, consider also allowing Smart Vault owners to remove Hypervisor tokens from their Vaults if they have been delisted from `SmartVaultYieldManager`, with a check that they are still sufficiently collateralized.

The Standard DAO: Acknowleged, not fixed as we believe a user can remove with `removeAsset()`. As long as the vault remains collateralised, there shouldn't be a problem. We are also not intending to remove Hypervisors if we can avoid it.

Cyfrin: Acknowledged, while removed Hypervisor tokens will continue to contribute to the collateralization value of a given Smart Vault, they can be removed by calling `SmartVaultV4::removeAsset` so long as the Vault remains sufficiently collateralized.

7.4.4 Dust amounts of swapped collateral tokens remain in `SmartVaultYieldManager`

Description: Due to rounding, swaps made via Uniswap-style routers with exact input parameters can result in residual dust amounts left in the calling contract. This is not an issue for deposits to Gamma Vaults, as all swapped tokens are sent to the corresponding Hypervisor contract; however, the use of `SmartVaultYieldManager::_swapToSingleAsset` called from `SmartVaultYieldManager::_withdrawUSDsDeposit` and `SmartVaultYieldManager::_withdrawOtherDeposit` during withdrawals can leave dust amounts of the input token.

Impact: Dust amounts of collateral tokens can accumulate in `SmartVaultYieldManager` and will be utilized by the next caller for a given token.

Recommended Mitigation: Consider checking for non-zero residual amounts of the input token(s) to swaps made during the withdrawal of yield positions and, if present, return them to the Smart Vault.

The Standard DAO: Fixed by commit `a62973e`.

Cyfrin: Verified, dust amounts of the unwanted token are now transferred back to the sender.

7.4.5 `WETH` collateral cannot be swapped in `SmartVaultV4`

Description: `SmartVaultV4::swap` allows Smart Vault collateral, specified by its `bytes32` symbol, to be swapped for other supported collateral tokens. The corresponding token address for a given symbol is returned by `SmartVaultV4::getTokenisedAddr` based on the output of `SmartVaultV4::getToken`:

```
function getToken(bytes32 _symbol) private view returns (ITokenManager.Token memory _token) {
    ITokenManager.Token[] memory tokens =
    ↪   ITokenManager(ISmartVaultManagerV3(manager).tokenManager()).getAcceptedTokens();
    for (uint256 i = 0; i < tokens.length; i++) {
        if (tokens[i].symbol == _symbol) _token = tokens[i];
    }
    if (_token.symbol == bytes32(0)) revert InvalidToken();
}

function getTokenisedAddr(bytes32 _symbol) private view returns (address) {
    ITokenManager.Token memory _token = getToken(_symbol);
    return _token.addr == address(0) ? ISmartVaultManagerV3(manager).weth() : _token.addr;
}
```

Native `ETH` is present in the list of accepted tokens; however, it returns `address(0)`. Hence, the symbols for both `ETH` and `WETH` correspond to the `WETH` address which is used as the `tokenIn` parameter for the Uniswap V3 Router swap instruction. This is the correct method for swapping native `ETH` via the Uniswap V3 Router which will first attempt to utilize any native balance to cover `amountIn`.

After the swap parameters are populated, execution of the actual swap occurs based on `SmartVaultV4::executeNativeSwapAndFee` or `SmartVaultV4::executeERC20SwapAndFee`, depending on the `inToken` address:

```
inToken == ISmartVaultManagerV3(manager).weth() ?
    executeNativeSwapAndFee(params, swapFee) :
    executeERC20SwapAndFee(params, swapFee);
```

Here, the first conditional branch will be executed if the caller intends to swap `WETH` or native `ETH`; however, this logic assumes that the caller exclusively wants to swap native `ETH`, so it will fail for `WETH` unless the Smart Vault has a sufficient balance of `ETH` to perform a native `ETH` swap.

Impact: It is impossible for `WETH` collateral to be swapped directly within a Smart Vault.

Proof of Concept: The following test can be added to `SmartVault.js`:

```
it('cant swap WETH', async () => {
  const ethCollateral = ethers.utils.parseEther('0.1')
  await MockWeth.connect(user).deposit({value: ethCollateral});
  await MockWeth.connect(user).transfer(Vault.address, ethCollateral);

  let { collateral } = await Vault.status();
  expect(getCollateralOf('WETH', collateral).amount).to.equal(ethCollateral);

  await expect(
    Vault.connect(user).swap(
      ethers.utils.formatBytes32String('WETH'),
      ethers.utils.formatBytes32String('WBTC'),
      ethers.utils.parseEther('0.05'),
      0)
    ).to.be.revertedWithCustomError(Vault, 'TransferError');
});
```

Recommended Mitigation: Consider handling `WETH` with `SmartVaultV4::executeERC20SwapAndFee` by modifying the conditional logic in `SmartRouterV4::swap`:

```
-   inToken == ISmartVaultManagerV3(manager).weth() ?
+   _inToken == NATIVE ?
        executeNativeSwapAndFee(params, swapFee) :
        executeERC20SwapAndFee(params, swapFee);
```

The Standard DAO: Fixed by commit fb965bd.

Cyfrin: Verified, `WETH` collateral can now be swapped; however, if the output token is specified as `NATIVE` then any existing `WETH` collateral in the Smart Vault will also be withdrawn. Also, `SmartVaultV4::executeNativeSwapAndFee` is now no longer used and can be removed.

The Standard DAO: Fixed by commit 589d645.

Cyfrin: Verified, now only the `WETH` output from the swap is withdrawn to native.

7.4.6 Liquidations could be blocked by reverting ERC-20 transfers

Description: When liquidations are performed via `SmartVaultV4::liquidate`, ERC-20 collateral tokens are handled within a loop:

```
function liquidate() external onlyVaultManager {
    /* snip: validation, state updates & native liquidation
    ITokenManager.Token[] memory tokens =
    ↪   ITokenManager(ISmartVaultManagerV3(manager).tokenManager()).getAcceptedTokens();
    for (uint256 i = 0; i < tokens.length; i++) {
        if (tokens[i].symbol != NATIVE) liquidateERC20(IERC20(tokens[i].addr));
    }
}
```

If the contract balance of a given ERC-20 is non-zero, it will proceed to perform a transfer to the protocol address, as show below:

```
function liquidateERC20(IERC20 _token) private {
    if (_token.balanceOf(address(this)) != 0)
    ↪  _token.safeTransfer(ISmartVaultManagerV3(manager).protocol(), _token.balanceOf(address(this)));
}
```

However, if any of these transfers revert, the whole call will revert and liquidation will be blocked. Analysis of the collateral tokens currently intended to be supported failed to identify any immediate risks, although it is prescient to note the following:

- `GMX` includes rewards distribution logic on transfers (that, however unlikely, could potentially revert).
- `WETH` and `ARB` are Transparent Upgradeable proxies.
- `WBTC`, `LINK`, `PAXG`, and `SUSHI` are Beacon proxies.
- `RDNT` is a LayerZero bridge token.

Impact: Liquidations for a given Smart Vault will be blocked if `GMX` collateral transfers revert. If any other collateral tokens are upgraded to introduce novel transfer logic, they could also make Smart Vaults susceptible to this issue. If an attacker can force a single collateral token transfer to revert, they can avoid being liquidated.

Recommended Mitigation: Consider separate handling of each ERC-20 transfer with `try/catch` to avoid blocked liquidations.

The Standard DAO: Fixed by commit efda8d2.

Cyfrin: Verified, liquidation will no longer revert if a single transfer fails. Direct use of `ERC20::transfer` instead of `SafeERC20::safeTransfer` appears to be okay because:

- The Smart Vault will always be calling a contract with code when looping through the accepted tokens
- The current list of accepted collateral tokens all return `true` or revert on failed transfer.

7.4.7 Potentially incorrect encoding of swap paths

Description: During fork testing, it became apparent that swap paths should use packed encoding; however, the existing mocked test suite does the following:

```
// data about how yield manager converts collateral to USDC, vault addresses etc
await YieldManager.addHypervisorData(
  MockWeth.address, MockWETHWBTCHypervisor.address, 500,
  new ethers.utils.AbiCoder().encode(['address', 'uint24', 'address'], [MockWeth.address, 3000,
  ↪  USDC.address]),
  new ethers.utils.AbiCoder().encode(['address', 'uint24', 'address'], [USDC.address, 3000,
  ↪  MockWeth.address])
)
```

Referring to the ethers documentation, this shows that `AbiCoder::encode` is the incorrect method for packed encoding. If extended to the real configuration of Hypervisor data for deployed contracts, this would result in all yield deposit functionality reverting due to failed swaps.

Impact: Yield deposit functionality would not work due to incorrect configuration of Hypervisor data.

Recommended Mitigation: Use tightly packed encoding for swap paths.

The Standard DAO: Acknowledged. We are aware that this kind of encoding would not work in production with real routers, but could not figure out how to decode the correct path types in the mock swap router. Will amend the tests & mock swap router if you are aware of a solution.

Cyfrin: Acknowledged. The solution would be to use the Uniswap V3 Path and BytesLib libraries; however, this additional complexity may not be desired for the mock tests.

7.4.8 Collateral tokens with more than 18 decimals are not supported

Description: Due to the existing decimals scaling logic within `PriceCalculator::getTokenScaleDiff`, any collateral tokens with more than 18 decimals will not be supported and will result in DoS of Smart Vault functionality:

```
function getTokenScaleDiff(bytes32 _symbol, address _tokenAddress) private view returns (uint256
↪   scaleDiff) {
    return _symbol == NATIVE ? 0 : 18 - ERC20(_tokenAddress).decimals();
}
```

Similar scaling is present in `SmartVaultV4::yieldVaultCollateral`; however, this would require another `USDs` Hypervisor with a problematic underlying token to be added, which is unlikely.

Impact: Smart Vault collateral cannot be calculated if a token with more than 18 decimals is added to the list of accepted tokens, resulting in denial-of-service.

Recommended Mitigation: Consider scaling to a greater number of decimals if collateral tokens with more than 18 decimals will be added.

The Standard DAO: Fixed by commit `cf871f7` – not suitable for hypervisor deposits, but should be ok for collateral.

Cyfrin: Verified, now supports collateral tokens with more than 18 decimals; however, division before multiplication for the `scale < 0` branch`could be problematic - it might be better to first scale all decimals to 36 and then divide back down to 18 in the return statement of`tokenToUSD‘.

The Standard DAO: Fixed in commit `2342302`.

Cyfrin: Verified, now scales decimals to 36 before rescaling back down to 18.

7.5 Informational

7.5.1 Unnecessary typecast of `msg.sender` to `address`

Description: There is an instance of the `msg.sender` context variable that is unnecessarily cast to `address` in `SmartVaultYieldManager::deposit`:

```
uint256 _balance = IERC20(_collateralToken).balanceOf(address(msg.sender));
```

Recommended Mitigation: Consider removing the `address` typecast as `msg.sender` is already an address.

The Standard DAO: Fixed by commit 1a9dc5f.

Cyfrin: Verified, typecast has been removed.

7.5.2 Comment incorrectly refers to € when it should be $

Description: The following comment is present when summing the stablecoin collateral in `SmartVaultV4::yieldVaultCollateral`:

```
// both USDs and its vault pair are € stablecoins, but can be equivalent to €1 in collateral
```

Here, the € symbol is used for USD instead of $.

Recommended Mitigation: Update the comment to use the $ symbol.

The Standard DAO: No longer applicable. Comment removed in commit 5862d8e.

Cyfrin: Verified, comment has been removed.

7.5.3 Inconsistent use of equivalent function parameter and immutable variable in `SmartVaultYieldManager::_withdrawUSDsDeposit` is confusing

Description: When withdrawing collateral from the `USDs` Hypervisor in `SmartVaultYieldManager::_withdrawUSDsDeposit`, the `_hypervisor` parameter will always be equal to the immutable `usdsHypervisor` variable due to the following conditional check in `SmartVaultYieldManager::withdraw`:

```
_hypervisor == usdsHypervisor ?
    _withdrawUSDsDeposit(_hypervisor, _token) :
    _withdrawOtherDeposit(_hypervisor, _token);
```

However, a mixture of both the `_hypervisor` parameter and the equivalent immutable variable is used within`SmartVaultYieldManager::_withdrawUSDsDeposit`:

```
    function _withdrawUSDsDeposit(address _hypervisor, address _token) private {
    IHypervisor(_hypervisor).withdraw(_thisBalanceOf(_hypervisor), address(this), address(this),
    ↪   [uint256(0),uint256(0),uint256(0),uint256(0)]);
    _swapToSingleAsset(usdsHypervisor, USDC, ramsesRouter, 500);
    _sellUSDC(_token);
}
```

This is confusing to the reader as it could imply that the `_hypervisor` parameter differs from the immutable `usdsHypervisor`, which is not the case.

Recommended Mitigation: Consider consistent utilization of either the `_hypervisor` parameter or the immutable `usdsHypervisor` variable.

The Standard DAO: Fixed by commit f601a11.

Cyfrin: Verified, the immutable variable is now used exclusively.

7.5.4 `USDC` cannot be added as an accepted collateral token

Description: At least 10% of each collateral deposit to Gamma must be directed toward the `USDs/USDC` pool underlying the `USDs` Hypervisor:

```
function _usdDeposit(address _collateralToken, uint256 _usdPercentage, bytes memory _pathToUSDC)
↪   private {
    _swapToUSDC(_collateralToken, _usdPercentage, _pathToUSDC);
    _swapToRatio(USDC, usdsHypervisor, ramsesRouter, 500);
    _deposit(usdsHypervisor);
}
```

During this process, `SmartVaultYieldManager::_swapToUSDC` swaps the collateral token to `USDC`; however, this would fail for `USDC` without additional handling as it has no path to itself. A similar issue is present in `SmartVaultYieldManager::_sellUSDC` when attempting to withdraw the `USDs` Hypervisor deposits to USDC.

Additionally, assuming the was correctly handled, broad use of `SmartVaultYieldManager::thisBalanceOf` would result in the entire balance of USDC being utilized for the `USDs` Hypervisor deposit within `SmartVaultYieldManager::_swapToRatio` without considering the subsequent Hypervisor deposit:

```
uint256 _tokenBBalance = _thisBalanceOf(_tokenB);
(uint256 _amountStart, uint256 _amountEnd) = IUniProxy(uniProxy).getDepositAmount(_hypervisor, _tokenA,
↪   _thisBalanceOf(_tokenA));
```

Furthermore, if `USDC` were to be added as an accepted collateral token, this would result in liquidations being blocked for blacklisted Smart Vaults. An attacker could deposit illegally-obtained `USDC` into their Smart Vault, borrowing `USDs` and avoiding ever being liquidated as the attempt by the protocol to transfer these tokens out would fail.

Impact: `USDC` cannot be added as an accepted collateral.

Recommended Mitigation: These issues should first be addressed if it is desired to add `USDC` as an accepted collateral token.

The Standard DAO: Acknowledged. Not fixing because we have no intentions to add `USDC` as a collateral type. If we were to add it, we believe it would still be fine, as long we didn't add hypervisor data for it. This seems acceptable to us.

Cyfrin: Acknowledged.

7.5.5 Native asset cannot be removed using `SmartVaultV4::removeAsset`

Description: `SmartVault::removeAsset` allows Smart Vault owners to remove assets from their Vault, including collateral assets so long as the Vault remains fully collateralized. This currently works for ERC-20 collateral tokens; however, there is no handling for the case where `_tokenAddr == address(0)`. This address corresponds to the `NATIVE` symbol in the list of accepted `TokenManager` tokens, but native transfers attempted by `SafeERC20::safeTransfer` fail because this edge case is not considered.

```
function removeAsset(address _tokenAddr, uint256 _amount, address _to) external onlyOwner {
    ITokenManager.Token memory token = getTokenManager().getTokenIfExists(_tokenAddr);
    if (token.addr == _tokenAddr && !canRemoveCollateral(token, _amount)) revert Undercollateralised();
    IERC20(_tokenAddr).safeTransfer(_to, _amount);
    emit AssetRemoved(_tokenAddr, _amount, _to);
}
```

While native collateral withdrawals are already correctly handled by `SmartVaultV4::removeCollateralNative`, this edge case results in an asymmetry between ERC-20 and native asset transfers within `SmartVault::removeAsset`.

Impact: Smart Vault owners cannot use `SmartVault::removeAsset` to remove native tokens from their Vault.

Recommended Mitigation: Consider handling the case where the native asset is attempted to be removed. Also, the use of events should be reconsidered depending on whether the asset removed is a collateral asset.

```
function removeAsset(address _tokenAddr, uint256 _amount, address _to) external onlyOwner {
    ITokenManager.Token memory token = getTokenManager().getTokenIfExists(_tokenAddr);
    if (token.addr == _tokenAddr && !canRemoveCollateral(token, _amount)) revert Undercollateralised();
+   if(_tokenAddr == address(0)) {
+       (bool sent,) = payable(_to).call{value: _amount}("");
+       if (!sent) revert TransferError();
+   } else {
-   IERC20(_tokenAddr).safeTransfer(_to, _amount);
+        IERC20(_tokenAddr).safeTransfer(_to, _amount);
+   }
    emit AssetRemoved(_tokenAddr, _amount, _to);
}
```

The Standard DAO: Fixed by commits 8257c4c & 57d5db4.

Cyfrin: Verified, native collateral can now be removed via either function.

7.6 Gas Optimization

7.6.1 Unnecessary call to `SmartVaultV4::usdCollateral` when depositing/withdrawing collateral to/from yield positions

Description: When depositing/withdrawing collateral to/from yield positions in `SmartVaultV4`, the Smart Vault is validated to remain sufficiently collateralized and the collateral value is validated to have not dropped by more than 10%:

```
if (undercollateralised() || significantCollateralDrop(_preDepositCollateral, usdCollateral())) revert
↪  Undercollateralised();
```

This logic calls `SmartVaultV4::usdCollateral` to obtain the value of the Smart Vault collateral; however, this is an expensive call that performs multiple loops over collateral tokens and is also invoked within `SmartVaultV4::undercollateralised`:

```
function undercollateralised() public view returns (bool) {
    return minted > maxMintable(usdCollateral());
}
```

Recommended Mitigation: Consider calling `usdCollateral()` only once after the deposit/withdrawal of collateral, then pass that value to `undercollaterlised()` and `significantCollateralDrop()`. The current implementation of `undercollateralised()` can be refactored into a public function that calls `usdCollateral()` and passes the result to an internal `_undercollateralised()` function that takes the collateral value as argument.

The Standard DAO: Fixed by commit 3fdefc8.

Cyfrin: Verified, a private function has been introduced.

7.6.2 Cached `_token0` not used

Description: In `SmartVaultYieldManager::_swapToSingleAsset`, the cached address `_token0` is not used in the condition of the ternary operation:

```
address _token0 = IHypervisor(_hypervisor).token0();
address _unwantedToken = IHypervisor(_hypervisor).token0() == _wantedToken ?
    IHypervisor(_hypervisor).token1() :
    _token0;
```

Recommended Mitigation: Use the cached `_token0` variable in the comparison.

The Standard DAO: Fixed by commit 1c30144.

Cyfrin: Verified, the cached address is now used.