Audit Findings

- ***High Vulnerabilities: 2 findings***
- ***Medium Vulnerabilities: 3 findings***
- ***Gas optimization: 8 findings***

Background:

Sturdy is a protocol for interest-free borrowing and high yield lending.

Lenders deposit stablecoins that they want to earn yield on while borrowers provide collateral and take out loans at no interest (interest rates will kick in above certain utilization rates, but this functionality is out of scope). Sturdy accomplishes this by staking collateral provided by borrowers in third party protocols like Yearn, Lido, and Convex. The yield from collateral staking is periodically harvested, swapped to stablecoins, and distributed to stablecoin lenders.

Sturdy forked Aave V2 contracts for its underlying accounting and lending pool mechanics. In order to convert non-interest bearing tokens ('external assets', e.g. ETH) to their interest bearing equivalent ('internal assets', e.g. stETH), Sturdy uses modules called 'vaults.' When users deposit external assets to a vault, the vault immediately stakes it, converting it to an internal asset, and deposits it to the lending pool. When users want to withdraw their collateral, the vault withdraws the internal asset from the lending pool, unstakes it, and returns it to the user. The user can also deposit or withdraw internal assets directly from the lending pool by interacting directly with the smart contracts.

`PoolAdmin` can harvest the yield by calling a function that distributes excess staked collateral to lenders after swapping it to stablecoins. This can take on a different form based on the specific collateral asset and staking strategy as described for each vault contract below.

The full repository (with tests) can be found here.

High Vulnerabilities

Hard-coded slippage may freeze user funds during market turbulence

Lines of code

https://github.com/code-423n4/2022-05-sturdy/blob/main/smart-contracts/GeneralVault.sol#L125

https://github.com/code-423n4/2022-05-sturdy/blob/main/smart-contracts/LidoVault.sol#L130-L137

Impact

GeneralVault.sol#L125

GeneralVault set a hardcoded slippage control of 99%. However, the underlying yield tokens price may go down.

If Luna/UST things happen again, users' funds may get locked.

LidoVault.sol#L130-L137

Moreover, the withdrawal of the lidoVault takes a swap from the curve pool. 1 stEth worth 0.98 ETH at the time of writing.

The vault can not withdraw at the current market.

Given that users' funds would be locked in the lidoVault, I consider this a high-risk issue.

Proof of Concept

1 stEth = 0.98 Eth

LidoVault.sol#L130-L137

Tools Used

Slither, remix.

Recommended Mitigation Steps

There are different ways to set the slippage.

The first one is to let users determine the maximum slippage they're willing to take.

The protocol front-end should set the recommended value for them.

```
 function withdrawCollateral(
    address _asset,
    uint256 _amount,
    address _to,
    uint256 _minReceiveAmount
  ) external virtual {
      // ...
    require(withdrawAmount >= _minReceiveAmount,
Errors.VT_WITHDRAW_AMOUNT_MISMATCH);
  }
```

The second one has a slippage control parameter that's set by the operator.

```
   // Exchange stETH -> ETH via Curve
    uint256 receivedETHAmount =
CurveswapAdapter.swapExactTokensForTokens(
      _addressesProvider,
      _addressesProvider.getAddress('STETH_ETH_POOL'),
      LIDO,
      ETH,
      yieldStETH,
      maxSlippage
    );

   function setMaxSlippage(uint256 _slippage) external onlyOperator {
        maxSlippage = _slippage;

        //@audit This action usually emit an event.
        emit SetMaxSlippage(msg.sender, slippage);
```

```
    }
```

These are two common ways to deal with this issue. I prefer the first one.
The market may corrupt really fast before the operator takes action.
It's nothing fun watching the number go down while having no option.

The check for value transfer success is made after the return statement in _withdrawFromYieldPool of LidoVault

Lines of code

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/LidoVault.sol#L142

Impact

Users can lose their funds.

Proof of Concept

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/LidoVault.sol#L142

The code checks transaction success after returning the transfer value and finishing execution. If the call fails the transaction won't revert since require(sent, Errors.VT_COLLATERAL_WITHDRAW_INVALID); won't execute.

Users will have withdrawn without getting their funds back.

Recommended Mitigation Steps

Return the function after the success check.

Medium Vulnerabilities

Possible lost msg.value

Lines of code

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L75-L89

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/LidoVault.sol#L79-L104

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/ConvexCurveLPVault.sol#L131-L149

Impact

Possible lost value in `depositCollateral` function call.

Proof of Concept

In call `depositCollateral` can will send value and the asset can be an ERC20(!= address(0)), if `LidoVault` and `ConvexCurveLPVault` contract receive this call the funds will be lost.

Also in LidoVault, L88, if sent as asset ETH(== address(0)) and send more value than `_amount`(msg.value > _amount), the accident will be lost.

Recommended Mitigation Steps

In GeneralVault, `depositCollateral` function:

- Check if the `msg.value` is zero when the `_asset` is ERC20(!= address(0))
- Check if the `msg.value` is equeal to `_amount` when the `_asset` ETH(== address(0))

```
function depositCollateral(address _asset, uint256 _amount) external payable virtual {
```

```
  if (_asset != address(0)) { // asset = ERC20
    require(msg.value == 0, <AN ERROR FROM Errors LIBRARY>);
  } else { // asset = ETH
    require(msg.value == _amount, <AN ERROR FROM Errors LIBRARY>);
  }

  // Deposit asset to vault and receive stAsset
  // Ex: if user deposit 100ETH, this will deposit 100ETH to Lido and
receive 100stETH TODO No Lido
  (address _stAsset, uint256 _stAssetAmount) =
_depositToYieldPool(_asset, _amount);

  // Deposit stAsset to lendingPool, then user will get aToken of
stAsset
  ILendingPool(_addressesProvider.getLendingPool()).deposit(
    _stAsset,
    _stAssetAmount,
    msg.sender,
    0
  );

  emit DepositCollateral(_asset, msg.sender, _amount);
}
```

Also can remove the `require(msg.value > 0, Errors.VT_COLLATERAL_DEPOSIT_REQUIRE_ETH);` in LidoVault, L88

`UNISWAP_FEE` is hardcoded which will lead to significant losses compared to optimal routing.

Lines of code

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L48

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L184

Impact

In `YieldManager`, `UNISWAP_FEE` is hardcoded, which significantly reduces the possibilities and will lead to non optimal routes. In particular, all swaps using ETH path will use the wrong pool as it will use the ETH / USDC 1% one due to this line.

Proof of Concept

For example for CRV / USDC, the optimal route is currently CRV -> ETH and ETH -> USDC, and the pool ETH / USDC with 1% fees is tiny compared to the ones with 0.3 or 0.1%. Therefore using the current implementation would create a significant loss of revenue.

Recommended Mitigation Steps

Basic mitigation would be to hardcode in advance the best Uniswap paths in a mapping like it's done for Curve pools, then pass this path already computed to the swapping library. This would allow for complex routes and save gas costs as you would avoid computing them in `swapExactTokensForTokens`.

Then, speaking from experience, as `distributeYield` is `onlyAdmin`, you may want to add the possibility to do the swaps through an efficient aggregator like 1Inch or Paraswap, it will be way more optimal.

`processYield()` and `distributeYield()` may run out of gas and revert due to a long list of extra rewards/yields.

Lines of code

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/ConvexCurveLPVault.sol#L105-L110

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L129-L136

Impact

Yields will not be able to be distributed to lenders because attempts to do so will revert.

Proof of Concept

The `processYield()` function loops overall of the extra rewards and transfers them

```
File: smart-contracts/ConvexCurveLPVault.sol   #1

105       uint256 extraRewardsLength =
IConvexBaseRewardPool(baseRewardPool).extraRewardsLength();
106       for (uint256 i = 0; i < extraRewardsLength; i++) {
107         address _extraReward =
IConvexBaseRewardPool(baseRewardPool).extraRewards(i);
108         address _rewardToken =
IRewards(_extraReward).rewardToken();
109         _transferYield(_rewardToken);
110       }
```

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/ConvexCurveLPVault.sol#L105-L110

There is no guarantee that the tokens involved will be efficient in their use of gas, and there are no upper bounds on the number of extra rewards:

```
   function extraRewardsLength() external view returns (uint256) {
        return extraRewards.length;
```

```
    }


    function addExtraReward(address _reward) external returns(bool){
        require(msg.sender == rewardManager, "!authorized");
        require(_reward != address(0),"!reward setting");


        extraRewards.push(_reward);
        return true;
    }
```

https://github.com/convex-eth/platform/blob/main/contracts/contracts/BaseRewardPool.sol#L105-L115

Even if not every extra reward token has a balance, an attacker can sprinkle each one with dust, forcing a transfer by this function

`_getAssetYields()` has a similar issue:

```
File: smart-contracts/YieldManager.sol   #X

129       AssetYield[] memory assetYields =
_getAssetYields(exchangedAmount);
130       for (uint256 i = 0; i < assetYields.length; i++) {
131         if (assetYields[i].amount > 0) {
132           uint256 _amount =
_convertToStableCoin(assetYields[i].asset, assetYields[i].amount);
133           // 3. deposit Yield to pool for suppliers
134           _depositYield(assetYields[i].asset, _amount);
135         }
136       }
```

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L129-L136

Tools Used

Code inspection.

Recommended Mitigation Steps

Include an offset and length as is done in `YieldManager.distributeYield()`.

Gas Optimization

Loop gas optimization

Impact

Severity: Gas-Optimization

---ARRAY.LENGTH SHOULD NOT BE LOOKED UP IN EVERY LOOP OF A FOR-LOOP

---IT COSTS MORE GAS TO INITIALIZE VARIABLES TO ZERO

---PREFIX INCREMENTS INSTEAD OF POSTFIX INCREMENTS

Lines Affected

_There are nine lines of code that can utilize these particular gas optimization fixes. _

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L130

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/ConvexCurveLPVault.sol#L106

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L218

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L120

Recommendation

---.LENGTH: Caching the length changes each of these to a DUP (3 gas), and gets rid of the extra DUP needed to store the stack offset.In particular, in for loops, when using the length of a storage array as the condition being checked after each loop, caching the array length in memory can yield significant gas savings if the array length is high.

---Initialization: Let the default zero be applied instead of initializing the default variable.

-uint256 value; is cheaper than uint256 value = 0;.

-for (uint i; i counter; ++i) is cheaper than for (uint i=0; i <counter; i++)

---++I Cost less gas than I++, especially when it's used in For-Loops (--I/I-- TOO)

-Saves 6 gas PER LOOP

Using Private rather than public for constants, saves gas.

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table.

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/CollateralAdapter.sol#L22

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L55

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L41

Public Functions not called by the contract should be declared external instead.

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public` and can save gas by doing so.

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/CollateralAdapter.sol#L35

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L61

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L60

Using >0 costs more than !=0 when used on a uint in a require() statement.

This change saves 6 gas per instance
https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L179

Internal functions not called by the contract should be removed to save deployment gas.

If the functions are required by an interface, the contract should inherit from that interface and use the `override` keyword.

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L235

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L204

Using private rather than public for constants, saves gas.

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table.

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/CollateralAdapter.sol#L22

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L55

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L41

Empty blocks should be removed or emit something.

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be `abstract` and the function signatures be added without any default implementation. If the block is an empty if-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...}` => `if(!x){if(y){...}else{...}}`).

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L153

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/LidoVault.sol#L24

Function guaranteed to revert when called by normal users can be marked payable.

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE`(2),`DUP1`(3),`ISZERO`(3),`PUSH2`(3),`JUMPI`(10),`PUSH1`(3),`DUP1`(3),`REVERT`(0),`JUMPDEST`(1),`POP`(2), which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/CollateralAdapter.sol#L43-L47

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/ConvexCurveLPVault.sol#L37

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/GeneralVault.sol#L165

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/LidoVault.sol#L30

https://github.com/code-423n4/2022-05-sturdy/blob/78f51a7a74ebe8adfd055bdbaedfddc05632566f/smart-contracts/YieldManager.sol#L64