CERTORA

SMART CONTRACT SECURITY ANALYSIS REPORT AND FORMAL VERIFICATION PROPERTIES OF *MIMIC FINANCE*

Table of contents

Summary

This document describes the specification and verification of **mimic-fi / v2-core contracts** using the Certora Prover, and manual code review findings. The work was undertaken from **15 November 2022** to **11 January 2023**. The latest commit that was reviewed and run through the Certora Prover was **d2e8f4f**.

The scope of our verification includes the following contracts:

- Helpers: all contracts listed here except the ones in the test directory
- Registry: all contracts listed here except the ones in the test directory
- `PriceOracle.sol` (here)
- `PriceFeedProvider.sol` (here)
- `SwapConnector.sol` (here)
- `SmartVault.sol` (here)
- `CompoundStrategy.sol` (here)
- `AaveV2Strategy.sol` (here)

The Certora Prover proved the implementation of the contracts above are correct with respect to the formal rules written by the Certora team. The team also performed a manual audit of the contracts. During the verification process and the manual audit, the Certora Prover discovered bugs in the code listed in the table below.

All the rules will be publicly available in the project's repository.

Main Issues Discovered

ID	***Vulnerability Category***	***Vulnerability Description***	***Requirements***	***Attack Complexity***	***Impact***
H-01	*Lack of checks + reentrancy*	*Unsanitized swap data enables a smart-contract swapper to bypass the oracle protections and create an arbitrary swap that can result in a drain of funds without withdraw permissions*	*Authorized to call swap()* *3 or more assets authorized to be swapped* *Swapper is a smart-contract*	*Mid+*	*Highest*

H-02	*Lack of Checks*	*The bridger can drain all the funds by providing a malicious address as the bridge in the unchecked bytes calldata*	*Authorized to call bridge()* *Calldata not parsed nor checked in the action*	*Lowest*	*Highest*
M-01	*Precision error*	*getPrice() returns zero for a base-quote pair even though the price of the reverse pair is positive*	*The oracle value of the base is significantly higher than the quote* *There is a difference in the decimals of the tokens*	*High*	*High*
M-02	*Revert of a getter*	*Querying the price of a base-quote pair reverts, but querying the reciprocal price does not revert*	*Quote decimals are larger than the base decimals by more than 18*	*Medium*	*Medium*
M-03	*Precision error*	*Even if the price of both the base-pivot and quote-pivot are equal, the getPrice() of a base-quote pair can still be different*	*There is a large difference in the decimals of the tokens*	*Medium*	*High*
M-04	*Revert on safeApprove()*	*OpenZeppelin's ERC20.safeApprove() will revert if the approval amount is not currently 0 (or resetting it)*	*External call able to not transfer the stated amount*	*Lowest*	*Medium+*

H-01: Draining funds without withdraw permissions via a reentrancy + arbitrary swap data

Severity: High

Category: Lack of checks, reentrancy

File(s): ParaswapV5Connector.sol, OneInchV5Connector.sol

Property broken:

Taking out all the Vault's funds requires withdraw permissions

The bug:

There is no storage accounting of assets, so SmartVault relies on checks right before and after each `swap()`. But since there are no reentrancy guards, it's possible to circumvent those checks. Reentrancy, paired with arbitrary actual swap data, enables draining funds from the vault.

Description:

The attacker constructs a swap that trades some real asset against a second. The checks before the swap will remember that current state. But then, inside the swap, the attacker would gain execution back (due to a malicious token or 1inch's implementation), allowing them to create a second trade with a different source asset that would be checked as if this trade was the original. The attacker made two withdrawals but paid for only one.
This exploit scales with the number of reentrancy times, which is limited by the number of different tokens in the pool.

Exploit Scenario: (Example illustration here)

1. The attacker creates a trade between the two tokens, but with the data of a malicious token that enables reentrancy in a malicious pool.
2. `swap()` is called. The expected post amount is calculated for the transition between state0 (before any trade) and state1 (after a single trade).
3. Reentrancy happens by the malicious token, and the swapper creates another `swap()` call again.
    a. The expected post amount is still unchanged from the first trade.
4. Repeat N times, where N is the number of different tokens in the pool minus one.
5. On the last swap call, make a genuine swap between the two declared tokens. The expected post amount is valid.
6. The SmartVault is now in state1, and as execution "recursion" unfolds, all the checks pass.

Mimic's response:
A reentrancy guard was added: https://github.com/mimic-fi/v2-core/pull/50

H-02: Bridge address can be chosen arbitrarily to drain bridged funds

Severity: High

Category: Lack of checks

File(s): HopConnector.sol (and probably all future BridgeConnectors)

Property broken:

Taking out all the Vault's funds requires withdrawal permissions.

The bug:

The bridge address can be chosen arbitrarily, instead of being initialized at construction time.

Description:

A malicious actor with permission to call the `bridge()` function (bridger) can drain all the SmartVault's funds by choosing themselves as the bridge address. The bridge address is not validated.

Exploit Scenario:

1. The bridger sets themselves or their wallet as the bridge address.
2. Next time someone calls `bridge()`, all the fund amounts that should've been bridged will be sent directly to the bridger.

Mimic response:
A reentrancy guard was added: https://github.com/mimic-fi/v2-core/pull/50

M-01: Base-quote pair returned zero but the reverse pair did not

Severity: Medium

Category: Precision error

File(s): SmartVault.sol, PriceOracle.sol

Property broken:

The price of a base-quote pair is zero if and only if the price of the reverse pair is also zero.

The rule:

```
rule pricesEqualZeroMutually(address base, address quote) {
    requireInvariant tokensPriceReciprocity(base, quote);
    matchDecimals(base, quote);
    matchDecimals(quote, base);

    // Additional assumptions for a more realistic scope
    require getFeedPrice(base, quote) >= 100000;
    require getFeedPrice(quote, base) >= 100000;
    requireValidDecimals(quote);
    requireValidDecimals(base);

    assert getPrice(base, quote) == 0 <=> getPrice(quote, base) == 0;

    // this checks that returned price cannot be zero
    assert getPrice(base, quote) != 0;
}
```

Exploit Scenario:
According to the example given in the verification report,
`getPrice()` returned 0, because it used the method
`_scalePrice(price=0x4c4b40, priceDecimals=11, resultDecimals=4)` that returned 0

The simulated `(price, decimals)` obtained by the Oracle were
`(0x2863c1f5cdae42f9540000000, 25)`

`_getFeedData()` returned `(0x2863c1f5cdae42f9540000000, 25)`
`_getInversePrice()` returned `(0x4c4b40, 11)`

The above represents a case where the ratio between base/quote is enormous, and there is a difference between the decimals of the tokens.

Now when `_scalePrice()` is called with `(price=0x4c4b40, priceDecimals=11, resultDecimals=4)` in order to scale the inverse price, the calculation used, forces the scaled price to become zero:
`return (price / 10**(priceDecimals.uncheckedSub(resultDecimals)`
0x4c4b40 / 10**(11-4) = 0x4c4b40 / 10,000,000 = 5,000,000 / 10,000,000 = 0 since it is uint256 which causes the result to round down to zero.

Implications:
When `getPrice()` returns zero, the following negative impacts can occur:

1. **Unprofitable swaps:** in `swap()` if the `limitType` is `Slippage`, the expected `minAmountOut` of the swap will be nulled. Therefore the user can get an extremely bad rate or even nothing. Because of that, the protocol will also get fewer fees from the swap.
2. **Fees Overcharge:** in `_payFee()` (the function that pays fees to the protocol and that can be called on `withdraw`, `exit` and `swap`), the `feeAmountInFeeToken` will be nulled, so the `Fee.totalCharged` is not changed, but fees are transferred. As a result, the representation of the `totalCharged` fees is less than the reality.

   A similar scenario can also happen during the call of `_setFeeConfiguration()` when the `fee.totalCharged` is nullified.

Mimic's response:
We will contemplate this for feature versions, but not now since we are mainly working with known tokens like native tokens or stable coins.

M-02: Base-quote pair reverted but the reverse pair did not

Severity: Medium - `getPrice()` reverts while it should not

Category: Precision error

File(s): SmartVault.sol, PriceOracle.sol

Property broken:

Querying the price of a base-quote pair reverts, if and only if, querying the reciprocal price also reverts.

The rule:

```
rule getPriceMutuallyRevert(address base, address quote) {

    getPrice@withrevert(base, quote);
    bool revert1 = lastReverted;
    getPrice@withrevert(quote, base);
    bool revert2 = lastReverted;

    assert revert1 <=> revert2;
}
```

The exploitation scenario:
According to the example given in the verification report,
`getPrice(base, quote)` did not revert, but the inverse `getPrice(base, quote)` did revert.

The reason for the revert was, as mentioned in the report:
`require(baseDecimals <= quoteDecimals.uncheckedAdd(FP_DECIMALS), 'BASE_DECIMALS_TOO_BIG')`

The prover selected such a pair that for one token the decimals were 6, while for the other token, the decimals were 25, meaning the difference between the decimals of the tokens was 25 - 6 = 19.

The `require` statement in the code allows for a difference between the decimals that is not bigger than `FP_DECIMALS` (defined as a constant `FP_DECIMALS == 18`). Yet it checks the requirement only in one direction, meaning that:

`baseDecimals must be smaller or equal to quoteDecimals + 18`

But there is no check that
`quoteDecimals must be smaller or equal to baseDecimals + 18`

That asymmetry resulted in `getPrice(base, quote)` not reverting and `getPrice(quote, base)` reverting.

Implications:
When `getPrice()` reverts, the following negative impacts can occur:

1. **Denial of service:** During `withdraw()`, `exit()`, and `swap()`, the fees are paid with `_payFee()`, which uses `getPrice()`. If `getPrice()` reverts, these functions will revert too.
2. **Inability to change fee rates:** `_setFeeConfiguration()`, which is used by all the different fee setter functions, contains a call to `getPrice()` that will revert. As a result, the fee configuration could not be modified.

Mimic's response:
We will contemplate this for feature versions, but not now since we are mainly working with known tokens like native tokens or stable coins..

M-03: The price of two tokens with respect to the pivot token is one, but their relative price is zero

Severity: Medium

Category: Precision error

File(s): SmartVault.sol, PriceOracle.sol

Property broken:

If `getPrice(base, pivot) == getPrice(quote, pivot) == ONE`
then: `getPrice(base, quote) == ONE`.

The rule:

```
rule pivotUnitPrice(address base, address quote) {
    usePivotForPair(base, quote);
    address pivot = oracle.pivot();

    // View prices from feed
    uint256 priceBase = getFeedPrice(base, pivot);
    uint256 priceQuote = getFeedPrice(quote, pivot);

    requireValidDecimals(pivot);
    matchDecimals(base, pivot);
    matchDecimals(quote, pivot);

    assert
    (getPrice(base, pivot) == FixedPoint_ONE() &&
    getPrice(quote, pivot) == FixedPoint_ONE()) =>
    getPrice(base, quote) == FixedPoint_ONE();
}
```

The exploitation scenario:
If there is no feed available for `getPrice(base, quote)` but there are feeds available for `getPrice(base, pivot)` and `getPrice(quote, pivot)`, the calculation of `getPrice(base, quote)` will be done using the pivot.

According to the example given in the verification report, we see a violation of the rule: `getPrice(base, quote)` is zero, even though both `getPrice(base, pivot)` and `getPrice(quote, pivot)` were a unit price.

The bug is similar to M-01, and it stems from the big difference between the decimals of `baseToken` and `quoteToken`. That difference later creates a big ratio between the feed data for the quote vs. feed data for the base, and dividing a tiny number by a vast number produces zero. It's a combination of rounding down to zero and a large decimal difference.

The prover found the following scenario:
`baseToken.decimals()` = 4
`quoteToken.decimals()` = 34
`pivotToken.decimals()` = 18
`_getFeedData(baseFeed) = (0x2710 (same as priceBase), 18) = (priceBase, decimalsBase)`
`_getFeedData(quoteFeed) = (0x1ed09bead87c037b115566fc0ffff (same as priceQuote), 18) = (priceQuote, decimalsQuote)`

That will result in:
`getPrice(base, pivot)` = `_scalePrice(price=0x2710 (same as priceBase), priceDecimals=18, resultDecimals=32)` = `0xde0b6b3a7640000`

`getPrice(quote, pivot)` = `_scalePrice(price=0x1ed09bead87c037b115566fc0ffff (same as priceQuote), priceDecimals=18, resultDecimals=2)` = `0xde0b6b3a7640000`

`getPrice(base, quote)` = `_getPivotPrice → basePrice.divDown(quotePrice)` = `0x2710.divDown(0x1ed09bead87c037b115566fc0ffff)` = **0**

`getPrice(base, quote)` returns zero due to the big difference between `basePrice` and `quotePrice`. As a result, the division results in a tiny number rounded down to zero. To better understand the results above, we will convert them to decimal values:

```
0x2710 == 10,000 = 1e5
0x1ed09bead87c037b115566fc0ffff ==
10,000,000,000,000,009,999,999,999,999,999 ~ 1e34
0xde0b6b3a7640000 == 1,000,000,000,000,000 = 1e18 = FixedPoint.ONE
```

Implications:
Although the chances of having simultaneously: `getPrice(base, pivot) == getPrice(quote, pivot) == ONE` are extremely low, the implication of `getPrice()` returning zero can be significant, as previously discussed in the M-01 bug.

Mimic's response:
We will contemplate this for feature versions, but not now since we are mainly working with known tokens like native tokens or stable coins.

M-04: Unexpected reverts from ERC20.safeApprove()

Severity: Medium

Category: DoS, Logic Flaw

File(s): Potentially all connector files. Especially ParaswapV5Connector.sol, OneInchV5Connector.sol, and HopConnector.sol

The bug:

The approval was not reset to 0 after an external call that should have used said approval.

Description:

OZ's `SafeERC20.safeApprove()` assumes the current approval is 0. Otherwise, it assumes a malicious play and reverts.
There could be many situations where this would happen naturally. For instance, when a swap didn't need to use all the `amountIn` that was approved (if that's the kind of swap requested, e.g. `swapTokensForExactTokens`).
This could also happen intentionally, like in Paraswap and 1inchSwap, where the actual swap data is different from the declared amounts.

The implication is a DoS to the relevant swap. The approved amount can be set to zero via a `call()` primitive externally to the SmartVault.

Exploit Scenario:

1. `1inchSwap()` was called, but it was efficient and didn't use all the needed approval.
2. The following `1inchSwap()` calls for that token would always revert since the current approval to the 1inch address on that token is not 0 when trying to approve the new swap.

Mimic's response:
The approval is now set to zero before any calls to `approve()`:
https://github.com/mimic-fi/v2-core/pull/49 and
https://github.com/mimic-fi/v2-smart-vaults/pull/57/files

Assumptions and Simplifications Made During Verification

We made the following assumptions during our verification:

- We unroll loops. Violations that require a loop to execute more than twice will not be detected.
- We have checked the contracts only against the AAVE strategy, and we substantially simplified the AAVE protocol.
- Properties and verification are done based on AAVE's Strategy implementation with harnesses. We do so since an array as a return value is not currently supported in CVL.

Notations

✅Indicates the rule is formally verified.
❌Indicates the rule is violated.
⏳Indicates the rule is timing out.

Formal Verification Properties

Since the protocol consists of different contracts, we will present the relative properties for each contract. The name of the rule will be shown as: (`ruleName`)

Smart Vault

1. ✅ Only authorized users can run all the methods in the contract without reverting. For all the other (not authorized) users, if they try to execute any method, it should revert (`onlyAuthUserCanCallFunctions`)
2. ✅ `collect()` integrity:
   The balances of any user or token not participating in the method call are not affected. Also, assets don't disappear nor are created out of thin air (`collectTransferIntegrity`)
3. ✅ `withdraw()` integrity:
   The balances of any user or token not participating in the method call are not affected. Also, assets don't disappear nor are created out of thin air. For simplification, we assumed there are no fees. Also we split the verification in two separate rules: one rule when the transferred token is not native, and second rule for transfer of a native token. (`withdrawTransferIntegrity`, `withdrawTransferIntegrityOfNativeToken`)
4. ✅ After calling `wrap()` and then `unwrap()` successfully, the balance of the native wrapped token stays the same, and the unwrapped amount is the same as the initially

wrapped
(`wrapUnwrapIntegrity`)
5. ✅ After calling `unwrap()` and then `wrap()` successfully, the balance of the native wrapped token stays the same, and the wrapped amount is the same as the initially unwrapped
(`unwrapWrapIntegrity`)
6. ✅ `unwrap()` cannot revert after successfully calling `wrap()`
(`unwrapCannotRevertAfterWrap`)
7. ✅ `wrap()` cannot revert after successfully calling `unwrap()`
(`wrapCannotRevertAfterUnwrap`)

Registry

1. ✅ clone() integrity
(`cloneIntegrity`)
2. ✅ If an implementation is set as ‘deprecated’, it cannot be changed (see also rule 8)
(`ImplDataModifiedOnlyOnce`)
3. ✅ Cannot register the same instance twice
(`cannotRegisterSameInstanceTwice`)
4. ✅ Cannot register a cloned implementation
(`cannotRegisterClonedImplementation`)
5. ❌ A clone is never an implementation
(`cloneIsNotImplementation`)
6. ✅ Cannot deprecate twice the same implementation
(`cannotDeprecateTwice`)
7. ❌ Front-running (does any of the following actions block a sequential action from being performed?):
    a. ❌ `register()` (`frontRunning_register`)
    b. ❌ `deprecate()` (`frontRunning_deprecate`)
    c. ❌ `clone()` (`frontRunning_clone`)
    Mimic response: The first two functions are authorized and controlled by the Mimic DAO. It doesn't make a lot of sense for the DAO to front-run itself. Regarding the last function, would there be a problem if you can front-run a clone?
8. ✅ Implementation data can only be changed once
(`ImplDataModifiedOnlyOnce`)
9. ✅ Only `clone()` can set the instance implementation
(`onlyCloneChangesInstanceImplementation`)

Price Oracle

1. ❌ PriceOracle should not return a price of 0
(`pricesEqualZeroMutually`)
2. ❌ The price of a base-quote pair is zero if and only if the price of the reverse pair is also zero.
(`pricesEqualZeroMutually`)
3. ❌ Querying the price of a base-quote pair reverts if and only if querying the reciprocal price also reverts.
(`getPriceMutuallyRevert`)
4. ❌ Price reciprocity: `getPrice(base,quote) * getPrice(quote,base) = ONE`
(`ONE` = Fixed point library 1).
(`getPriceReciprocity`)
5. ❌ Price feed quote decimals cannot be changed.
The reason for violation is the havoc caused by a low-level call which changes the decimals in the contract where they are stored. In reality, the decimals are stored and fetched from a chain-link oracle contract. Therefore the possibility of a call changing them is negligible.
(`feedDecimalsCannotChange`)
6. ❌ Unity of pivot price:
If `getPrice(base, pivot) == getPrice(quote, pivot) == ONE`
then: `getPrice(base, quote) == ONE`.
(`pivotUnitPrice`)

Strategy

Properties and verification were done based on AAVE's Strategy implementation with harnesses.

1. Integrity of `join()`:
    a. ✅ `investedValue` of a strategy and the aToken balance of SmartVault should be changed according to `amountOut` value.
    (`joinIntergrity_investedValueAndATokenBalance`)
    b. ✅ The token balance of the SmartVault and aToken contracts should be changed according to `amountOut` value. `amountOut` should match `amountsIn[0]`. (`joinIntergrity_tokenBalance`)
    c. ✅ 2 small joins shouldn't bring more profit than one large join (useless with a ratio of 1:1 but might be good for other future strategies).
    (`joinIntergrity_bigVsSmalls`)

   d. ✅ No other balance, except involved in `join()`, of Token, aToken and `investedValue` was touched. (`joinIntergrity_untouchableBalance`)
2. Integrity of `claim()`:
   a. ✅ `investedValue` should be the same after a claim. (`claimIntergrity_investedValue`)
   b. ✅ The `unclaimedRewards` of a SmartVault before claiming rewards should be greater than or equal to its `unclaimedRewards` after claiming everything. (`claimIntergrity_unclaimedRewards`)
   c. ✅ Cannot claim more than `unclaimedRewards`. (`claimIntergrity_noMoreThanUnclaimedRewards`)
   d. ✅ Balance of IncentivesController and SmartVault in `rewardToken` should be updated correctly. (`claimIntergrity_balancesUpdate`)
   e. ✅ No other balance of `rewardToken` was touched by `claim()`. (`claimIntergrity_untouchableBalance`)
   f. ✅ Cannot claim if the strategy wasn't allowed. (`claimIntergrity_uselessTheSecondClaim`)
   g. ✅ The second claim in two consecutive claims should bring 0 rewards. (`claimIntergrity_uselessTheSecond`)
3. Integrity of `exit()`:
   a. ✅ The token balance of the SmartVault and aToken contracts should be changed according to `amountOut` value. `amountsIn[0]` should be equal to the sum `amountOut` and `paidFees`. (`exitIntergrity_tokenBalance`)
   b. ✅ The aToken balance of SmartVault should be changed according to the `amountOut` value. The `investedValue` of a strategy shouldn't increase. (`exitIntergrity_investedValueAndATokenBalance`)
   c. ✅ No other balance, except involved in `exit()`, of Token, AToken and investedValue was touched. (`exitIntergrity_untouchableBalance`)
   d. ✅ Checking if conditions for successful and unsuccessful strategies are correct (`exitIntergrity_checkingConditions`):
      i. If there were losses, then no fees are charged
      ii. If there were profits and withdraw more than gains, `investValue` is decreased
4. Scenarios:
   a. ✅ join/claim/exit on one strategy shouldn't affect join on another strategy (`frontrunJoinCheck`)
   b. ✅join/claim/exit on one strategy, shouldn't affect claim on another strategy (`noClaimFrontrun`)

Swap Connector

1. ✅ Addresses' balances involved in a swap should be updated correctly regarding the DeX's token, `paidFees` and `amountOut` (`swapIntergrity`)
2. ✅ Correctness of `tokenIn` balance updates by `swap()` (`swapIntergrityTokenIn`)
3. ✅ Any other address than the swap actors shouldn't be affected by `swap()` (`swapIntergrityOthersUntouchable`)
4. ✅ During `swap()`, no additional `tokenIn` tokens should be gained (`swapConsistencyTokenIn`)
5. ✅ During `swap()`, no additional `tokenOut` tokens should be gained (`swapConsistencyTokenOut`)

Disclaimer

Appendix 1: Operational Weak Points and Potential Trust Issues

In this appendix, we'd like to highlight some issues or considerations that have to be kept in mind and addressed while using, deploying, and configuring a SmartVault.

Mimic Finance's SmartVault is designed to help automate DAO operations and reduce the trust the DAO puts in their managers and operators. The SmartVault is not designed as a trustless solution by default, nor is it completely secure. It can be considered safe only if used in particular ways, along with specific "action" contracts (which were out of scope for this audit).

For a DAO to leverage Smart Vaults, they must add those checks on-chain to achieve more security and less trust in their managers and Mimic's bots infrastructure (compared to just using trusted managers).

Although this was out of scope, the security in using SmartVaults relies on those operational (or runtime) considerations and checks. These are not immediately clear from Mimic's documentation, so an independent DAO looking at this open-source could get confused.

By its design, SmartVault has a permission mechanism (the `Auth` modifier), and a registry to whitelist the components. Every DAO should have its SmartVault instance.
The ideal (while not necessarily the most practical) trust model would have only the DAO as the fully trusted party (change "the DAO" with "fully trusted parties" in the following considerations when using other trust models).

The operational considerations and checks are as follows:

1. Only the DAO should have permission to give and set other permissions (meaning, being the **only** full owner of the vault). - **critical impact**.
    a. E.g, `authorize()`, `unauthorize()`
2. Only the DAO should have permission to add or change components. Those components must be fully trusted (otherwise, a single party that could add a component would compromise the entire vault) - **critical impact**.
    a. E.g. `setStrategy()`, `setPriceOracle()`, `setSwapConnector()`, `setBridgeConnector()`
    b. The DAO must review any new component approved by the registry for a strict interface. Since the whitelist is per address and not per function/type, the DAO must prevent adding a strategy with swap functions into the wrong role.
3. Any component MUST NOT touch the storage! Components are always called by a delegate call, so a single component can compromise the vault - **critical Impact**
4. Most of SmartVault's primitives should only be configured with AUTH such that they are callable only via an action contract (that is fully trusted), that would filter and narrow its possible inputs. - **critical impact**

a. For example - if `swap()` could be called directly by some party, they could make a draining swap, effectively getting the funds out even if they don't have permission to withdraw.
b. Because of the above, action implementations should be verified and then fully trusted by the DAO.
c. Consider if it's ok that anyone could call a certain action, or have it restricted with another authentication mechanism. Note to treat the mimic bots' addresses as if it was another manager at best (since it's an external party with their keys).
d. We also suggest highlighting that in Mimic's documentation, *and in the suggested proposals*, so the DAOs won't violate that in future configurations of their instance.

5. The Oracle Manager (one who calls `setPriceFeed()`) must be the DAO. Otherwise a malicious oracle would allow a bad swap that can drain the vault. Also, that's the only current way to configure a whitelist of assets outside of the actions - **high impact**
6. Reentrancy Protections should be in place, at least in the implemented actions (or add them in a custom Vault instance, as the original SmartVault code didn't have any). For some functionality, this is crucial (like swap operations). Note that all the actions must share the same reentrancy lock state. - **high impact**
7. Vault Fees are configured at runtime instead of construction time (or registry, for that matter). This means that the entity with the AUTH to call `setFees()` must be the DAO or fully trusted (as it can change the fees to drain the vault). Mimic can protect themselves by whitelisting in their infra-only vaults that are configured correctly. - **high-impact**
8. The registry should be trusted not to DOS an attempt to change or add a component to SmartVault - **medium impact**
9. Mimic documentation/proposals claim that the Vaults are pausable, while they're not inherently so. A SmartVault can be configured to have pause functionality with a custom instance or via actions that look into an Oracle that the DAO controls. - **informational**