

Sommelier A-14

Security Audit

Dec 15, 2023

Version 1.0.0

Table of Contents

- [Introduction](#)
- [Overall Assessment](#)
- [Specification](#)
- [Source Code](#)
- [Issue Descriptions and Recommendations](#)
- [Security Levels Reference](#)
- [Disclaimer](#)

Introduction

This document includes the results of the security audit for Sommelier's smart contract code as found in the section titled 'Source Code'. The security audit was performed by the Macro security team periodically between November 23, 2023 to December 13, 2023.

The purpose of this audit is to review the source code of certain Sommelier Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro's review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
High	3	-	-	3
Medium	2	-	-	2
Low	2	1	-	1
Code Quality	6	-	-	6
Informational	2	-	-	-
Gas Optimization	1	-	-	1

Sommelier was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions with the Sommelier team.
- Available documentation in the repository.

Trust Model, Assumptions, and Accepted Risks (TMAAR)

Trusted entities:

- Strategists:
 - User that can manage positions in the cellar. Is trusted to act in the benefit of the cellars shareholders, and earns a portion of the cellars profits. All actions made by the strategist are approved by Sommelier governance. Has the ability to shutdown the cellar in case of an emergency.
- Governance:
 - Sommelier governance responsible for approving strategist actions, as well as adding or removing trusted positions for cellars.
- Multisig:
 - Approves adaptors and positions in the registry as well as adding and updating price feeds for assets in the priceRouter. Can pause cellars, which can be unpaused by governance.
- Chainlink:
 - Responsible for a majority of pricing, as well as running automated tasks for share price oracles. It is trusted that the data it provides is correct.

The goal of the system is to have checks and balances for each permissioned action, where if any one permissioned entity acts malicious, the others can remedy the situation, requiring multiple points failure before it can negatively impact users.

Assumptions:

- There is an assumption that permissioned entities will not act maliciously.
- It is assumed that the protocols that a cellar interacts with wont act maliciously, and will operate as intended.

Accepted Risks:

- Share price varies based on market conditions, and there is no guarantee share price will increase.
- Protocols that a cellar interacts with could be exploited. There are ways trusted entities can help mitigate the effect of such exploits, but there may be a negative effect on share price and a loss of funds for users.

Source Code

The following source code was reviewed during the audit:

- **Repository:** [cellar-contracts](#)
- **Commit Hash (initial):** f8f4b51e9d2bfa5c2b4627b031ecc28f47ccf0a4
- **Commit Hash (final):** 5be96d5e226ac693b29a04912e8511a1356564e9

Specifically, we audited the following contracts within this repository.

Contract	SHA256
src/modules/adaptors/Curve/CurveAdaptor.sol	7007ed798f720c912d6dac25a5d57ce56d7db0fa8e36859a4ad5ca3afd03c4af
src/modules/adaptors/Curve/CurveHelper.sol	555a9ca4cfc7e5c2ab482c40dac50f055240f31e087e4588055fcc5f96164142
src/modules/price-router/Extensions/Curve/Curve2PoolExtension.sol	4f3627e4a6db9af71bafc38aca6c0e6b6b2f5912e0895cf625da036d77d1a9fa
src/modules/price-router/Extensions/Curve/CurveEMAExtension.sol	c231a1e23bdb5ad52bec7a45c3556e34c843ab81a19ff222b6be6b13c0165654
src/modules/adaptors/Convex/ConvexCurveAdaptor.sol	93591aee6545b25971deb581f00437da66c85e96c7a517125f0865a3dc6638f7
src/modules/withdraw-queue/SimpleSolver.sol	6e349d899bda75d2488b610477df5275fd1f1bffaaf8325b5df915984e7c8feb8
src/modules/withdraw-queue/WithdrawQueue.sol	82ecc2848af2427f9565ae90bce9ff346ef620a148259246be6f9c035dd66b81

Contract	SHA256
src/modules/adaptors/ERC20Adaptor.sol	32a435b666990d0363ad4f088593b478e1a865e1eeb458d44cd083ef32e17472
src/modules/SimpleSlippageRouter.sol	efd65b3d4edbea6100bb5b183d660e5f0cf53939e936785f8f7f39af59f1ca3b

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

- ~~H-1~~ Strategist can steal assets based on slippage
- ~~H-2~~ Unverified underlying tokens
- ~~H-3~~ Curve pricing manipulation
- ~~M-1~~ Sandwich griefing of rebalance execution
- ~~M-2~~ Reentrancy guard for `CurveHelper`
- ~~L-1~~ Ensure native ETH isn't overridden
- L-2 `WithdrawalRequest` with zero price can be solved
- ~~Q-1~~ Missing slippage input constraints
- ~~Q-2~~ Use `_maxAvailable()` for `withdrawFromBaseRewardPoolAsLPT`
- ~~Q-3~~ Use curly braces to wrap branches and loops
- ~~Q-4~~ Use `uint256` for `_deadline`
- ~~Q-5~~ `SimpleSlippageRouter` error naming
- ~~Q-6~~ Use `IERC4626` instead of `Cellar`
- ~~G-1~~ Use delta share balance for slippage checks
- I-1 `ConvexCurveAdaptor` only works for mainnet
- I-2 Use caution when basing price off Curve pools

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:

- How bad things can get (for a vulnerability)
- The significance of an improvement (for a code quality issue)
- The amount of gas saved (for a gas optimization)

2. The high/medium/low **likelihood** of the issue:

- How likely is the issue to occur (for a vulnerability)

3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client must fix the issue, no matter what, because not fixing would mean significant funds/assets WILL be lost.
(H-x) High	We recommend the client must address the issue, no matter what, because not fixing would be very bad, or some funds/assets will be lost, or the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to seriously consider fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albeit not in an existential manner.
(L-x) Low	<p>The risk is small, unlikely, or may not be relevant to the project in a meaningful way.</p> <p>Whether or not the project wants to develop a fix is up to the goals and needs of the project.</p>
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

H-4 Strategist can steal assets based on slippage

TOPIC	STATUS	IMPACT	LIKELIHOOD
Input Validation	Fixed ↗	High	Medium

In `CurveAdaptor.sol` the functions `addLiquidity()` and `addLiquidityETH()` gives approvals to a provided `pool` address before making a call to it.

```
// Approve pool to spend amounts, and check for max available.
for (uint256 i; i < underlyingTokens.length; ++i)
    if (orderedUnderlyingTokenAmounts[i] > 0) {
        orderedUnderlyingTokenAmounts[i] = _maxAvailable(underlyingTokens[i], or
        underlyingTokens[i].safeApprove(pool, orderedUnderlyingTokenAmounts[i]);
    }

pool.functionCall(data);
```

Reference: [CurveAdaptor.sol#L222-L229](#)

However, this `pool` address is not validated to be associated with any positions in the cellar. This can allow a malicious strategist to pass in a custom contract and or custom `lpToken` that steals assets up to the slippage value set.

Remediations to Consider

Verify the passed in `pool` and `lpToken` are associated with one of cellars positions to ensure the contracts have been approved by the multi-sig and governance to not be malicious.

H-2 Unverified underlying tokens

TOPIC	STATUS	IMPACT	LIKELIHOOD
Input validation	Fixed ↗	High	Medium

When a strategist interacts with a curve pool, by either depositing or withdrawing liquidity, an array of `underlyingTokens` is passed in but not verified to be associated with the pool. If a strategist were to pass in some but not all of the underlying tokens of a pool when calling `withdrawLiquidityETH()`, then only the underlying tokens passed in would be received by the cellar from the `CurveHelper.sol` contract. This means that if the left out tokens received are worth less than what slippage allows, then execution would continue but the tokens would remain within the helper contract, and would be free to be taken by anyone if they call it using a custom pool. If assets of a provided underlying token are directly sent into the helper contract prior to withdrawing, the slippage check could be consistently passed, leaving at most the slippage to be taken, allowing a way to slowly drain a cellar.

Remediations to Consider

Pull the underlying tokens directly from the pool to ensure they are correct and none are missing. Additionally if the `CurveHelper.sol` contract uses the delta balance of tokens received before and after interacting with the pool rather than its total balance then assets cannot be sent in to help get over the slippage checks.

H-3 Curve pricing manipulation

TOPIC	STATUS	IMPACT	LIKELIHOOD
Oracle Manipulation	Fixed ↗	High	Low

`Curve2PoolExtension.sol` and `CurveEMAExtension.sol` allowing pricing assets associated with Curve. However Curve has been found to have some pools where the `lp_price()` and `oracle_Price()` values used for these pricing to be potentially manipulatable, which could be exploited to extract value from the cellar.

Remediations to Consider

Set bounds on each asset priced with either `Curve2PoolExtension.sol` or `CurveEMAExtension.sol`, setting a limit on how much the price can diverge from expected values. Also consider making sure assets that are priced using these extensions are illiquid and only used by cellars with share price oracles, since it has multiple guards on rapid price changes.

M-4 Sandwich griefing of rebalance execution

TOPIC	STATUS	IMPACT	LIKELIHOOD
Griefing	Fixed ↗	Medium	Low

When dealing with Curve pools that use native ETH, the `CurveHelper.sol` contract is used as a proxy to receive ETH and wrap it into WETH so the cellar can properly handle it. It will then send all the assets of the provided token types the helper contract holds to the cellar after successfully interacting with the pool and wrapping/unwrapping the ETH.

```
pool.functionCall(data);

// Iterate through tokens, update tokensOut.
tokensOut = new uint256[](underlyingTokens.length);

for (uint256 i; i < underlyingTokens.length; ++i) {
    if (address(underlyingTokens[i]) == CURVE_ETH) {
        // Wrap any ETH we have.
        uint256 ethBalance = address(this).balance;
```

```

        IWETH9(nativeWrapper).deposit{ value: ethBalance }();
        // Send WETH back to caller.
        ERC20(nativeWrapper).safeTransfer(msg.sender, ethBalance);
        tokensOut[i] = ethBalance;
    } else {
        // Send ERC20 back to caller
        ERC20 t = ERC20(underlyingTokens[i]);
        uint256 tBalance = t.balanceOf(address(this));
        t.safeTransfer(msg.sender, tBalance);
        tokensOut[i] = tBalance;
    }
}
}

```

Reference: [CurveHelper.sol#L152-L172](#)

However, since the entire balance of the helper contract is sent back to the cellar for each token an attacker could sandwich the rebalance call using this adaptor with ETH positions to send tokens expected to be received by the call directly to the curve helper. This would cause additional tokens to be sent into the cellar and with enough assets cause the rebalance deviation to be exceeded after the rebalance calls are executed, causing rebalancing to revert. Then the attacker could immediately call one of the curve helper functions using a custom pool, and retrieve the tokens they initially sent in.

This exploit could be made to prevent the cellar from executing important calls, but typically there would not be an incentive to attempt to pull this off.

Remediations to Consider

Instead of sending the entire balance of the helper for each token, get the balance before and after the interaction with the pool, ensuring the cellar only receives the tokens from the curve pool, preventing the possibility of griefing.

M-2 Reentrancy guard for CurveHelper

The `CurveAdaptor.sol` makes effort to prevent reentrancy from occurring when interacting with `Curve` pools as there has been known to be issues in the past with `Curve`. However, the `CurveHelper.sol` interacts with curve pools and directly holds assets that could be taken if a custom call is made when executing, up to the slippage tolerance.

Remediations to Consider

Add a reentrancy guard to the deposit and withdraw functions in `CurveHelper.sol`, to prevent the potential for assets to be stolen if reentrancy occurs via a curve pool. Note that this reentrancy guard should use unstructured storage to prevent a cellar delegate calling one of these functions and inadvertently writing to a used storage slot.

↳ Ensure native ETH isn't overridden

TOPIC	STATUS	IMPACT	LIKELIHOOD
Error recovery	Fixed ↗	Low	Low

In `CurveHelper.sol` when depositing liquidity in `addLiquidityETHViaProxy()` the amount of native ETH is tracked to send with the call to the pool.

```
uint256 nativeEthAmount;

// Transfer assets to the adaptor.
for (uint256 i; i < underlyingTokens.length; ++i) {
    if (address(underlyingTokens[i]) == CURVE_ETH) {
        // If token is CURVE_ETH, then approve adaptor to spend native wrapper.
        ERC20(nativeWrapper).safeTransferFrom(msg.sender, address(this), orderedUnderlyingTokenAmounts[i]);
        // Unwrap native.
        IWETH9(nativeWrapper).withdraw(orderedUnderlyingTokenAmounts[i]);
    }
}
```

```

        nativeEthAmount = orderedUnderlyingTokenAmounts[i];
    }
}

```

Reference: [CurveHelper.sol#L87-L98](#)

However, if there happens to be multiple `underlyingTokens` provided that are the `CURVE_ETH` address, then the tracked `nativeEthAmount` will be overridden, sending an improper value to the pool.

Remediations to Consider

Check if the `nativeEthAmount` is non-zero before setting it, and revert if that's the case.

L-2 **WithdrawalRequest with zero price can be solved**

TOPIC	STATUS	IMPACT	LIKELIHOOD
User inputs	Acknowledged	High	Low

The `WithdrawQueue` contract allows users to set specific request parameters to exit ERC4626 positions by relaying complex withdrawal operations to third-party solvers.

Additionally, through the function `isWithdrawRequestValid` it's possible to check whether a specific `userRequest` is valid or not depending on multiple parameters such as `sharesToWithdraw`, `deadline`, its corresponding share allowance, and `executionSharePrice`:

```

function isWithdrawRequestValid(
    ERC4626 share,
    address user,
    WithdrawRequest calldata userRequest
) external view returns (bool) {
    // Validate amount.
    if (userRequest.sharesToWithdraw > share.balanceOf(user)) return false;
}

```

```

    // Validate deadline.
    if (block.timestamp > userRequest.deadline) return false;
    // Validate approval.
    if (share.allowance(user, address(this)) < userRequest.sharesToWithdraw) re
    // Validate sharesToWithdraw is nonzero.
    if (userRequest.sharesToWithdraw == 0) return false;
    // Validate sharesToWithdraw is nonzero.
    if (userRequest.executionSharePrice == 0) return false;

    return true;
}

```

Reference: [WithdrawQueue.sol#L121-138](#)

However, there are no checks in `solve()` to verify whether the `executionSharePrice` of a request is not `0`; only the `inSolve`, `deadline`, and `sharesToWithdraw` are validated:

```

if (request.inSolve) revert WithdrawQueue__UserRepeated(users[i]);
if (block.timestamp > request.deadline) revert WithdrawQueue__RequestDeadlineEx
if (request.sharesToWithdraw == 0) revert WithdrawQueue__NoShares(users[i]);

```

Reference: [WithdrawQueue.sol#L185-187](#)

This could cause users to mistakenly set their share price to `0`, and their shares get drained without anything in return.

It is worth it to note that `updateWithdrawRequest` does not perform any sanity checks on the `userRequest` parameters and is used to add, update, or remove user withdrawal requests.

Remediations to Consider

Consider checking the `executionSharePrice` of each request before solving requests or documenting this potential behavior as a known pitfall.

Missing slippage input constraints

TOPIC	STATUS	QUALITY IMPACT
Input Validation	Fixed 	Low

In `CurveAdaptor.sol` slippage constraints are set in its constructor as immutable values, with a comment suggesting a range on this bound.

```
/**
 * @notice Number between 0.9e4, and 1e4 representing the amount of slippage th
 *         tolerated when entering/exiting a pool.
 *         - 0.90e4: 10% slippage
 *         - 0.95e4: 5% slippage
 */
uint32 public immutable curveSlippage;

constructor(address _nativeWrapper, uint32 _curveSlippage) CurveHelper(_nativeW
    addressThis = payable(address(this));
    curveSlippage = _curveSlippage;
}
```

Reference: [CurveAdaptor.sol#L69-L80](#)

Consider constraining the `_curveSlippage` to be between the values suggested to ensure slippage is setup as intended.


~~Q-2~~ Use `_maxAvailable()` for `withdrawFromBaseRewardPoolAsLPT`

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed 	Low

In `ConvexCurveAdaptor.sol`'s `withdrawFromBaseRewardPoolAsLPT()` function, if the passed in amount is set to `uint256.max` it sets the amount to be the balance of the

cellar. This pattern is typically handled using `baseAdaptor` 's `_maxAvailable()` function. Consider using `_maxAvailable()` to keep with the same pattern followed by the other adaptors.

Q-3 Use curly braces to wrap branches and loops

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed 	High

In Contracts like `CurveAdaptor.sol` , there are branches and for loops that are large statements of code that are not wrapped within curly braces:

```
// Approve pool to spend amounts, and check for max available.
for (uint256 i; i < underlyingTokens.length; ++i)
    if (orderedUnderlyingTokenAmounts[i] > 0) {
        orderedUnderlyingTokenAmounts[i] = _maxAvailable(underlyingTokens[i],
        underlyingTokens[i].safeApprove(pool, orderedUnderlyingTokenAmounts[i]
    }

pool.functionCall(data);
```

Reference: [CurveAdaptor.sol#L222-L229](#)

Although this compiles, it can be confusing to read and is error prone if additional lines are added or edited. Consider wrapping longer segments of code with curly braces to make it more clear what is occurring.

Q-4 Use uint256 for _deadline

TOPIC	STATUS	QUALITY IMPACT
-------	--------	----------------

In `SimpleSlippageRouter.sol` a `_deadline` parameter is used to ensure the calls execution occurs before a set time, is of type `uint64` . Using variable sizes smaller than 32 bytes can increase gas cost, or cause callers to unnecessarily cast the value down to `uint64` .

Consider using `uint256` for `_deadlines` .

Q-5

SimpleSlippageRouter error naming

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed 	Low

In `SimpleSlippageRouter.sol` , the errors follow a naming convention of `SimpleSlippageAdaptor__` .

```
error SimpleSlippageAdaptor__ExpiredDeadline(uint64 deadline);
```

Reference: [SimpleSlippageRouter.sol#L26](#)

Consider keeping with the name of the contract and use `SimpleSlippageRouter__` to preface errors.

Q-6

Use IERC4626 instead of Cellar

TOPIC	STATUS	QUALITY IMPACT
-------	--------	----------------

In `SimpleSlippageRouter.sol`, each function takes in a `Cellar` which is interacted with. Since a `Cellar` shares the same interface and functionality as standard `ERC4626` vaults, using an `IERC4626` interface instead makes it more clear that the contract works for vaults in general.

Use delta share balance for slippage checks

TOPIC	STATUS	GAS SAVINGS
Gas Optimization	Fixed 	High

In `SimpleSlippageRouter.sol`, the functions `deposit()` and `withdraw()` preview the amount of shares received or burned for the provided `_assets` and use this value to compare slippage.

```
function deposit(Cellar _cellar, uint256 _assets, uint256 _minimumShares, uint256 _deadline) public {
    if (block.timestamp > _deadline) revert SimpleSlippageAdaptor__ExpiredDeadline();
    uint256 shares = _cellar.previewDeposit(_assets);
    if (shares < _minimumShares) revert SimpleSlippageAdaptor__DepositMinimumShares();
    ERC20 baseAsset = _cellar.asset();
    baseAsset.safeTransferFrom(msg.sender, address(this), _assets);
    baseAsset.approve(address(_cellar), _assets);
    _cellar.deposit(_assets, msg.sender);
    _revokeExternalApproval(baseAsset, address(_cellar));
}
```

Reference: [SimpleSlippageRouter.sol#L73-L82](#)

The `previewDeposit()` or `previewWithdrawal()` functions are quite gas intensive, especially when called on cellars which don't use a share price oracle. Instead of

previewing, the change in shares of the caller, before and after the deposit or withdraw call can be used to determine slippage with reduced gas cost.

I-1 **ConvexCurveAdaptor** only works for mainnet

TOPIC

Chain compatibility

IMPACT

Informational *

The `ConvexCurveAdaptor.sol` is setup to interact with Ethereum mainnet Convex contracts, since Convex contracts [differ on other chains](#).

I-2 **Use caution when basing price off Curve pools**

TOPIC

Curve Pricing

IMPACT

Informational *

Curve has been known to have multiple exploits and effort has and should be made to reduce the effect of potential exploits from negatively impacting Cellars. In the case of manipulating the valuation of Liquidity tokens: The `Curve2PoolExtension` is pricing is based on the values of the Curve pool, so caution and additional safety measures should be made to ensure pricing this way remains safe. Notably, Cellars that include positions priced using Curve pricing extensions should use the

`ERC4626SharePriceOracle` which has additional safe guards, including a circuit breaker during volatile share price changes, as well as a time weighted average price that is used instead of the latest price if it benefits the protocol. This ensures that price manipulation has a limited effect on extracting value from the Cellar, and requires significant capital and time to do so, which should be caught beforehand. It is also

suggested that positions priced using Curve should be made illiquid and not be a Cellars main asset, to prevent manipulation to over/under evaluate the value of these positions when users deposit/withdraw.

In the case of reentrancy exploits: For each user deposit or withdrawal into Curve positions, checks are made to ensure the Curve pool isn't in a reentered state. This ensures that no reentrancy attacks can be made by users to drain additional assets out of the cellar in case of a reentrancy exploit is found in a Curve pool a Cellar has a position in.

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Sommelier team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.