Audit Report

General Galactic Skyteller V1

Delivered: 2023-04-18

Prepared for General Galactic Corporation by Runtime Verification, Inc.

Summary

Runtime Verification, Inc. has audited the smart contract source code of General Galactic Corporation for Skyteller project. The review was conducted from 2023-04-04 to 2023-04-18.

General Galactic Corporation engaged Runtime Verification in checking the security of their Skyteller project, which allows users to off-ramp their assets to liquidity providers such as banks. In this aspect, users of Skyteller contracts can perform token swap operations where the swapped amount is directed to the liquidity provider's contract defined by the user.

The issues which have been identified can be found in the sections Findings and Informative findings.

Scope

The audited smart contracts are:

- `SkytellerErrors.sol`
- `SkytellerProxyFactory.sol`
- `SkytellerProxy.sol`
- `SkytellerRouterV1.sol`
- `SkytellerSweeperV1.sol`

The audit has focused on the above smart contracts, and has assumed correctness of the libraries and external contracts they make use of. The libraries are widely used and assumed secure and functionally correct.

The review encompassed `generalgalactic/skyteller-solidity-contracts` private code repository. The code was frozen for review at commit 7193fa1adf215412ee5f8bf17640e155e03d8560

The review is limited in scope to consider only contract code. Off-chain and client-side portions of the codebase are *not* in the scope of this engagement.

Assumptions

The audit assumes that all addresses assigned a role must be trusted for as long as they hold that role. Apart from the deployer that is responsible to create and deploy the contracts, an `owner` role (see OpenZeppelin's access control) is present for routers and sweepers who can set important parameters and perform administrative duties. At a time, each router is owned by a single user that can transfer funds to the router and perform `sweep` operations. Sweeper contract, on the other hand, is owned by Galactic who is able to set important parameters of the protocol such as the rate for minimum amount to get from a swap, swap fee, arithmetic precision, etc.

Galactic plans to deploy their contracts behind upgradable proxies. Consequently, the operator of the proxy contract has the capability to replace the implementation contract at any time.

Further, Skyteller protocol benefits from services provided by Chainlink and Uniswap. Skyteller can be negatively affected for the cases where these services are not fully operational.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we met with the Galactic team to provide feedback and suggested development practices and design improvements.

This report describes the **intended** behavior of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Skyteller: Contract Description and Properties

This section describes the Skyteller project at a high level and which invariants we expect it to always respect at the end of a contract interaction.

Overview

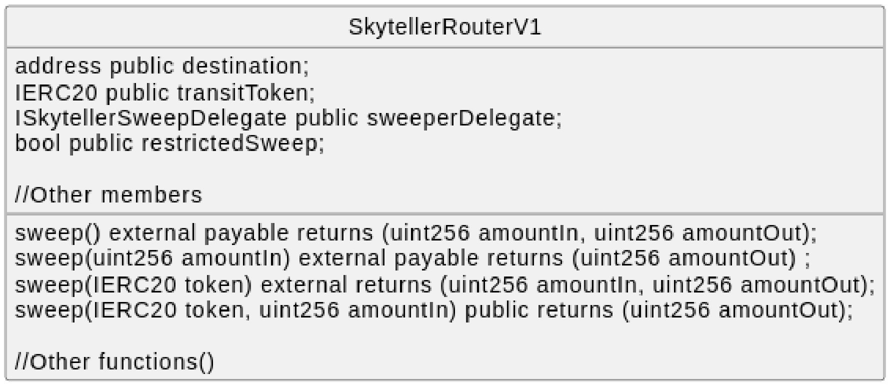
Skyteller project is an off-ramping protocol where users define contracts that belong to a liquidity provider (such as a bank) to swap their crypto-assets to the liquidity provider's accepted asset and obtain off-chain liquidity. Protocol's major functionality is to provide the on-chain services of the off-ramping protocol behind a user interface and perform better user experience for the users unfamiliar with the underlying technologies.

Skyteller protocol contains a number of contracts but the major functionality is contained in two contracts named "`SkytellerSweeper`" and "`SkytellerRouter`".

`SkytellerRouter`

`SkytellerRouter` is designed to be tied to a single customer, potentially going under the KYC process of the liquidity provider. At a time, there exists a 1:1 relation between each router and individual customers, though this is not limited by the protocol. Router contract acts as a service delegate for the SkytellerSweeper where a customer can perform so-called "sweep" operations and direct their off-ramping process to the liquidity provider.

Below, a diagram is presented that states the members of the SkytellerRouter contract mentioned in this section.

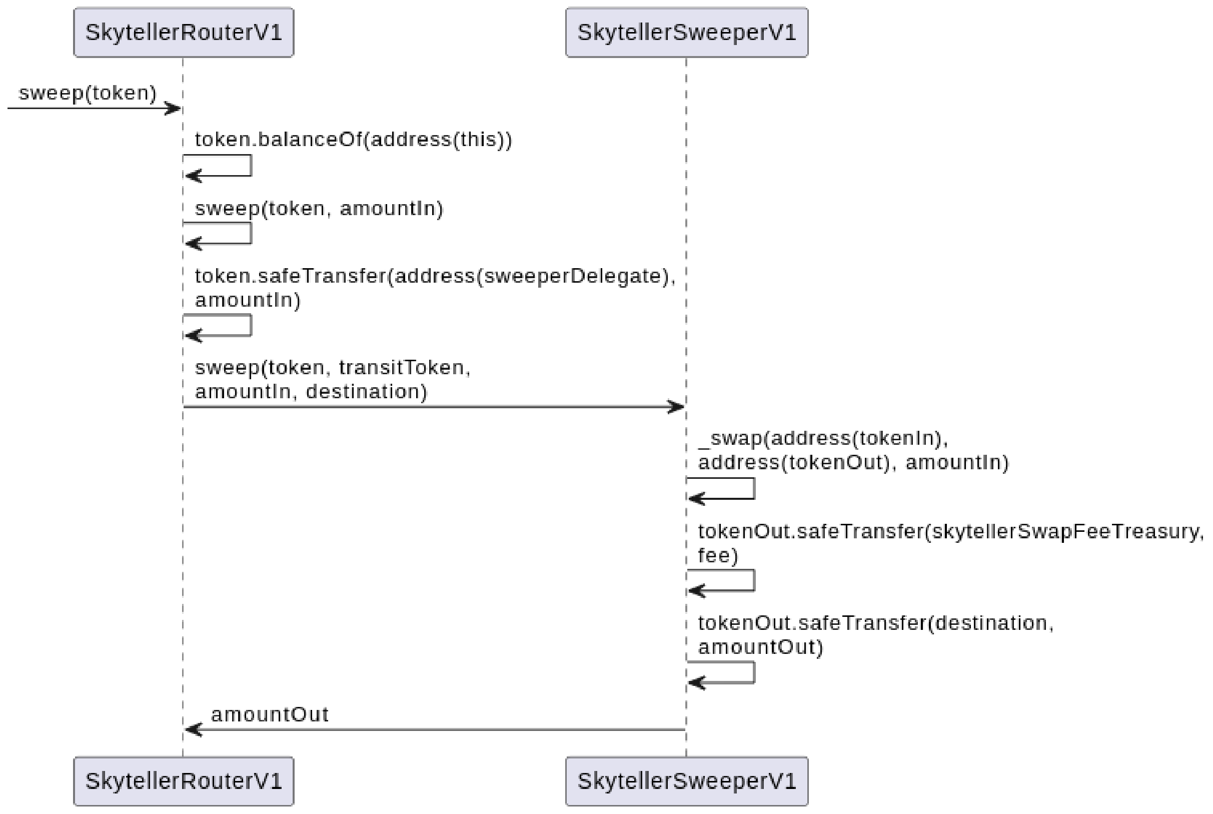


```
SkytellerRouterV1
address public destination;
IERC20 public transitToken;
ISkytellerSweepDelegate public sweeperDelegate;
bool public restrictedSweep;

//Other members
sweep() external payable returns (uint256 amountIn, uint256 amountOut);
sweep(uint256 amountIn) external payable returns (uint256 amountOut) ;
sweep(IERC20 token) external returns (uint256 amountIn, uint256 amountOut);
sweep(IERC20 token, uint256 amountIn) public returns (uint256 amountOut);

//Other functions()
```

`SkytellerRouter` exposes an initialization function and four variants of sweep function. Sweep functions perform necessary operations and checks before performing a sweep over the `SkytellerSweeper`. Implementations of the three sweep variants call the fourth variant where an input ERC20 token and an amount to be sweeped is provided as parameters. The three variants basically accept different combinations of these two parameters and complete the missing parameters accordingly. For instance the sweep variant with no parameters calls the fourth variant with WETH and all of the contract balance.

The main sweep function of the router performs a couple of checks and calls the sweep function of the `SkytellerSweeper` delegate contract. First, it is checked if the router is performing in a special "restricted mode". If restricted mode is active then the sweep function is only available to be called by the owner of the contract. Secondly, it is checked if the token to be sweeped is the same as the "`transitToken`" which is an intermediate token used by the `SkytellerSweeper`. If the token to be swept is the `transitToken` then the tokens are directly transferred to the liquidity provider without performing a sweep by `SkytellerSweeper`.

Following diagram summarizes the interactions between the router and the sweeper for a typical token sweep operation.

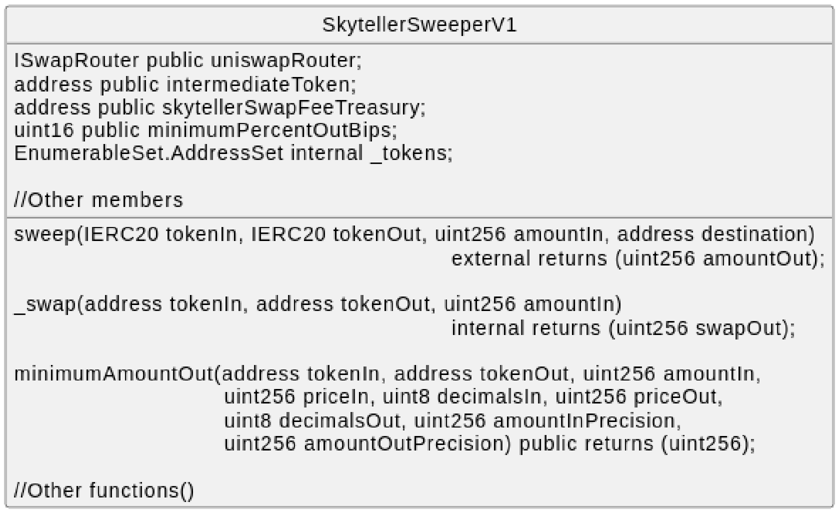

SkytellerRouterV1
SkytellerSweeperV1
sweep(token)
token.balanceOf(address(this))
sweep(token, amountIn)
token.safeTransfer(address(sweeperDelegate), amountIn)
sweep(token, transitToken, amountIn, destination)
_swap(address(tokenIn), address(tokenOut), amountIn)
tokenOut.safeTransfer(skytellerSwapFeeTreasury, fee)
tokenOut.safeTransfer(destination, amountOut)
amountOut
SkytellerRouterV1
SkytellerSweeperV1

Other than the main sweep functionality, router contract also exposes owner-only functions to switch the restricted mode, set destination address for the sweeps, set sweeper delegate contract address and withdraw tokens from the router balance. Additionally, router owners are allowed to destruct the contract via a `selfDestruct` call.

SkytellerSweeper

`SkytellerSweeper` is operated by a party that offers sweep functionality by performing appropriate swap operations over Uniswap. Initially, Galactic intends to operate the sweeper. Sweeper exposes initialization and many different view functions. The main functionality is exposed by the `sweep` function. `sweep` function is mainly called by the routers but it is also open to be called by the so called “keepers” that may perform sweeps independent from any router.

Below, a diagram is presented that states the members of the `SkytellerSweeper` contract mentioned in this section.



```
SkytellerSweeperV1

ISwapRouter public uniswapRouter;
address public intermediateToken;
address public skytellerSwapFeeTreasury;
uint16 public minimumPercentOutBips;
EnumerableSet.AddressSet internal _tokens;

//Other members

sweep(IERC20 tokenIn, IERC20 tokenOut, uint256 amountIn, address destination)
                                        external returns (uint256 amountOut);

_swap(address tokenIn, address tokenOut, uint256 amountIn)
                                        internal returns (uint256 swapOut);

minimumAmountOut(address tokenIn, address tokenOut, uint256 amountIn,
                 uint256 priceIn, uint8 decimalsIn, uint256 priceOut,
                 uint8 decimalsOut, uint256 amountInPrecision,
                 uint256 amountOutPrecision) public returns (uint256);

//Other functions()
```

Sweeper’s `sweep` function accepts an input-output token pair together with an amount to be swept and a target destination address where the output tokens from the sweep operation is directed to. Sweep operation requires the necessary sweeping amount to be transferred to the `SkytellerSweeper`‘s balance before the sweep function. This transfer is provided by the router within the same transaction before the sweep call to the sweeper. Keepers on the other hand should perform this transfer before calling sweep.

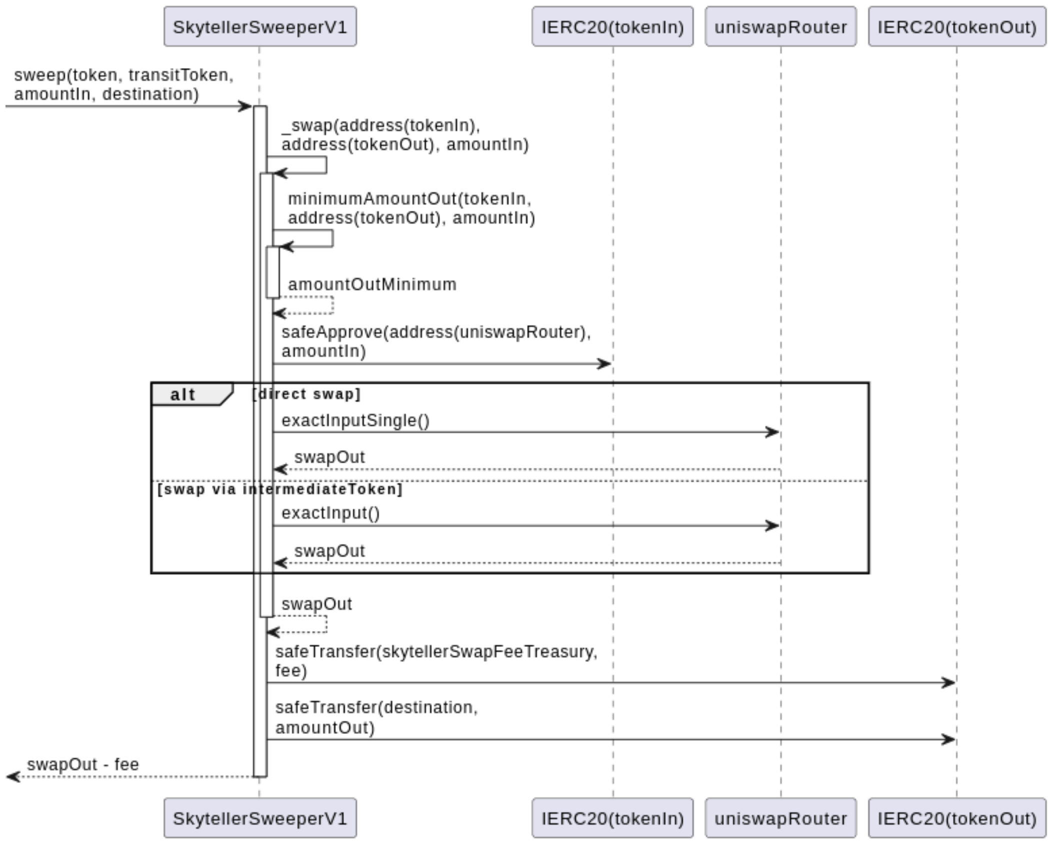
Once the appropriate preconditions are met, sweep operation performs a swap over Uniswap and transfers the tokens obtained by the swap to the provided destination address. During the swap, different paths of execution can be performed according to the parameters provided to the sweeper.

A very basic case is when the sweep function is called with the same input-output token pair. For this case the amount is directly transferred without a swap over Uniswap. If a direct swap is possible over Uniswap the swap is performed directly and if a direct swap is not possible

`SkytellerSweeper` uses an intermediate token (set at initialization) to perform the swap via two swaps.

Fees applied during the sweep operations are the swap fee by Uniswap and the fee collected by Skyteller. Skyteller uses a global percentage to designate the minimum amount to be received after a swap over Uniswap. Before a swap takes place, this percentage is used in combination with prices from an external oracle (such as Chainlink) to calculate the minimum amount expected from a swap and used accordingly.

Following diagram summarizes the interactions within the sweeper for a typical sweep operation that contains swaps over Uniswap.


SkytellerSweeperV1
IERC20(tokenIn)
uniswapRouter
IERC20(tokenOut)
sweep(token, transitToken, amountIn, destination)
_swap(address(tokenIn), address(tokenOut), amountIn)
minimumAmountOut(tokenIn, address(tokenOut), amountIn)
amountOutMinimum
safeApprove(address(uniswapRouter), amountIn)
alt
[direct swap]
exactInputSingle()
swapOut
[swap via intermediateToken]
exactInput()
swapOut
swapOut
safeTransfer(skytellerSwapFeeTreasury, fee)
safeTransfer(destination, amountOut)
swapOut - fee

Together with the main sweep functionality, sweeper contract also exposes owner-only functions to pause the sweeps, set intermediate token, set amount and treasury for Skyteller fees, set the expected minimum swap percentage and set the price feed and Uniswap router addresses. Additionally, sweeper owners can transfer tokens from the contract by using the provided rescue functions in case unexpected or residual token balance resides in the contract's balance.

Properties

Several important properties should be satisfied by the contract at all times. Here will only be listed properties related to the contracts under the audit. Properties related to the use of external libraries are assumed to hold.

In the following, we list several properties that the contract should satisfy. These are not the only properties, but they are fundamental for the correctness of the protocol and, as such, deserve special attention.

Properties related to SkytellerRouter:

P1 A sweep operation should transfer a desired amount (in ETH or in an ERC20 compatible-token) less than or equal to the contract's balance to a destination address in terms of a desired ERC-20 compatible token (`transitToken`)

P2 When the balance being swept is the same as the transit token, tokens are directly transferred without swapping

P3 When the token being swept is the same as the transit token, the transferred amount (including transfer fees, if any) should be exactly the same as the input amount

Properties related to SkytellerSweeper:

We assume the system is closed for this set of properties, i.e., transfers outside the system are not considered.

P4 When a new token is added to the supported list either (USD/ETH) of its price feed address is set.

P5 When an `owner` removes a token from its supported list, related price feeds are also removed.

P6 Pool fee for a token pair should be in the percentage range [0,100]

P7 Minimum amount that is required to be obtained from a sweep should be in the percentage range [0,100]

P8 Sweep fee collected by Skyteller should be in the percentage range [0,100]

P9 For a sweep to take place it is necessary to have a Uniswap route either directly between the input-output token pair or a route between both the input-transit token pair and transit-output token pair.

P10 For a sweep to take place it is necessary for the corresponding price oracles to be set for either USD or ETH token prices for both the input and the output tokens

P11 During a sweep operation, if the input token is the same as the output token, the transferred amount (including transfer fees, if any) should exactly be the same as the input amount

P12 The sweep operation should be completed within the same block of the transaction

P13 For a sweep operation, in terms of payment tokens (`tokenIn`), the balance of the sweeper should be the same before and after the sweep if the swap goes through.

P14 For a sweep operation, in terms of destination tokens (`tokenOut`), the balance of the sweeper should be the same before and after the sweep if the swap goes through.

Finally, properties related with access control can be listed as follows:

P15 At any given time, each router contract is affiliated with exactly a single user as the `owner`

P16 The `owner`, and the `owner` only, can perform the following operations for the router: set the restricted mode, set the destination and the transition token, set sweeper delegate contract, withdraw tokens/ETH, `selfDestruct` the contract

P17 At any given time, each sweeper contract is affiliated with exactly a single user as the `owner`

P18 The `owner`, and the `owner` only, can perform the following operations for the sweeper: pause/unpause the sweeps, set the treasury address, set the swap fee, set the minimum percentage of the provided amount for the sweep operation to complete, set the uniswap router, set pool fees and routes, set price feeds and accepted tokens, remove the token form the supported tokens list, rescue (by withdrawing) tokens from the sweeper contract

Findings

A01: Sweeps are vulnerable to price manipulation attacks, natural slippage and MEV

[Severity: High | Difficulty: High | Category: Security]

The Sweeper contract uses price oracles to mitigate the risk of price manipulation attacks. Client uses Chainlink as a price Oracle because it provides prices gathered from multiple sources resulting in more robust exchange rates over time. The mitigation makes a price manipulation attack more difficult because an attacker needs to manipulate the spot price of Uniswap and the feed price of Chainlink. However, the mitigation is insufficient to prevent price manipulation attacks altogether. The scenario below demonstrates such a price manipulation attack.

Further, there exists a timespan between the submission of the sweep transaction and its execution. Consequently, there can be a difference in the price at submission and execution time, commonly called slippage. The slippage can be amplified by miners to their own benefit.

Scenario

Price Manipulation Attack

1. Alice wants to sell 100 TokenA for TokenB
2. Uniswap and Chainlink report the same price of 2 TokenB per 1 TokenA
3. Alice hopes to buy 200 TokenB and initiates the sweep
4. An attacker sees Alice's transaction on the mempool and attempts to front-run her
5. Assume that the attacker manages to manipulate the price of Uniswap and Chainlink
6. The new price reported by Uniswap and Chainlink is 1 TokenB per TokenA
7. When Alice's tx is processed, she receives only 100 TokenB.

MEV opportunities caused by slippage

Let's consider the scenario for a possible slippage, defined below:

1. Alice wants to sell 100 TokenA for TokenB
2. Uniswap and Chainlink report the same price of 2 TokenB per 1 TokenA
3. Alice hopes to buy 200 TokenB and initiates the sweep
4. Alice submits her transaction with very low gas fees.
5. Because of the low gas price, it takes some time before a miner includes Alice's transaction.
6. The new price reported by Uniswap and Chainlink is 1 TokenB per TokenA

7. When Alice's tx is processed, she receives only 100 TokenB.

This scenario incentivizes a miner, who is also selling TokenB for TokenA on Uniswap, to take advantage of the slippage.

1. Alice wants to sell 100 TokenA for TokenB
2. Uniswap and Chainlink report the same price of 2 TokenB per 1 TokenA
3. Alice hopes to buy 200 TokenB and initiates the sweep
4. Alice submits her transaction
5. A miner sees Alice's transaction on the mempool.
6. The miner now waits until the next round from Chainlink enters the mempool.
7. If the price reported by Chainlink goes up, the miner will order Alice's tx after Chainlink's tx.
8. The miner ultimately made more profit to the disadvantage of Alice.

Scenarios intentionally use extreme price changes and/or manipulations for demonstration purposes.

Recommendation

When initiated by the owner of the router, the user interface should allow the user to specify the minimum amount he wants to get out of the sweep. The sweep function should take this minimum amount as a parameter and revert if the actual amount is lower. The user interface should expose this parameter to the end user. This mitigation ensures that the trade is only executed when the user agrees with the pricing.

When using oracles to protect the interest of the router owners it is not advised to rely only on the last round from Chainlink. Instead, the best price from the previous X rounds/hours can be inspected and used. It might be considered to add support for market orders, where router owners can specify their minimum acceptable price, and the trade is immediately executed via Uniswap if possible.

Another option to consider may be to add support for limit orders, where router owners can specify their minimum acceptable price, and the trade is executed by a keeper via Uniswap if the terms are satisfied. Moreover, the router can also provide functionality similar to a derivative-trading market: Owners sell options/futures to potential buyers. For this alternative, in comparison with the limit order functionality, the main difference is that the counterparty becomes a buyer instead of Uniswap.

Many additional design alternatives may be considered to mitigate this issue. Differentiating between the router owners and keepers and introducing a fee to incentivise keepers can be considered. The fee should be sufficiently high to counteract against the MEV potential, and it should not be higher than the gas costs that the owner would pay if he swept directly. Though, it is hard to design a robust fee mechanism that can guarantee this property even in volatile market situations. Further, notice that this incentive would not protect against griefing attacks.

Another complementary solution would be giving the router's owner control of the risk parameters. Providing order book functionality by the router can be another design alternative. Such design alternatives may be implemented as different router variants that support additional functionality such as auction houses where buyers can bid on non-USDC assets from router owners.

In summary, providing further financial incentives to the users of the contract and evolving the current design to support features that enable such incentives may reduce the risks mentioned in this finding.

Status

Client is aware of pricing as an attack surface both via market manipulation and oracle exploits. In addition to current mitigation by using Chainlink oracle feeds, the client is looking into possible approaches to evolve the protocol to mitigate possible attacks. In future development iterations, finding will be handled with a favorable approach considering the price and benefits of additional security. The reason it is currently not implemented is to reduce the overall complexity of the protocol and to ensure gas efficiency.

A02: Inconsistencies and calculation errors may arise for `amountOut` value

[Severity: Medium | Difficulty: Low | Category: Functional Correctness]

An inconsistency between the actual amount of tokens spend and the reported amountOut value can exist for the following two scenarios:

Fee-on-transfer tokens

If the supplied token to the `SkytellerRouterV1.sweep()` is the transit token, the function sets `amountOut = amountIn`. This does not hold for fee-on-transfer tokens.

```
if (token == transitToken) {
      token.safeTransfer(destination, amountIn);
      amountOut = amountIn;
```

Reports by a malicious delegate

Routers emit a `SkytellerRouterSweep` event on every successful sweep action. The event contains an `amountOut` field. This field reflects what the delegate returned as `amountOut`. However, this can be different from the correct value.

In particular, a malicious delegate could keep all the tokens and report a fake value to the router. The fraud can be hard to detect for the end user because he cannot rely on the integrity of the event log.

Recommendation

Routers should not blindly include the reported `amountOut` values from delegates. Instead, the router should compute the actual balance difference.

```
uint256 destinationBalanceBefore = token.balanceOf(destination);
// ...
amountOut = token.balanceOf(destination) - destinationBalanceBefore;
```

Status

The client is fully aware of this and going to handle the situation during their upcoming development iterations. Client is also planning to provide documentation that emphasizes the importance of sweep delegate security for the protocol users to avoid such risks.

A03: Contracts depend on off-chain security

[Severity: Medium | Difficulty: Low | Category: Input Validation]

The Solidity contracts could benefit from more input validations. The contract's behavior is unclear when a user provides invalid or unintended parameters. In many cases, an invalid input will put the contract into a temporary DoS state. Some examples can be seen in the initializer below:

```
function initialize(
     address _weth9,
     address _owner,
     address _destination,
     address _transitToken,
     address _sweeperDelegate
) external reinitializer(VERSION) {
     __Ownable2Step_init_unchained();
     __UUPSUpgradeable_init_unchained();
     _transferOwnership(_owner);
     weth9 = IWETH9(_weth9);
     destination = _destination;
     transitToken = IERC20(_transitToken);
     sweeperDelegate = ISkytellerSweepDelegate(_sweeperDelegate);
}
```

- Example: `_weth9 == address(0)`
  The `sweep()` and `sweep(uint256)` functions stop working.
- Example: `destination == address(0)`
  Sweeps can lead to the permanent loss of assets.
- Example: `owner == address(0)`
  All functions that require elevated permissions from the owner are not operable.

Another example may be the tokens that revert on zero transfers. Consider the following code example:

```
fee = skytellerSwapFee * swapOut / FEE_PRECISION;
amountOut = swapOut - fee;
tokenOut.safeTransfer(skytellerSwapFeeTreasury, fee);
```

Notice that some tokens will revert on `transfer(receiver, 0)` calls. Consequently, the sweep action can fail if `skytellerSwapFeeTreasury = 0`. This can happen due to rounding errors for very small amounts or when `skytellerSwapFee = 0`.

It should be emphasized that it is not always the user's fault or intention to provide invalid parameters. User interfaces may fail to provide the correct parameters. The zero address is a prime example: Many programming languages and APIs silently fall back to neutral elements when encountering an error scenario. Imagine there is an undiscovered bug in your UI or backend that leads to padding a transaction's calldata with zero bytes. The situation can go unnoticed and propagate into your on-chain code, where it can cause severe damage, e.g., permanent loss of assets.

Of course, any other unused address can be equally problematic, but it is less likely that a silent failure propagates with a non-zero value. A sanity check for the zero address can go a long way and avoid many unintended transactions.

For these reasons, it has become a security best practice to not depend on the security of the user interface and backend too much but to perform on-chain input validation. It is recommended to follow this best practice and add input validations to all externally callable functions, particularly state-modifying functions like initializers and setters.

Recommendation

Performing necessary validations are vital to mitigate the vulnerability identified by the finding. Furthermore, using the functions provided by libraries following the recommended best practices would also help mitigate this problem. An example to this recommendation is using `transferOwnership()` function in `initialize()` function(s) instead of `_transferOwnership()` would result in performing necessary checks.

Status

The client is aware of the possible ramifications and plans to handle the appropriate checks in contract initialization and setters during their upcoming development iterations.

A04: `Routers` are destructible

[Severity: High | Difficulty: High | Category: Security]

The owner of a router is allowed to destroy it at any time without any further sanity checks. Destroying a router can lead to permanent loss of assets. When a contract is destroyed the remaining ETH balance is sent to the owner but all remaining assets (ERC20, ERC721, ERC1155,..) are permanently lost.

Further, `selfdestruct` is about to be deprecated (see EIP-6780). In the future, `selfdestruct` will not remove storage and contract code. It will continue to send the account balance to the caller. Even if these proposals are not accepted, changes to the semantics of `selfdestruct` are very likely to be performed in the future. Avoiding this feature is to be advised until the debate has settled.

Recommendation

`selfdestruct` feature should be considered for removal form the contracts. If that is not possible or not wanted, it may be considered to be transformed into a two-step process to avoid accidental destruction and permanent loss of assets. For example, a `proposeSelfdestruct(uint256 deadline)` function may be exposed together with `acceptSelfdestruct()` and `revokeSelfdestruct()` functions. A user wishing to destroy the contract must call `proposeSelfdestruct(deadline)` followed by `acceptSelfdestruct()` within the given deadline. Furthermore, these calls should not be possible within the same transaction. A change in the destruction intention in the meantime can result in waiting for the deadline to expire, or explicitly revoking the proposal.

Status

The client is aware of the possible outcomes of the changes in the `selfdestruct` feature, and intends to remove the `selfDestruct()` function in future upgrades.

A05: Push based fund transfers to `Sweeper`

[Severity: High | Difficulty: Medium | Category: Functional Correctness]

Token transfers to SkytellerSweeper from routers and other contracts are performed in a push-based style. For the routers, in transition a `sweep()` call is performed which may be dangerous in case of a re-entrancy. For keeper contracts it becomes possible to sweep another user's assets.

The router delegates sweep in two phases:

1. The router sends the asset to the delegate
2. The router invokes the delegate's sweep function

```
token.safeTransfer(address(sweeperDelegate), amountIn);
amountOut = sweeperDelegate.sweep(token, transitToken, amountIn,
destination);
```

The first action performs an untrusted external call. The target of the external call, and all transitively called contracts, can re-enter into the delegate contract to steal the user's assets.

Scenario

Transfers from Routers

1. Alice calls `Router.sweep(token, amountIn)` with `token != transitToken`.
2. Router transfers the assets to the sweeperDelegate.
3. Assume that the untrusted transfer calls into a malicious contract.
4. The malicious contract re-enters into the delegate: `sweeperDelegate.sweep(token, transitToken, amountToSteal, Bob)`
5. Alice's assets in the Sweeper will be transferred to Bob.
6. After the re-entrant call returns, the router eventually calls `sweeperDelegate` again.
7. If enough assets are left in the Sweeper to pay the Skyteller protocol fee, the transaction will be successful.

The difficulty of this attack mostly depends on the ability of the attacker to inject a malicious call during step 3.

Transfers by Keepers

1. Alice transfers some funds to the sweeper address e.g. `token.transfer(address(sweeper), amount)`
2. Bob calls `sweep()` with some amount less than or equal to Alice's amount and a custom destination e.g. `sweeper.sweep(paymentToken, transitToken, amount, address(bob))`

Recommendation

It may be considered to lock the `sweeperDelegate.sweep` function before sending it user assets. The function can be unlocked once the sweep is completed and all assets have been returned.

Using pull based token transfers would also decrease such (and alike) risks.

Additionally, keeping per-user assets transferred to sweeper can be developed to control such cross-user asset sweeping issues.

Status

The client is fully aware of this and going to handle the situation during their upcoming development iterations. They plan to develop pull based transfers and add reentrancy locks to mitigate potential attacks.

A06: Oracle data may be outdated

[Severity: High | Difficulty: High | Category: Security]

Chainlink oracles are not necessarily always up-to-date. For instance, Chainlink has paused oracles in the past because of abnormal market situations (see Blizz finance post mortem for an example). Hence, it is recommended to check the `updatedAt` timestamp whenever a price is obtained from an Oracle. If this value is too old, appropriate action should be taken.

```
(, int256 answer,,,) = AggregatorV3Interface(feed).latestRoundData();
```

Recommendation

Either one of the following actions can be taken:

1. Consider removing the dependency on the Chainlink oracle entirely. If it is only used to derive the lower limit of what a user accepts to get out of a trade, then it is safer to let the user specify his lower bound directly.
2. If (1) is impossible or not wanted, consider reverting if the `updatedAt` timestamp is too old.

Status

The client is fully aware of this and plans to handle the situation by checking the `updatedAt` timestamp and revert accordingly.

Informative findings

B01: Mitigation from edge-cases in arithmetic operations

[Severity: - | Difficulty: - | Category: Informative]

Reverts caused by arithmetic overflow might occur for the following cases where potentially large integers are multiplicated:

```
return (priceIn * PRICE_DERIVATION_PRECISION / priceOut,
        PRICE_DERIVATION_DECIMALS);
```

```
priceOut *= 10 ** (decimalsIn - decimalsOut);

priceIn *= 10 ** (decimalsOut - decimalsIn);
```

```
return
   (minimumPercentOutBips * amountOutPrecision * derivedPrice * amountIn) /
         (BIPS_PRECISION * amountInPrecision * 10 ** derivedPriceDecimals);
```

Especially for the third example, four large integers are being multiplied that might result in an integer overflow.

Moreover, there exists three points of division where proper rounding may be revised.

Minimum amount calculation:

```
return
   (minimumPercentOutBips * amountOutPrecision * derivedPrice * amountIn) /
         (BIPS_PRECISION * amountInPrecision * 10 ** derivedPriceDecimals);
```

Rounding down is used when calculating the minimum amount expected to be swept. Rounding up should be considered since users experience less than expected minimum amount when round down. If rounded up, the users always get more than expected minimum amount.

Skyteller fee calculation

```
fee = skytellerSwapFee * swapOut / FEE_PRECISION;
```

Rounding down is used when calculating sweep fee. Rounding up should be considered since the treasury gets less than expected fee when round down. If rounded up, the treasury always gets more than the expected fee.

Recommendation

Add prominent documentation that clearly states the overflow-safe operation ranges for the critical values of Skyteller's protocol. An alternative approach may be to re-arrange the terms in the mentioned calculations at loss for precision.

Also consider using rounding up during `minimumAmountOut` and `skytellerSwapFee` calculations.

Status

The client is aware of this and they plan to consider the severity of the possible errors and handle the situation accordingly if necessary.

B02: Constructors and Intializers should emit events

[Severity: - | Difficulty: - | Category: Informative]

No event is being emitted during the construction and initialization of the contracts in the project. This situation makes it hard to monitor and track the statuses of the contracts.

Recommendation

It is recommended to emit events from all externally callable, state-changing functions. This includes constructors and initializers. An exhaustive event log makes it easier to monitor contract interactions.

Status

The client is aware of this and they plan to consider the necessity and handle the situation accordingly if desired.

B03: Redundant code in `SkytellerSweeperV1.derivePrice()`

[Severity: - | Difficulty: - | Category: Informative]

The variable `decimals` that is also used as the return value from the `derivePrice` either is shadowed by the actual return value or calculated redundantly for some execution branches.

```
if (decimalsIn > decimalsOut) {
   decimals = decimalsIn;
   priceOut *= 10 ** (decimalsIn - decimalsOut);
} else if (decimalsOut > decimalsIn) {
   decimals = decimalsOut;
   priceIn *= 10 ** (decimalsOut - decimalsIn);
} else {
   decimals = decimalsIn;
}
return (priceIn * PRICE_DERIVATION_PRECISION / priceOut,
                               PRICE_DERIVATION_DECIMALS);
```

For the code snippet above, `PRICE_DERIVATION_DECIMALS` is a constant value, returned from the function which was previously declared as `decimals`. According to the contract logic `PRICE_DERIVATION_DECIMALS` is the correct value which makes the last else redundant.

Recommendation

Function should be modified considering the desired behavior.

Status

The client is aware of this and the situation is going to be addressed in future iterations.