Schwap Security Review

Version 0.2

July 3, 2023

Conducted by:
Georgi Georgiev (Gogo), Independent Security Researcher
deadrosesxyz, Independent Security Researcher

Audited through the **Hyacinth** platform.

Table of Contents

1 About Gogo and deadrosesxyz

Georgi Georgiev, known as Gogo, and deadrosesxyz are independent security researchers specializing in Solidity smart contract auditing and bug hunting. Having conducted numerous solo and team smart contract security reviews, they always strive to deliver top-quality security auditing services. For security consulting, you can contact them on Twitter, Telegram, or Discord - *@gogotheauditor* & *@deadrosesxyz*.

2 Disclaimer

Audits are a time, resource and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Schwap
Repository	https://github.com/0xSchwap/schwap-contracts
Commit hash	527a5f685f94eab8704a0c328005b6bef80ece0f
Documentation	https://docs.schwap.app
Methods	Manual review

Scope

src/SCH.sol
src/SchwapMarket.sol
src/SimpleMarket.sol
src/UniswapSimplePriceOracle.sol
src/interfaces/

Issues Found

Critical risk	1
High risk	1
Medium risk	5
Low risk	9

5 Findings

5.1 Critical severity

5.1.1 An adversary can set high minimum sell amount and close all active offers

Severity: *Critical Risk*

Context: SchwapMarket.sol#L189-#L200, UniswapSimplePriceOracle.sol#L13-#L16

Description: To prevent the market from being flooded with offers of little value, the Schwap market has implemented a minimum sell amount for each market. This amount is calculated based on the price of a default `dustToken` asset which is set upon deployment (WETH). The problem is that the UniswapV2 oracle that is used to calculate the `dust` value in the other tokens, simply uses the reserves of the corresponding token pool in UniswapV2:

```
function getPriceFor(address tokenA, address tokenB, uint256 tokenAAmt) public view
    returns (uint256 dust) {
    (uint reserve0, uint reserve1) = UniswapV2Library.getReserves(uniswapFactory,
        tokenA, tokenB);
    dust = UniswapV2Library.getAmountOut(tokenAAmt, reserve0, reserve1);
}
```

Furthermore, the `setMinSell` function lacks access control, allowing anyone to call it at any time.

```
function setMinSell(IERC20 pay_gem) public {
    require(msg.sender == tx.origin, "No indirect calls please");
    require(address(pay_gem) != dustToken, "Can't set dust for the dustToken");

    uint256 dust = PriceOracleLike(priceOracle).getPriceFor(dustToken, address(
        pay_gem), dustLimit);

    _setMinSell(pay_gem, dust);
}
```

This creates an attack vector where a malicious user can manipulate the reserve of the underlying UniswapV2 pool using a flash loan. They can then increase the `_dust` value for any token to a level high enough to cancel any/all offers using the public `cancel` method which performs the following checks:

```
modifier can_cancel(uint id) override {
    require(isActive(id), "Offer was deleted or taken, or never existed.");

    require(
        msg.sender == getOwner(id) || offers[id].pay_amt < _dust[address(offers[id].
            pay_gem)], // @audit An attacker can manipulate the `_dust` amount.
        "Offer can not be cancelled because user is not owner nor a dust one."
    );
    _;
}
```

Recommendation: Implement a TWAP oracle, such as an UniswapV3 oracle.

Resolution: Unresolved.

5.2 High severity

5.2.1 UniswapV2 library incorrectly casts types to retrieve factory address

Severity: *High Risk*

Context: UniswapLibrary.sol#L36

Description: Although the UniswapV2 library was out of scope for this audit, we still reviewed it and noticed a vulnerability introduced when the Schwap team changed the type castings in order to update the solidity compiler version from the original >=0.5.0 to ˆ0.8.13.

```
-    pair = address(uint(keccak256(abi.encodePacked(
+    pair = address(uint160(bytes20(keccak256(abi.encodePacked(
```

Since solidity version `0.8.0`, there are new restrictions on explicit type conversions. This means that casting from type `uint256` to type `address` is no longer possible unless the `uint256` variable is first casted to another 20-byte type such as `uint160`. The Schwap team decided to mitigate this by first casting the result of `keccak256` (which is of type `bytes32`) to `bytes20` and then moving on to casting to `address`.

However, the correct way to perform the above casting should be as follows:

```
-    pair = address(uint(keccak256(abi.encodePacked(
+    pair = address(uint160(uint256(keccak256(abi.encodePacked(
```

The problem with the first version is that when the type cast is done using a `bytesNN` type, only the most significant (left-most) bytes are saved in the new variable.

Consider the following example, where the actual factory address is represented by `0xaaa...aaa`:

```
wrong:      address(uint160(bytes20(0xfff...aaa)));
correct:    address(uint160(uint256(0xfff...aaa)));
```

The first line would extract the 20 left-most bytes (40 hex characters) when casting to `bytes20`, so the result would be `0xffffffffffffffffffffffffaaaaaaaaaaaaaaaa`.

The first line would extract the 20 right-most bytes (40 hex characters) when casting from `uint256` to `uint160`, so the result would be `0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`.

Recommendation: Change #L36 in UniswapLibrary.sol to the following:

```
-    pair = address(uint160(bytes20(keccak256(abi.encodePacked(
+    pair = address(uint160(uint256(keccak256(abi.encodePacked(
```

Resolution: Unresolved.

5.3 Medium severity

5.3.1 _matcho can revert due to Out-Of-Gas error if there are too many offers to execute

Severity: *Medium Risk*

Context: SchwapMarket.sol#L471-#L497

Description: When creating an offer, if there are too many offers to be executed in the `_matcho` function, it can result in the transaction reverting due to an Out-of-Gas exception.

Recommendation: Implement a mechanism to limit the number of offers that can be executed within the `_matcho` function. This can be done by setting a predefined limit or allowing users to specify a limit when creating an offer.

Resolution: Unresolved.

5.3.2 User can create the best offer in a market they're blacklisted in and heavily limit project's functionalities

Severity: *Medium Risk*

Context: src/SchwapMarket.sol

Description: Many ERC-20 tokens have a blacklisting feature. If a user is blacklisted, they can disrupt the proper execution of functions such as `_matcho()`, `sellAllAmount()`, `buyAllAmount()`, `getBuyAmount()`, `getPayAmount()`, and other functions that rely on them.

For example, let's consider a user who is blacklisted in the USDT market. This user can create the best offer for every market where they set `buy_gem` to USDT. When another user attempts to execute the offer, it will revert since the user is blacklisted. As the offer is the best one, this means that `_matcho()`, `sellAllAmount()`, and `buyAllAmount()` will always revert, and `getBuyAmount()` and `getPayAmount()` will provide false information as the offer cannot be executed.

Recommendation: Implement a pull-over-push method for offer withdrawals. Upon filling an offer, store the sent `buy_gem` tokens in the contract and allow the offer maker to withdraw their assets in a separate transaction.

Resolution: Unresolved.

5.3.3 There should be a way to change the default dust token

Severity: *Medium Risk*

Context: SchwapMarket.sol

Description: Currently, the minimum sell amount for the dust token cannot be changed. This limitation poses a problem when the price of the token experiences a significant change. In such cases, users are required to make offers that may be too high, which restricts the availability of the protocol.

For example, let's consider the current dust token as WETH, and the dust limit is set to 0.005 WETH at a current price of $2000. This means that users must trade for at least $10. However, if the price of WETH increases over time (e.g. after 3-4 years) to $200k, the dust limit will be equivalent to $1000, which is unreasonably high.

Recommendation: Consider implementing a method which allows the owner of the contract to update `_dust[dustToken]`.

Resolution: Unresolved.

5.3.4 An adversary can grief users who attempt to buy the entirety of an offer for very little cost

Severity: *Medium Risk*

Context: SimpleMarket.sol#L96-#L100

Description: When a user tries to fill an offer completely, an adversary can front-run them and purchase 1 wei, causing the innocent user's transaction to revert due to the following lines of code:

```
if (quantity == 0 || spend == 0 ||
    quantity > _offer.pay_amt || spend > _offer.buy_amt)
{
    return false;
}
```

Recommendation: If a user wants to fill an offer completely, allow them to pass `quantity` of type(uint256).max. Add the following line:

```
if (quantity == type(uint256).max) quantity = _offer.pay_amt;
```

Resolution: Unresolved.

5.3.5 Offer fills, cancels and inserts can revert unexpectedly causing temporary DoS

Severity: *Medium Risk*

Context: SchwapMarket.sol#L616-L619

Description: When a user creates an order in the SchwapMarket, it is initially saved in the "unsorted list" represented by the `_near` mapping and the `_head` variable.

Then, when the order maker wants to cancel the order or a taker wants to fill it, the `_hide` function needs to iterate through the below while loop. If the `id` to be hidden is too far away from the `_head` , this method can revert due to the excessive number of iterations, effectively causing a temporary Denial-of-Service (DoS) situation until higher IDs are inserted.

```
uint256 uid = _head;
...
while (uid > 0 && uid != id) {  // @audit if the ‘id‘ to hide is too far away from
    the ‘_head‘, this method will revert
    pre = uid;
    uid = _near[uid];
}
```

Recommendation: A potential fix is to allow users to pass the value of `uid` similarly to how `_findpos` works.

Resolution: Unresolved.

5.4 Low severity

5.4.1 Using block.number to measure time may lead to unexpected results on different chains

Severity: *Low Risk*

Context: SchwapMarket.sol#L178

Description: In case the project is deployed on other network different from Ethereum mainnet, using `block.number` is an unreliable way to measure time, as

1. Different chains have different time between blocks
2. Most chains do not have a fixed, exact time between blocks.

Recommendation: To ensure accurate time tracking, it is recommended to use `block.timestamp` instead.

Resolution: Unresolved.

5.4.2 Usage of floating pragma may lead to incompatibility with other chains

Severity: *Low Risk*

Context: src

Description: When deploying the project on other chains, compatibility issues may arise due to the use of a floating pragma version. For example, Arbitrum currently does not support solidity 0.8.20. While the Schwap protocol is initially deployed on Ethereum, there are plans to support other chains in the future, making this issue relevant but low severity.

Recommendation: Lock the pragma at a specific version up to 0.8.19.

Resolution: Unresolved.

5.4.3 sellAllAmount may revert in cases where it shouldn't necessarily do so

Severity: *Low Risk*

Context: SchwapMarket.sol#L269-L289

Description: In some cases, `sellAllAmount` might have gone through all available offers without having spent all of `pay_amt`, but still obtaining more tokens than `min_fill_amount`. At #L271, instead of reverting when `offerId == 0`, it would be better to `break` instead. If the check at #L288 passes, the user will receive more tokens than what they were willing to settle for anyway.

Recommendation:
Rewrite the following line:

```
require(offerId != 0);
```

to the following:

```
if (offerId == 0) break;
```

Resolution: Unresolved.

5.4.4 Incompatibility with fee-on-transfer and rebasing tokens

Severity: *Low Risk*

Context: SimpleMarket.sol, SchwapMarket.sol

Description: The SimpleMarket and SchwapMarket contracts do not properly account for deposited and withdrawn tokens in the case of fee-on-transfer or rebasing tokens.

Recommendation: Consider either mitigating the issue by properly tracking the transferred funds for fee-on-transfer tokens or explicitly stating that such tokens are not supported by the Schwap protocol.

Resolution: Unresolved.

5.4.5 In getBuyAmount and getPayAmount, there should be a check if getBestOffer returns offerId == 0

Severity: *Low Risk*

Context: SchwapMarket.sol#L319, SchwapMarket.sol#L332

Description: Both `getBuyAmount` and `getPayAmount` do not check if the returned value from `getBestOffer` is 0.

Recommendation: Add the following line of code after #L319 and #L332:

```
require(offerId != 0);
```

Resolution: Unresolved.

5.4.6 isOfferSorted should check if _rank[id].delb != 0

Severity: *Low Risk*

Context: SchwapMarket.sol#257

Description: If an offer is fully executed and `_rank[id].delb != 0`, `isOfferSorted` should not return true as it may lead to unexpected behavior.

Recommendation: Add the following check:

```
if (‘_rank[id].delb != 0‘) return false
```

Resolution: Unresolved.

5.4.7 del_rank should be called instantly after unsorting an offer.

Severity: *Low Risk*

Context: SchwapMarket.sol#L173

Description: If an offer is executed and unsorted, its rank should be cleared up instantly. If this doesn't happen, it might lead users to unexpected results when calling `getBetterOffer`/`getWorseOffer` with an id that is unsorted but hasn't had its rank cleared up yet.

Recommendation: Execute the `del_rank` logic when unsorting an offer.

Resolution: Unresolved.

5.4.8 Unclear specification regarding access control of some functions

Severity: *Low Risk*

Context: SchwapMarket.sol#L74, SchwapMarket.sol#L509

Description: The following comment is placed above the `_offeru` and `offer` functions:

```
// Available to authorized contracts only
```

However, there are no access control checks. Considering that the underlying logic does not actually require any access control, the issue is assigned low severity.

Recommendation: Consider whether it is indeed intended that the above-mentioned functions should be restricted to authorized contracts only.

Resolution: Unresolved.

5.4.9 Missing modifier implementation

Severity: *Low Risk*

Context: SimpleMarket.sol#L36-L38

Description: The `can_offer` modifier is declared and used in the SimpleMarket contract. However, an implementation is missing in both the SimpleMarket and SchwapMarket contracts.

Recommendation: Either implement the intended logic or remove the modifier.

Resolution: Unresolved.