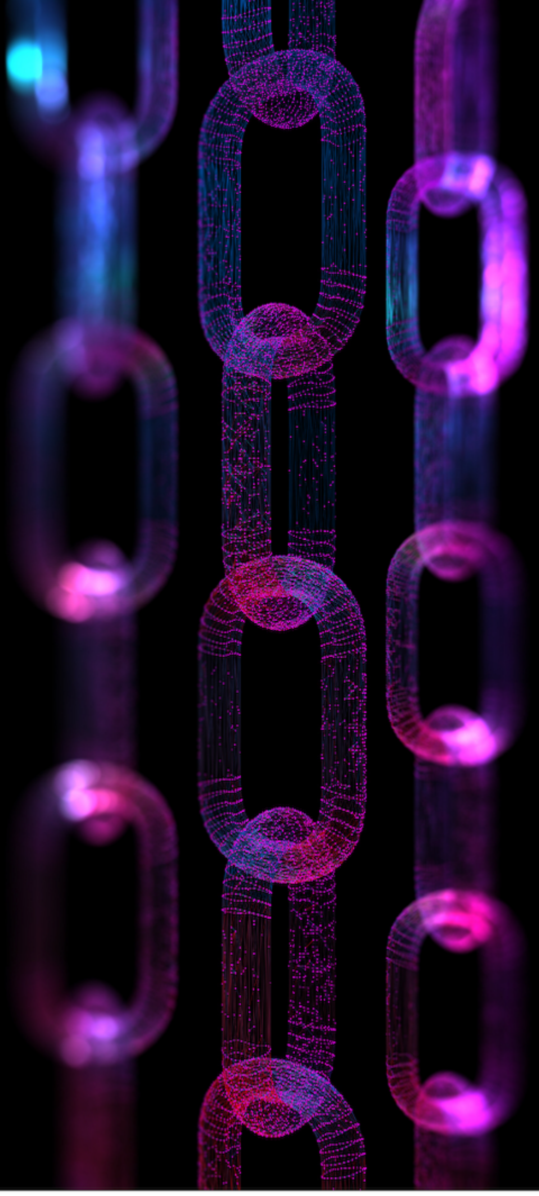
COMPOSABLE SECURITY



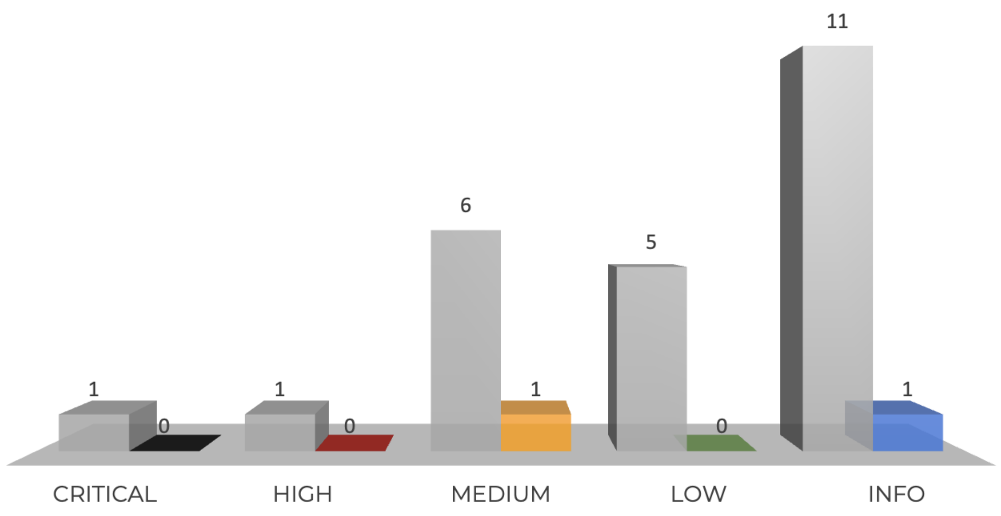
REPORT

Smart contract security review for codefunded services

Prepared by: Composable Security
Report ID: CFND-290d84c
Test time period: 2023-11-07 - 2023-11-14
Retest time period: 2023-12-12 - 2023-12-19
Report date: 2023-12-19
Version: 1.5
Visit: composable-security.com

Contents

1. Retest summary (2023-12-19)



The description of the current status for each retested vulnerability and recommendation has been added in its section.

1.1. Results

The **Composable Security** team was involved in a one-time iteration of verification whether the vulnerabilities detected during the tests (between 2023-11-07 and 2023-11-14) were removed correctly and no longer appear in the code.

The current status of detected issues is as follows:

- **critical** vulnerability has been fully fixed.
- **high** vulnerability has been fully fixed.
- 6 vulnerabilities with a **medium** impact on risk were handled as follows:
  - 5 has been fixed,
  - 1 has been acknowledged.
- 5 vulnerabilities with a **low** impact on risk were handled as follows:
  - 4 has been fixed before the retest,
  - 1 has been fixed during the retest, after consultation (commits: db5ca52, 7fb73d7, 20f311f).
- 11 security **recommendations** were handled as follows:
  - 10 have been implemented,
  - 1 have been acknowledged.

During the retest additional safeguard has been proposed by the testing team and implemented by the developers team in commit: c238961f7d25fb913f26e7377f42081cf241ca42.

1.2. Scope

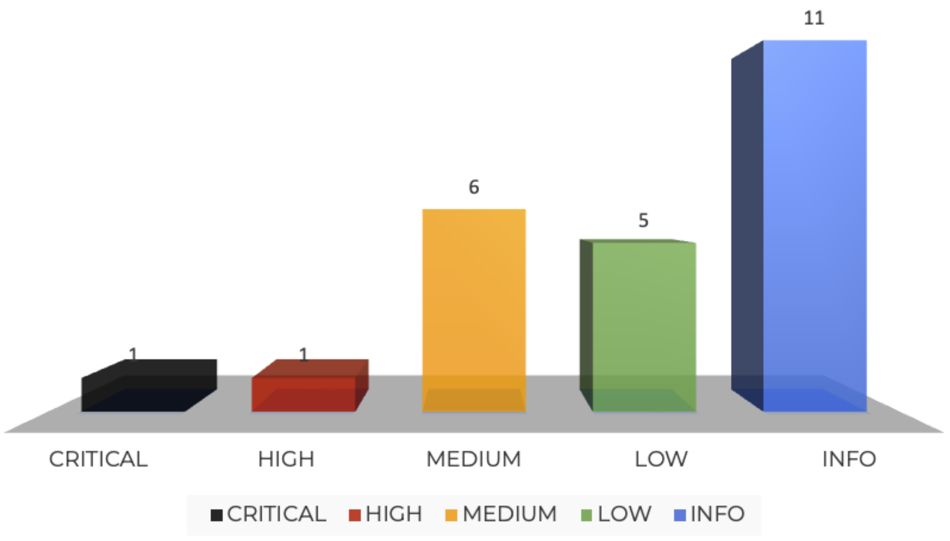
The retest scope included the same contracts, on a different commit in the same repository.

GitHub repository: https://github.com/codefunded/sc-fundswap/
CommitID: b279c659c822415ddca778dc44a15c176ede68e5

2. Current findings status

ID	Severity	Vulnerability	Status
CFND-290d84c-C01	CRITICAL	Loss and theft of tokens in batch executor	FIXED
CFND-290d84c-H01	HIGH	Cancelling order via re-entrancy	FIXED
CFND-290d84c-M01	MEDIUM	Stealing treasury tokens via re-entrancy	FIXED
CFND-290d84c-M02	MEDIUM	Critical impact on FundSwap by malicious plugin	ACKNOWLEDGED
CFND-290d84c-M03	MEDIUM	Invalid fee used	FIXED
CFND-290d84c-M04	MEDIUM	Tokens theft via modified fee	FIXED
CFND-290d84c-M05	MEDIUM	Inconsistent use of modified order	FIXED
CFND-290d84c-M06	MEDIUM	Incorrect handling of non-standard ERC20 tokens	FIXED
CFND-290d84c-L01	LOW	Unfavorable rounding for the maker	FIXED
CFND-290d84c-L02	LOW	Lack of 2-step ownership transfer	FIXED
CFND-290d84c-L03	LOW	Using counter ids in chains with long reorgs	FIXED
CFND-290d84c-L04	LOW	Whitelist bypass via changed order	FIXED
CFND-290d84c-L05	LOW	Full order fill with partial fill	FIXED
ID	**Severity**	**Recommendation**	**Status**
CFND-290d84c-R01	INFO	Consider using the latest solidity version	IMPLEMENTED
CFND-290d84c-R02	INFO	Import specific contracts from the file	IMPLEMENTED
CFND-290d84c-R03	INFO	Use correct type uint16	IMPLEMENTED
CFND-290d84c-R04	INFO	Use proper prefixes for contracts	IMPLEMENTED
CFND-290d84c-R05	INFO	Use prefix increment	IMPLEMENTED
CFND-290d84c-R06	INFO	Inconsistent checks of balances and approvals for order	IMPLEMENTED
CFND-290d84c-R07	INFO	Remove maker key from mapping	IMPLEMENTED
CFND-290d84c-R08	INFO	Handle malicious string data	ACKNOWLEDGED
CFND-290d84c-R09	INFO	Use nested ifs	IMPLEMENTED
CFND-290d84c-R10	INFO	Store length before the loop	IMPLEMENTED
CFND-290d84c-R11	INFO	Avoid multiple calls with the same parameters	IMPLEMENTED

3. Security review summary (2023-11-14)



3.1. Results

The **codefunded** engaged Composable Security to review security of **FundSwap**. Composable Security conducted this assessment over **1 week with 2 engineers**.

The summary of findings is as follows:

- 1 **critical** risk impact vulnerability was identified. Its potential consequences is:
  - Stealing tokens from the chain of sequential orders filled by `FundSwapBatchExecutor` contract.
- 1 vulnerability with a **high** impact on risk was identified. Its potential consequence is:
  - Stealing tokens from the pool and keeping the burnt order data in `orders` mapping (defined in `FundSwapOrderManager`)
- 6 vulnerabilities with a **medium** impact on risk were identified.
- 5 vulnerabilities with a **low** impact on risk were identified.
- 11 **recommendations** have been proposed that can improve overall security and help implement best practice.
- The project is highly centralized, but that was also its intention. Owner addresses have a lot of power, especially through the selection of plugins used.
- The most important issue with medium impact on risk concerns plugins and their ability to influence core business logic in an unrestricted way.
- There are risks in the project that are worth communicating to the community or noting in the documentation due to the limited technical possibilities to prevent them. Some of the operations may be abused due to the nature of blockchain. An example of such a function is `cancelOrder`, which can be front-run by another user.

- The team was engaged and the communication was very good.

Composable Security recommends that **codefunded** complete the following:

- Address all reported issues.
- Extend unit tests with scenarios that cover detected vulnerabilities where possible.
- Consider whether the detected threats and vulnerabilities may exist in other places (or ongoing projects) that have not been detected during engagement.
- Review dependencies and upgrade to the latest versions.

3.2. Scope

The scope of the tests included selected contracts from the following repository.

GitHub repository: https://github.com/codefunded/sc-fundswap/
CommitID: 290d84c77167058280119ee829ebc114c18955e6

The detailed scope of tests can be found in Agreed scope of tests.

4. Project details

4.1. Projects goal

The Composable Security team focused during this audit on the following:

- Perform a tailored threat analysis.
- Ensure that smart contract code is written according to security best practices.
- Identify security issues and potential threats both for **codefunded** and their users.
- The secondary goal is to improve code clarity and optimize code where possible.

4.2. Agreed scope of tests

The subjects of the test were selected contracts from the **codefunded** repository.

GitHub repository:
https://github.com/codefunded/sc-fundswap/
CommitID: 290d84c7716705828011 9ee829ebc114c18955e6

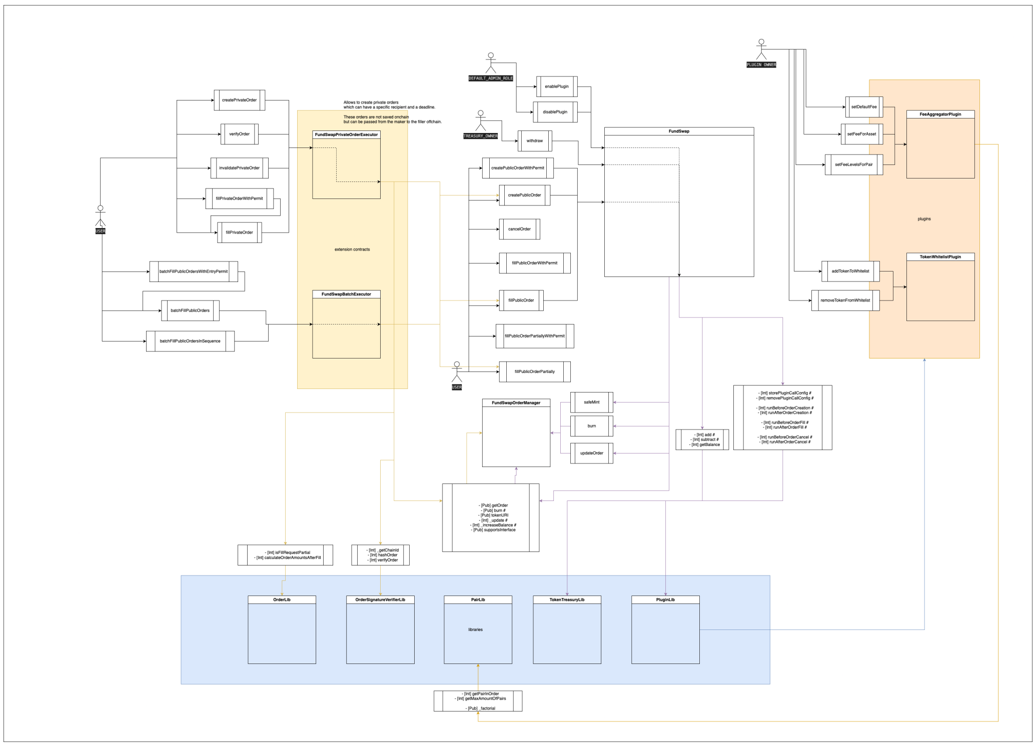
Files in scope:

```
├── contracts/FundSwap.sol
├── contracts/FundSwapOrderManager.sol
├── contracts/IFundSwap.sol
├── contracts/OrderStructs.sol
├── contracts/libraries
│   ├── contracts/libraries/OrderLib.sol
│   ├── contracts/libraries/OrderSignatureVerifierLib.sol
│   ├── contracts/libraries/PairLib.sol
│   ├── contracts/libraries/PluginLib.sol
│   └── contracts/libraries/TokenTreasuryLib.sol
├── contracts/periphery
│   ├── contracts/periphery/FundSwapBatchExecutor.sol
│   └── contracts/periphery/FundSwapPrivateOrderExecutor.sol
└── contracts/plugins
    ├── contracts/plugins/FeeAggregatorPlugin.sol
    ├── contracts/plugins/IPlugin.sol
    └── contracts/plugins/TokenWhitelistPlugin.sol
```

Documentation: Not provided. A brief description of contracts is included in the README.md file.

4.3. Threat analysis

This section summarizes the potential threats that were identified during initial threat modeling performed before the audit. The tests were focused, but not limited to, finding security issues that could be exploited to achieve these threats.


DEFAULT_ADMIN_ROLE
enablePlugin
disablePlugin
TREASURY_OWNER
withdraw
PLUGIN_OWNER
setDefaultFee
setFeeForAsset
setFeeLevelsForPair
FeeAggregatorPlugin
plugins
addTokenToWhitelist
removeTokenFromWhitelist
TokenWhitelistPlugin
createPrivateOrder
verifyOrder
invalidatePrivateOrder
fillPrivateOrderWithPermit
fillPrivateOrder
USER
Allows to create private orders which can have a specific recipient and a deadline.
These orders are not saved onchain but can be passed from the maker to the filler offchain.
FundSwapPrivateOrderExecutor
extension contracts
FundSwapBatchExecutor
batchFillPublicOrdersWithEntryPermit
batchFillPublicOrders
batchFillPublicOrdersInSequence
FundSwap
createPublicOrderWithPermit
createPublicOrder
cancelOrder
fillPublicOrderWithPermit
fillPublicOrder
fillPublicOrderPartiallyWithPermit
fillPublicOrderPartially
USER
FundSwapOrderManager
safeMint
burn
updateOrder
- [Int] add #
- [Int] subtract #
- [Int] getBalance
- [Int] storePluginCallConfig #
- [Int] removePluginCallConfig #
- [Int] runBeforeOrderCreation #
- [Int] runAfterOrderCreation #
- [Int] runBeforeOrderFill #
- [Int] runAfterOrderFill #
- [Int] runBeforeOrderCancel #
- [Int] runAfterOrderCancel #
- [Pub] getOrder
- [Pub] burn #
- [Pub] tokenURI
- [Int] _update #
- [Int] _increaseBalance #
- [Pub] supportsInterface
- [Int] isFillRequestPartial
- [Int] calculateOrderAmountsAfterFill
- [Int] _getChainId
- [Int] hashOrder
- [Int] verifyOrder
OrderLib
OrderSignatureVerifierLib
PairLib
libraries
TokenTreasuryLib
PluginLib
- [Int] getPairInOrder
- [Int] getMaxAmountOfPairs
- [Pub] _factorial

Potential attackers goals:

- Stealing tokens
- Loosing tokens
- Minting orders NFT without limits
- Non-cancelable orders
- Unauthorized plugin
- Unauthorized token
- Lack of transfer when creating or filling order
- Deanonymization of orders
- Invalid calculations
- Bypassing access control
- Denial of Service
- Centralization risk

Potential scenarios to achieve the indicated attacker's goals:

- Double-fill of the order because of lack of replay attacks
- Double transfer of token during filling the order via re-entrancy
- Cancelling the order during the filling process (via re-entrancy)

- Impersonating the maker with someone else's signature of the order
- Stealing tokens via malicious chain of orders
- When cancelling order, malicious treasury owner withdraws order's tokens before they are returned to maker
- Re-submitting different order with the same id after Polygon's fork and sending the filling transaction for worse parameters
- Filling the order created for other filler
- Insufficient signature verification for the order
- Malicious plugin updating order to increase fees
- Out of gas (DoS) because of expensive order fulfillment
- Using non-whitelisted tokens
- Invalid calculation of fee amount
- Unauthorized mint of orders' NFT
- Unauthorized burn of orders' NFT
- Updating the order to the detriment of the user
- Invalid amount of tokens sent after cancellation
- Using incorrect fee from the fees list
- Private key compromise, rug-pull

4.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the **codefunded** development team to thoroughly understand intentions and assumptions of the project.
- Initial threat modeling to identify key areas and focus on covering the most relevant scenarios based on real threats.
- Automatic tests using slither.
- Custom scripts (e.g. unit tests, PoCs) to verify scenarios from initial threat modeling.
- **Manual review of the code.**

4.5. Disclaimer

Smart contract security review **IS NOT A SECURITY WARRANTY**.

During the tests, the Composable team makes every effort to detect any occurring problems and help to address them. However, it is not allowed to treat the report as a security certificate and assume that the project does not contain any vulnerabilities. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we encourage the implementation of security mechanisms at all stages of

development and maintenance.

5. Vulnerabilities

[CFND-290d84c-C01] Loss and theft of tokens in batch executor

CRITICAL **FIXED**

Retest (2023-12-12)

The vulnerability has been removed as recommended. The `minAmountOut` field has been added to the `OrderFillRequest` struct and is verified after the order is filled. Additionally, left-over output tokens are sent out to the filler.

Affected contracts

- FundSwapBatchExecutor.sol

Description

The `batchFillPublicOrdersInSequence` function allows users to fill a chain of orders. The contract will pull tokens from the user only for the first order and then use the output tokens in the next one.

In the end, the `batchFillPublicOrdersInSequence` function transfers the output token to the user, but only from the last filled order. If there are unused output tokens received from the orders in the middle of the chain, they are not transferred to the user and stay in the `FundSwapBatchExecutor` contract.

Additionally, the attacker is able to front-run taker's transaction and partially fill the last order and make user get less of the last output token than expected.

Later those left-over tokens can be stolen by other users using a different malicious chain of two orders.

Attack scenario

1. Some makers added the following orders:
   (a). buying 1000 tokens A for 100 tokens B,
   (b). buying 200 tokens B for 200 tokens C,
   (c). buying 100 tokens C for 100 tokens D.
2. The victim taker fills the following chain of orders:
   (a). selling 1000 tokens A for 100 tokens B,

(b). selling 100 tokens B for 100 tokens C,
(c). selling 100 tokens C for 100 tokens D.

3. The victim taker assumes he will receive 100 tokens D.
4. The attacker front-runs the taker's batch transaction and fills the third order partially (sells 99 tokens C for 99 tokens D).
5. When the taker's transaction is executed, in the last fill the function `_fillPublicOrder` pulls only 1 token C from `FundSwapBatchExecutor` contract and sends back 1 token D (and closes the order).
6. In the end, `FundSwapBatchExecutor` contract sends only 1 token D to victim taker and 99 tokens C are left in `FundSwapBatchExecutor`.
7. Next, the attacker sends one transaction that firstly adds the following orders (tokens X and Y):
   (a). buying 1 token Y for 1 token X,
   (b). buying 99 tokens C for 1 tokens X.
   and then fills them though `FundSwapBatchExecutor` with the following chain:
   (a). selling 1 token Y for 1 token X,
   (b). selling 99 tokens C for 1 token X.
8. When the `FundSwap` contract fills the second order, the 99 tokens C are sent to the attacker who was the maker of this order.

Vulnerable scenario

There is also a scenario when the taker can loose tokens due to invalid chain of fill requests:

1. Some makers added the following orders:
   (a). buying 1000 tokens A for 100 tokens B,
   (b). buying 200 tokens B for 200 tokens C,
   (c). buying 100 tokens C for 100 tokens D.
2. The victim taker fills the following chain of orders:
   (a). selling 1000 tokens A for 100 tokens B,
   (b). selling 100 tokens B for 100 tokens C,
   (c). selling 50 tokens C for 50 tokens D.
3. The victim taker assumes he will receive 50 tokens C and 50 tokens D.
4. The victim receives only 50 tokens D.

Result of the attack: Stealing tokens from the chain of sequential orders filled by `FundSwapBatchExecutor` contract.

Recommendation

- The `FundSwapBatchExecutor` contract should either return all tokens that are left on it after filling the whole chain of orders or make sure that the output tokens from the previous order is equal to the input tokens for the current order and equal to the value provided by taker in fill request, and revert otherwise.

- To protect from changing the orders via partial fill and front-running, we recommend adding a `minAmountOut` field in each `OrderFillRequest` and verifying this at the end of fill, so that the taker can be sure that they received at least as much as they asked for.

References

1. SCSVS G4: Business logic

[CFND-290d84c-H01] Cancelling order via re-entrancy

HIGH **FIXED**

Retest (2023-12-12)

The vulnerability has been removed as recommended. The `nonReentrant` modifier has been added. The `updateOrder` function verifies whether the order exists.
The whitelist plugin is not added by default in the Solidity code, but it is included in the deployment script.

Affected contracts

- FundSwap.sol
- FundSwapOrderManager.sol

Description

The `cancelOrder` function is not protected from re-entrancy (like other functions are with `nonReentrant` modifier) and can be called from the tokens' callback function (1,2) (e.g. ERC-777).

This allows the attacker to cancel their order during the filling proces and retrieve twice as much tokens. In fact, they would get their tokens, but and the same amount from other makers.

Note: Tokens are meant to be whitelisted, but it's achieved with plugin that is not required. Additionally, selected functions in the `FundSwap` contract are protected from re-entrancy (with `nonReentrant` modifier), therefore the testing team assumes that re-entrancy has been included in the internal threat model.

Attack scenario

The attackers might take the following steps in turn:

1. Some user adds order (e.g. selling 10 WETH for 20k USDC). This is required because the attacker must steal someone's tokens.
2. Attacker adds the order via their contract (selling 1 WETH for 1000 token X, which has a callback on receiver or is simply malicious and does any callback).
3. Attacker calls the `fillPublicOrderPartially` function to partially fill the order (999 X for 0.999 ETH).
4. The `FundSwap` contract calls `safeTransferFrom` function on market buy token, which calls back the attacker's contract (re-entrancy in `safeTransferFrom` function).
5. The attacker's contract calls the `cancelOrder` function on order from point 1:
   (a). `FundSwap` sends 1 WETH back to attacker.
   (b). `FundSwap` subtracts the treasury.
   (c). `FundSwap` burns the order in order manager.
6. After re-entrant call the `FundSwap` contract calls the `safeTransfer` function on the maker sell token and transfers 0.999 ETH to the attacker.
7. `FundSwap` contract subtracts the balance of `tokenTreasury` for maker sell token which does not revert, because there is an order added by other user.
8. `FundSwap` contract updates the order which does not revert because the update function does not verify whether the NFT for given order still exists, but it simply updates the orders mapping.

Result of the attack: Stealing tokens from the pool and keeping the burnt order data in `orders` mapping (defined in `FundSwapOrderManager`).

Recommendation

- Protect `cancelOrder` function from re-entrancy (using modifier).
- Consider adding whitelisting plugin by default or implementing whitelisting in the `FundSwap` contract.
- Check whether order's NFT exists in the `updateOrder` function.

References

1. SCSVS G6: Communications

[CFND-290d84c-M01] Stealing treasury tokens via re-entrancy

MEDIUM FIXED

Retest (2023-12-12)

The vulnerability has been removed as recommended. The `nonReentrant` modifier has been added.
However, the `tokenTreasury` is not subtracted for `makerSellToken` after the transters in the `_fillPublicOrder` and `_fillExactInputPublicOrderPartially` functions.

Affected contracts

- FundSwap.sol

Description

The `withdraw` function is used by the treasury owner to withdraw outstanding balance of any provided token. The function makes sure that after withdrawal of the requested amount there are enough tokens left to cover all active orders.

However, the required amount of tokens can be manipulated using re-entrancy. When filling the order, the treasury owner can withdraw more tokens than should be allowed.

Note: Tokens are meant to be whitelisted but it's achieved with plugin that is not required. Additionally, selected functions in the `FundSwap` contract are protected from re-entrancy (with `nonReentrant` modifier), therefore the testing team assumes that re-entrancy has been included in the internal threat model.

Attack scenario

The malicious treasury owner might take the following steps in turn:

① Some user adds order (e.g. selling 10 WETH for 20k USDC). This is required because the attacker must steal someone's tokens.

② Attacker (malicious treasury owner) adds the order via their contract (e.g. selling 1 WETH for 1000 token X, which has a callback on receiver or is simply malicious and does any callback).

③ Attacker fills the order from previous point.

- (a). `FundSwap` contract subtracts the balance of `makerSellToken` in the `tokenTreasuruy`
- (b). `FundSwap` contract calls `safeTransferFrom` function on `marketBuyToken`, which calls back the attacker's contract (re-entrancy in `safeTransferFrom` function).
- (c). The attacker's contract calls the `withdraw` function for `makerSellToken`.
- (d). `FundSwap` sends 1 WETH to attacker because the balance of `FundSwap` was not yet decreased by the transfer, but the balance in `tokenTreasury` struct has already been subtracted.
- (e). After re-entrant call the `FundSwap` contract calls the `safeTransfer` function and continues filling the order.

④ In the end, the attacker withdrew 1 WETH and bought back 1 WETH from his order, thus the attacker has stolen 1 WETH.

Result of the attack: Stealing tokens from the orders as treasury owner.

Recommendation

- Protect `withdraw` function from re-entrancy.
- Subtract `makerSellToken` in `tokenTreasury` after transfer.

References

1. SCSVS G6: Communications

[CFND-290d84c-M02] Critical impact on FundSwap by malicious plugin

MEDIUM **ACKNOWLEDGED**

Retest (2023-12-12)

The development team has implemented the `_checkPluginChanges` function, designed to ascertain the stability of token addresses following modifications by plugins.
Furthermore, the introduction of the `minAmountOut` parameter serves as a safeguard against unauthorized alterations of output amounts.
However, it is important to note that this update does not include measures to protect against Denial of Service (DoS) attacks. Consequently, we strongly advise conducting a thorough security review for any newly integrated plugins to ensure system integrity and security.

Affected contracts

- FundSwap.sol

Description

Malicious plugins can change `makerBuyToken` and `makerSellToken` (not for partial fills), steal other approved tokens from takers and makers or cause a Denial of Service on the selected functionalities in the protocol.

Note: The impact on risk has been reduced because the plugins are mean to be controlled by the trusted third parties.

Attack scenario

The malicious plugin might take the following steps to **steal the tokens**:

① Maker adds order (e.g. selling 10 WETH for 20k USDC).
② `Fundswap` calls `PluginLib.runBeforeOrderCreation` function.
③ Malicious plugin changes the `makerBuyToken` from USDC to some fake token or the `makerBuyTokenAmount` to 1.
④ `Fundswap` stores the modified order.
⑤ The owner of malicious token back-runs the maker and fills the order and gets 10 WETH for 1 USDC or for a fake token.

The malicious plugin might take the following steps to do the **denial of service** attack:

① Maker adds order (e.g. selling 10 WETH for 20k USDC).
② After some time, maker wants to cancel the order and calls `cancelOrder` function.
③ `Fundswap` calls `PluginLib.runBeforeOrderCancel` function which reverts and blocks the cancellation.
④ Later, some taker fills the order.
⑤ `Fundswap` calls `PluginLib.runBeforeOrderFill` function which reverts and blocks the filling.
⑥ In the end, the tokens are stuck in the contract.

Result of the attack: Stealing tokens by malicious plugins and Denial of Service (until the owner disables the malicious plugin).

Recommendation

Consider one of the following:

- not allow plugins to change all order parameters (use the immutable ones from the original order),
- add the parameter `minAmountOut` and `outToken` to `OrderFillRequest` struct that allows the taker to specify minimum amount of `outToken` to be received and revert the filling operation if the output amount is smaller.

Note: if the team decides to implement a recommendation to now allow plugins to change token addresses, the `outToken` parameter can be ommited.

Long term: make sure each plugin is reviewed for security issues that could influence the whole project.

References

1. SCSVS G6: Communications

[CFND-290d84c-M03] Invalid fee used

MEDIUM **FIXED**

Retest (2023-12-12)

The vulnerability has been removed as recommended.
The `setFeeLevelsForPair` function accepts fee levels in ascending order (ordered by `minAmount` parameter), but does not check whether the fees are lower for greater amounts.
Additionally, the `_calculateFeeAmount` still iterates over the whole list instead of stopping when the the greater `minAmount` is found.
The `pairsForAsset`, `assetsWithPairFee` variables and the `getFeesForAllAssets`, `getFeeLevelsForAllPairs` functions have been removed.
Note: The developers team has been informed about the missing changes and introduced them during the retest in commit: `82c0e14545534dc011de4f76e85d5a80001f3c3c`
.

Affected contracts

- FeeAggregatorPlugin.sol

Description

The `_calculateFeeAmount` function calculates the fee for a specific pair of assets and an input amount.

Each pair can have multiple levels of fees that are based on the `amount`. When searching for the fee level, the `_calculateFeeAmount` function iterates over all levels (FeeAggregatorPlugin.sol#L291) and checks whether the `amount` is greater than the current level's `minAmount` and its fee is lower than the current one. If those checks are passed, the current fee is updated to the fee from current level.

However, this search can lead to invalid fees being used.

Vulnerable scenario

The following steps lead to the described result:

① The owner of fee plugin adds the following levels for a specific pair of tokens:
  - `minAmount`: 100 tokens, `fee`: 1%,
  - `minAmount`: 1000 tokens, `fee`: 2%.

② The taker fills the order for that pair with the amount of 1001 tokens.

③ The taker will pay 1% fee instead of 2%.

Result of the attack: Using invalid fee for the pair.

Recommendation

- We recommend to change the `setFeeLevelsForPair` to accept only sorted levels by `minAmount` in ascending order (keeping the requirement of the first `minAmount` to be qual to 0).
  Then, the function `_calculateFeeAmount` should be changed to take the first fee from the first level (index equal to 0) as the final fee and iterate over the levels until the `amount` is smaller than the `minAmount` and update the final fee.
- Unnecessarily, both assets are being stored in `assetsWithPairFee` and in both variations of the pair in `pairsForAsset`. These fields are private, hence they cannot be directly retrieved by anyone, and their use is confined solely to the `setFeeLevelsForPair` and `getFeeLevelsForAllPairs` functions, where the code becomes unreadable due to this complexity.
  We recommend to store the smaller address `asset1` in `assetsWithPairFee` and add only the mapping of `asset1` -> `asset2` in `pairsForAsset`. This approach will eliminate the need for a temporary variable in `getFeeLevelsForAllPairs` and reduce the process to a single loop.

References

1. SCSVS G7: Arithmetic

[CFND-290d84c-M04] Tokens theft via modified fee

MEDIUM FIXED

Retest (2023-12-12)

The vulnerability has been removed as recommended.

Affected contracts

- FeeAggregatorPlugin.sol

Description

Plugin owner can front-run the filling transactions and update the fees to 100%.

Attack scenario

The attacker (malicious plugin owner) might take the following steps in turn:

① Maker adds an order (e.g. selling 1 WETH for 2000 USDC).
② Victim taker fills the order and hopes to receive 1 WETH minus small fee.
③ Fee plugin's owner front-runs the filling transaction and updates the fee to 100%.
④ Victim pays 2000 USDC but receives no WETH.

Result of the attack: Stealing order's tokens by the fee plugin owner and treasury owner who might be the same person.

Recommendation

- Add timelock to functions that update fees and other critical parameters in plugins.
- Set reasonable maximum fee (e.g. 3%).
- Similarly to the recommendation in CFND-290d84c-H01, add the parameter `minAmountOut` to `OrderFillRequest` struct that allows the taker to specify minimum amount to be received and revert the filling operation if the output amount is smaller.

References

1. SCSVS I1: Basic

[CFND-290d84c-M05] Inconsistent use of modified order

MEDIUM **FIXED**

Retest (2023-12-12)

The vulnerability has been removed as recommended. The token addresses cannot be changed by plugins.

Affected contracts

- FundSwap.sol

Description

Plugins are able to change orders before and after they are created or filled. In the case of a partial fill the modified order is treated differently from the full fill.

- **In case of a partial fill:** it transfers `makerBuyToken` from the modified order, but the `makerSellToken` is taken from the original order.

- **In case of a full fill:** both tokens are taken from the modified order.

Attack scenario

The attackers might take the following steps in turn:

1. Maker adds an order.
2. One of plugins changes both tokens before the fill.
3. Taker partially fills the order.
4. `FundSwap` transfers `makerSellToken` from the original order instead of the modified one.

Result of the attack: Transfer of incorrect token.

Recommendation

- The modified order should be treated the same on all cases, therefore partial fill should transfer `makerSellToken` taken from the modified order.
- However, the general recommendation is to not allow to change tokens by the plugins and take token addresses from the original order in all cases.

References

1. SCSVS G4: Business Logic

[CFND-290d84c-M06] Incorrect handling of non-standard ERC20 tokens

MEDIUM FIXED

Retest (2023-12-12)

The vulnerability has been removed as recommended.
The developers team do not plan to use fee-on-transfer tokens. When creating an order, the factual amount transferred is calculated and compared with the amount parameter. If they do not match, the function reverts.
Additionally, the team plans to use whitelist plugin in all cases and do not put non-standard ERC20 tokens on the list.

Affected contracts

- FundSwap.sol
- FundSwapPrivateOrderExecutor.sol

Description

Using non-standard ERC20 tokens, such as **fee-on-transfer tokens**, can lead to Denial of Service, inability to cancel the order and withdraw the `makerSellToken`.

When transferring tokens from the `msg.sender` in `createPublicOrder`, the contract assumes that full `order.makerSellTokenAmount` amount has been received and increases the balance in `tokenTreasury`. In fact, the amount has been reduced by the fee.

Later, the functions `_fillPublicOrder` and _fillExactInputPublicOrderPartially try to send the `order.makerSellTokenAmount` amount, but the balance is insufficient or tokens are taken from the deposit related with a different order.

The same situation happens when the treasury owner wants to withdraw tokens or the maker wants to cancel the order. Last case where this issue is present is the `fillPrivateOrder` function, where the `FundSwapPrivateOrderExecutor` contract tries to transfer out the `privateOrder.makerBuyToken` that was pulled in the same function, but the transaction reverts due to insufficient balance.

Similar problems can arise with **rebasing tokens**. During the period when they are kept on the `FundSwap` contract, their amount might change (increase or decrease). However, the order's `makerSellTokenAmount` is not updated accordingly.

When sending out the changed amount, the transfer would revert (or took the missing amount from other orders) if the amount decreased or the difference would be left on the `FundSwap` contract if the amount increased.

Vulnerable scenario

The following steps lead to the described result:

1. Maker adds an order for 100 tokens X (with 1% fee on transfer).
2. `FundSwap` pulls 100 tokens (and stores 100 tokens in `makerSellTokenAmount`), but receives 99 tokens.
3. `FundSwap` adds 100 tokens to `tokenTreasury`.
4. Makers wants to cancel his order and calls `cancelOrder` function.
5. `FundSwap` reads `makerSellTokenAmount` and makes a transfer to the maker with 100 tokens, but has only 99 tokens and the **transaction reverts**.

Result of the attack: Denial of Service and loss of tokens for fee-on-transfer and rebasing tokens.

Recommendation

- If you are not planning to use non-standard ERC20 tokens, make sure that the the whitelist plugin is used and such tokens are never added to it.

- When pulling the tokens from maker, check the balance before and after the transfer to calculate the factual amount transferred and either update the order accordingly (`makerSellTokenAmount`) or revert if the transferred amount and `makerSellTokenAmount` are not equal.

References

1. SCSVS I2: Token
2. SCSVS G4: Business Logic

[CFND-290d84c-L01] Unfavorable rounding for the maker

LOW FIXED

Retest (2023-12-12)

The vulnerability has been removed as recommended.

Affected contracts

- OrderLib.sol

Description

During the partial fill, the `OrderLib` library calculates how many tokens should be left after the order is partially filled. The amount is calculated using division and is rounded down.

That leads to a situation when taker does not have to transfer whole `makerBuyTokenAmount` to retrieve all `makerSellTokenAmount` tokens. As the taker controls the amount of tokens that is exchanged, the rounding should be in favor of the maker.

Attack scenario

The attackers might take the following steps in turn:

① Maker adds an order that sells 100 token A for 1000 token B.
② Taker submits a partial fill with 991 tokens B.
③ `FundSwap` transfers 991 token B to maker and 100 token A to taker.

Result of the attack: Rounding in favor of taker.

Recommendation

- Add `Rounding.Ceil` to `mulDiv` function.

References

1. SCSVS G7: Arithmetic

[CFND-290d84c-L02] Lack of 2-step ownership transfer

LOW FIXED

Retest (2023-12-12)

The vulnerability has been removed as recommended.

Affected contracts

- FeeAggregatorPlugin.sol
- TokenWhitelistPlugin.sol

Description

The `TokenWhitelistPlugin` and `FeeAggregatorPlugin` contracts inherit from `Ownable` contract which allows to instantly transfer ownership to any address except `address(0)`. In case of invalid address passed as the new owner (e.g. a contract that cannot make calls to protected functions and the `transferOwnership` function) there is no way to get the ownership back.

The `Ownable` contract is also inherited by the `FundSwapOrderManager` contract, but its owner is `FundSwap` contract and there is no way to transfer or renounce it.

Vulnerable scenario

Faulty operation scenario:

① Owner transfers ownership to the wrong address.

Result of the attack: No possibility to call the functions protected by `onlyOwner` modifier.

Recommendation

- Use `Ownable2Step` contract instead of `Ownable`.
- Consider overriding the `renounceOwnership` function to make it always revert if

you do not plan to do it.

References

1. SCSVS G5: Access control

[CFND-290d84c-L03] Using counter ids in chains with long reorgs

LOW **FIXED**

Retest (2023-12-19)

The vulnerability has been removed as recommended. The `tokenIdCounter` variable has been removed and the `tokenHash` is used as token identifier.
Additionally, token id to hash mappings in `FundSwapOrderManager` were removed and the order is identified with `uint256` type (compliant with ERC721) instead of `bytes32`.

Retest (2023-12-14)

The developers team has used order hash as order's identifier. However, the `FundSwapOrderManager` ERC-721 contract still mints tokens with incremental numbers, that can be exploited in the following way:

- User A creates an order O1 that gets `tokenId` = 10.
- User A creates an order O2 that gets `tokenId` = 11.
- User A transfers token 10 (order O1) to user B.
- The chain reorgs.
- Attacker resends transaction from step 2, but the order O2 gets `tokenId` = 10.
- Attacker resends transaction from step 3 and transfers the order O2 instead of O1, because of the `tokenId` swap.

We recommend to use `tokenHash` as the identifier of `FundSwapOrderManager` (ERC721) token (casted to `uint256`).

Affected contracts

- FundSwap.sol

Description

The `createPublicOrder` function creates a new order and its identifier is generated via `safeMint` function (from order manager contract) using the `tokenIdCounter` counter.

The `FundSwap` project is going to be deployed on Polygon chain where reorgs happen quite often and can be longer that a couple of minutes. An example of almost a minute reorg was on 22th of October: 49016661

With the block reorg, the attacker can submit a different order that will reuse the identifier and resubmit the taker's transaction that will fill the new order.

Note: The vulnerability requires that the order is created and filled within the reorged transactions.

Attack scenario

The attackers might take the following steps in turn:

① Before reorg:
  - Attacker adds an order (e.g. 1 WETH for 2000 USDC) that gets the id 1.
  - Victim fills the order with id 1.
  - `FundSwap` sends 2000 USDC to attacker and 1 WETH to victim.

② After reorg (all previous transactions are cancelled):
  - Attacker adds an order (e.g. 0.1 WETH for 2000 USDC) that gets the id 1.
  - Attacker resends victim's transaction that fills the order with id 1.
  - `FundSwap` sends 2000 USDC to the attacker and 0.1 WETH to the victim.

Result of the attack: Theft of taker's tokens via unfavorable order.

Recommendation

- Use hash of the order as its identifier.

References

1. SCSVS G4: Business Logic

[CFND-290d84c-L04] Whitelist bypass via changed order

LOW **FIXED**

Retest (2023-12-12)

The vulnerability has been removed as recommended. Plugins cannot change token addresses.

Affected contracts

- PluginLib.sol
- TokenWhitelistPlugin.sol

Description

The `TokenWhitelistPlugin` plugin makes sure that the exchanged tokens are on the list of allowed tokens (TokenWhitelistPlugin.sol#L43, TokenWhitelistPlugin.sol#L50). However, other plugins can change those tokens and bypass the whitelist.

Attack scenario

The attackers might take the following steps in turn:

1. A new plugin is added (after whitelisting plugin), which replaces the token inside the order to the non-whitelisted token.

Result of the attack: Exchanging tokens that are not on the whitelist.

Recommendation

- The general recommendation is to not allow to change tokens by the plugins and take token addresses from the original order in all cases.
- However, if the dev team plans to allow to change tokens by plugins, make sure that the whitelisting plugin is the first one on the plugins list.

References

1. SCSVS G5: Access control

[CFND-290d84c-L05] Full order fill with partial fill

LOW **FIXED**

Retest (2023-12-12)

The vulnerability has been removed after the code freeze, before the audit started.

Affected contracts

- FundSwap.sol

Description

The `_fillPublicOrderPartially` function allows to fill full amounts of the order that does not result in burning the order and does not emit the `PublicOrderFilled` event.

The taker can later call `fillPublicOrder` function with amount equal to 0 to emit that event, but there is no incentive for them to do that.

Note: The codefundedteam also reported this vulnerability independently after the code freeze.

Attack scenario

The attackers might take the following steps in turn:

1. Maker adds an order (e.g. selling 1 WETH for 2000 USDC).
2. Taker calls the `fillPublicOrderPartially` with `amountIn` equal to 2000 USDC.
3. `FundSwap` makes the partial fill and the order remains with amounts equal to 0, but no `PublicOrderFilled` is emitted.

Result of the attack: Lack of `PublicOrderFilled` event when the order is filled fully.

Recommendation

- Detect a case when taker calls `fillPublicOrderPartially` function with the full amount of `makerBuyTokenAmount` and revert.
  - FundSwap.sol#L356

References

1. SCSVS G4: Business Logic

6. Recommendations

[CFND-290d84c-R01] Consider using the latest solidity version

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented. The version used is `0.8.23`.

Description

In accordance with the best security practices, it is recommended to use the latest stable versions of major Solidity releases. Very often, older versions contain bugs that have been discovered and fixed in newer versions.

Moreover, it is worth remembering that the version should be clearly specified so that all tests and compilations are performed with the same version.

Recommendation

Use a specific version of Solidity compiler (latest stable):

```
pragma solidity 0.8.23;
```

WARNING: If you plan to deploy on multiple chains, be aware that some of them don't support PUSH0 opcode, which will be in your bytecode if you use solc >=0.8.20. In this situation, it is recommended to choose 0.8.19.

References

1. SCSVS V1: Architecture, design and threat modeling

[CFND-290d84c-R02] Import specific contracts from the file

INFO **IMPLEMENTED**

Retest (2023-12-19)

The recommendation has been implemented.

Retest (2023-12-12)

The recommendation has been partially implemented. The libraries still import full files. We present imports from `OrderLib.sol` file:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

import '@openzeppelin/contracts/utils/math/Math.sol';
import '../OrderStructs.sol';
```

Description

Import declarations should import specific identifiers, rather than the whole file. Using import declarations of the form `import <identifier_name> from "some/file.sol"` avoids polluting the symbol namespace making flattened files smaller, and speeds up compilation (but does not save any gas).

Recommendation

Use import declarations of the form `import <identifier_name> from "some/file.sol"`.

References

1. SCSVS G11: Code clarity

[CFND-290d84c-R03] Use correct type uint16

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented. Now the `uint16` type is used for the `lowestFee` variable.

Description

The `lowestFee` variable is of type `uint256` while the fees are kept in `uint16` type. Later, the `lowestFee` variable is also casted to `uint16`.

Recommendation

Use `uint16` type for `lowestFee` variable.

References

1. SCSVS G11: Code clarity

[CFND-290d84c-R04] Use proper prefixes for contracts

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented. Now the abstract contract name is PluginBase.

Description

Do not use `I` prefix for abstract contracts (`IPlugin`). The best practice is to use it with interfaces.

Recommendation

Use `BasePlugin` instead of `IPlugin` for the abstract contract.

References

1. SCSVS G11: Code clarity

[CFND-290d84c-R05] Use prefix increment

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented.

Description

Consider using the prefix increment expression whenever the return value is not needed. The prefix increment expression is cheaper in terms of gas.

Recommendation

Use `++i` instead of `i++`.

- FundSwapBatchExecutor.sol#L69

- FundSwapBatchExecutor.sol#L97
- PluginLib.sol#L98
- PluginLib.sol#L116
- PluginLib.sol#L132
- PluginLib.sol#L148
- PluginLib.sol#L164
- PluginLib.sol#L180
- FeeAggregatorPlugin.sol#L168
- FeeAggregatorPlugin.sol#L208
- FeeAggregatorPlugin.sol#L246
- FeeAggregatorPlugin.sol#L248
- FeeAggregatorPlugin.sol#L258
- FeeAggregatorPlugin.sol#L264
- FeeAggregatorPlugin.sol#L291

References

1. Semgrep: Use prefix increment
2. SCSVS G11: Code clarity

[CFND-290d84c-R06] Inconsistent checks of balances and approvals for order

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented. Now also taker balance is verified.

```
if (
  IERC20(privateOrder.makerBuyToken).balanceOf(_msgSender()) <
  privateOrder.makerBuyTokenAmount
) {
  revert FundSwap__InsufficientTakerBalance();
}
```

Description

The function `fillPrivateOrder` has inconsistent checks of balances and approvals. Only the balance of the maker is verified.

```
if (
  IERC20(privateOrder.makerSellToken).balanceOf(privateOrder.maker) <
  privateOrder.makerSellTokenAmount
) {
  revert FundSwap__InsufficientMakerBalance();
```

Recommendation

- The `fillPrivateOrder` function should also verify appropriate approvals and taker side of the order.
- Due to the large number of checks here, it is recommended grouping and separating them with comments describing specific sections to increase the readability of the code.

References

1. SCSVS G4: Business Logic

[CFND-290d84c-R07] Remove maker key from mapping

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented.

Description

It is not necessary to have maker as the key because the maker is part of the hash and cannot be spoofed as their signature is checked.

```
/// @dev token => order hash => is executed
  mapping(address => mapping(bytes32 => bool)) public executedPrivateOrderHashes
     ;
```

should be:

```
/// @dev token => order hash => is executed
  mapping(bytes32 => bool)) public executedPrivateOrderHashes;
```

Recommendation

Remove the address from the mapping and limit yourself to the order hash as the key.

References

1. SCSVS G11: Code clarity

[CFND-290d84c-R08] Handle malicious string data

INFO **ACKNOWLEDGED**

Retest (2023-12-12)

The developers team has acknowledged the recommendation and is aware of potential XSS on the front-end side.

Description

When handling strings, remember to handle them appropriately on the front-end and back-end sides of web2 applications. They can be a path to attack an application via Cross-Site Scripting.

Recommendation

Validate strings passed through the smart contracts on the web2 part.

- FundSwapOrderManager.sol#L84

References

1. SCSVS I1: Basic

[CFND-290d84c-R09] Use nested ifs

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented. Currently, the nested ifs are used instead of &&.

Description

Using nested `if`s is cheaper than using && multiple check combinations. There are more advantages, such as easier to read code and better coverage reports.

Recommendation

Use nested `if`s.

- FundSwapPrivateOrderExecutor.sol#L155
- FundSwapPrivateOrderExecutor.sol#L158
- FeeAggregatorPlugin.sol#L252
- FundSwap.sol#L262
- FundSwap.sol#L265
- FundSwap.sol#L361

References

1. Semgrep: Using nested if
2. SCSVS G11: Code clarity

[CFND-290d84c-R10] Store length before the loop

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented. Currently, the length in detected LOCs is stored before loops..

Description

Caching the array length outside a loop saves reading it on each iteration, as long as the array's length is not changed during the loop.

Recommendation

Save the length of the array in a variable and use the variable in the loop.

- FundSwapBatchExecutor.sol#L69
- FundSwapBatchExecutor.sol#L97
- PluginLib.sol#L98
- PluginLib.sol#L116
- PluginLib.sol#L132
- PluginLib.sol#L148
- PluginLib.sol#L164
- PluginLib.sol#L180
- FeeAggregatorPlugin.sol#L168
- FeeAggregatorPlugin.sol#L246

- FeeAggregatorPlugin.sol#L248
- FeeAggregatorPlugin.sol#L291

References

1. Semgrep: Array length outside the loop
2. SCSVS G11: Code clarity

[CFND-290d84c-R11] Avoid multiple calls with the same parameters

INFO **IMPLEMENTED**

Retest (2023-12-12)

The recommendation has been implemented. Currently neither `calculateOrderAmountsAfterFill` nor `hashOrder` is called twice.

Description

Calling the same function with identical parameters reduces code readability and costs more gas.

The `OrderLib.calculateOrderAmountsAfterFill` function is called twice in `_fillPublicOrderPartiall` function (directly, and nested).

The `OrderSignatureVerifierLib.hashOrder` function is unnecessarily called two times in `fillPrivateOrder` (1,2), because there exists a `orderHash` variable.

Recommendation

Store the value after the first call and reuse it.

References

1. SCSVS G11: Code clarity

7. Impact on risk classification

Risk classification is based on the one developed by OWASP [1], however it has been adapted to the immutable and transparent code nature of smart contracts. The Web3 ecosystem forgives much less mistakes than in the case of traditional applications, the servers of which can be covered by many layers of security.

Therefore, the classification is more strict and indicates higher priorities for paying attention to security.

OVERALL RISK SEVERITY				
Impact on risk	HIGH	CRITICAL	HIGH	MEDIUM
	MEDIUM	MEDIUM	MEDIUM	LOW
	LOW	LOW	LOW	INFO
		HIGH	MEDIUM	LOW
		Likelihood		

[1] OWASP Risk Rating methodology

8. Long-term best practices

8.1. Use automated tools to scan your code regularly

It's a good idea to incorporate automated tools (e.g. slither) into the code writing process. This will allow basic security issues to be detected and addressed at a very early stage.

8.2. Perform threat modeling

Before implementing or introducing changes to smart contracts, perform threat modeling and think with your team about what can go wrong. Set potential targets of the attacker and possible ways to achieve them, keep it in mind during implementation to prevent bad design decisions.

8.3. Use Smart Contract Security Verification Standard

Use proven standards to maintain a high level of security for your contracts. Treat individual categories as checklists to verify the security of individual components. Expand your unit tests with selected checks from the list to be sure when introducing changes that they did not affect the security of the project.

8.4. Discuss audit reports and learn from them

The best guarantee of security is the constant development of team knowledge. To use the audit as effectively as possible, make sure that everyone in the team understands the mistakes made. Consider whether the detected vulnerabilities may exist in other places, audits always have a limited time and the developers know the code best.

8.5. Monitor your and similar contracts

Use the tools available on the market to monitor key contracts (e.g. the ones where user's tokens are kept). If you have used code from another project, monitor their contracts as well and introduce procedures to capture information about detected vulnerabilities in their code.

Damian Rusinek
Smart Contracts Auditor
@drdr_zz
damian.rusinek@composable-security.com

Paweł Kuryłowicz
Smart Contracts Auditor
@wh01s7
pawel.kurylowicz@composable-security.com