Futureswap V2 Audit

Refactor Audit

OPENZEPPELIN SECURITY | JANUARY 3, 2021

Security Audits

Futureswap is an on-chain futures exchange that offers up to 20x leverage. The Futureswap team asked us to audit their smart contracts. We reviewed the contracts with a team of 2 auditors over the course of 4 weeks. Here we publish our results.

Scope

We audited commit `96255fc4a550a5f34681c117b5969b848d07b3a3` of the `futureswap/fs_core` repo. All files in the `contracts` directory (and all subdirectories) were in scope except for the `Migrations.sol` file and the `mocks` subdirectory.

Update: *After the initial delivery of the report, the Futureswap team made several code changes to address the issues in this report. The pull requests for those changes are linked to in the updates for the corresponding issues in the report below. All the fixes mentioned in this report are present in commit `cd0d7038cd0e39cb22b94d02faf8a6b458df7c78` of the `futureswap/fs-core` repo. The FsToken was pulled out of the `futureswap/fs_core` repo and moved into the `futureswap/fs_token` repo with fixes present in commit `6c0dd5e65bef41f13043a380f6a875315b25bcd7` of that repo.*

Update: *(February 2, 2021) The Futureswap team noticed that miners sometimes use a `block.timestmp` value that is slightly behind the real present time, and that this was causing the MessageProcessor to reject some valid signature submissions. They adjusted the*

Compiled bytecode of audited contracts

Following a special request from the FutureSwap team, we have compiled the audited Solidity files and reference in this report the resulting bytecode. The compilation settings were:

- Compiler version `0.5.17+commit.d19bba13.Linux.g++` with SHA256 checksum `c35ce7a4d3ffa5747c178b1e24c8541b2e5d8a82c1db3719eb4433a1f19e16f3`
- Optimizer: enabled
- Number of runs: 100
- EVM Version: Istanbul

We have not verified that the audited version of the code matches the deployed system on mainnet. To reduce the need to trust third parties, we encourage users to conduct this verification by themselves when the system is open-sourced, using their most trusted set of tools and infrastructure of choice to read information from the Ethereum blockchain. Users and developers willing to conduct such a process must be aware of the following caveats.

Verifying deployed bytecode is not sufficient to ensure the correct behavior of a smart contract system – creation code and initialization parameters should be understood and verified as well.

When compiling contracts without linking libraries, the Solidity compiler inserts placeholders for the addresses of libraries in the bytecode. Therefore, to obtain the executable bytecode, one needs to link contracts to the addresses of deployed libraries in mainnet (read more about it in the Solidity documentation). We are including the compiled bytecode with both unlinked and linked libraries. The addresses used for linking were provided privately by the FutureSwap team.

Even in the case where the FutureSwap team has deployed the exact latest audited version of the in-scope contracts (corresponding to commit `be8d371a40f21b863ec7a99445bad92e6c8d17f1` of the `fs-core` repository and commmit `6c0dd5e65bef41f13043a380f6a875315b25bcd7` of the `fs_token` repository), there might be expected differences between the actual deployed code in mainnet and the compiled bytecode referenced in this report. In particular, one should be aware that:

reference will have 20-bytes-long sequences of zeros instead of the actual address of the library.

- From the Solidity documentation, since the resulting bytecode of the compiled contracts contains the metadata hash, any change to the metadata results in a change of the bytecode. This includes changes to a filename or path, and since the metadata includes a hash of all the sources used, a single whitespace change results in different metadata, and different bytecode. In practice, this translates to the fact that the bytecode of libraries and contracts we reference may have a different metadata hash attached at the end of the executable bytecode, as the absolute paths of the compiled files inevitably differs.

The audited contracts' creation and runtime bytecode can be found here.

High level overview

Wallet

All user funds enter and exit through Futureswap's `Wallet` contract. The Futureswap exchanges work only with ERC20 tokens that have 18 decimals. Futureswap can support tokens that have fewer than 18 decimals because the `Wallet` contract wraps them in a `DecimalPaddingToken` contract, which the exchanges will use in lieu of the original, decimal-deficient token.

Meta transactions and off-chain oracles

Futureswap leverages meta-transactions and a trusted off-chain oracle in a novel way in order to prevent oracle frontrunning issues. First, the user signs a message containing the parameters that describe the action they want to take. For example, if they want to open a long position, they would sign parameters that indicate which asset they want exposure to, how much capital they want to put at risk, how much leverage they want to use, the min and max asset prices they're willing to tolerate, etc.

Next, the user (or a relayer) passes this signature and message to the off-chain oracle, which appends price information for the asset and stablecoins being used, the time at which the oracle signed the message, etc. In this way, the oracle's signed message *includes* the user's message

Finally, the oracle's message and signature can then be used to execute the operation by passing them along as parameters in a function call to the Futureswap `MessageProcessor` contract. The `MessageProcessor` contract enforces the user-chosen bounds. In this way, the oracle cannot force a user to take any action without the user agreeing.

Each user message contains a user-chosen nonce (`userInteractionNumber`), which is tracked by the meta transactions system to prevent signature replays.

Update: *The meta transaction system now requires a signature from a "verifier" that double-checks the information provided by the oracle. It also now requires a signature from a "stamper" that restricts which addresses can relay transactions in the case that the user does not submit their own transaction. These updates are from commit* `3e3a05646d6cac72d07b0dea5db8df9da95df654` *and PR #494, respectively.*

Exchanges

A Futureswap `Exchange` is defined by an asset token and a stablecoin. An exchange can be created by governance via the `ExchangeFactory`. Liquidity providers can add liquidity to the exchange, in exchange for non-transferable liquidity tokens. These liquidity tokens represent a share of the exchange's liquidity and can be redeemed.

Traders can open leveraged long or short positions with up to 20x leverage.

Rewards

Liquidity providers who keep their liquidity in an exchange for an entire week-long window earn rewards in the form of Futureswap tokens (FST). Similarly, users who trade on a Futureswap exchange also earn FST.

When users sign messages to close a trade, their message contains a "referral" address. This address also receives FST rewards, and is a way for third-party UIs to profit from users who user their UI. (Users can, of course, add their own address as the `referral` address).

The FST are non-transferable tokens, and are used to vote on governance decisions.

Not all tokens are compatible with the Futureswap platform. The choice of tokens used with Futureswap must be considered very carefully.

The `wrap` function of the `DecimalPaddingToken` contract assumes standard balance-updating behavior during the underlying token's `transferFrom` function. The `unwrap` function makes a similar assumption about the underlying token's `transfer` function. This means Futureswap is not compatible with tokens that do periodic "rebasing" (e.g., AMPL, BASED, and YAM). It is also not compatible with tokens that extract a fee from the recipient or the receiver during a `transfer` or `transferFrom`. It should not be used with tokens that implement any kind of demurrage.

The `constructor` function of the `DecimalPaddingToken` contract assumes that the underlying token has a `decimals()` function. However, the ERC20 specification says the `decimals` function is optional, and that contracts MUST NOT expect the `decimals` value to be present. This means that Futureswap is not compatible with a fully-compliant ERC20 token that does not expose a `decimal` value. This is rare in practice, but should be kept in mind.

Futureswap can handle ERC20 tokens that have `decimals = 18` "natively". Tokens with fewer than 18 decimals are handled by wrapping them with a `DecimalPaddingToken`. However, tokens with more than 18 decimals are not compatible with Futureswap.

Caution is advised before using any tokens that are "pauseable", that can be minted arbitrarily by a trusted third party, that can impose "blacklists", or that can be upgraded. Any of these actions could result in loss of funds from a Futureswap exchange that supports such a token. Tokens like these should not be used in a Futureswap exchange unless their "owners" are fully trusted.

For the purposes of this audit, we assume that the Futureswap developers and governance system will take these factors into consideration, and only integrate tokens that will behave well with the Futureswap contracts.

Oracles

The off-chain oracles are a critical component of Futureswap, and it is important to understand their powers and responsibilities. Oracles must remain honest (not lie about the price, the time they

An oracle cannot force a user to engage in a trade outside of the bounds that the user set for the trade. The user's bounds are enforced by the contracts. However, a dishonest or compromised oracle can lie about asset and stablecoin prices. They thus have the power to steal money from an exchange by opening a position themselves and then lying about the price when closing it.

Additionally, the oracle must remain online. During normal operation, users require signatures from a valid oracle in order to open and close positions, add and remove liquidity, add collateral to an existing position, liquidate a trade, etc. If the off-chain oracles stop responding to user requests, users cannot perform most critical actions. So it is imperative that the oracles stay online. In the event that oracles go offline, it is possible for the governance system to disable the exchanges (a disruptive emergency measure) to allow users to withdraw their assets.

Some of the behaviors required of an honest oracle are obvious, such as "don't lie about the price". Others are less obvious. For example, if an oracle can be made to sign two user messages from the same user that both have the same `userInteractionNumber` (a seemingly innocent action), then the user can use this to get a "free option" at the expense of liquidity providers. They could do this, for example, by getting the oracle to sign a message that would open a long position, and another message that would open a short position (using the same `userInteractionNumber`), and then watch the asset price for a few minutes before submitting whichever message would be most profitable. This guarantees the user profit, and nullifies the remaining transaction, because it uses the same `userInteractionNumber`, which is now invalid.

For the purposes of this audit, we assume that the oracles are honest. That is, that they will not sign any false information (e.g., price information). We also assume they are aware of the subtleties of the meta transaction system and will not take any actions (e.g., the one described above) that could allow a user to effectively "cancel" a previously signed, but not yet mined, transaction.

The Futureswap team is aware of these subtleties and has outlined correct oracle behavior in an Oracles Rules document. We assume oracles will follow these rules.

Governance system

The voting system has the ability to arbitrarily update many critical variables and contracts. This includes the ability to add new oracles, which means the voting system has all the powers of an oracle, along with many more.

Voters use FST to vote. FST is non-transferable at the contract level, and it is paid out to liquidity providers, traders, and referrers. In this audit, we assume that the voting system behaves honestly and does not pass proposals that would harm Futureswap.

Registry owner and registry holder

All important Futureswap contract addresses are stored in the `Registry` contract. The registry contract has an `owner`. This `owner`, the voting system, and the `ExchangeFactory` contract all have access to change some of the variables in the `Registry`. This includes the ability to add oracles.

Additionally, there is a `RegistryHolder` contract. This is the contract that all other contracts query to find out the address of the `Registry` contract. The `RegistryHolder` has an `owner`, and its `owner` is the most powerful role in the Futureswap system. The `owner` (and the voting system) has the ability to update the `Registry` contract address. This means the `owner` can make arbitrary system changes, including changing the address of the voting system. A compromise of the `RegistryHolder` `owner` key would be critical, and could result in permanent loss of control of the system.

We assume that the `Registry` contract `owner` and the `RegistryHolder` contract `owner` remain honest and uncompromised.

Findings and recommendations

Critical severity

None.

High severity

An attacker who sees an honest user's call to `MessageProcessor.instantWithdraw` in the mempool can grab the `oracleMessage` and `oracleSignature` parameters from the user's transaction, then submit their own transaction to `instantWithdraw` using the same parameters, a higher gas price (so as to frontrun the honest user's transaction), and carefully choosing the gas limit for their transactions such that the internal call to the `callInstantWithdraw` will fail on line 785 with an out-of-gas error, but will successfully execute the `if(!success)` block.

The result is that the attacker's instant withdraw will fail (so the user will not receive their funds), but the `userInteractionNumber` will be successfully reserved by the `ReplayTracker`. As a result, the honest user's transaction will revert because it will be attempting to use a `userInteractionNumber` that is no longer valid.

Consider adding an access control mechanism to restrict who can submit `oracleMessage`s on behalf of the user.

Update: *Fixed in PR #494, where the relayer who transmits the transaction must be approved by a trusted offline "stamper", and in PRs #533 and #537, where the user can now specify a minimum amount of gas that must be passed along by the relayer.*

[H02] Liquidity rewards are computed incorrectly for a week if any liquidity provider removes liquidity during the week

The `calculateLiquidityProviderPayout` function computes the amount of FST rewards that a liquidity provider should earn in proportion to the share of `totalLiquidity` that they provided. However, the function assumes that the `totalLiquidity` provided for the week is `liquidityToken.totalSupplyAt(weekEntry.snapShotId)`. This assumption is incorrect if any liquidity providers have removed liquidity after the snapshot was taken and before the end of the week. The error results in liquidity providers receiving fewer FST rewards than they ought to receive. The size of the error scales with the total amount of liquidity that has been removed, and so can be exacerbated by a malicious whale.

This is a known issue that is described in a code comment within the `payoutLiquidityProvider` function. In the comment, a valid solution to the problem is

> maximum of their starting balance.

However, this solution is not implemented in the code. Consider implementing this solution.

Update: *Fixed in PR #526, where the solution in the code comment mentioned above has been implemented.*

[H03] MessageProcessor interactions can be frontrun for profit

Anyone can take the `oracleMessage` and `oracleSignature` from a valid transaction in the mempool — whether it is being broadcast by an oracle, or by a user who sets their `userInteractionNumber` to an odd number — and rebroadcast it in their own function call using a one-wei-higher gas price. This frontruns the honest sender's transaction and gives the frontrunner the reward from `maybePaySender`. It also results in the honest sender wasting gas (because their transaction will revert when `reserve` is called by `ensureUnusedUserInteractionNumber` in the `verifyCommonParams` function.

This is profitable for the frontrunner as long as the `amount` of the reward from `maybePaySender` is greater than the cost of gas used to frontrun. Under these conditions, one could expect bots to frontrun all `MessageProcessor` interactions of this type. While the user's trades would execute as intended, the honest senders would operate at a loss.

Consider adding access controls so that only `userMessage.signer`, or a valid oracle, can successfully call the functions that invoke the `maybePaySender` function.

Update: *Fixed in PR #494, where the relayer must now be approved by a trusted offline "stamper" in order to submit the user's message and signature.*

Medium severity

[M01] Not using upgrade safe contracts in FsToken inheritance

The `FsToken` contract is intended to be an upgradeable contract, used behind a proxy (namely, the `FsTokenProxy` contract).
However, the contracts `ERC20Snapshot`, `ERC20Mintable` and `ERC20Burnable` in the

From the README file of the upgrades safe library:

> **you must use this package** and not @openzeppelin/contracts if you are writing upgradeable contracts.

In particular, using the upgrades safe library in this case will ensure the inheritance from `Initializable` and the other contracts is always linearized as expected by the compiler (see this forum post for more info about it).

Update: *Fixed in PR #534, where the FsToken was moved to another repo and adjusted to use the upgrade safe version.*

[M02] Users can add collateral to closed trades

During our audit, the Futureswap team independently discovered that users were capable of adding collateral to closed trades. Consider calling the `ensureTradeOpen` function during the `Trading.addCollateral` function to prevent this.

Update: *Fixed in PR #503.*

[M03] Unchecked output of the ECDSA `recover` function

The `ECDSA.recover` function (in version `2.5.1`) returns `address(0)` if the signature provided is invalid. This function is used twice in the Futureswap code: Once to recover an `oracleAddress` from an `oracleSignature`, and again to recover the user's address from their signature.

If the oracle signature was invalid, the `oracleAddress` is set to `address(0)`. Similarly, if the user's signature is invalid, then the `userMessage.signer` or the `withDrawer` is set to `address(0)`.

This can result in unintended behavior. For example, it allows users to perform some interactions on behalf of the zero address, or (in the unlikely event that `address(0)` were ever added as an oracle) it could allow all invalid `oracleSignature`s to be accepted as valid.

Update: *Fixed in PR #493.*

[M04] The `updatePayoutDistribution` function does not correctly update the `sumOfExchangeWeights` on all exchanges

The `for loop` in the `Incentives.updatePayoutDistribution` function is intended to update the `sumOfExchangeWeights` value for every exchange. However, the code is incorrect, and results in only a single exchange updating the its `sumOfExchangeWeights` value to the same value `allExchangeAddresses.length` many times.

The result is that all but one exchange will have an incorrect `sumOfExchangeWeights` until the next time the `advanceWeek()` function is called.

Consider refactoring this loop to correctly update the `sumOfExchangeWeights` on all exchanges.

Update: *Fixed in PR #514.*

[M05] The `instantWithdraw` function's `userMessage` may be generic enough to introduce replay issues between platforms

The user's signature for a call to the `instantWithdraw` function is over generic data that does not include any Futureswap-specific data. In particular, the user signs a struct that contains only a public token `address`, a `uint256 amount`, and a `uint256 userInteractionNumber`.

It is not unlikely that a user may interact with another DeFi platform (e.g., a platform that mimics Futureswap's meta transactions pattern) that also asks them to sign messages of this type. If that happens, then anyone can replay the user's signature — which was intended for another platform — to Futureswap, causing the user's funds to be instant-withdrawn to their wallet.

This does not result in any theft of funds, because the funds are withdrawn to the user's own external account. However, it could be a griefing vector.

Consider requiring that the `userMessage` include something Futureswap-specific.

[M06] Lack of event emissions during important actions

There are several actions that perhaps should be emitting events but aren't, such as: adding/removing wallet access, adding/removing price oracles, adding/removing exchanges, calling the `doFireRegistryUpdateEvent` function, setting the various addresses via the `owner`, approving/vetoing in the `Voting` contract, etc.

Event emission is particularly important when adding/removing oracles and adding/removing wallet access because oracles and wallet accessors are not tracked in iterable objects. Without events to search, users will have a difficult time learning which addresses are oracles and which addresses have wallet access.

Consider reviewing all actions on the platform and adding event emissions when changes are made to important state variables.

Update: *Fixed in PR #528 where events are now emitted when mappings in the registry are updated. It may still be worthwhile to add more events during other important actions, but the this fix covers the most important changes (changes to important variables stored in non-iterable data types).*

[M07] Refunded tokens can become stuck in the Wallet contract

The `Wallet.refund` function can be called by anyone. It transfers `_amount` of `_tokenAddress` tokens from `msg.sender` to the Wallet contract, and credits `_userAddress`. If `_tokenAddress` has `decimals != 18` then the tokens transferred to the contract via this function may become stuck in the Wallet contract and require an upgrade to remove them.

Consider having the `refund` function wrap the input token if necessary. Otherwise, consider documenting that the `refund` function should not be called with any `_tokenAddress` that does not have `decimals == 18`.

Update: *Fixed in PR #515, where the `refund` function now reverts if the token does not have 18 decimals.*

In the `MessageProcessor` contract, if the `callAddLiquidity`, `callRemoveLiquidity`, `callOpenTrade`, and/or `callCloseTrade` function uses more than 64 times the amount of gas used by the `takePenalty` function, then an attacker can cause honest users to be penalized when they shouldn't be.

The attack works as follows. The attacker can monitor the mempool for incoming transactions from honest users (to, say, the `addLiquidity` function), grab the parameters out of the transaction, frontrun the user's transaction with the attacker's own transaction that submits the same parameters, but carefully chooses the `gasLimit` for the attack transaction such that the call to `callAddLiquidity` (within the `addLiquidity` function) would fail with an out-of-gas error during its call to the `Exchange` contract. This would not cause the transaction to revert, but instead cause `success` to be `false`. Then, because only 63/64 of the remaining gas was forwarded during the call to the exchange, there would be enough gas left over to execute the `if (!success)` block, which penalizes the user.

This vulnerability is not present in the current code with the current opcode pricing. However, it could be an issue if Ethereum opcodes are repriced, or if future upgrades change the gas usage of these functions.

Consider adding a `minGas` parameter to the `userMessage`, and reverting during the `verifyCommonParams` function if `gasleft() < minGas`. This way, the user/UI can set the minimum amount of gas a relayer can use to process their transaction.

Update: *Fixed in PRs #533 and #537, where the user can now specify a minimum amount of gas that the relayer must provide when submitting the user's message and signature.*

Low severity

[L01] The public snapshot function of `FsToken` and `LiquidityToken` may be abused to increase gas costs

The `FsToken` and `LiquidityToken` contracts inherit from the `ERC20Snapshot` contract, which has a public `snapshot` function that anyone can call at any time. There are potential risks to having this `snapshot` function be public. The risk are outlined in the comments of the OpenZeppelin Contracts v3.1.0 of `ERC20Snapshot.sol`, quoted below.

```
* While an open way of calling {_snapshot} is required for certain
* you must consider that it can potentially be used by attackers in
*
* First, it can be used to increase the cost of retrieval of values
* logarithmically thus rendering this attack ineffective in the lon
* specific accounts and increase the cost of ERC20 transfers for th
* section above.
```

If this is a concern, then consider upgrading to use `v3.1.0` of the `ERC20Snapshot` contract, which makes the `snapshot` function internal. Alternatively, consider modifying the `v2.5.1` contract to make the `snapshot` function internal.

Update: *Fixed in PR #532, where the `snapshot` function is now access-controlled.*

[L02] Unbounded Loops

There are a few unbounded loops in the codebase, such as the for loops in the doFireRegistryUpdateEvent and getIndexOf functions. These loops can exceed the block gas limit if the exchanges array is sufficiently large. This could prevent the voting system from being able to update addresses in the registry, and could also prevent the removal of exchanges from the exchanges array. We believe the Futureswap team is aware of this issue. We recommend keeping this issue in mind when supporting a large number of exchanges.

Update: *Unchanged. Comment from the Futureswap team: "The Futureswap team does not intend to launch a large number of exchanges and would update contracts beforehand if this was ever necessary. No change done."*

[L03] Unchecked input in `Registry.addExchange`

The `Registry.addExchange` function does not require that the input `_exchange` has not already been added. This means it is possible that a given exchange can be pushed to the `exchanges` array several times. If this happens, then the `removeExchange` function will remove only one instance of the exchange from the `exchanges` array — resulting in a state where an exchange is present in the `exchanges` array but does not have wallet access and does not appear in the `exchangeMapping` mapping.

Update: *Fixed in PR #520.*

[L04] Unchecked input in `Registry.removeExchange`

The `Registry.removeExchange` function does not require that the input `_exchange` is already in the `exchangeMapping`. Consider adding such a `require` statement to the `removeExchange` function. This will prevent an event (which doesn't exist yet, but should as per the "Lack of event emissions during important actions" issue) from firing when the `removeExchange` function is called but no exchange has been removed.

Update: *Fixed in PR #520.*

[L05] Unreachable code in `Wallet.maybeWrapToken`

The second conditional expression in the `Wallet.maybeWrapToken` function will always be `false` unless the `_publicTokenAddress` is malicious and returns inconsistent values for its `decimals`. Consider removing this `if` statement to reduce the deployment size of the contract.

Update: *Fixed in PR #521.*

Notes & Additional Information

[N01] The voting window and resolving window overlap at the time boundary

The `vote` function of the Voting contract allows voting as long as the current block's timestamp is less than or equal to the `votingEnds` value of the proposal, as seen below:

```
function vote(uint256 _proposalId, bool _isYesVote) public {
    ...
    require(now <= p.votingEnds, "vote is not open");
    ...
}
```

```
function requireProposalCanBeResolved(Proposal memory p) private v
    ...
    require(now >= p.votingEnds || p.ownerApproved, "vote is still c
    ...
}
```

In other words, the voting and resolving windows overlap at `p.votingEnds`. So at this timestamp, a proposal could be resolved before the voting window closes. Consider modifying the `requireProposalCanBeResolved` function's `require` statement to `now > p.votingEnds`.

Update: *Fixed by PR #527.*

[N02] ProposalRejected event might never be emitted

The `resolve` function of the `Voting` contract emits a `ProposalRejected` event when the proposal fails. However, there does not seem to be any incentive to call the `resolve` function when a proposal has failed. The caller would spend gas to emit an event without getting anything in return. If the event is never emitted, it may make it difficult to track the state of proposals off-chain via emitted events (such as in tools like The Graph).

If this is a concern, consider adding an incentive for calling the `resolve` function.

Update: *Unchanged. Comment from the Futureswap team: "Not addressed, State of proposals can be tracked via view functions and the Futureswap team will resolve non succeeding proposals."*

[N03] Wallet does not renounce minting privileges over DecimalPaddingToken contract

The `doEnableToken` function of the `Wallet` contract takes care of deploying new `DecimalPaddingToken` contracts when necessary. During deployment, since the `DecimalPaddingToken` contract is an `ERC20Mintable`, it will set the `Wallet` contract as an address with minting privileges.

having the `Ownable` contract ensures the minter role.

Update: *Fixed in PR #529, where the `DecimalPaddingToken` contract no longer inherits from `ERC20Mintable` or `ERC20Burnable`.*

[N04] Duplicate conditionals in the `calculateClosingData` function

There are two `if` statements with the same conditional in the `calculateClosingData` function. One begins on line 338, and the other begins on line 353. Consider combining these statements into one.

Update: *Fixed in PR #518.*

[N05] ensureClosingAccess function can be restricted to view

The `ensureClosingAccess()` function on line 287 of `Trading.sol` does not modify any state variables. Considering restricting the function to `view`.

Update: *Fixed in PR #487.*

[N06] Incorrect code comment

The comment on line 25 of `Wallet.sol` says `tokenAddress -> userAddress -> amount` but it should say `userAddress -> tokenAddress -> amount`

Update: *Fixed in PR #519.*

[N07] Lack of explicit visibility for some state variables and constants

Some state variables and constants do not have explicit visibility. See constants in `MessageProcessor`, the `replayNumberByUserAddress` in `ReplayTracker`, etc. To improve readability, consider making the visibility of all state variables and constants explicit throughout the code.

Update: *Fixed in PR #535.*

[N08] Typos

- exchange/Exchange.sol L86 `occure` should be `occur, occurred`
- exchange/Exchange.sol L187 `incase` should be `in case`
- exchange/Liquidity.sol L262 `recieve` should be `receive`
- exchange/Liquidity.sol L270 `recieve` should be `receive`
- exchange/Liquidity.sol L290 `recieve` should be `receive`
- exchange/Liquidity.sol L298 `recieve` should be `receive`
- exchange/Trading.sol L328 `exlusive` should be `exclusive`
- exchange/Trading.sol L440 `exlusive` should be `exclusive`
- exchange/VoteCasting.sol L66 `exludes` should be `excludes, exudes`
- external/WrappedETH.sol L7 `implemenetation` should be `implementation`
- FSToken/FsProxyAdmin.sol L6 `effectly` should be `effectively`
- incentives/Incentives.sol L323 `begining` should be `beginning`
- incentives/Incentives.sol L337 `payed` should be `paid`
- messageProcessor/MessageProcessor.sol L611 `recieve` should be `receive`
- messageProcessor/MessageProcessor.sol L620 `recieve` should be `receive`
- messageProcessor/MessageProcessor.sol L418 `withDrawer` should be `withdrawer`
- registry/IRegistryUpdateConsumer.sol L3 `objets` should be `objects`
- registry/KnowsRegistry.sol L7 `classs` should be `class`
- registry/Registry.sol L13 `Wether` should be `Weather, whether`
- voting/Voting.sol L34 `upgrate` should be `upgrade`
- voting/Voting.sol L61 `lenght` should be `length`
- voting/Voting.sol L137 `intial` should be `initial`
- wallet/DecimalPaddingToken.sol L27 `correspoding` should be `corresponding`
- wallet/DecimalPaddingToken.sol L36 `correspoding` should be `corresponding`
- wallet/Wallet.sol L120 `retuns` should be `returns`
- wallet/Wallet.sol L195 `to large` should be `too large`
- wallet/Wallet.sol L235 `widthdrawl` should be `withdraw, withdrawal`
- wallet/InternalWallet.sol L4 `only every` should be `only ever`

Update: *Fixed in PR #516.*

[N09] Unused Parameters

Update: *Fixed in PR #517.*

[N10] Variable Naming Issues

- The `tokenAddress` parameter of the `Wallet.doWithdraw(address tokenAddress, uint256 amount, bool isWrapped, address to)` function is always an "internal token address". Consider renaming it to `_internalTokenAddress`.
- In MessageProcessor.sol, the `verifyMaxmarketAssetPrice` and `verifyMinmarketAssetPrice` function names are not using camelCase.

Update: *Fixed in PR #524.*

Conclusions

0 critical and 3 high severity issues were found. Several recommendations were made to improve the project's overall quality and reduce its attack surface.

Related Posts

Beefy Zap Audit

OpenBrush Contracts Library Security Review

Linea Bridge Audit

Security Audits

Security Audits

Security Audits

OpenZeppelin

Defender Platform

Secure Code & Audit
Secure Deploy
Threat Monitoring
Incident Response
Operation and Automation

Services

Smart Contract Security Audit
Incident Response
Zero Knowledge Proof Practice

Learn

Docs
Ethernaut CTF
Blog

Company

About us
Jobs
Blog

Contracts Library

Docs

Privacy | Terms of Use