version 1.0.0

—

Presented by

Manuel Araoz
CTO, Zeppelin

March 12th, 2018

zeppelin audits

01. Introduction

This document includes the results of the audit performed by the Zeppelin team on the Augur Core project, at the request of the Augur team. The audited code can be found in the public augur-core Github repository, and the version used for this report is commit

`3b5a63d372d205a0214e3061293d5bca0fd5636a`

Some fixed and partially fixed issues from a previous audit can also be found in Appendix A.

The goal of this audit is to review Augur's solidity implementation for its decentralized prediction market, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

— Disclaimer

Note that as of the date of publishing, the contents of this document reflect the current understanding of known security patterns and state of the art regarding smart contract security. Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

— Methodology

Augur's whitepaper was analyzed and synthesized into a series of specifications and expected behaviours, and the codebase was studied in detail in order to acquire a clear impression of how such specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

— Structure of the document

This report contains a list of issues and comments on different aspects of the project: General Observations, Trading, Reporting, Forking, and Miscellaneous. Each issue is assigned a severity level based on the potential impact of the issue, as well as a small example to reproduce it and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

For this audit, we used the following sources of truth about how the Augur Core system should work:

> http://docs.augur.net/
> Whitepaper
> https://augur.stackexchange.com/

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Augur team or reported an issue.

02. About Zeppelin

Zeppelin Solutions is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Zeppelin Solutions has developed industry security standards for designing and deploying smart contract systems.

Zeppelin Solutions is the creator, maintainer, and major contributor of OpenZeppelin, the standard framework for secure smart contract development, maintained by a community of 3000+ developers distributed around the globe.

Over $600 million have been raised with Zeppelin's audited smart contracts. Clients include Golem, Brave, Augur, Blockchain Capital, Status, Cosmos, and Storj, among others.

More info at: https://zeppelin.solutions

03. Severity level reference

Every issue in this report was assigned a severity level from the following:

CRITICAL

Critical severity issues need to be fixed as soon as possible.

HIGH

High severity issues will probably bring problems and should be fixed.

MEDIUM

Medium severity issues could potentially bring problems and should eventually be fixed.

LOW

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

04. List of issues by severity

CRITICAL

HIGH

MEDIUM

LOW

05. List of issues by topic

APPENDIX A - Fixed and partially fixed issues

06. Issue Descriptions and Recommendations

A. General Observations

CRITICAL

Use safe math (new)

Arithmetic operations on integers may overflow silently causing bugs.
As a critical example, the function `derivePayoutDistributionHash` of the `Market` contract uses an unsafe addition. This operation can overflow and still be equal to the required number of ticks, yielding an invalid set of payout numerators. This can be used by an attacker to manipulate the Universe open interest via `calculateProceeds` of the `ClaimTradingProceeds` contract. This can reduce the open interest to zero, effectively freezing the universe by causing all subsequent calls to reducing open interest to throw.

46 additional unsafe operations were found:

```
source/contracts/libraries/math/RunningAverage.sol
  12:15    Avoid using arithmetic operation '/' directly.

source/contracts/libraries/token/StandardToken.sol
  19:53    Avoid using arithmetic operation '-' directly.

source/contracts/reporting/DisputeCrowdsourcer.sol
  34:28    Avoid using arithmetic operation '/' directly.
  34:28    Avoid using arithmetic operation '*' directly.
  35:35    Avoid using arithmetic operation '/' directly.
  35:35    Avoid using arithmetic operation '*' directly.
  49:30    Avoid using arithmetic operation '-' directly.
  65:35    Avoid using arithmetic operation '/' directly.
  65:35    Avoid using arithmetic operation '*' directly.

source/contracts/reporting/FeeWindow.sol
  97:31    Avoid using arithmetic operation '+' directly.

source/contracts/reporting/Market.sol
  34:58     Avoid using arithmetic operation '/' directly.
  35:47     Avoid using arithmetic operation '-' directly.
  76:21     Avoid using arithmetic operation '/' directly.
  87:40     Avoid using arithmetic operation '+' directly.
```

```
  176:53    Avoid using arithmetic operation '-' directly.
  215:65    Avoid using arithmetic operation '/' directly.
  252:28    Avoid using arithmetic operation '-' directly.
  252:28    Avoid using arithmetic operation '*' directly.
  252:50    Avoid using arithmetic operation '*' directly.
  392:28    Avoid using arithmetic operation '-' directly.
  473:58    Avoid using arithmetic operation '+' directly.
  479:25    Avoid using arithmetic operation '+' directly.
  480:25    Avoid using arithmetic operation '+' directly.

source/contracts/reporting/RepPriceOracle.sol
  10:40    Avoid using arithmetic operation '*' directly.

source/contracts/reporting/Reporting.sol
  10:50    Avoid using arithmetic operation '*' directly.
  10:50    Avoid using arithmetic operation '*' directly.
  12:53    Avoid using arithmetic operation '/' directly.
  13:51    Avoid using arithmetic operation '/' directly.

source/contracts/reporting/ReputationToken.sol
  76:25    Avoid using arithmetic operation '/' directly.

source/contracts/reporting/Universe.sol
  296:41    Avoid using arithmetic operation '*' directly.
  301:15    Avoid using arithmetic operation '*' directly.
  364:27    Avoid using arithmetic operation '/' directly.
  366:24    Avoid using arithmetic operation '+' directly.
  370:26    Avoid using arithmetic operation '/' directly.
  373:24    Avoid using arithmetic operation '+' directly.
  379:21    Avoid using arithmetic operation '-' directly.
  403:33    Avoid using arithmetic operation '/' directly.
  403:33    Avoid using arithmetic operation '*' directly.
  425:45    Avoid using arithmetic operation '*' directly.
  425:45    Avoid using arithmetic operation '*' directly.

source/contracts/trading/FillOrder.sol
  321:67    Avoid using arithmetic operation '-' directly.
  322:44    Avoid using arithmetic operation '+' directly.
  326:66    Avoid using arithmetic operation '-' directly.
  335:35    Avoid using arithmetic operation '-' directly.

source/contracts/trading/Orders.sol
  182:20    Avoid using arithmetic operation '+' directly.
  185:20    Avoid using arithmetic operation '+' directly.

source/contracts/trading/TradingEscapeHatch.sol
  70:36    Avoid using arithmetic operation '/' directly.
  70:36    Avoid using arithmetic operation '-' directly.
```

Consider using the existing `SafeMathUint256` and `SafeMathInt256` libraries for all arithmetic operations.

***Update**: Fixed in [f164ac53644795072753c99bb7e391f7b2a42493](f164ac53644795072753c99bb7e391f7b2a42493).*

HIGH

Extractable functionality is not necessary and error prone

Augur's codebase presents an horizontal feature that allows many contracts to return tokens or Ether back to the owner in case they were wrongly transferred to them. This functionality is held within the `Extractable` contract, and is extended by many contracts like `Cash`, `Orders`, `Universe`, among others.
Given many of the contracts that inherit said functionality can hold some token balances, they need to declare a set of protected tokens to exclude those balances from the `Extractable` functionality. This is error prone, and relies entirely on the developer to have declared that set properly.
For example, `Cash` and `FeeToken` are not declared as protected tokens for the `InitialReporter` and the `DisputeCrowdsourcer` contracts, although these contracts can own said tokens balances. The same happens for the `Cash` contract, it doesn't mark Ether as a protected token. This means that the `Controller` can extract these tokens at will.
As shown, this functionality can cause several problems if it is not well implemented. Given it is not a core functionality for Augur, consider removing the whole feature from Augur's codebase to reduce the attack surface.

***Update**: Fixed in 53c15f956580caa67771e60b5fa2cc5d76474e82.*

MEDIUM

Favor pull payments over push payments (new)

Many ETH transfers are executed using a low level call. This allows the recipient to execute arbitrary code upon the transfer (due to the gas stipend allocated), and also to throw upon receiving a payment, thus blocking the application flow. For more info on this problem, please see `this note`.

10 places were found were a low level call is triggered:

- `DisputeCrowdsourcer#redeem (L39)`
- `InitialReporter#redeem (L35)`
- `InitialReporter#withdrawInEmergency (L62)`

- `FillOrder#fillOrder` (L390)
- `CashAutoConverter#cashToEth` (L35)
- `FeeWindow#redeemInternal` (L124)
- `ClaimTradingProceeds#claimTradingProceeds` (L49)
- `Market#initialize` (L89)
- `Market#withdrawInEmergency` (L314)
- `Mailbox#withdrawEther` (L33)

Even though no attacks were found regarding this issue, consider using transfer instead of a low-level call in all these scenarios. Moreover, most cases use `Cash#withdrawEtherTo` which in turn makes the low-level call. Consider using a `Cash` token transfer instead.

***Update**: Addressed in 65561b0a0b7064c9f83bad6b8f9883911576ce65. Augur's comments: "This is intentional in order to support smart wallets, but worth discussing again to make sure we agree with the tradeoff. The note about transferring Cash tokens would require users interact with Cash, which we've explicitly decided is too onerous a UX".*

MEDIUM

Integer index types are unnecessarily small

There are several places where a for loop is done using an index variable of type `uint8`. In some cases, like `Market#isContainerForReportingParticipant`, the iterated array is guaranteed to be of size less than 256. In others, such as `Universe#redeemStake`, in which the array is an external input, the array could have more than 256 elements. If this happens, the loop will get stuck due to the index variable overflowing after exceeding 255, the maximum number for integers of that size.

Regardless, the EVM word size is 256 bits, so there is no additional benefit to using a smaller integer variable. There is, in fact, an additional cost due to the operations required to simulate overflow semantics.

Consider using `uint256` for all loop index variables.

***Update**: Fixed in 3dc92eff16dc121e38abd0ac11447ca135bbcdc7.*

MEDIUM

Unbound iteration in arrays (new)

There are several loops in the contracts which can eventually grow so large as to make future operations of the contract cost too much gas to fit in a block. Additionally, gas exhaustion could be used as an exploit to block the application flow.

For example, the `finishedCrowdsourcingDisputeBond` function of the `Market` contract iterates over an unbounded participants array.

Consider ensuring that array maximum lengths are checked on iteration.

***Update**: Fixed in 16065f39efedadf11ca3ccaaa3bc4f04cb97b143.*

MEDIUM

Unbound iteration in arrays

There are several loops in the contracts which can eventually grow so large as to make future operations of the contract cost too much gas to fit in a block. Additionally, gas exhaustion could be used as an exploit to block the application flow. Furthermore, if the index used for iteration is an 8-bit integer, the array's length must be checked to be under 256, to prevent infinite loops.

For example, `Market.sol#derivePayoutDistributionHash` in `L415` iterates over an unbounded array using an `uint8`.
Consider reviewing all for-loops and ensure that array maximum lengths are checked on iteration.

***Update:** The Augur team decided not to fix this problem: "These were reviewed and found to have implicit bounds".*

LOW

Naming issues

Some functions in the Augur codebase are named in a way that does not describe their actual behavior. For example:

- The `Market` contract has a `getTotalStake` function that actually returns a subset of the total amount of staked tokens. Consider the name `getParticipantsStake`.
- The function `assertReputationTokenIsLegit` of the `ReputationToken` contract actually checks that a given `ReputationToken` is a sibling, i.e. that it belongs to a child universe of its parent universe. This means that a valid `ReputationToken` can still return false to this function. Consider the name `assertReputationTokenIsLegitSibling`.

Consider reviewing all function names and fixing those that don't describe their actual behavior to reduce confusion.

***Update**: Fixed in 137e28c606dbce1363f315e23d2c610465c9a281.*

LOW

Repeated code for factory contracts

There is a lot of repeated or very similar code in the factory folder, for Factory contracts. This makes any change to the way factories work tedious to implement and error-prone.

Consider creating a low-level generic factory contract in assembly, and retaining only interfaces for the particular implementations.

***Update**: Augur team decided not to follow this suggestion.*

LOW

Unused boolean return values

Many functions in the codebase return boolean values which are never used. For example the `BaseReportingParticipant` contract defines a `fork` function that returns a hardcoded true value. This function is called from `Market#finishedCrowdsourcingDisputeBond` but the return value is never used. Another example are the functions `ethToCash` and `cashToEth` of the `CashAutoConverter` contract.

Consider removing these unused return values to avoid confusion and reduce the amount of code to be deployed.

***Update**: Augur team decided not to follow this suggestion.*

LOW

Unsolved TODO comments

Some `TODO` comments were found in the contracts:

```
source/contracts/Controller.sol
  29:4    'TODO' comment.

source/contracts/reporting/ReputationToken.sol
  82:4     'AUDIT' comment.
  88:4     'AUDIT' comment.
  94:4     'AUDIT' comment.
  100:4    'AUDIT' comment.

source/contracts/trading/ClaimTradingProceeds.sol
  16:0    'AUDIT' comment.

source/contracts/trading/FillOrder.sol
  66:8     'TODO' comment.
  393:8    'AUDIT' comment.

source/contracts/trading/ShareToken.sol
  15:4    'FIXME' comment.
```

Consider having all these reminders removed by the time these contracts are deployed to avoid confusion.

***Update**: Fixed in 137e28c606dbce1363f315e23d2c610465c9a281.*

LOW

Instances of Map contract left in blockchain storage

The `Map` contract provides a mapping with a count of items. It is used, for example, in the `Market` contract to save all of the `DisputeCrowdsourcer` instances corresponding to each given payout distribution. When the mapping needs to be deleted because the crowdsourcing dispute was finished, it is simply overwritten with the address of a new instance of `Map`. This is in fact mentioned in the documentation for Map: *"allows for a clean*

way to clear an existing map by simply creating a new one". However, this results in potentially several "garbage" `Map` instances left behind in blockchain storage.

Consider clearing the `Map` storage when it becomes unused. To do this, define a new function in `Map` that calls `selfdestruct`, and call it when an instance becomes unused. If there are more complex situations such as a `Map` instance being necessary by more than one contract, consider a reference counting mechanism.

***Update**: Augur team decided not to follow this suggestion.*

LOW

Unused Set library

The Augur codebase includes a `library` to manage data sets. Although this library is `imported` and `declared` in the `FeeWindow` contract, it is never used.

Consider removing this whole library to reduce the amount of code deployed and the attack surface.

***Update**: Fixed in 137e28c606dbce1363f315e23d2c610465c9a281.*

LOW

Inconsistent usage of getter functions and state variables

Many contracts define getter functions to abstract the way some behavior is implemented. However, there are some cases where these getters are not used and state variables are queried manually.

For example, the `Universe` contract declares a `forkingMarket` variable to keep track of the forking `Market`. Then, it defines a function called `isForking` to tell whether said `Universe` is forking or not. However there are some places where the `forkingMarket` variable is queried manually within the Universe contract. The same occurs within some Market functions.

Another example is the `supply` variable defined in the `BasicToken` contract to keep track of the total supply of it. This variable is accessed manually in the `redeem` and `withdrawInEmergency` functions of the `DisputeCrowdsourcers` contract.

This pattern is error-prone. Consider using the defined getter functions consistently instead of related state variables, to avoid confusion and reduce the attack surface.

***Update**: Fixed in c13fa15ab625b070754df87375d2a40db60c5e14.*

LOW

Use a standard toolchain for building contracts

Script `CompileSolidity.ts` manually walks the Solidity contracts in the project and builds them for deployment purposes. The same is `reimplemented` in conftest.py as part of the test suite. Furthermore, the methods executed from the deployment scripts are defined manually in `ContractInterfaces.ts`, which is cumbersome and highly error prone (for instance, method `StandardToken#approve_` should be marked as constant in its ABI for consistency).

Instead of reimplementing these features, consider using an existing build tool such as `truffle`, to simplify operations and leverage other tools compatible with it.

In particular, tools such as `solidity-coverage`, which require a standard setup for both compilation and for running the tests via testrpc, could be used to measure automated tests code coverage and detect untested paths.

Alternatively, an ad-hoc coverage solution for this codebase could be implemented:

1. Instrument contracts using solidity-coverage
2. Compile them from conftest.py
3. Install a `log_listener` in the testing chain to capture all events from all tests
4. Output all captured events during the tests run to an `allFiredEvents` file
5. Use solidity-coverage to generate the coverage info from the `allFiredEvents` file

***Update**: The Augur team prefers to keep using their custom tools.*

LOW

No assertions for detecting broken invariants

Augur stores important data elements in its state such as:

- The balance of tokens
- The open interest
- The list of orders in the order book
- The universe tree structure
- etc.

Many elements of such state maintain specific relationships of value between each other and define the integrity of Augur's state. Some examples are:

- There is a particular relationship between the amount of minted `ShareTokens` and the value escrowed in a market.
- The value of `openInterestInAttoEth` must exactly match the sum of all escrowed amounts at different markets.
- The size of the dispute bond for all participants must ensure a 50% ROI for the winning outcome.

Consider implementing a mechanism to assert the integrity of invariants via:

1) Assertions in solidity carried out after important operations that change state, or
2) External assertions that read the state and evaluate its integrity.

With such a mechanism in place, Augur would have a clearer understanding of when to apply emergency mechanisms, when to upgrade contracts, etc.

Update: *The Augur team confirmed they will add this kind of assertions in a near future.*

Update 2*: Fixed in 51b78bc40d1756a1af69f94f86fee495a6f0e2b7.*

LOW

Install OpenZeppelin via NPM

`Ownable`, `ERC20Basic`, `ERC20`, `BasicToken`, and `StandardToken` appear to have been copied from the OpenZeppelin repository. This violates OpenZeppelin's MIT license, which requires the license and copyright notice to be included if its code is used, and makes it difficult and error-prone to update to a more recent version.

Consider following the recommended way to use OpenZeppelin contracts, which is via the zeppelin-solidity NPM package. This allows for any bug-fixes to be easily integrated into the codebase.

***Update**: Even though a license file was included in d075d9c0176a08fea521ff31a373776f134828d5, the Augur team made clear that managing Solidity dependencies with NPM is not aligned with their plans.*

LOW

OpenZeppelin standard tokens were modified

Additionally to copying OpenZeppelin's contracts instead of installing them via NPM, some of them were modified. For example, the contract `StandardToken` was modified to implement the eternal approval functionality, the BasicToken contract was modified adding an internalTransfer method.

This is not the way OpenZeppelin standard contracts should be used. Making changes to open-source libraries, instead of using them as is, can be dangerous and won't allow you to integrate bug-fixes into the codebase easily.

Consider inheriting from the OpenZeppelin standard contracts to implement additional functionality.

***Update**: Even though a note listing which files were modified was included in d075d9c0176a08fea521ff31a373776f134828d5, the Augur team prefers to use their own version of OpenZeppelin contracts.*

LOW

Outdated OpenZeppelin's contracts

Apart from copying and modifying some OpenZeppelin's contracts, these seem to be outdated. For example, the contract `StandardToken` does not include the `increase approval` and `decrease approval` mitigations included since one of the latest releases of OpenZeppelin.

Consider using the version of the contracts included in the latest release of OpenZeppelin.

***Update**: Partially fixed in 534b92e5f12ad5974572d4ecb2abf0f524ccb36c.*

LOW

Outdated documentation

Augur public documentation is outdated: *"These docs are currently being updated as we approach the launch of Augur. The Augur Team plans to have these docs fully updated prior to launching Augur"*.

Consider updating the documentation as mentioned, to make sure users understand how Augur works without confusion.

***Update**: The Augur team is still working on this.*

B. Trading

MEDIUM

Users are allowed to place orders for a market independently of their state

The `Trade` contract allows users to place a short or long orders through the `publicSell` or `publicBuy` functions respectively. Both alternatives will end attempting to fulfill any existing order with the incoming one. However, there is no precondition validating that the given Market is not finalized, or being disputed, or actually in a state that allow users to place orders.

The same thing happens with the `publicCreateOrder` function of the `CreateOrder` contract. It allows users to create an order without validating the market state. Event this may not be a vulnerability per se, it won't prevent people from wasting their money due to an inconsistent market.

Consider adding a precondition in the `Trade#trade` function, that is where all the Trade public functions converge, to validate that the given Market is in a valid state. Add an additional precondition to the `CreateOrder#createOrder` function to check this too.

***Update**: The Augur team considers this a valid scenario.*

MEDIUM

Unclear relation between `MIN_ORDER_VALUE` and `MINIMUM_GAS_NEEDED`

CreateOrder checks that `order_price * order_amount > MIN_ORDER_VALUE`. This makes spamming orders to the order book expensive for an attacker.

Trade's fillBestOrder, has `FillOrder` perform a series of expensive state changing operations that cancels bids and asks on a one-to-one basis using a while loop, and to avoid this loop from running out of gas and reverting, `msg.gas >= MINIMUM_GAS_NEEDED` is used.

It is important to note that the relationship between `MIN_ORDER_VALUE` and `MINIMUM_GAS_NEEDED` is a very delicate one, and must be meticulously balanced. Any mis-calibration in this value pair will result in making a spam attack on the order book feasible.

We recommend that a test for this specific situation is implemented in order to ensure that the relation between `MIN_ORDER_VALUE` and `MINIMUM_GAS_NEEDED` is correctly set. Alternatively, consider adding the ability to manually adjust such values.

***Update**: The Augur team clarified our understanding, and it's not an issue.*

MEDIUM

Reentrancy risk in FillOrder

Consider the following situation: An attacker performs a trade operation such as a `publicSell` which fills an order. Such operation will call `fillOrder` which will be entered a first time, resulting in the selling of complete sets via `tradeMakerSharesForFillerShares`, which will send ether to the attacker.

Now, the attacker's malicious fallback method can't re-enter fillOrder via publicSell again because it is protected from re-entrancy with the `nonReentrant` modifier, but it can re-enter via `publicFillOrder`, thus entering `fillOrder` a second time. This cannot be done consecutively because publicFillOrder itself is protected by another `nonReentrant` guard.

Even if the damage that can be done in the situation described above is not significant, it illustrates how internal methods that can be accessed from different locations are moderately exposed to complex re-entrancy attacks. Using `nonReentrant` guards on the public methods that reach such internal methods may not be enough for situations that reach a given level of complexity.

We recommend that internal methods are protected by `nonReentrant` guards instead of the public methods that use them.

***Update**: The Augur team mentioned this issue was fixed avoiding calls to external accounts. However, this is still an issue. Notice that an external call is performed through the `fillOrder` function of the `FillOrder` contract.*

***Update 2**: Fixed in 39718842e8362366b4af167f4e2e8ffbbd65354a.*

LOW

Orders are vulnerable to front-running

Calls to `Trade#publicTakeBestOrder` drives the market price up/down (see Trade `L59`) as orders are filled. Upon observing a call to `publicTakeBestOrder` in a pending transaction, an attacker could call that function with a higher gas price, in order to front-run the buyer. This allows him to purchase the shares before the price increase from the original transaction.

This issue is mitigated by having a `_price bound` in the call, which limits up to how much the original caller is willing to pay to fill the orders. A narrow `_price` margin would abort the purchase if it gets front-runned. However, it still signals the intention from the original buyer to make the purchase, which the attacker can leverage.

See this post for an explanation on frontrunning on the Bancor protocol, and a list of possible solutions and their trade-offs. As a simple solutions, consider adding a `maxGasPrice` check to mitigate the issue, which only allows front-running by malicious miners.

***Update**: The Augur team decided not to fix this issue: "We're getting rid of market orders in the UI so this will only be available initially via the API, so anyone putting themselves in this position was extremely aware of their actions"*

LOW

Basic token implementation allows transfers to the zero address

The `BasicToken` contract is a basic implementation of the ERC20 standard extended by all the different token contracts of Augur. The `transfer` function implemented by this contract allows to transfer funds to the zero address. This could derive in undesired token transfers.

Consider adding a precondition to validate the recipient is not the zero address. To keep supporting burn operations, which are done as a transfer to the zero address, consider using the `VariableSupplyToken` contract, or performing a transfer to another hardcoded address (such as `0x1`).

***Update**: The Augur team decided not to fix this issue.*

C. Reporting

HIGH

Non-potential-winning dispute crowdsourcers can redeem their REP tokens

When a `DisputeCrowdsourcer` reaches its size, it is considered as a new potential winning participant and the rest of the `crowdsourcers` of that `FeeWindow` are disavowed. However, there is no precondition to prevent disavowed `crowdsourcers` to redeem their REP tokens and get proportional funds in return. This can be done since anyone can call the `redeem` function for a disavowed `DisputeCrowdsourcer`. The whitepaper is unclear about how the system should behave in this scenario.

Consider excluding disavowed `crowdsourcers` in the `redeemInternal` function of the `FeeWindow` where the proportional funds are calculated.

***Update**: Augur team informed this is intended behavior. Whitepaper updated.*

HIGH

Market number of ticks can be zero

There is no precondition to check that the `numTicks` for a `Market` are a non-zero number, which can be set up by creating a scalar market in `Universe#createScalarMarket`. This causes odd behaviours in the Market, such as having only one possible valid outcome (all zeros), not being able to create orders for that Market (see `Order.sol L59`), or being able to purchase complete sets of shares at zero cost (see `CompleteSets.sol L35`).

Consider adding a precondition to check that the given number of ticks is greater than zero in the Market constructor.

***Update**: Fixed in 3ee0fae51d68c2d23d8c3ac56e5659fcfc6fdeb2.*

HIGH

Self-reference in market nudging mechanism

The market nudging mechanism implies that Augur uses self-reference to operate. It uses one of its own markets to report on the price of REP, and then uses such reported price to determine the fees reporters will have on future windows. This self-reference seems dangerous in general, as the dynamics are not clear, but we found some concrete problems.

For example, reporters are incentivized to report on smaller prices, given this would make reporting fees higher, making REP tokens gain value with time.

Despite the whitepaper's claim that the game theory backing this nudging mechanism holds, the fact that such mechanism is implemented in a completely automated form is concerning. Consider making the nudging mechanism customizable by the Augur team at least during the initial dev mode phase.

However, when looking at the implementation, found in contract `RepPriceOracle`, we see that the nudging mechanism is not implemented via an Augur market, but from an external source, we recommended. As such, it's not affected by the problem described above, but the issue is pointed out to signal concern over the designed mechanism.

***Update**: The Augur team decided not to include this fix by now, since the nudging mechanism is currently implemented in a centralized way.*

MEDIUM

Markets can be initially reported in a locked universe

It is unclear from the white paper whether a `Market` can be *initially* reported in a locked Universe, as no reporting should occur during a fork, and no rewards should be paid (initial reporting pays out bonds).

Consider either clarifying on the white paper whether initial reporting is accepted during a fork, or add a restriction in `doInitialReport` to prevent invocation during forks.

Update: *Augur team clarified this is intended and will update the whitepaper to specify this behavior.*

LOW

Lack of Report abstraction

Large part of the reporting module needs the report itself to carry out different functionalities. However, reports are not modeled explicitly forcing many functions to send and receive the attributes of every report separately.

For instance, many functions receive the payout numerators set with the invalid attribute and then call the `Market` contract to get the corresponding distribution hash. For example, the `doInitialReport`, `contribute` and `createChildUniverse` handle a payout numerators set with the invalid attribute and call the `derivePayoutDistributionHash` of the `Market` to carry out their behavior.

Consider modelling a report explicitly including all the needed attributes to improve the interfaces and reduce code complexity.

Update*: The Augur team decided not to fix this issue*

LOW

Universe open interest is not decremented in bad times

Whenever shares are exchanged for ETH in a market, such as in `ClaimTradingProceeds#claimTradingProceeds` or `CompleteSets#sellCompleteSets`, the corresponding universe open interest is decremented via a call to `decrementOpenInterest` (see `ClaimTradingProceeds` `L38` and `CompleteSets` `L57`).

However, when shares are burned in exchange for ETH in `TradingEscapeHatch#claimSharesInUpdate`, the universe open interest is not decremented, and may reflect an invalid value.

Note that this issue is mitigated by the fact that the trading escape hatch can only be used in the event of an emergency stop.

Consider adding a call to `decrementOpenInterest` with `_amountToTransfer` value to reduce the open interest as needed.

***Update**: The Augur team considers this a valid scenario in order to keep this logic as simple as possible.*

***Update 2**: Fixed in [c13fa15ab625b070754df87375d2a40db60c5e14](c13fa15ab625b070754df87375d2a40db60c5e14).*

D. Forking

CRITICAL

An attacker can manipulate the tentative winning outcome in a fork

The `Universe` contract has a function named `updateTentativeWinningChildUniverse` that is called every time some REP tokens are migrated to a child Universe to keep track of the tentative winning payout distribution and finalize the forking market if possible. This function declares an `uint256` local variable called `_currentTentativeWinningChildUniverseRepMigrated` to store the amount of REP tokens migrated to the current tentative winning child `Universe` temporary. However, said amount is queried but never stored in the local variable mentioned above. **This means that any payout distribution hash could be tracked as the tentative winning one**, given it just requires to have an amount of REP tokens migrated greater than the amount stored in the `_currentTentativeWinningChildUniverseRepMigrated` variable, which will always be true.

Then, notice that the function `getWinningChildUniverse` would assume that the tentative winning payout distribution is the final one if the fork end time has passed.

Fix the `updateTentativeWinningChildUniverse` logic to use the local variable `_currentTentativeWinningChildUniverseRepMigrated` correctly.

***Update**: Fixed in 1dbaa4307a699729bf33133f09d4d23c9c425237.*

CRITICAL

An attacker can prevent forking-market traders from claiming their fees

The `Market` contract has a `finalizeFork` function which is called every time a forking market is finalized. This function sets the `finalizationTime` state variable of the `Market` with the current timestamp.

On the other hand, the `ClaimTradingProceeds` contract has only one public function allowing traders to claim their fees three days after a Market was finalized. It performs a precondition to check that it's been at least 3 days from the finalization time.

Given the `finalizeFork` function is public and there are no preconditions to guarantee it is not called many times, an attacker can call it more than once to get the `finalizationTime` updated every time. This means, they will need to call this function once every three days for a forking market to prevent traders from claiming their fees.

Add a precondition in the `finalizeFork` function to prevent anyone from calling it more than once.

***Update**: Fixed in 1dbaa4307a699729bf33133f09d4d23c9c425237.*

CRITICAL

Markets can be migrated after finalization

Function `Market#migrateThroughOneFork` migrates the market from its current Universe to the winning one in the event of a fork, removing all reports except for the initial report.

If the Market was already finalized, this method can still be invoked, which will remove all reports except for the initial one, effectively changing the return value of `getWinningReportingParticipant` and `getWinningPayoutNumerator` back to the initial report, which are used to determine how share tokens fees are paid out (see `ClaimTradingProceeds#calculateProceeds`). Furthermore, since

`winningPayoutDistributionHash` is not cleared out, the market is kept finalized and cannot be disputed.

The same happens when calling `Market#disavowCrowdsourcers`, which clears out all participants except for the initial reporter, effectively changing the winning payout numerator.

Check that a market is not finalized when attempting to migrate it.

***Update**: Fixed in 1dbaa4307a699729bf33133f09d4d23c9c425237.*

MEDIUM

Forking market can be migrated

The `migrateThroughOneFork` function of the `Market` contract has no checks to ensure that it cannot be called for the forking market itself. The only condition preventing this is the call to _initialParticipant.resetReportTimestamp, which fails if the initial participant was `forked` beforehand. Note that the contracts do not enforce `fork` to be called in the reporting participants the event of a fork, meaning that this call could be missed.

Consider adding a check to ensure that the forking market cannot be migrated.

***Update**: Fixed in 1dbaa4307a699729bf33133f09d4d23c9c425237.*

MEDIUM

Fork values for child universes must be manually updated

Each Universe has its own `reputation goal`, `fork threshold` and `initial report value`. When a fork occurs many child Universes are created and this values

need to be recalculated since they are derived from the REP total supply. The function in charge of updating this values is `updateForkValues`.

Given that `updateForkValues` is not called from any contract, it must be called manually. Moreover, it has to be called every time some REP tokens are migrated to a child Universe. This implies that the platform relies on manual work which is error prone. For example, if the function is not called, the fork reputation goal won't be updated causing new forks to be cheap.

Consider calling this function within the REP tokens migration flow.

***Update**: Augur team replied: "Doing this in the suggested automated fashion would be impossible as it involves iterating an unbounded array (all sibling universes)".*

LOW

Markets can fork into more than N+1 universes, N being the number of outcomes (new)

Based on the whitepaper: *"When a market forks, new universes are created. Forking creates a new child universe for each possible outcome of the forking market (including Invalid). For example, a "binary" market has 3 possible outcomes: A, B, and Invalid."* (Augur Whitepaper, 9. Fork, Page 6)

However, any participant can report multiple valid outcomes through the `contribute` function of the `Market` contract. This means there are multiple combinations of possible reports for a binary market, which in turn means multiple child universes can be created in case of a fork.

Even this is not a real problem, the way the code works does not correspond to the one described in the documentation. Consider updating the documentation to reflect actual behavior.

***Update**: Augur team will update the whitepaper.*

LOW

Markets may fork in more than N+1 universes, N being the number of outcomes

Reporters can stake their REP tokens buying `StakeTokens` for a market. This tokens are created using an array of `payoutNumerators`, which is required to have the same length as the number of outcomes, and that the sum of all the contained values is equal to the number of ticks of the given *market*. This is how a dispute works too through the `disputeDesignatedReport` or the `disputeFirstReporters` functions of the `Market` contract, buying `StakeTokens` for a `payoutNumerators` set. Then, there are a lot of possible `StakeTokens` for the same market, at least more than the number of outcomes.

Besides, based on the documentation when a market forks, many child universes are created as the number of outcomes with the possibility of one more universe in case of an invalid outcome.

However, when a fork occurs, child universes are created based on , which means that there may be an amount greater than the number of outcomes. For example, suppose a binary market with 1000 Ticks, it has three different possible outcomes: A, B and Invalid. Then, we can stake REP tokens on the outcome A submitting a `payoutNumerators` set like `[1000, 0]`, and we can also stake tokens on B submitting `[0, 1000]`. However, if we submit a `payoutNumerators` set like `[500, 500]`, it will pass the precondition checks, meaning that if a fork happens, more than three possible universes will be created.

Even this is not a real problem, the way the code works does not correspond to the one described in the documentation. Consider updating the documentation to reflect actual behavior.

***Update**: The Augur team confirmed they are going to update the whitepaper to match the actual behavior.*

LOW

When a market forks, stake tokens and disputes of other markets are reset

When a market forks, the rest of the markets are set to the `AWAITING_FORK_MIGRATION` state. In order to move forward, the `migrateThroughOneFork` function of each market has to be called. This function migrates a market to the winning universe, and it also clears its reference to the `StakeTokens` and to the designated, first and last dispute bonds.

Moreover, there is a public function called `disavowTokens` in the market contract that seems to be unused since the migrate function is already carrying out that logic.

Then, reporters would need to call the `redeemDisavowedTokens` function of the `StakeToken` contract to receive their REP tokens back. They also need to call the `withdrawDisavowedTokens` function of the `DisputeBond` contract to do so in case of a dispute.

However, if reporters forget to call any of those functions, they won't be able to claim their reporting fees, since market won't recognize their `StakeTokens` based on the preconditions needed to redeem them. This also means that reporters will have to re-stake their REP tokens, or re-dispute an outcome.

Consider implementing an alternative flow to allow users to report on a fork without getting their `StakeTokens` reset.

***Update**: The Augur team confirmed they will not change how this works now, but doesn't discard reviewing it in the future.*

***Update 2**: Intended. This is done currently so that more REP may participate in the fork.*

D. Miscellaneous

MEDIUM

Trading contracts upgradeability may become useless

One part of the deployment script is to whitelist all the trading contracts. Then, during dev mode, the `Controller` owner will be the only address whitelisted besides all these contracts. Once dev mode is turned off, the `Controller` owner is removed from the whitelist, meaning that only the trading contracts will be able to whitelist new contracts, but none of them implement such behavior.

This means that if the Controller owner does not whitelist another account controlled by the Augur team, then none of the trading contracts will be able to be upgraded, since no one will be able to whitelist it.

Consider either whitelisting an account controlled by the Augur team explicitly in the deployment script or to use the Controller owner address only for this purpose.

***Update**: Fixed in 821bee17bff898ef449fe498bc7a5180e95bcd64.*

MEDIUM

Controller does not guarantee that dev mode cannot be turned on again

"Augur will launch with three temporary security measures, or "training wheels". These measures will be removed once Augur's developer community feels that the platform has been thoroughly tested."

The dev mode is a particular status of the `Controller` contract and it is considered turned on only when the Controller's owner is whitelisted. Given the only way to whitelist a new address is from a whitelisted caller, and none of the whitelisted contracts implement such functionality, there must be a whitelisted account controlled by Augur to whitelist

future contracts. This means that the owner address can be whitelisted again in a future, turning on the dev mode again.

Consider implementing the dev mode explicitly through a state variable within the Controller contract instead of combining the whitelisting and ownable features to avoid confusion and hold what's mentioned in the whitepaper.

***Update**: Intended. Augur team replied: "This is not an issue as there needs to be an on chain audit to determine if we have actually relinquished control no matter the mechanism. Since we can effectively rewrite all the contracts until that point there is no simple way to determine malicious actions by us without a comprehensive on chain audit."*

LOW

Unchecked token transfers and approvals

All the tokens of the platform inherit from `StandardToken`, which is an implementation of the ERC20 token standard. This standard makes clear that the `transfer`, `transferFrom` and `approve` functions must return a boolean value to indicate whether the transaction was successful or not.

In the Augur codebase, every time a token transfer or approval is performed, its return value is not checked. For example, the `redeemInternal` function of the `FeeWindow` contract performs a REP token transaction without checking its result.

Consider wrapping all the transfers and approvals within a **`require`** statement.

***Update:** Fixed in 65561b0a0b7064c9f83bad6b8f9883911576ce65.*

LOW

ShareToken is unnecessarily whitelisted

The `Controller` contract includes a simple authorization flow that can be used by many contracts to call each other in a secure way. This feature is used by all the trading contracts, in fact, one part of the deployment script is to whitelist all of them.

However, the `ShareToken` contract is the only trading contract that doesn't call any other whitelisted contract, meaning that there is no need to have it whitelisted. Moreover, note

this contract is used through the `Delegator`, so it does not make sense to have the deployed implementation whitelisted.

Consider removing the ShareToken contract from the whitelisted contracts to reduce the attack surface.

***Update**: Fixed in 137e28c606dbce1363f315e23d2c610465c9a281.*

E. Notes & Additional Information

- Augur contracts are annotated with version pragma solidity 0.4.18. At the time of writing, the latest update released for that version was 5 months ago. Bear in mind that version 0.4.20 was released recently. Consider upgrading to a more recent version to enforce the use of an updated compiler.
- The `ShareToken` contract declares a decimals state variable of type uint256. Consider changing it to uint8 type to be ERC20 compliant.
- Some of the ERC20 token contracts declare a decimals state variable using a `uint256` type. For example `Cash`, `ShareToken` and `ReputationToken`. Consider changing those state variables to a `uint8` type following the ERC20 standard.
- FeeToken and FeeWindow contracts are ERC20 tokens, yet they do not define the optional `decimals`, `name` or `symbol` public fields. Consider adding these fields for better compatibility with user-facing software.
- Some unnecessary inheritance relations between contracts were found. For example, the `CancelOrder` contract inherits from `MarketValidator` unnecessarily. Consider removing those relations to reduce code complexity and avoid confusion.
- Some unused functions were found. For example, the `Controller` contract declares a function called `assertOnlySpecifiedCaller` that is never used. Consider removing all unused functions to reduce the attack surface and code complexity.
- Some code duplication was found in the Augur codebase. For example, the `Cash` contract defines two functions `withdrawEther` and `withdrawEtherTo` to carry out Ether transfers, which repeat most of the code. Consider reducing code duplication to reduce code complexity and the probability of making a mistake.
- There is a library called `DirectionExtentions` in the `FillOrder.sol` that is never used, although it is `declared` in the `FillOrder` contract. Consider dropping this unused library to reduce code complexity and the attack surface.

- The ERC20 specification suggests emitting a Transfer event from the address `0x0` when minting new tokens. Consider emitting such event in the `VariableSupplyToken#mint` function. Additionally, we suggest also emitting a Transfer event to the address 0x0 when burning tokens in the `VariableSupplyToken#burn` function.

- Keep in mind that there is a possible attack vector on the `approve/transferFrom` functionality of ERC20 tokens, described here. Consider using the mitigations implemented in OpenZeppelin's `StandardToken`.

- There is no precondition to avoid creating a `Market` using the zero address as the designated reporter. Consider adding a validation to check that the designated reporter is a non zero address in the `initialize` function of the `Market` contract.

- Many functions in the project return a hardcoded boolean when they are actually calling another function with a return value of their own. For example, the Augur contract defines a `trustedTransfer` function that returns always true besides what the internal call to `transferFrom` returns. Consider returning the result of those internal calls instead of a hardcoded boolean.

- Many functions handle strings through a `bytes32` typed variable. Based on Solidity docs, string variables should be used for arbitrary-length UTF-8 data.

- There are some view functions that could be defined as `pure` ones. For example, the `Orders` contract defines a `getOrderId` function that does not perform any state read or write. Consider declaring those functions as pure instead.

- Comment *"intentionally not a safeSub since minValue may be negative"* in `TradingEscapeHatch` `L57` is confusing, since there is no subtraction operation in the following lines, and there is no `minValue` variable in the context. Consider removing or fixing the comment.

- Avoid code repetition/naming in things like `fillOrder` in `FillOrder` `L380` and `fillOrder` in `Orders` `L174`.

- Consider using scientific notation to declare numeric constants to avoid typos.

  ***Update**: Fixed in 137e28c606dbce1363f315e23d2c610465c9a281 and 55c89e94a1bba5f06da3264601c3a057d614e1a6.*

08. Appendix A - Fixed and partially fixed issues

A. General Observations

HIGH

Tight coupling between contracts

The general architecture of Augur possesses a high degree of afferent (incoming) and efferent (outgoing) couplings between its contracts. This is often referred to as tight coupling and is a measure of how fragile the software is to replacing one of its components. This associated degree of interconnections is also known to increase the attack surface of the software, which is particularly relevant in smart contracts.

For example, `Market's tryFinalize` calls `ReportingWindow's updateMarketPhase`, which in turn calls back to `Market's getReportingState`, which in turns calls other methods in `ReportingWindow` again.

An established flow of information with unidirectional pathways leads to a more loosely coupled software architecture which is more robust, more secure and more resilient to change and scaling.

It is recommended that the Augur team studies the degree of tight coupling / software complexity systematically, and that evaluate what changes in the architecture could create a more unidirectional flow of information, or what design could be used as a foundation framework that simplifies such flow.

Also, consider evaluating the possibility of splitting up the entire code base into completely standalone modules that can be independently tested: i.e. a trading module, a reporting module and a central module to unify the other two.

***Update**: Even though the trading module is still part of the whole codebase, the Augur team reduced the reporting module coupling in `1de558ec7aa0583750e2a90200e13bfc28d35fb1`.*

HIGH

Anyone can trigger Augur events

Most of the events of Augur are triggered from the Augur contract itself. All functions are public, and many of them perform a precondition to check the sender, but not all of them. All those preconditions are performed using the `Universe` given by parameters, which could be a malicious contract to bypass those checks. Then, an attacker can trigger invalid events.

This may not cause a problem directly, but it can flood the blockchain with invalid events, making it unreliable for those who want to browse it.

Consider adding a precondition to check that the msg.sender is a trusted one to those log functions that doesn't have it yet. Additionally, add another precondition to validate that the given universe is a valid one.

Update: Fixed in a0ba05f3d229c796090c2c5dbcbc7c2bee668468.

MEDIUM

Whitelisted contracts are not explicit to the user

The backbone of Augur is composed of a series of whitelisted contracts of singleton nature, which compose a network of allowed interactions on key elements of the architecture. A whitelisted contract has extraordinary privileges within the code, such as the ability to mint `ShareTokens` or even add other addresses to the whitelist itself. Given that there is no explicit tracking of which addresses are whitelisted, there is potential for mistrust form the user's perspective, which can't know the exact member addresses or contracts conforming the whitelist.

Consider making the members of the whitelist explicit to the user. This could be achieved by using an array of whitelisted addresses apart from the mapping that is already used. The mapping provides quick verification that an address is whitelisted and the array exposes the list to the public. Alternatively, add events to signal changes to the whitelist. This would have the additional benefit of being able to detect if the whitelist mechanism has been deployed incorrectly or has been compromised.

Update: Fixed in d075d9c0176a08fea521ff31a373776f134828d5.

MEDIUM

Favor pull payments over push payments

All ETH transfers are executed via call.value. This allows the recipient to execute arbitrary code upon the transfer (due to the gas stipend allocated), and also to throw upon receiving the payment, thus blocking the application flow. For more info on this problem, see this note.

This enables the following issues listed in the document:

- All reporting fees can be frozen by a market creator
- Shareholders fees can be frozen by the market creator
- A market owner can block the Participation token purchase
- Markets ether balance can be stolen by the first reporter

Consider using OpenZeppelin's `PullPayment contract` to implement pull payments, or use the safer transfer keyword for ETH sending.

***Update**: Partially fixed in 2c7c1dd36b1512b440faea205111520f5e9a37e7. There still are some low level calls to be fixed (see the newly reported issue).*

LOW

Use safe math

Given that arithmetic operations on integers may overflow silently, causing bugs, consider using the existing `SafeMathUint256` and `SafeMathInt256` libraries for all arithmetic operations.

For instance, `Market#derivePayoutDistributionHash` uses unsafe addition in `L417`. This operation can overflow and still be equal to the target number of `numTicks`, yielding an invalid set of payout numerators.

As an example, given `numTicks == 1000`, the `array [2**256-1, 1001]` is deemed as valid. Nevertheless, this issue is not propagated since the constructor of `StakeToken`, which replicates exactly the same check, does use safe math (see `StakeToken L32`), and raises an error when attempting to create a token for such invalid payout distribution.

The following unsafe operations were found:

```
trading/Order.sol
    128  Arithmetic operation (++)

libraries/Extractable.sol
     28  Arithmetic operation (++)

libraries/collections/Map.sol
     24  Arithmetic operation (+=)
     37  Arithmetic operation (-=)

libraries/arrays/AddressArrays.sol
     18  Arithmetic operation (-)
     23  Arithmetic operation (++)
     24  Arithmetic operation (+)

libraries/arrays/Bytes32Arrays.sol
     18  Arithmetic operation (-)
     23  Arithmetic operation (++)
     24  Arithmetic operation (+)

libraries/arrays/Uint256Arrays.sol
     18  Arithmetic operation (-)
     23  Arithmetic operation (++)
     24  Arithmetic operation (+)

libraries/collections/Set.sol
     23  Arithmetic operation (+=)
     37  Arithmetic operation (-=)

libraries/math/RunningAverage.sol
     12  Arithmetic operation (/)
     16  Arithmetic operation (++)
     17  Arithmetic operation (+=)

reporting/DisputeBond.sol
     34  Arithmetic operation (*)

reporting/Market.sol
     85  Arithmetic operation (/)
     90  Arithmetic operation (++)
     95  Arithmetic operation (+)
    117  Arithmetic operation (++)
    120  Arithmetic operation (++)
    127  Arithmetic operation (/)
    163  Arithmetic operation (+=)
    189  Arithmetic operation (+=)
```

```
    323  Arithmetic operation (-)
    408  Arithmetic operation (+=)
    415  Arithmetic operation (++)
    417  Arithmetic operation (+=)
    579  Arithmetic operation (+)
    587  Arithmetic operation (+)
    677  Arithmetic operation (+)
    678  Arithmetic operation (++)
    683  Arithmetic operation (+)
    684  Arithmetic operation (+)

reporting/ReportingWindow.sol
     49  Arithmetic operation (*)
     65  Arithmetic operation (+=)
    155  Arithmetic operation (-=)
    204  Arithmetic operation (+)
    212  Arithmetic operation (+)
    216  Arithmetic operation (+)
    221  Arithmetic operation (-)

reporting/StakeToken.sol
     31  Arithmetic operation (++)
     89  Arithmetic operation (*)
     89  Arithmetic operation (/)
    116  Arithmetic operation (*)
    116  Arithmetic operation (/)
    145  Arithmetic operation (-)
    168  Arithmetic operation (*)
    168  Arithmetic operation (/)
    210  Arithmetic operation (++)

reporting/Universe.sol
     59  Arithmetic operation (+)
    109  Arithmetic operation (/)
    113  Arithmetic operation (+)
    125  Arithmetic operation (+)
    125  Arithmetic operation (+)
    125  Arithmetic operation (+)
    125  Arithmetic operation (+)
    129  Arithmetic operation (-)
    137  Arithmetic operation (+)
    261  Arithmetic operation (*)
    266  Arithmetic operation (*)
    375  Arithmetic operation (*)
    375  Arithmetic operation (*)
    380  Arithmetic operation (+)

trading/ClaimTradingProceeds.sol
```

```
     26  Arithmetic operation (+)
     32  Arithmetic operation (++)

trading/CompleteSets.sol
     36  Arithmetic operation (++)
     64  Arithmetic operation (++)

trading/FillOrder.sol
     95  Arithmetic operation (++)
    110  Arithmetic operation (-=)
    111  Arithmetic operation (-=)
    134  Arithmetic operation (-=)
    135  Arithmetic operation (-=)
    158  Arithmetic operation (-=)
    159  Arithmetic operation (-=)
    186  Arithmetic operation (++)
    190  Arithmetic operation (-=)
    191  Arithmetic operation (-=)
    320  Arithmetic operation (-)
    321  Arithmetic operation (+)
    321  Arithmetic operation (++)
    325  Arithmetic operation (-)
    334  Arithmetic operation (-)

trading/Orders.sol
    189  Arithmetic operation (-=)
    190  Arithmetic operation (-=)
    191  Arithmetic operation (-=)

trading/TradingEscapeHatch.sol
     24  Arithmetic operation (++)
     55  Arithmetic operation (++)
     83  Arithmetic operation (++)
```

***Update**: This issue was partially fixed in [d075d9c0176a08fea521ff31a373776f134828d5](). The Augur team decided not to use SafeMath library for those impossible-to-overflow scenarios to avoid extra gas usage. Note there are still scenarios that could indeed overflow (see the newly reported issue).*

LOW

Remove unused code

There are some contract interfaces that are never used in the project. For example the `IRegistrationToken` and `ITrade` contracts. This increases the code complexity.

Additionally, some of the contracts of the codebase define functions that are never used. For example, the `FillOrder` contract defines `getShortShareBuyerSource` and `getLongShareBuyerSource`, which are unused. This increases the code complexity, the attack surface and the size of the bytecode to be deployed.

Consider removing the unused contracts and functions to avoid the kind of problems mentioned above.

***Update**: Fixed in 1de558ec7aa0583750e2a90200e13bfc28d35fb1.*

B. Trading

CRITICAL

Markets are not sanity-checked in trading module

Several functions from contracts in the trading folder accept an `IMarket` as a parameter. Multiple properties are obtained from that market, such as its *denomination* and *share* tokens, its outcomes, its winning outcome, its universe, its reporting window, etc.

These functions do not validate that the market was indeed created from a valid Augur `MarketFactory`. As such, they are subject to manipulation through maliciously crafted contracts, that implement the same `IMarket` interface, though with different semantics.

The following public functions from contracts in trading accept an IMarket:

- `CancelOrder.sol#cancelOrder`
- `ClaimTradingProceeds.sol#claimTradingProceeds`
- `CompleteSets.sol#publicBuyCompleteSets`
- `CompleteSets.sol#publicSellCompleteSets`
- `CreateOrder.sol#publicCreateOrder`
- `Order.sol#create`
- `Trade.sol#publicBuy`
- `Trade.sol#publicSell`
- `Trade.sol#publicTrade`
- `Trade.sol#publicTakeBestOrder`
- `TradingEscapeHatch.sol#claimSharesInUpdate`

These contracts are set as *whitelisted callers*, as per `ContractDeployer#whitelistTradingContracts (L167)`, which implies that all functions decorated with the `onlyWhitelistedCallers` modifier are callable by them. Such methods often make sensitive modifications to a contract's state, not available to the public.

Inserting a malicious `IMarket` contract in the `Trade.sol public methods` has no apparent effects, since only the address of the contract is used as an identifier for the order.

However, given that the remaining functions do rely on information returned by the contract, they are potentially vulnerable.

Consider adding a check in all these methods that the market is a valid `Market` created by a valid `ReportingWindow` contract through a `createMarket` call. This can be checked by a `isContainerForMarket` call to the market's universe, which must also be validated. The latter can be done by tracking all created universes in the Augur contract singleton, and validating that the universe being checked is listed in it.

The following critical vulnerabilities, enabled by this issue, were detected:

- *Universe open interest can be manipulated by an attacker*
- *Complete sets of shares can be purchased for free*

***Update**: Fixed in a0ba05f3d229c796090c2c5dbcbc7c2bee668468.*

CRITICAL

Universe open interest can be manipulated by an attacker

The `universe` contract keeps track of the open interest from all markets through the state variable `openInterestInAttoEth`. This variable can be incremented or decremented by *whitelisted callers* through methods `incrementOpenInterest (L244)` and `decrementOpenInterest (L250)`.

Function `claimTradingProceeds` from `ClaimTradingProceeds.sol` can be publicly invoked, and accepts an `IMarket` without performing any validation on it.

An attacker can submit a maliciously crafted `IMarket` implementation that returns a fake stake token, in which the `msg.sender` has balance on the market's winning outcome. This causes the call to `divideUpWinnings` in `L35` to return a non-zero amount of `_proceeds` for the caller.

If the market returns any valid Augur universe in the call to `getUniverse` in `L38`, then the contract will call decrementOpenInterest into such universe, which will succeed as ClaimTradingProceeds is a *whitelisted caller*.

This allows an attacker to arbitrarily call `decrementOpenInterest` in any Augur universe. This has the immediate effect of decrementing the `targetRepMarketCap` in

L266, which increases the `currentFeeDivisor` in L355 up to a global maximum, thus decrementing incrementing the reporting fees.

A more critical side effect of calling `decrementOpenInterest` arbitrarily is that an attacker could reduce the value of a universe's `openInterest` down to zero. Given that decrementOpenInterest uses `SafeMathUint256.sol#sub`, which reverts the transaction if the subtrahend is larger than the minuend, any subsequent calls to `decrementOpenInterest` will then fail. This means that any user legitimately invoking claimTradingProceeds to collect their proceeds, will cause a call to `decrementOpenInterest` in L38, causing the transaction to revert.

This effectively prevents all users from all markets in the universe from collecting their winnings.

Consider implementing the suggestion described in "Markets are not sanity-checked in trading module" to fix this issue.

***Update**: Fixed in a0ba05f3d229c796090c2c5dbcbc7c2bee668468.*

CRITICAL

Complete sets of shares can be purchased for free

Function `CompleteSets#publicBuyCompleteSets` handles purchases of complete sets of shares for a given market. Given that no check is performed on that market, an attacker could submit any contract that adheres to the required interface.

In particular, an attacker could invoke the function with a market that returns any *denomination token* with no intrinsic value, and returns the *share token* for a different market.

On L35 of `CompleteSets`, the contract transfers `denominationToken (ie Cash)` from the caller to the market. In return, on L37, the contract creates shares of the market ShareToken (for each outcome) for the caller.

If the market contract returns a mock ERC20 token in the `getDenominationToken` call in L31 (in which the msg.sender has a positive balance), and returns the share tokens of another market in L37, then `CompleteSets` will create shares of an arbitrary market for

the attacker, at the expense of an arbitrary (and potentially valueless) Cash token. This allows an attacker to buy complete sets of shares from any market at no cost.

Consider implementing the suggestion described in "Markets are not sanity-checked in trading module" to fix this issue.

***Update**: Fixed in a0ba05f3d229c796090c2c5dbcbc7c2bee668468.*

CRITICAL

Alternative denomination tokens can be stolen from a Reporting Window

Uses of alternative *denomination tokens* (ie tokens used as `Cash` in markets, that are not the controlled `Cash` instance) are not properly managed. Their uses are intermixed in the code with the global Cash, making it difficult to ensure that the correct token is used in each scenario.

As an example, contract `ClaimTradingProceeds` `L55` transfers a reporter's share from a market's *denomination token* to a `ReportingWindow`. Since `ReportingWindow` contract is Extractable, and only the global Cash instance is marked as protected as per `L416`, any user can steal all alternative *denomination tokens* from a `ReportingWindow`.

Note that several comments in the `ReportingWindow` contract (`L264`, `L285`, `L306`) seem to imply that the feature of managing multiple denominations is not fully implemented.

To prevent these issues, consider restricting markets to work only with the controlled global `Cash` instance, which is a wrapped ETH implementation, or work with ETH directly.

***Update**: Fixed in 6c4941bbfce7f574cc0d601b6787076041291df6.*

CRITICAL

Order info is repeated as arguments when cancelling an order

Function `CancelOrder#cancelOrder` receives the ID of the order to cancel. However, it also receives the order type, market, and chosen outcome, which are used to determine

how the order should be refunded. All that information is already present in the order itself, and a user could supply different parameters to `cancelOrder`.

As a grave consequence of this, a malicious trader could send the opposite order type to attempt to collect shares from the opposite outcomes as their refund.

For instance, given an Ask order with escrowed shares for an outcome A, the malicious trader could cancel it using Bid as the order type. Then, the call to `getOrderSharesEscrowed` in `L35` would return the original number of escrowed shares for A, but `L59` would return shares for all outcomes except A, given that the order type parameter is Bid.

Consider retrieving order type, market and outcome from the order itself given the order ID, using the getters available in Orders, rather than accepting them as parameters.

***Update**: Fixed in 6c4941bbfce7f574cc0d601b6787076041291df6.*

HIGH

Cancelling an order with share tokens in escrow will fail

Function `CancelOrder#refundOrder` refunds the share and denomination tokens escrowed back to its creator when an order is cancelled. In `L59` and `L64`, the contract executes a transfer of share tokens to the order creator. However, the `CancelOrder` contract does not hold share tokens itself, so both of those transfers should fail, making it impossible to cancel an order with share tokens in escrow.

Note that the tests in `test_cancelOrder.py` that exercise that code do not test for shares refund, only for ETH refund.

Consider changing the transfer call in `L59` and `L64` to a `transferFrom(market)`, since it is the associated market who holds the shares in escrow, and add a test to verify its correct implementation.

***Update**: Fixed in 6c4941bbfce7f574cc0d601b6787076041291df6.*

HIGH

Markets can be created with malicious Cash tokens

Markets are created by invoking `ReportingWindow.sol#createMarket`. Among the parameters received is the address of the token to be used as a *denomination token* within the market, this is, a wrapper for Ether.

A malicious user could initialize a market with any implementation of `ICash`, including a malicious one that sends a fraction of token to the attacker for every transfer, as an example.

Another possible exploit is submitting an `ICash` token controlled by the attacker, which throws at the attacker's will. This can be used to block the application flow at any time the denomination token is used; for instance, when attempting to finalize an invalid market (see `Market L283`), thus causing `ReportingWindow#allMarketsFinalized` to never be true.

To prevent these issues, consider restricting markets to work only with the controlled global Cash instance, which is a wrapped ETH implementation, or work with ETH directly.

***Update**: Fixed in 6c4941bbfce7f574cc0d601b6787076041291df6.*

HIGH

Shareholders fees can be frozen by a malicious market creator

Once a `Market` is finalized, shareholders can claim their winning share fees calling the `claimTradingProceeds` function of the `ClaimTradingProceeds` contract. This function will divide the winnings between the requesting shareholder and the market creator, and transfer those amounts to them.

A malicious market creator could own a market through a controlled contract defining a payable fallback function that throws every time someone transfers ether to it. Then, the market creator would be effectively freezing the fees of all the shareholders.

Consider using pull payments to avoid handling ether transfers in the same flow of the claiming fees logic. Alternatively, the `claimTradingProceeds` flow could be split to allow shareholders and the market owner to claim their fees separately.

***Update**: Fixed in 2c7c1dd36b1512b440faea205111520f5e9a37e7.*

MEDIUM

It is possible to create orders for untrusted markets

There are two possible ways for a user to create orders for a given `Market`. A user can use any of the options that the `Trade` contract provides, and the `CreateOrder` contract also defines a `publicCreateOrder` function to do so. In fact, Trade functions call the `CreateOrder` contract if it couldn't fill any of the existing orders with the incoming one.

No function in the create order flow validates that the given market is a trusted one. Therefore, an attacker could ask someone to place an order for a self-controlled malicious market to steal their money.

Consider adding a precondition in `createOrder` to validate that the given market is a valid one using the `isContainerForMarket` function from `Universe`, and checking that the universe returned by the market is a valid one.

***Update**: Fixed in a0ba05f3d229c796090c2c5dbcbc7c2bee668468.*

LOW

The Trade logic treats a lack of gas as a complete order fill

The `documentation` makes clear that a trading transaction will return 1 if the order was filled completely. The trade function of the Trade contract has a statement that halts the execution and returns 1 if the amount of given gas is not enough to cover the transaction, based on the `MINIMUM_GAS_NEEDED` constant.

Consider using another return value to handle this scenario in order to avoid confusing a successfully order filled case with an insufficient gas provided one.

***Update**: The Augur team fixed this statement in the documentation.*

LOW

Market creators may not be able to collect their corresponding fees

Once a Market is finalized, shareholders can claim their winning share fees calling the `claimTradingProceeds` function of the `ClaimTradingProceeds` contract. This function will divide the winnings between the requesting shareholder and the market creator, and transfer those amounts to them.

This means that market creators will receive their corresponding fees only if shareholders claim their fees too. Then, it would take just one shareholder that does not claim their fees, to make sure that the market creator does not receive their whole corresponding fees. For example, there could be a shareholder with a huge amount of money invested that loses his account private key.

Consider splitting the `claimTradingProcees` flow allowing shareholders and the market owner to claim their fees separately.

***Update**: Fixed in d075d9c0176a08fea521ff31a373776f134828d5.*

C. Reporting

CRITICAL

It may not be possible to stake tokens on an invalid outcome

As per `StakeToken#isInvalidOutcome`, an outcome is considered invalid if all its `payout numerators` are equal. The `StakeToken` constructor checks for this condition, and requires that if the invalid flag received as a parameter is set, then the numerators are invalid (see `L35`).

This check is incorrectly implemented, since `L35` checks for the value of the invalid state variable, which is only assigned afterwards in `L38`. As such, the state variable will always be false, and not reflect the value of the parameter.

Check the value of the `_invalid` parameter instead of the invalid state variable in `L35`. Alternatively, evaluate removing the invalid parameter altogether, and assign the state variable from the result of `isInvalidOutcome`. Note that this last option forbids having a *valid* outcome where all payout numerators are equal, which may or may not be desirable.

Furthermore, the requirement of all payout numerators being equal may not be feasible, due to the additional restriction of having their sum equal to `numTicks` (see `L34`). For example, a market set up with 3 outcomes and 1000 `numTicks` cannot be marked as invalid, since 1000 is not divisible by 3.

Consider adding a check on market construction that `numTicks % numOutcomes` must equal to zero, or relax the requirement on the payout numerators for invalid outcomes.

Additionally, consider writing a special case for invalid outcome payouts, which doesn't require payout numerators as an argument.

***Update**: Fixed in 1de558ec7aa0583750e2a90200e13bfc28d35fb1.*

CRITICAL

Markets ether balance can be stolen by the first reporter

The `Market` contract defines the `firstReporterCompensationCheck` public function which is called from the *buy* function of the `StakeToken` contract every time a buy occurs. This is how first reporters get compensated with the reporting gas costs and the no-show REP bond. This function transfers said tokens to the first reporter, and then the reporting gas costs through a `call.value(reporterGasCostsFeeAttoeth)`.

An attacker could call the `StakeToken buy` function from a smart contract that defines a payable fallback function to perform a reentrance to the `buy` function every time it receives ether. The only precondition to transfer those fees to the first reporter is that the `tentativeWinningPayoutDistributionHash` is not set, which will be always true since it gets updated after the `firstReporterCompensationCheck` is called from `StakeToken#buyTokens`. This allows an attacker to withdraw all the ether balance of a `Market`.

Consider using pull payments to avoid handling Ether transfers in the same flow of the market logic. Alternatively, consider adding a reentrancy guard to the `StakeToken buy` function.

***Update**: Fixed in 1de558ec7aa0583750e2a90200e13bfc28d35fb1.*

CRITICAL

All reporting fees can be frozen by a Market creator

There are three different possible reporting fees that reporters can claim. First, they can call the `redeemWinningTokens` function of the `StakeToken` contract to claim their fees based on their staked REP tokens. Then, they may also use the withdraw function of the `DisputeBond` contract to claim fees in case of a dispute. And finally, there is also a chance to claim ParticipationTokens fees, in case there were no markets on which to report.

An attacker could create (and thus own) a market through a malicious contract, defining a payable fallback function that only throws. In this case, market owners can prevent their markets from closing, which means a single market owner can freeze all the reporting fees of a `ReportingWindow`.

All the fee payment mechanisms described above use the `internalCollectReportingFees` function of the `ReportingWindow` contract. This function performs a precondition to check that all the markets of the `ReportingWindow` are finalized. If the malicious `Market` is unfinalized, it will prevent all the reporters of a `ReportingWindow` from claiming their fees.

Consider using pull payments to avoid handling ether transfers in the same flow of the market logic.

***Update**: Fixed in 2c7c1dd36b1512b440faea205111520f5e9a37e7.*

CRITICAL

A market owner can block the Participation token purchase

The `ParticipationToken` contract defines a `buy` function to allow reporters to exchange their REP tokens for Participation ones in case there are no markets on which to report. Indeed, this function performs a precondition to check that all the markets of a `ReportingWindow` are finalized.

The only way to finalize a `Market` is through the `tryFinalize` function. It performs some state changes over the market and transfers the validity bond to the owner. Then, a malicious market creator could own a market through a contract, defining a payable fallback function that throws every time it receives ether. In this case, a market owner will block the `ParticipationToken` purchases.

Consider using pull payments to avoid handling ether transfers in the same flow of the market logic.

***Update**: Fixed in 2c7c1dd36b1512b440faea205111520f5e9a37e7.*

C. Forking

MEDIUM

Markets can be created in a locked universe

As described in the whitepaper, when a fork occurs *"the parent universe becomes permanently locked. In a locked universe, no new markets may be created [...]"*. In `ReportingWindow's createMarket(...)` we observe that, during a fork, the code does not appear to explicitly check for this. Moreover, there are no tests implemented for this requirement.

Explicitly check for the condition specified in the whitepaper that *the reporting window in question does not belong to a lock universe*, and exit early if it resolves to true. Additionally, we recommend creating a test for this specification. As a general practice, using state for explicitly and declaratively early exiting during functions makes the code more robust, easier to understand and resilient to change.

***Update**: Fixed in 1de558ec7aa0583750e2a90200e13bfc28d35fb1.*

MEDIUM

Eventually it will not be possible to produce further forks

At the time of writing, it would cost approximately 4.3 M dollars (149,600 REP) to fork a universe, considering 1,100 REP to dispute the designated report, 11,000 REP to dispute the first report, and 137,500 REP (1.25% of total supply) to dispute the last one.

Consider a heavily disputed market. There could be black swan scenario where most people think outcome A will happen, and outcome B actually happened, but questionably. For example, a president election market may cause a fork if a lot of people think there was fraud, since there is no incentive to vote for truths. If many such scenarios come through, causing a considered amount of forks until we have a universe without the necessary

supply of REP tokens to afford a dispute, there may be a time where reporters won't be able to dispute a market anymore.

Consider using an amount of REP tokens proportional to the supply of the universe for dispute bonds, rather than using fixed amounts.

***Update**: Fixed in [44b757f74c7c000a3446bb143d1412764fda7358](44b757f74c7c000a3446bb143d1412764fda7358).*

D. Miscellaneous

HIGH

Spender contracts cannot be re-approved if updated

A comment in function `Market.sol#approveSpenders` correctly reads: *"This will need to be called manually for each open market if a spender contract is updated"*. However, the function is marked as private, and cannot be invoked except from the market's constructor, thus making it impossible to approve updated contracts. This breaks all existing markets when one of the contracts listed in `L116` or `L121` is updated.

Change the private modifier to public, and add a check that it is invoked only by a trusted entity.

***Update**: Fixed in 5ae5f9685b3795c1ff1a20b0542cc9ccc91c6260.*

LOW

Delegator memory allocation not working for arguments larger than 32 bytes

`Delegator.sol` Uses a very simple algorithm to pad memory allocation to 32 bytes block which however ignores cases when the argument size is larger than 32 bytes.

Consider using an algorithm that handles padding to multiples of 32 for all of the cases, like:

```
_size := and(add(calldatasize, 0x1f), not(0x1f))
```

For more info on how this works, please see this Solidity documentation example.

***Update**: Fixed in d075d9c0176a08fea521ff31a373776f134828d5.*

LOW

Delegator not working for return data greater than 32 bytes

The `Delegator` contract defines a payable fallback function that delegates calls to its controller. The way the `delegatecall` is implemented, it is assuming that return data will always have a length of 32 bytes. This may not always be true since the return data size may be of arbitrary size.

Consider dropping this assumption and updating the code to accept an arbitrary length return value. It is possible to use a mechanism of manual data allocation with `returndatacopy` and `returndatasize` Solidity assembly opcodes added in the Byzantium hard fork with EIP211.

***Update**: Fixed in 3e8ee86cdd900b0e6c8c130c0e2147dcd0a8bc3c.*

E. Notes & Additional Information

- A lot of contracts define public getter functions for public state variables. There is no need to do that since, based on the docs, a public getter variable is generated automatically for each public variable. For example, the `StakeToken` contract defines a `getMarket` function that returns the `market` state variable. Consider removing those getter functions.
  ***Update**: Fixed in 3e8ee86cdd900b0e6c8c130c0e2147dcd0a8bc3c.*

- The following constants are unused:

  - `Reporting#DEFAULT_DESIGNATED_REPORT_NO_SHOW_BOND`
  - `Reporting#REGISTRATION_TOKEN_BOND_AMOUNT`
  - `ReportingWindow#BASE_MINIMUM_REPORTERS_PER_MARKET`

  Note that `DEFAULT_DESIGNATED_REPORT_NO_SHOW_BOND` should be returned from the getter `getDefaultDesignatedReportNoShowBond`, which currently returns `DEFAULT_DESIGNATED_REPORT_STAKE` instead. Consider removing the other ones for clarity.
  ***Update**: Fixed in 14501676bd8129c2d3830d73f4b344888e9d7698.*

- The library `ContractExists` declares only one function and it's being used just once in the `LegacyReputationToken` contract. Unless this library is being shared somewhere else, consider inlining that function to reduce code complexity.
  ***Update**: Fixed in fb5749cbe25c2e8f3aa217c612729ba79143ba64.*

- Function `Market.sol#migrateThroughOneFork` in `L335` resets the market's `designatedReportReceivedTime` to `block.timestamp-1` if it was set. Consider adding a comment to explain the rationale behind this.
  ***Update**: Fixed in 3e8ee86cdd900b0e6c8c130c0e2147dcd0a8bc3c.*

- The `Market` contract defines three functions for disputing, `disputeDesignatedReport`, `disputeFirstReporters` and `disputeLastReporters`, but the first is the one that triggers a migration in case of a fork, through the `triggersMigration` modifier. Consider adding that modifier to the rest of the dispute functions.
  ***Update**: Fixed in 3e8ee86cdd900b0e6c8c130c0e2147dcd0a8bc3c.*