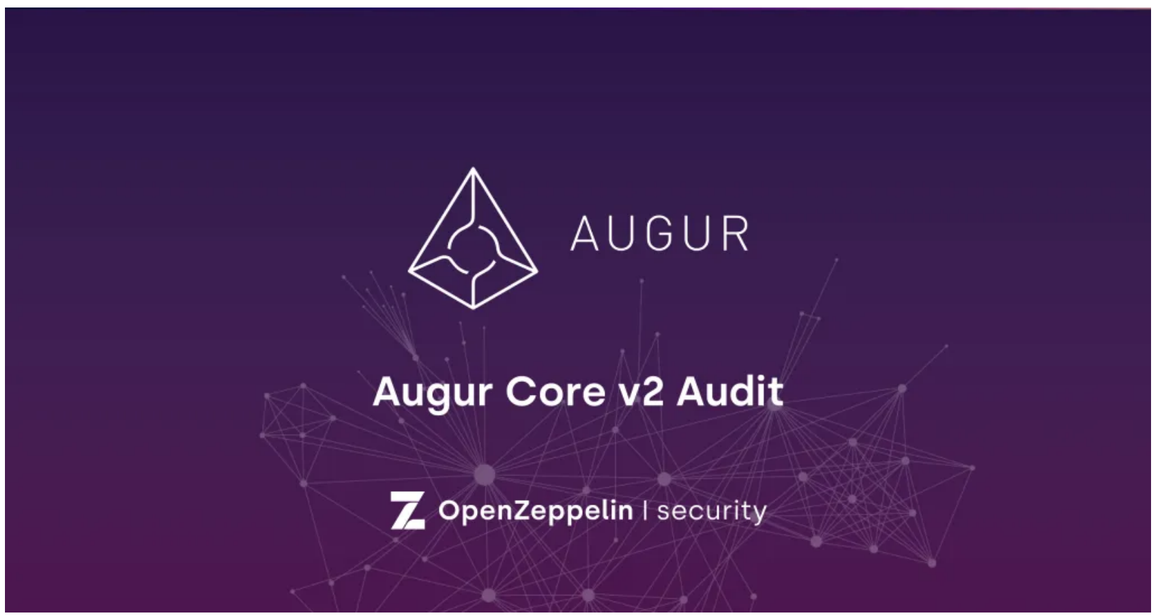
Augur Core v2 Audit

FEBRUARY 27, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY



The Augur team asked us to review and audit their Augur Core v2 project. We looked at the code and now publish our results.

The audited commit is `9a33c3269e812d0cb66d49b61a72db58e32e4749` and all contracts in the AugurProject/augur/packages/augur-core/source/contracts/ folder were in scope, with the following exceptions and caveats.

- Contracts `TestNetReputationToken.sol`, `TestOrders.sol`, `ERC820Registry.sol` and `reporting/DisputeCrowdsourcer.sol` were left entirely out of scope.

- All contracts in the trading directory were treated differently, as we did not perform our standard audit in full-depth due to the time requirements imposed on this particular audit. In this case, we only focused specifically on changes implemented since the last audited version and on particular major threats identified by the team.

Before moving to the full list of issues found in the project, some introductory remarks about the project's current status are in order.

First, the entire Augur Core v2 project is still under heavy development, so major features such as the DAI integration are still to be included in the protocol, and were therefore not reviewed.

Second, while the Augur team was responsive and helpful enough throughout the audit, we found documentation for all current (and future) changes in v2 to be meager (as highlighted in several parts of the report), which hindered the auditing process to some extent, considering the difficulties entailed by auditing code with little specifications on its intention. Great efforts must be made in terms of end-user documentation before releasing the final version of the protocol. By going through and explicitly documenting all functionality in the contracts, the Augur team may surface unintended code or comments that were simply left over from v1 and are not meant to exist in v2.

Finally, it is necessary to stress that the audit performed does not include any migration mechanisms from Augur's current live version on mainnet to the to-be-deployed second version of the protocol.

Next, our audit assessment and recommendations, in order of importance.

Update: *the Augur team applied several fixes based on our recommendations. We address below the fixes introduced as part of this audit.*

Critical severity

[C01] All CASH tokens approved to Augur can be emptied

A `MarketFactory` has a `createMarket` public function that allows anyone to create a new `Market`. After creating the market, and before initializing it, the factory first transfers all its REP token balance to the market. Then, it calls the given `IAugur` `contract` to execute a "trusted transfer" of CASH tokens, for an amount given by `_universe.getOrCacheValidityBond()` (being `_universe` the address of a potentially user-controlled `IUniverse` contract).

In a scenario where a victim has approved `N` CASH tokens to the `Augur` contract, and the `MarketFactory` is registered as a `trustedSender` in such `Augur` contract, an attack vector has been identified where tokens approved by the victim to the `Augur` contract can be freely emptied by an attacker, sending them to any newly created market.

All the attacker needs to do is call the `createMarket` function of the trusted `MarketFactory` with the following relevant parameters:

- `_universe`: an address of a contract controlled by the attacker that implements the `getReputationToken` function and `getOrCacheValidityBond` function
- `_sender`: address of the victim that approved `N` "Cash" tokens to `Augur` contract

As a consequence:

1. The `require` in line 16 of `MarketFactory.sol` passes regardless of how many tokens the `MarketFactory` contact owns.
2. The `require` in line 17 of `MarketFactory.sol` makes a `trustedTransfer` of CASH tokens. The amount of tokens to be transferred is controlled by attacker, as attacker controls `_universe.getOrCacheValidityBond()`.
3. Upon its `trustedTransfer` function being called, the `Augur` contract recognizes `MarketFactory` as a `trustedSender`, and executes a `transferFrom` from the victim's address, to the market's address, of as much tokens as the attacker set.
4. Finally, back in the `MarketFactory` contract, the created market is initialized.

As a result, the victim lost all CASH tokens approved to `Augur`, which were freely moved by an attacker to the newly created market. It is important to highlight that in the issue **"CASH tokens can be stolen from the Market contract"** a related attack vector is thoroughly described, where the attacker is able to actually steal the tokens moved to the market.

Even if a user approves a certain amount of tokens to a contract (*e.g.* the `Augur` contract), under no circumstances should those tokens be allowed to be moved at the will of a potentially malicious external actor. Therefore, consider implementing the necessary restrictions in the `MarketFactory` contract to prevent a scenario as the one described above. Related unit tests to thoroughly cover such dangerous cases must also be implemented.

Update: *fixed in `814b390` by checking that the universe address passed into the `createMarket` function is a known universe, thus eliminating the possibility of it being malicious.*

[C02] CASH tokens can be stolen from the Market contract

By exploiting the critical vulnerability described in "**[C01] All CASH tokens approved to Augur can be emptied**", any malicious user can move all CASH tokens approved to Augur, sending them to Market contracts. But the attacker does not *directly* benefit from the attack.

However, a severe attack vector has been identified in the Market contract that would allow the attacker to actually steal all tokens previously moved to the Market contract. Therefore, this issue should be considered as an extension of "**[C01] All CASH tokens approved to Augur can be emptied**". Following, we set out to describe step by step how an attacker can leverage a malicious universe and the lack of input validations to finally move the CASH tokens to any address in their control.

Once the approved CASH tokens are moved to the `Market` contract (described in "**[C01] All CASH tokens approved to Augur can be emptied**"), the `MarketFactory` contract calls the market's `initialize` function. This means that:

- 1.1) Market's `owner` and `repBondOwner` are set to the victim's address (*i.e.* the `_sender` parameter).
- 1.2) Market's `universe` is set to an attacker controlled `IUniverse` contract.
- 1.3) The market adds a new `InitialReporter` to the p`articipants` `array`, the attacker being the `designatedReporter` of the added `InitialReporter` contract.
- 1.4) Market's `repBond` is set to `0`, as `universe.getOrCacheMarketRepBond()` is controlled by the attacker.
- 1.5) Market's `validityBondAttoCash` is set to the entire CASH balance of the `Market` contract.

Now, when the market's `endTime` passes (this parameter is attacker-controlled too, and set during initialization), the attacker can either submit an initial report or just wait the 24 hours until public reporting is allowed. In any case, the `Market` has to be reported as "Invalid".

- 2.1) Regardless of when the attacker submits the initial report, the transfer operations in lines 151 and 152 or line 154 of `Market.sol` will move `0` REP tokens, as `_initialReportStake` is the same as `repBond` and `repBond` was set to zero (see 1.4).
- 2.2) In line 140 the `disputeWindow` of the `Market` is initialized by the attacker-controlled `universe`.
- 2.3) In line 141 (then lines 46 and 47 of `InitialReporter.sol`) the attacker gets registered as the `owner` and `actualReporter` of the `InitialReporter` contract.
- 2.4) Then, in line 50 of `InitialReporter.sol` the `payoutNumerators` is set to the attacker-controlled `_payoutNumerators`, such that `payouNumerators[0] > 0`, so the `Market` is reported as "Invalid". From now on, the function `getPayoutNumerator(0)` (inherited from the `BaseReportingParticipant` contract) will always return a value greater than zero.
- 2.5) In line 51 of `InitialReporter.sol` the `size` is set to zero.

Finally, the attacker calls the `finalize` function on the `Market` contract.

- 3.1) The `require` statement in line 236 of `Market.sol` is verified correctly.
- 3.2) In line 237, the attacker-controlled `universe` returns the necessary value to make `true` the conditional clause, bypassing (*i.e.* never executing) all logic in the `else clause`.
- 3.3) The attacker-controlled `universe` sets `winningPayoutDistributionHash` to some value other than zero in line 239. From now on, every call to `Market`'s `isFinalized` function will return `true`.
- 3.4) In line 250, the execution flow goes into the `distributeValidityBondAndMarketCreatorFees` function, where in line 310 the `marketCreatorFeesAttoCash` is set to `validityBondAttoCash`, which in turn is all `Market`'s CASH balance (see 1.5).
- 3.5) Inside the `distributeMarketCreatorFees` function, the call to `isInvalid` in line 315 will return `true` (see 3.3 and 2.4), therefore the execution flow will jump straight to line 321, where the `Market` contract will finally send `marketCreatorFeesAttoCash` amount of tokens (the

entire CASH balance – see 3.4 and 1.5) to an attacker-controlled address (as the attacker controls `universe.getOrCreateNextDisputeWindow(false)`).

As explained, by combining the vulnerability reported in the issue "**[C01] All CASH tokens approved to Augur can be emptied**" with the correct execution of the steps above, any malicious user can effectively steal CASH tokens that are approved to the Augur contract. While initially the attack here described seems to be prevented by properly verifying the legitimacy of the `universe` address set in the `Market` contract, a thorough review of the entire attack vector, which includes complex interactions between the `Augur`, `MarketFactory`, `Market` and `InitialReporter` contracts, is in order, so as to determine the most appropriate course of action to mitigate the critical vulnerability.

***Update**: the Augur team correctly pointed out that this attack is not actually possible because the transaction will be reverted upon calling the `logInitialReportSubmitted` function in the `Augur` contract (which checks that the passed universe address is legitimate).*

[C03] Anyone can remove all market's dispute crowdsourcers

After an initial report is done, the tentative outcome for an Augur market can be disputed by any REP token holder during the market's dispute round. Such dispute consists of staking REP tokens on an outcome other than the market's current tentative outcome. A dispute is considered successful when the total amount of dispute stake on some outcome meets the dispute bond size required for the current dispute round. It is important to highlight that dispute bonds do not need to be paid in their entirety by a single user; instead, users can crowdsource them.

During the dispute round, a `Market` keeps track of contributions to each possible outcome by instantiating `DisputeCrowdsourcer` contracts, storing references to them in the `crowdsourcers` `collection`. These `DisputeCrowdsourcer` contracts act as escrows for the staked REP tokens for each possible outcome, and are expected only to be cleared (via the `clearCrowdsourcers` function) when either the crowdsourcing for a particular dispute bond finishes or existing crowdsourcers need to be disavowed during a fork. However, as the `clearCrowdsourcers` function is `public`, anyone can, at any time, silently delete all crowdsourcer addresses registered in a `Market` contract. This renders the entire disputing process pointless, as anyone can just delete all crowdsourcers before a dispute bond is about to be filled, thus avoiding any dispute over the market's tentative outcome.

Consider restricting the visibility of the `clearCrowdsourcers` function to `private`. Furthermore, to prevent this issue from being reintroduced in future changes to the code base, consider adding related unit tests.

***Update**: fixed in `99d06e8` by restricting the `clearCrowdsourcers` function's visibility to `private`.*

[C04] DisputeCrowdsourcer contract does not allow for the purchase of overload tokens once its size is filled

The `DisputeCrowdsourcer`'s `contribute` function is expected to mint DISP tokens for contributors until the funding goal is reached. After the funding goal has been reached, further contributions are rewarded with "overload tokens" to enable participants to quickly raise the dispute

stakes without waiting through multiple dispute rounds. This mechanism prevents malicious parties from dragging out disputes.

However, in line 78 of `DisputeCrowdsourcer.sol`, `size.sub(_curStake)` will inevitably revert if `_curStake` is larger than `size`. `size` is the funding goal of the `DisputeCrowdsourcer` and is the threshold at which DISP tokens stop being minted, replaced by the minting of overload tokens. The value of `_curStake` is calculated by the `getStake` function which sums the total supply of all DISP and overload tokens. As a consequence, only the first purchase of overload tokens that brings the total stake above the funding goal will succeed. Every subsequent attempt to contribute to the tentative outcome (*i.e.* by purchasing overload tokens) will revert because the total number of overload tokens and DISP tokens will exceed the funding goal (*i.e.* `size`).

Consider refactoring the `contribute` function to handle the described scenario, where the total number of overload tokens and DISP tokens exceeds the funding goal, so as to ensure disputes cannot be easily dragged out by malicious parties.

Note: The `DisputeCrowdsourcer` contract is not within the scope of this audit but has many interactions with the contracts covered by this report.

Update: *fixed in `2ea5753` by removing the concept of overload tokens.*

[C05] Unfillable orders can be placed to constrain or halt markets

In the `FillOrder` contract, the `fillOrder` function calls `tradeMakerTokensForFillerTokens` in the `Trade library` located in the same file. The `tradeMakerTokensForFillerTokens` function makes a transfer of `longShareToken` and `shortShareToken` to the `_longBuyer` and `_shortBuyer` respectively. Either the `_longBuyer` or the `_shortBuyer` will be the order `creator` depending on the direction of the trade. Both transfers will trigger the `tokensReceived` hook on the recipient if the recipient is an ERC820-registered contract. This means that if the order `creator` is a contract that reverts when `tokensReceived` is called, the order will be unfillable because it will cause `fillOrder` to revert.

According to Augur v2 whitepaper, “Orders are never executed at a worse price than the limit price set by the trader, but may be executed at a better price.” An unfillable order may severely limit or completely disable a market because it must be filled before an order with a worse price can be filled. This applies to orders on both sides of the order book.

Any malicious user can exploit this vulnerability by placing an unfillable order at a price they do not want the market to trade above or below (depending on the direction of the order), thus creating an artificial ceiling or floor on the market. Furthermore, just by placing two unfillable orders, one long, one short, with a very tight spread, the attacker can effectively halt all trading operations in any Augur market.

Consider using the `trustedFillOrderTransfer` function instead of `transfer`, both in line 197 and line 199 of `FillOrder.sol`, to ensure the `ERC777 hooks` are not called, which should mitigate the described critical vulnerability.

Update: *fixed in `98a3f36` by modifying transfers of Augur's Share tokens to no longer call the ERC 777 hooks.*

[C06] Affiliate's fees can be withdrawn in an invalid, finalized Market

The `Market` contract implements the `recordMarketCreatorFees` function, which can only be called by "known fee senders" and is in charge of calculating the amount of fees to be assigned to the creator and the affiliate (if exists). When the market is finalized, the function takes care of calling the `distributeMarketCreatorFees` function. This last function either:

- A) If the market is valid, transfers the current owner all corresponding fees (*i.e.* `marketCreatorFeesAttoCash`). Additionally, if there is an affiliate, `distributeMarketCreatorFees` sends the corresponding fees to the affiliate too by calling the `withdrawAffiliateFees` function.
- B) If the market is invalid, sends all market creator fees (*i.e.* `marketCreatorFeesAttoCash`) to a fee pool.

In case (B), the amount of CASH tokens sent to the fee pool does not include any affiliate fees (they are always subtracted from `marketCreatorFeesAttoCash` in line 300). That means that the transfer of CASH tokens in line 321 does not actually transfer the affiliate's fees to the pool, and therefore an affiliate can still call `withdrawAffiliateFees` to cash out the fees.

In a scenario where addresses controlled by the market creator are set as the affiliate address in trading operations, and the `affiliateFeeDivisor` is set to 1 (so that in line 298 the `_affiliateFees` are as high as possible – equal to `marketCreatorFees`), then the market creator will not lose any fees if the market ends up being invalid, as they will be able to get away with all affiliate fees by calling the `withdrawAffiliateFees` function.

Similar to what is currently done with market creator fees, consider implementing the necessary changes in the `distributeMarketCreatorFees` function to effectively send all affiliate fees to the fee pool if the market is invalid. Unit tests to cover this sensitive scenario are in order, which should prevent this issue from being reintroduced in future changes to the code base.

***Update**: fixed in `8707454` . Affiliate fees are now transferred to the dispute window for invalid markets.*

[C07] All dispute bonds can be held hostage or permanently frozen when a Market is in a forking universe

When a non-forking market is in a forking universe, its `disavowCrowdsourcers` function must be called either directly or through the `migrateThroughOneFork` function before dispute bond holders can redeem and migrate their REP tokens to a child universe. This is due to the fact that the `DisputeCrowdsourcer` contract must be disavowed or the market must be finalized before redemption is allowed, and the market cannot be finalized in a forking universe.

In the `disavowCrowdsourcers` function, a transfer of REP tokens is made to the `repBondOwner` address. As the REP token is similar to an ERC777 token, its `transfer` function will call the `tokensReceived` hook on the `repBondOwner` address if it is an ERC820-registered contract.

A malicious `repBondOwner` account could leverage this hook to revert any calls to the `tokensReceived` function, rendering it impossible to successfully call the `disavowCrowdsourcers` or `migrateThroughOneFork` functions on that market. As a consequence, this attack will effectively freeze all dispute bond funds and therefore prevent the market from being migrated to the winning fork. It would be trivial for the malicious `repBondOwner` account to only release funds when a ransom has been paid to the contract which can then be collected by the attacker. This can also be exploited just as a griefing attack to cause Augur reporting participants to lose funds.

When making ERC777-like transfers, consider strictly implementing the withdrawal payments pattern (a.k.a. "pull style" payments) or ensuring the ERC777 transfer hooks are not called to mitigate attack vectors associated with making calls to external addresses.

***Update**: fixed in `c6eed38`. The transfer of REP token in the `disavowCrowdsourcers` function no longer calls the ERC 777 hooks.*

[C08] Fork reputation goal threshold can be decreased during fork

Once a universe is forking, all REP token holders are expected to migrate out their entire REP balance to one of the child universes. The migration of tokens can be accomplished calling the `migrateOut` or `migrateOutByPayout` functions in the `ReputationToken` contract corresponding to the forking universe. These will then trigger the burning of REP tokens in the forking universe, and by calling the `migrateIn` function of the child universe's REP token, the corresponding REP tokens will be minted in the child universe. The forking stage will be considered finished either when a certain amount of REP tokens have been migrated (*i.e.* `forkReputationGoal` tokens) to a child universe (which will be considered the winning child universe), or when a fixed amount of time has passed (*i.e.* when the system's time is greater than `forkEndTime`).

While the `forkReputationGoal` was expected to be a fixed threshold during a fork, an attack vector has been identified in a second-generation forking universe where any malicious user can turn the `forkReputationGoal` into an ever-decreasing moving target, thus dangerously allowing a holder of tokens to potentially manipulate the outcome of a fork in an unintended way.

Consider a Universe B, winning child of a locked Universe A that forked in the past (*i.e.* in Universe B, `augur.getTimestamp() >= parentUniverse.getForkEndTime()` equals `true`). Now consider that the Universe B is forking to several universes, but still has not settled on which child universe is the winning one; that is, users are actively migrating their Universe-B-REP tokens to one of the many child universes by calling the `migrateOut` function of the Universe-B-REP token. Therefore, the total supply of Universe-B-REP tokens is decreasing (because Universe-B-REP tokens are burned).

The `ReputationToken` contract implements the `updateTotalTheoreticalSupply` function that can be called by anyone at any time. In our scenario, calling this function will always execute line 157 of `ReputationToken.sol`, thus updating the `totalTheoreticalSupply` of Universe-B-REP tokens to the last registered `totalSupply` of tokens. This means that, as the `totalSupply` of Universe-B-REP tokens is decreasing due to tokens being migrated and burned, the `totalTheoreticalSupply` will also decrease each time `updateTotalTheoreticalSupply` is called after at least one Universe-B-REP token is migrated out to a child universe.

The `Universe` contract implements the `updateForkValues` function that can be called by anyone at any time, and updates the `forkReputationGoal` with the last registered value of `totalTheoreticalSupply`. In our scenario, if the `updateForkValues` function is called in Universe B after calling `updateTotalTheoreticalSupply` in the Universe-B-REP token, then the `forkReputationGoal` threshold will be decreased, as the `totalTheoreticalSupply` was decreased (explained in previous paragraph).

As a consequence, any user can leverage the migration of Universe-B-REP tokens to effectively turn the `forkReputationGoal` threshold of Universe B into a moving target, thus each time requiring less and less tokens for a child universe to be considered the winning universe, not only

because the amount of REP migrated is increasing (which is expected), but also because the threshold is being lowered.

Consider analyzing the need of having a public function that updates critical system parameters such as `updateForkValues`. If it is to remain public to be called from off-chain clients, then its execution during a fork should be halted, making sure it is called one last time before entering the fork stage. As a starting point, one potential (untested) solution for this issue could be adding a `require(!isForking())` statement at the beginning of the `updateForkValues` function, and adding a call to it in the `fork` function of the `Universe` contract. Regardless of the course of action taken, thorough unit tests to cover the described scenario should be included to prevent this issue from being reintroduced in future changes to the code base.

Update: *fixed in* `a9560f4`. *The fork reputation goal threshold can no longer be changed once a fork has begun.*

High severity

[H01] Implementation of EIP 777 does not fully match the specification

The following mismatches between the EIP 777 specification and the related implementation contracts (*i.e.* `ERC777Token`, `ERC777BaseToken`, `ERC777TokensRecipient` and `ERC777TokensSender`) were identified.

- In `ERC777Token`, the following functions are missing from the ERC777 interface: `name()`, `symbol()`, `totalSupply()`, `balanceOf(address)`, `granularity()`, `burn(uint256,bytes)` and `operatorBurn(address,uint256,bytes,bytes)`. It is worth highlighting that `burn(uint256,bytes)` and `operatorBurn(address,uint256,bytes,bytes)` are the only functions never implemented in child contracts.
- In `ERC777Token`, `ERC777TokensSender` and `ERC777TokensRecipient`, all function and event parameters defined as type `bytes32` should be defined as `bytes`.
- In `ERC777Token`, the `authorizeOperator(address)`, `revokeOperator(address)`, `send(address,uint256,bytes)` and `operatorSend(address,address,uint256,bytes,bytes)` functions return a success `bool` while the EIP 777 standard does not specify a return value for any of those functions.
- The `ERC777BaseToken` contract does not implement the `burn(uint256,bytes)` and `operatorBurn(address,uint256,bytes,bytes)` functions, which as mentioned, are missing from the `ERC777Token` interface. Augur developers acknowledge this situation in an inline comment.
- The `ERC777BaseToken` contract implements a public `sendNoHooks` function that allows the caller to transfer tokens bypassing the `tokensReceived` and `tokensToSend` hooks of the sender and receiver, a feature completely opposite to the EIP 777 specification.
- The `mint` function of `VariableSupplyToken` does not call the `callRecipient` function of `ERC777BaseToken`, thus never calling the receiver's `tokensReceived` hook. According to the EIP 777 spec, calling such hook is a MUST when minting tokens.
- The `Minted` event defined in `ERC777Token` is never emitted when minting tokens in the `mint` function of `VariableSupplyToken`. According to the EIP 777 spec, emitting such event is a MUST when minting tokens.

- According to the spec, the `tokensToSend` hook MUST be called *before* the token's state is updated. Similarly, the `tokensReceived` hook MUST be called *after* the token's state is updated. However, the function `transferFrom` of the `StandardToken` contract modifies the token's state (*i.e.* the allowances) before the `tokensToSend` hook is called.

Many of these particular noncompliances seem to be known by Augur's development team, who still decided to move forward with their custom implementation of EIP 777. This kind of decisions come with trade-offs. While the deviations from the spec may be more suitable for the Augur protocol, they might potentially cause errors in clients interacting with Augur that expect a fully-compliant implementation of the EIP 777. Therefore, it is advisable to either avoid calling the implemented token ERC777 altogether (its similarities and differences with the standard spec could be described in end-user documentation), or instead fully comply with the EIP's specification by following OpenZeppelin's ERC777 implementation, released in the 2.3.0 version.

Update: *in an attempt to fix this issue, where the* `mint` *function of the* `VariableSupplyToken` *was modified to call the ERC777* `tokensReceived` *hook, a critical regression error has been introduced. The* `migrateBalanceFromLegacyRep` *function of the* `OldLegacyRepToken` *contract must be called for every token holder to complete the migration, and as* `mint` *now calls the* `tokensReceived` *hook, any holder can revert attempts to mint new tokens for them. As a result, malicious tokens holders can prevent the migration from finishing (i.e. potentially never allowing the* `isMigratingFromLegacy` *flag to be set to* `false`*).*

[H02] EIP 820 implementation must be updated to EIP 1820

EIP 1820 has superseded EIP 820, the former being the latter's adaptation for Solidity 0.5, due to a bug that was found in ERC820 which would prevent certain calls made in the `staticcall` to never return a `true`.

Therefore, consider updating the specification to ERC1820. This will update the `insize` parameter of the `staticcall` from `0x08` to `0x24`. Contracts in `ERC820Implementer.sol`, `IERC820Registry.sol` `ERC820Registry.sol`, `ReputationToken.sol`, `DisputeWindow.sol`, and a number of other contracts using the ERC820 registry must all be updated to reflect the changes.

Update*: the ERC820 contract has been replaced with the ERC1820 contract in* `484e3ac`.

[H03] Affiliate fees are not accumulated in multiple trading operations

In Augur v2, market creators can set an `affiliateFeeDivisor` value which corresponds to the proportion of market creator fees that will be assigned to the market's affiliate when trading occurs.

For this purpose, the `Market` contract implements the function `recordMarketCreatorFees`, which can only be called by "known fee senders" and is in charge of calculating the amount of fees to be assigned to the creator and the affiliate (if exists). As trading can occur multiple times in a market, this function is expected to be called multiple times with fees for creator and affiliates accumulating each time, and it should distribute the fees to the corresponding parties only when the market is finalized.

However, while the fees are correctly accumulated for market creators, affiliates' fees are instead overwritten (*i.e.* they are assigned to the `affiliateFeesAttoCash` mapping while they should be added using `affiliateFeesAttoCash[_affiliateAddress] = affiliateFeesAttoCash[_affiliateAddress].add(_affiliateFees);`). As a consequence, if more than one trading operation of shares occurs in a market involving the same affiliate address, the affiliate will receive less fees than expected once the market is finalized.

Consider accumulating affiliate fees the same way market creators fees are currently accumulated. Furthermore, it is highly recommended to include unit tests that cover the described scenario, so as to make sure this issue is not reintroduced in future changes to the code base.

***Update**: fixed in `1b07fa4`. Affiliate fees are now properly incremented.*

[H04] Reentrancy vulnerabilities in Market contract

In the `Market` contract there are two reentrancy issues that allow for the removal of all of the contract's REP tokens. In the contract's current state, neither of these issues are exploitable because the contract never holds a REP token balance beyond the `repBond` being transferred in both instances.

In the `disavowCrowdsourcers` function, the `repBond` is transferred to the `repBondOwner` before the `repBond` is set to `0`. Because it is an ERC777-like transfer, the `tokensReceived` hook is called on `repBondOwner` if `repBondOwner` is an ERC820-registered contract. Should the `repBondOwner` address actually be an attacker-controlled contract, it could reenter the `Market` contract by calling back into the `disavowCrowdsourcers` function from the `tokensReceived` hook and continue to do this until all REP tokens are drained from the market.

A similar issue was identified in the `distributeInitialReportingRep` function. In this case, the attacker could call the `doInitialReport` function which calls the `doInitialReportInternal` function, which in turn calls the `distributeInitialReportingRep` function, finally executing an ERC777-like transfer to the designated reporter address. In a scenario where the designated reporter address is a malicious attacker-controlled contract, it could reenter the `Market` contract by calling back into the `doInitialReport` function and continue to do so until all REP has been drained from the market.

Despite neither of these vulnerabilities being exploitable, as in the current implementation the amount of REP held in a market is limited to the `repBond`, precautions should be taken to prevent an exploitable vulnerability from being introduced in the future. In this regard, consider always following the Checks-Effects-Interactions pattern to mitigate the chances of calls to external addresses resulting in an exploitable reentrancy vulnerability.

***Update**: fixed in `c6eed38`. Transfers of REP tokens in the `Market` contract no longer call the ERC777 hooks.*

[H05] Not following the Checks-Effects-Interactions pattern

Solidity recommends the usage of the Check-Effects-Interactions Pattern to avoid potential vulnerabilities, such as reentrancy. In several locations throughout the Augur code base (*e.g.* in `Market` contract functions `distributeInitialReportingRep`, `distributeMarketCreatorFees` and

`disavowCrowdsourcers`), calls to external, potentially attacker-controlled, contracts are made *before* making the necessary checks and modifications in the contract's storage. This can potentially expose serious reentrancy vulnerabilities, as discussed in the **"[H04] Reentrancy vulnerabilities in Market contract"** issue.

Strictly following the Check-Effects-Interactions pattern is of utmost importance in systems like Augur where most of the token transfers involve ERC777-like tokens, which explicitly call hook functions in the sender and receiver of each transfer operation. Therefore, the development team must ensure all interactions with external contracts are performed after all checks and state changes are made.

***Update**: Some instances of this issue have been fixed in `9f6ca45`.*

Update: [H06] Legacy REP tokens can be migrated to child universes

When reviewing the originally reported **"[L07] REP token allows migration of legacy tokens after universe fork"**, finally identified as a non-issue, the Augur team uncovered a serious vulnerability that would allow legacy REP tokens to be migrated to child universes in v2. In Augur team's words: *"This would allow v1 REP tokens to abstain from v2 forks safely and therefore bypass the v2 use-or-lose mechanism"*.

The fix entails allowing the migration of legacy REP tokens only in Genesis universes, and it was implemented in `48744d6`.

Medium severity

[M01] Missing docstrings throughout

Most of the contracts and functions in Augur's code base lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. In general, docstrings should explicitly explain the purpose or intention of functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

***Update**: fixed across a series of commits.*

[M02] Missing error messages in require statements

Most `require` statements in Augur contracts are lacking error messages. Consider including specific and informative error messages in all require statements, as they greatly improve code readability, making the code base more self-explanatory.

***Update**: partially fixed across a series of commits.*

[M03] Undocumented mathematical operations to compute Augur payouts

The Augur platform relies on a number of complex economic incentives to effectively align users' behavior, thus preventing misuse and attacks on the system. Such incentives are mostly based on stakes and fees that users and stakeholders deposit and receive in several different scenarios. Intending to make the platform as transparent as possible, the Augur team has implemented most of the calculations for fees, bonds, stakes and payouts in their smart contracts. However, such sensitive operations were found to be undocumented, rendering them extremely hard to follow and understand. Refer to the reported issues "**[M01] Missing docstrings throughout**" and "**[L02] Use of magic constants**" for related problems.

While attempts to map all calculations spread throughout the code base to the Augur's in-progress v2 whitepaper were made, still the manual process was unreliable and error-prone. Assessing for correctness becomes difficult when there is no way to straightforwardly understand the intentions behind each calculation, regardless of their simplicity.

Some examples of this issue can be found in:

- `Market.sol` : lines 216, 278 and 343
- `Universe.sol` : lines 83 to 84, 324 and calculations in functions `getOrCacheValidityBond` , `getOrCacheDesignatedReportStake` and `getOrCacheDesignatedReportNoShowBond`
- `DisputeCrowdsourcer.sol` : line 81 and several calculations made in the `redeem` function
- `AuctionToken.sol` : line 46
- `Auction.sol` : line 101, lines 203 to 205

Consider thoroughly documenting all sensitive calculations made in the code base, making explicit the rationale behind them where appropriate. As a starting point, this can be done in inline comments and docstrings; although it is highly advisable to include references to external more-detailed end-user documentation for v2 once it is ready. All of this will greatly improve the readability of the code, which should add to the platform's transparency and the users' overall experience.

***Update**: fixed across a series of commits.*

[M04] Undocumented REP price auction mechanism

The `Auction` contract, together with the `AuctionToken` contract, implement the logic behind the built-in auction-based REP price oracle introduced in Augur Core v2. Although the logic appears to be fairly straightforward, no detailed documentation (apart from the unit tests) was found regarding how the auction is expected to work. The lack of explicit explanations about the functions' intended behavior and the rationale behind arithmetic operations (observed throughout the entire Augur code base, as more generally described in "**[M01] Missing docstrings**

throughout" and "**[M03] Undocumented mathematical operations to compute Augur payouts**") prevent us from thoroughly assessing the correctness and security of these important mechanisms that will govern the REP price in the future.

Consider thoroughly documenting the entire REP price auction mechanism, both in docstrings and external end-user documentation. This should greatly add to the project's maintainability and transparency, helping future developers, users and auditors evaluate the code's level of security and correctness.

***Update**: fixed by removing th* `Auction` *contract in* `f641c42` .

[M05] Lack of input validation

Several functions in the Augur code base lack explicit checks of user-controlled parameters. While this practice is often used as a way to reduce gas costs, under no circumstances should the lack of input validation undermine security nor functionality. Some examples of issues of varying severity that steam from unsanitized input are "**[C01] All CASH tokens approved to Augur can be emptied**", "**[L04] Externally-owned accounts can be registered as contracts in the Augur contract**" or even "**[M08] Factories may unexpectedly fail to create proxies**".

Consider implementing require statements where appropriate to validate all user-controlled input. Including clear user-friendly error messages (as reported in "**[M02] Missing error messages in require statements**") is highly recommended as well.

***Update**: input validation has been implemented across a series of commits.*

[M06] Lack of event emission when market's dispute phase starts pacing on

Following what is stated in Augur's whitepaper, if a market's tentative outcome is disputed with a bond greater than 0.02% of all REP but less than 2.5% of all REP, then the market enters the waiting for next fee window to begin phase, before undergoing another dispute round. The purpose of this is simply to slow down the dispute process as the bonds get larger, therefore giving honest participants more time to crowdfund the larger dispute bond.

The `Market` contract implements such behavior toggling a boolean flag called `disputePacingOn`, which once set to `true`, will prevent further contributions to the market's current tentative outcome. However, the `Market` contract never emits an event in such scenario.

Considering how important and sensitive this stage in a market's dispute phase is, consider defining and emitting an event in order to effectively notify off-chain clients about it.

***Update**: fixed in* `27be8c5` *. The* `DisputeCrowdsourcerCompleted` *event now logs a flag that indicates whether the market's dispute phase is pacing on.*

[M07] Unchecked return value in Ownable contract

In the `Ownable` contract's `transferOwnership` function, the abstract internal function `onTransferOwnership` is called but the boolean value it returns is never checked. Hence, if a subclass of `Ownable` were to return `false` from `onTransferOwnership` to prevent ownership transfers, all transfers would still succeed.

In the Augur code base, there are no contracts currently using `onTransferOwnership` to prevent ownership transfers. Yet, it is advisable to either check the return value in a `require` statement (to prevent introducing issues in future changes) or remove the return boolean value altogether.

Update: fixed in `5bc7904` . The internal `onTransferOwnership` function no longer returns a boolean flag.

[M08] Factories may unexpectedly fail to create proxies

Augur's factories are `CloneFactory` contracts in charge of creating clones of minimal proxies (following EIP 1167). The target addresses of such proxies are queried from the `Augur` contract by calling its `lookup` function with a particular key.

A problem may arise when a key has not been registered in the `Augur` contract, since the `lookup` function always defaults to return the zero address in such case. This address will then be set as the target address of the created minimal proxy, which will therefore attempt to delegate all calls to the zero address. After wrapping the proxy with the corresponding interface, all factories call the target's `initialize` function and expect a boolean value in return. As this value will *not* be present in the data returned by the proxy's target contract, the transaction will be unexpectedly reverted without a clear nor informative reason.

Consider implementing, in all factories, the necessary validations on the address returned by the `lookup` function to make sure it is different from the zero address, thus avoiding unexpected failures in Augur's factories.

***Update**: The Augur team decided not to move forward with our recommendation, since the transaction is reverted when calling the `initialize` function on a clone that points to the zero address.*

[M09] Calls to sellCompleteSets function in FillOrder contract can be unexpectedly reverted

In the `CompleteSets` contract, the `sellCompleteSets` function is used to burn a complete set of outcome tokens in exchange for the underlying CASH token for a given market. This function is used by the `FillOrder` contract's `fillOrder` function when the parameter `_ignoreShares` is set to false.

The `sellCompleteSets` function, in line 100 of `CompleteSets.sol` , calls the `destroyShares` function which then calls the `VariableSupplyToken` contract's `burn` function, that in turn calls the `ERC777` hook on the `_sender` if the `_sender` is an ERC820-registered contract. This can be leveraged by a malicious `_sender` by reverting transactions when `sellCompleteSets` is called. This includes any calls to the `fillOrder` function when the `_ignoreShares` parameter is set to false.

Consider not calling the ERC777 token hooks when outcome tokens are burned by the `sellCompleteSets` function. This will eliminate the risk of a malicious party unexpectedly reverting transactions.

***Update**: Fixed in `c6eed38` . The call to the `VariableSupplyToken` contract's `burn` function no longer triggers the ERC777 hook.*

Low severity

[L01] Lookup key strings are not centrally defined

Known Augur contracts are tracked in the `registry mapping` of the `Augur` contract. New entries can be added by a privileged address via the `registerContract` function, and the `lookup` function acts as a public getter to query the registry providing a string-type key. While this registry is used by several different contracts to get the addresses of legitimate Augur contracts, the strings used as keys to query the registry are not centrally defined. The identified strings are: `"ReputationTokenFactory"`, `"AuctionFactory"`, `"MarketFactory"`, `"DisputeWindowFactory"`, `"CompleteSets"`, `"UniverseFactory"`, `"CreateOrder"`, `"CancelOrder"`, `"FillOrder"`, `"Trade"`, `"ClaimTradingProceeds"`, `"Orders"`, `"Time"`, `"Cash"`, `"ProfitLoss"`, `"Map"`, `"Market"` and `"ShareToken"`.

This issue does not pose a security risk, but the approach taken is very error-prone and difficult to maintain. Therefore, consider factoring out all mentioned constant strings to a single library, which can be then imported in the necessary contracts. This will ease maintenance and make the code more resilient to future changes.

***Update**: the Augur team decided not to move forward with our recommendation, arguing that centrally defining the constants would add unnecessary complexity and some additional gas costs related to the resulting size of the contracts.*

[L02] Use of magic constants

There are several occurrences of magic constants in the Augur code base, an issue deeply related with what is being reported in "**[M03] Undocumented mathematical operations to compute Augur payouts**". These values make the code harder to understand and to maintain.

The following is a non-extensive list of some locations in the code where magic constants can be found:

- Line 73 in `ReputationToken.sol`
- Line 216 and 228 in `Market.sol`
- Lines 81 to 84, 444 to 445, 451 to 452 and 460 in `Universe.sol`

Consider defining a constant variable for every magic constant (including booleans), giving it a clear and self-explanatory name. For complex values, or in cases where defining a constant variable does not seem appropriate, consider adding an inline comment explaining how they were calculated or why they were chosen. All of this will allow for added readability, easing maintenance.

***Update**: fixed in `fe4ccb2` by defining new constant state variables and adding explanatory inline comments in the `Universe` and `Reporting` contracts.*

[L03] The Cash contract should not be an ERC777 token

The `Cash` contract inherits from the `VariableSupplyToken` contract which is an ERC777 token (although with several major deviations, as reported in "**[H01] Implementation of EIP 777 does not fully match the specification**"). Augur's contracts are built to be able to use any ERC20 token (such as DAI) as the "Cash token", which is used for trading as well as other purposes. Should the Cash token be implemented as an ERC777 token in production, it would potentially render other Augur contracts vulnerable to serious unexpected exploits such as reentrancy and denial of service attacks. Therefore, consider removing all ERC777 functionality from the `Cash` contract to avoid any unexpected behaviors, clearly documenting that the Cash token should not implement ERC777-related features, but only ERC20.

***Update**: fixed in `c0ad321` by turning the `Cash` contract into an ERC20.*

[L04] Externally-owned accounts can be registered as contracts in the Augur contract

The `registerContract` > function in the `Augur` contract is a privileged function (only called by the `uploader address`) in charge of registering contract addresses in the `registry mapping`, by associating the address with a given `key`. However, while the function is only intended to register contract addresses, there is currently no validation on whether the address provided is either a contract or an externally-owned account.

While this issue does not pose a security risk, consider using the `exists` function of the `ContractExists` utility library to ensure the address being registered is indeed a contract.

***Update**: fixed in `eb10e94` by using the `exists` function to validate the address being registered is indeed a contract.*

[L05] Redundant Mint and Burn events in VariableSupplyToken contract

In the `VariableSupplyToken` contract, consider reusing the `Burned` and `Minted` events inherited from the `ERC777Token` contract, thus removing the defined `Burn` and `Mint` events.

***Update**: fixed in `484e3ac`. The `Burn` and `Mint` events in the `VariableSupplyToken` have been removed and the `burn` and `mint` functions now call the inherited `_mint` and `_burn` functions.*

[L06] LegacyReputationToken's decimals public getter does not return uint8

The ERC20 specification specifies an optional getter for the token's `decimals`, which must return a `uint8` type. While the `LegacyReputationToken` does implement this getter, it returns a `uint256` instead. Consider changing it to `uint8` to be fully ERC20 compliant. If `uint256` was used intentionally to match the currently deployed REP token implementation, consider thoroughly documenting that reasoning.

***Update**: fixed in `43a8d57` . Now the `decimals` public getter returns a `uint8` type.*

[L07] REP token allows migration of legacy tokens after universe fork

The `ReputationToken` contract implements the `migrateFromLegacyReputationToken` function that allows users to migrate legacy reputation tokens in. However, there is no validation in place to prevent this migration from occurring in a REP token whose universe has forked and that fork has already finished.

To prevent unexpected locking and loss of tokens, consider implementing the necessary validations in the `migrateFromLegacyReputationToken` function to ensure no legacy tokens can be migrated to a REP token of a locked universe whose fork period has finished.

***Update**: this has been identified as a non-issue by the Augur team, since even after a fork users should be able to migrate to the Genesis universe. Yet, our report led the Augur team to uncover a serious vulnerability which we included in this report, for the sake of completeness and transparency, in the new "**[H06] Legacy REP tokens can be migrated to child universes**" issue.*

[L08] Getter for REP total theoretical supply may be inaccurate

The `ReputationToken` contract implements the `getTotalTheoreticalSupply` public getter function to retrieve the value of the token's totalTheoreticalSuppy. While the `totalTheoreticalSuppy` is a fixed value in the genesis universe, it is variable in a child universe whose parent universe fork is already finished. However, for the `totalTheoreticalSupply` to change, the `updateTotalTheoreticalSupply` function must be manually called. Therefore, if the total supply of REP tokens has changed over a period of time during which the `updateTotalTheoreticalSupply` function was never called, the value returned by the `getTotalTheoreticalSupply` function will be outdated and potentially inaccurate.

Consider implementing a more general approach to always keep in sync the `totalTheoreticalSupply` with the total supply of REP tokens in a child universe whose parent's fork has already finished, as described in the reported **"[L09] System relies on off-chain clients calling unrestricted functions to stay up-to-date"** issue. For this case in particular, the `updateTotalTheoreticalSupply` function could be called before returning the `totalTheoreticalSupply` inside the `getTotalTheoreticalSupply` function.

***Update**: fixed in `48744d6` . The `getTotalTheoreticalSupply` function now calculates and returns the updated value of the total theoretical supply of REP tokens.*

[L09] System relies on off-chain clients calling unrestricted functions to stay up-to-date

The Augur system relies on off-chain clients to call unrestricted functions in order to update and sync system parameters. In particular, public functions whose purpose is to update state, like `updateTotalTheoreticalSupply` and `updateForkValues` may be dangerous if called at an unexpected time. If the contract relies on a certain state during various stages of execution, having these functions available to be called by anyone may result in an undesired state change that may render the system inconsistent or vulnerable to an attack. Consequences of this issue

can be seen in **"[C08] Fork reputation goal threshold can be decreased during fork"** or **"[L08] Getter for REP total theoretical supply may be inaccurate"**.

Consider removing any state variables used to store dynamically calculated values such as `totalTheoreticalSupply` in the `updateTotalTheoreticalSupply` function and `forkReputationGoal` and `disputeThresholdForFork` in the `updateForkValues` function. Instead, access these values through getter functions that recalculate the values as needed. This will guarantee they are up to date and accurate and remove any dependencies on off-chain systems.

***Update**: fixed in `48744d6` by removing the `totalTheoreticalSupply` state variable from the `ReputationToken` contract and modifying the `getTotalTheoreticalSupply` to recalculate and return the updated value of the total theoretical supply of REP tokens every time it is called.*

[L10] Current Market owner may not receive no-show bond after initial report

All market creators stake a no-show bond (paid in REP tokens) upon creating a market, which is to be automatically returned if the designated reporter reports a tentative outcome within the first 24 hours after the market's end time. When a `Market` is initialized, the `owner` and `repBondOwner` state variables (the first one inherited from the `Ownable` contract) are set to the market creator's address (see lines 78 and 79).

In a scenario where the ownership of a market has been transferred by calling the `transferOwnership` function and the designated reporter effectively shows up, the no-show bond is going to be still transferred back to the original owner of the market, tracked in the `repBondOwner` state variable, and not to the current owner of the market. This behavior is inconsistent with what occurs in the `InitialReporter` contract, where the stake received after the initial report is awarded to the current owner of the contract, and not to the original (*i.e.* the actual reporter).

Should this be the system's expected behavior, consider explicitly documenting such sensitive scenarios to raise end-user awareness. Adding related unit tests to increase coverage is advisable too.

***Update**: fixed in `142ed41` by adding a new `transferRepBondOwnership` function that must be called manually by the address in `repBondOwner` to transfer the ownership of the REP bond. However, the added function does not emit an event to notify off-chain clients of such sensitive change.*

[L11] Unnecessary code repetition in Auction contract

The `Auction` contract implements the functions `getAuctionStartTime` and `getAuctionEndTime`, which implement the exact same logic except for line 244. Therefore, to favor simplicity and modularization, consider factoring out the repeated logic into a private function.

***Update**: this issue is no longer valid as the `Auction` contract has been removed in `f641c42`.*

[L12] Unnecessary code repetition in CompleteSets contract

The `CompleteSets` contract includes the functions `publicSellCompleteSets` and `publicSellCompleteSetsWithCash`, which implement the exact same logic, both calling the `sellCompleteSets` function. Similarly, the functions `publicBuyCompleteSets` and `publicBuyCompleteSetsWithCash` also behave the same, in this case calling the `buyCompleteSets` function. Therefore, so as to favor simplicity, avoid confusions and reduce the code's attack surface, consider removing one in each pair functions.

***Update**: fixed in `ebed2db` by removing the `publicSellCompleteSetsWithCash` and `publicBuyCompleteSetsWithCash` functions.*

[L13] Outdated ReentrancyGuard contract in use

The `ReentrancyGuard` contract currently in use is out of date. While its `nonReentrant` modifier works as expected, consider updating the contract to the latest version in order to benefit from an optimized gas usage for small transactions. Refer to OpenZeppelin's issue #1056 for more details about such gas savings.

***Update**: the Augur team decided not to implement our recommendation because the latest version of the `ReentrancyGuard` contract has an additional local variable which causes stack depth issues in some of their contracts.*

[L14] Inconsistent use of afterInitialized modifier in contracts

Several contracts in Augur's code base are intended to be deployed behind the EIP 1167 minimal proxy contract. Therefore, to be able to replace the `constructor` function, they all inherit functionality from the `Initializable` contract. It provides two modifiers `beforeInitialized` and `afterInitialized`, which are used to allow / deny access to a function based on the `initialized` boolean flag, which can be toggled to `true` (and never be set back to `false`) by calling the internal `endInitialization` function.

An inconsistent use of the `afterInitialized` modifier was found throughout Augur's contracts. Where in many contracts `afterInitialized` seems to be correctly used to label every function that can only be called *after* initialization (*e.g.* in `DisputeWindow`), in other contracts (*e.g.* `Market`) the `afterInitialized` modifier is never used.

Consider making the necessary changes in the contracts to consistently use the `afterInitialized` modifier, adding related unit tests that prevent developers from reintroducing this kind of issues. Thorough testing is in order should the development team decide to remove the `afterInitialized` modifier from all functions in contracts that are deployed through factories (which call the implementation's `initialize` function), so as to ensure behavior is not affected and security not compromised in any sense.

***Update**: fixed in `aff5118` by removing all `afterInitialized` modifiers.*

[L15] Multiple unused return values

In multiple locations in the Augur code base, there are private and internal functions that return boolean values indicating the function's success. However, the functions often either revert or return true and their return values are not checked. Many public functions in the Augur code base

also follow this pattern but the Augur team clarified that the return values of the public functions are intended for off-chain systems. Consider removing any unnecessary return values to avoid confusion about their intended purpose.

***Update**: Fixed in `2921365` by removing several unused return values.*

[L16] Multiple getters for the same state variable

Several contracts in the Augur code base contain multiple public getter functions for the same state variable. For example:

- In the `ERC777BaseToken` contract, there are multiple getter functions that return the token's total supply. Namely, `supply` (automatically generated by Solidity) andtotalSupply.
- In the `Augur` contract, there are multiple getter functions that get the contract registered with a given key. Namely, `registry` (automatically generated by Solidity) and `lookup`.

To favor encapsulation and explicitness, ensure that there is at most one publicly exposed getter for each contract state variable.

***Update**: fixed in `66745b6` by restricting the visibility of `supply` and `registry`.*

[L17] Not using safe arithmetic operations

Several arithmetic operations in the code base are not using the available `SafeMathUint256` and `SafeMathInt256` libraries that would prevent arithmetic issues. See for example line 455 in `Market.sol`, line 147 in `Universe.sol` or lines 245, 246, 249 and 250 in `Orders.sol`.

While this issue does not pose a security risk, as no exploitable overflows or underflows were detected in the current implementation, this may not hold true in future changes to the code base if unit testing is not effectively covering all cases.

Given that arithmetic operations on integers may overflow / underflow silently, causing bugs, consider using the existing `SafeMathUint256` and `SafeMathInt256` libraries for all arithmetic operations. Unit tests to ensure correct behavior are of utmost importance in cases where safe arithmetic operations are not used in favor of gas efficiency.

***Update**: fixed in `b138335` for all of instances of the issue that we pointed out as examples.*

[L18] Undocumented assembly blocks

The `CloneFactory` contract in `CloneFactory.sol` includes a function `createClone` with an assembly block. Even though the function is taken from EIP 1167 where the functionality is documented, assembly is a low-level language that is harder to parse by readers. Consider including extensive inline documentation clearly explaining what every single assembly instruction does. This will make it easier for users to trust the code, for reviewers to verify it, and for developers to build on top of it or update it. Note that the use of assembly discards several important safety

features of Solidity, which may render the code less safe and more error-prone. Hence, consider implementing thorough tests to cover all potential use cases of the `createClone` function to ensure it behaves as expected.

Update: *fixed in* `fe274f5` *by adding explanatory inline comments for each assembly instruction.*

[L19] Outdated and inaccurate README file

The README.md file in the augur-core repository is out of date and inaccurate. It does not reflect the v2 changes, some of the suggested commands do not work, and the build is failing. In particular, some of the identified issues are:

- The "Source code organization" section is missing the `legacy_reputation` folder.
- In the "Docker—Test" subsection there is a link pointing to Augur's old, deprecated repository. This is the same in the "Worst-case-loss for trades" section.
- The Reporting flow diagram mentioned in "Reporting diagrams" is outdated.
- There are instructions to run Oyente, a smart contract analysis tool that has not been updated in more than a year and is unclear if it works properly with Solidity +0.5. According to the Oyente's output when run with a test contract: *"The latest supported version is 0.4.19"*.
- The list of files and folders in the "Tests" section is clearly outdated, with many inconsistencies (*e.g.* `delegation_sandbox.py` and `test_mutex.py` are listed but do not exist in the `tests` folder).
- The coverage report is outdated and the suggested command does not work.

README files on the root of git repositories are the first documents that most developers often read, so they should be complete, clear, concise and accurate. To define the structure and contents of this file, consider following Standard Readme. Furthermore, it is highly advisable to include instructions for the responsible disclosure of any security vulnerabilities found in the project.

Update*: fixed in* `c629b50` *by updating the README file where appropriate.*

[L20] Outdated test coverage report

The test coverage report has not been updated for Augur Core v2 code base. Without this report it is impossible to know whether there are parts of the code never executed by the automated tests; so for every change, a full manual test suite has to be executed to make sure that nothing is broken or misbehaving.

Consider updating the test coverage report, and making it reach at least 95% of the source code.

Update*: fixed in* `88e7aa0` *by revamping the entire test coverage setup.*

Notes & Additional Information

***Update**: several of the following soft recommendations have been implemented across various commits.*

Naming

- The `Cash` contract should be renamed to `CashMock` to denote it is just a testing contract with no relevant functionality for the Augur system. Moreover, consider moving it to a `mocks` folder.
- The `TimeControlled` contract implements functionality to handle a private timestamp. On the assumption that it is just a mock contract used for testing purposes (as it does not implement any security validations), consider renaming it to `TimeControlledMock` to explicitly denote *it is not* going to be used in production. Additionally, consider moving it to a `mocks` folder.
- In the `Market` contract, consider renaming the `APPROVAL_AMOUNT` constant to `MAX_APPROVAL_AMOUNT` or similar, so as to better denote it represents the maximum amount of tokens that can be approved.
- In the `Market` contract, consider renaming the `distributeMarketCreatorFees` function to `distributeMarketCreatorAndAffiliateFees` or similar, to explicitly denote the function also takes care of distributing the affiliate's fees when appropriate.
- In the `Auction` contract, consider renaming the `auctionOver` function to `isAuctionOver`, so as to better denote it returns a boolean flag indicating whether an auction is over or not.
- In the `Auction` contract, inside the `initializeNewAuction` function, consider renaming the `_currentAuctionIndex` local variable to `_newAuctionIndex`, which should prevent confusions with the `currentAuctionIndex` state variable. This will add to the code's readability, avoiding expressions such as require(currentAuctionIndex != _currentAuctionIndex);.
- In the `Auction` contract, consider renaming the `getRoundType` function to `getCurrentRoundType`.
- In the `DisputeWindow` contract, consider renaming the functions `buy` and `redeem` to `buyParticipationTokens` and `redeemParticipationTokens` respectively.
- In the `Universe` contract, consider renaming the `createChildUniverse` function to `getOrCreateChildUniverse`, as it does not always create a child universe, but under certain circumstances just retrieves an existing one.
- In the `Universe` contract, consider renaming the mappings `validityBondInAttoCash`, `designatedReportStakeInAttoRep` and `designatedReportNoShowBondInAttoRep` in order to make them more self-explanatory. The naming should denote that they contain stake and bond values for different dispute windows.
- For consistency in naming in the `Augur` contract, consider renaming the function `disputeCrowdsourcerCreated` to `createDisputeCrowdsourcer`.
- In the `Augur` contract, consider renaming the `isValidMarket` function to `isKnownMarket`. This should avoid confusions between the notion of a legit Augur market (which is what the function verifies) and the potential valid or invalid outcomes of a market.

- Consider changing the `ERC20Token` contract to be an `interface`, then renaming it to `IERC20` to explicitly denote it is an interface. For consistency, this change should also be reflected in the file's name.
- The `_amountToMint` parameter in the `burnForAuction` function of the `IV2ReputationToken` contract should read `_amountToBurn`.

Typos

- There is a typo in an inline comment in the `Auction` contract. In line 40, `Indicies` should read `Indices`.
- There is a typo in an inline comment in the `AuctionToken` contract. In line 21, `recieved` should read `received`.
- There is a typo in an inline comment in the `ReputationToken` contract. In line 62, `tenative` should read `tentative`.
- There is a typo in an inline comment in the `Universe` contract. In line 84, `occuring` should read `occurring`.

Coding style

- Following the Solidity Style Guide wherever possible is advisable. There are a few occurrences in the Augur code base where the guide is not strictly followed, the most relevant point to highlight being long function declarations that should be stacked vertically for readability; from the Solidity Style Guide, *"keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code"*. This can be consistently enforced across the entire code base by means of an automatic linter, such as Solhint.
- In order to keep a consistent style throughout the project, consider prepending an underscore to the parameter `newOwner` in the `transferOwnership` function of the `IOwnable` interface, to match its counterpart `_newOwner` in the Ownable contract.
- Consider removing the named return variable `_market` in the `createMarket` function of the `MarketFactory` contract, and instead declare it as a regular local variable in line 14 of `MarketFactory.sol`. This suggestion should be applied in all functions were named return variables are used.
- The file `OldLegacyRepToken.sol` does not have the same name as the contract it contains; namely, `OldLegacyReputationToken`. According to the Solidity Style Guide: *"Contract and library names should also match their filenames"*. Consider changing the file's name to `OldLegacyReputationToken.sol` to match the contract.
- So as to explicitly denote their visibility and favor readability, consider prepending an underscore to the name of all `internal` and `private` functions.

Documentation

- The private `redistributeLosingReputation` function in the `Market` contract includes an inline comment that reads *"We burn 20% of the REP to prevent griefing attacks which rely on getting back lost REP"*. Considering that the following line just burns the entire REP balance of the market, further explanations should be added to better describe how that 20% is calculated, and how it matches the market's balance at that point.

- The public `doInitialReport` function in the `Market` contract allows a reporter to give a description for their report by providing a string in the function's `_description` parameter. However, the given description is never actually stored in any contract, but just logged in an event by calling the `logInitialReportSubmitted` function of the `Augur` contract. The same is true for `contributeToTentative` and `contribute` functions, which only log the passed description calling `Augur`'s `logDisputeCrowdsourcerContribution` function. Should this be the system's expected behavior, consider documenting it and adding related unit tests.
- To indicate the payout value for each outcome at market finalization, Augur uses an array called `payoutNumerators`. Although this fundamental data structure is extensively used across multiple contracts, its purpose nor its contents are ever clearly documented. An outdated description can be found in Augur v1 documentation, which should be modified to reflect the important changes that the payout array suffered in v2 (*e.g.* the "Invalid" outcome is now considered an explicit outcome). Therefore, for clarity, consider renaming this array to `payoutsForOutcomes` or similar. Furthermore, consider thoroughly documenting the purpose and contents of this data structure in Augur Core v2, either in external end-user documentation or in docstrings (refer to related reported issue **"[M01] Missing docstrings throughout"**).
- The `exists` function of the `ContractExists` library will return `false` if it is called from the constructor of a contract, due to the fact that extcodesize is `0` during the execution of the constructor of a contract. Ensure to clearly document this behavior in the function's docstrings, so as to prevent potential issues in future versions of the code base.
- In the `Auction` contract, there is an inaccurate inline comment that reads: *"This will raise an exception if insufficient CASH was sent"*. As the operation it refers to will execute a `transferFrom` of CASH tokens, consider rephrasing the comment as *"This will raise an exception if insufficient CASH was approved or the sender has an insufficient CASH balance"*.
- To avoid confusions, consider deleting the inline comment in line 17 of `FillOrder.sol`, and properly document it in the project's backlog of issues.
- The FXP formulas documented in inline comments in lines 386 and 394 of `Universe.sol` are incorrect and must be updated. Both formulas should have a parentheses after the initial `previous_amount` that encompass the remainder of the formula.

Code organization, simplicity and cleanliness

- In the `IAugur` contract, consider importing `IDisputeWindow.sol` directly and removing the import statement for `IDisputeCrowdsourcer.sol`. The `IDisputeCrowdsourcer` contract is never used in `IAugur`.
- In the `Augur` contract, consider removing the `IInitialReporter.sol` import as it is never used.
- To improve the contract's readability and organization, consider moving all events defined in the `Augur` contract to the `IAugur` interface.
- To favor simplicity and avoid confusion, consider deleting the `ICash` contract as it does not add functionality on top of the `ERC20Token` from which inherits. Then, all instances of `ICash` should be renamed to `ERC20Token` (which in turn should be renamed to `IERC20`, as suggested in another note in this report).
- The `ReputationToken` contract seems to unnecessarily redefine the functions `transfer` and `transferFrom`, inherited from the `StandardToken` contract. As they do not implement new functionality, but just call the parent's `transfer` and `transferFrom` functions

respectively, they could be removed from the `ReputationToken` contract in order to reduce its attack surface.

- Consider removing the `maxSupply` state variable in the `IAuction` contract, as it is never used in the child `Auction` contract. Similarly, in the `Auction` contract, consider removing the `feeBalance` state variable.
- In the `AuctionFactory` contract, consider importing `IAuction.sol` directly and removing the import statement for `Auction.sol`. The `Auction` contract is never used in `AuctionFactory`.
- In the `Universe` contract, consider removing the import statement for `IRepPriceOracle.sol`, as the contract `IRepPriceOracle` is never used in `Universe`.

Consistency

- The `AuctionFactory` and `ReputationTokenFactory` contracts do not inherit from the corresponding interfaces, namely `IAuctionFactory` and `IReputationTokenFactory`. Furthermore, in the case of the `AuctionFactory`, there is a mismatch between the interface and the contract; whereas the former declares a `createAuction(IAugur,IUniverse,IReputationToken)` function, the latter defines it as `createAuction(IAugur,IUniverse,IV2ReputationToken)`. Consider fixing these issues to favor consistency between interfaces and implementation contracts.
- Factories `ReputationTokenFactory` and `TestNetReputationTokenFactory` return references to contracts with different interfaces (`IV2ReputationToken` and `IReputationToken` respectively). Should this be the factories' expected behavior, consider clearly documenting this mismatch in interfaces between testnet and production tokens.

Refactoring

- In line 236 of `Market.sol`, there is a check to ensure the market has not been finalized before executing the rest of the `finalize` function. Consider using `require(!isFinalized());` for increased readability and to clarify intentions.
- In line 243 of `Market.sol`, consider reusing the `getWinningReportingParticipant` getter to retrieve the last element in the `participants` array.
- As it stands, the `getWinningPayoutNumerator` function in the `Market` contract will revert the transaction if the market is not finalized. As such behavior is uncommon for a simple getter function, consider removing the `require` statement and explicitly checking if the market is indeed finalized (using the `isFinalized` function) independently of the `getWinningPayoutNumerator` function.
- We recommend turning all instances of the redundant require statements into modifiers. In general, this is advisable whenever the same code is used in more than one location in the same file, which should help abide by DRY programming principles. For example, in the `Augur` contract, all instances of `require(msg.sender == uploader);` could be replaced with an `onlyUploader` modifier.
- In lines 359 and 449 of `Augur.sol`, consider reusing the `isKnownUniverse` function to verify that the address corresponds to a legit universe in the Augur system.

- In several locations in the Augur code base, external functions are called from within the same contract. See for example the line 36 in `CompleteSets.sol`. This pattern is confusing and unnecessary to achieve the desired effects. Consider separating out functionality into internal functions that may be called from the different external functions rather than having external functions call each other. This will make the intended function permissions more clear and be more in line with the intended use of Solidity's `external` and `internal` functions.

Explicitness

- To favor explicitness, all instances of `uint` should be declared `uint256` (*e.g.* see line 15 of `ERC777BaseToken.sol`).
- Consider always explicitly casting all instances of `this` using `address(this)`. Similarly, all typed variables that refer to a contract or interface should also be explicitly casted when used as an `address` (*e.g.* in line 164 of `OldLegacyRepToken.sol`, `_token` should be `address(_token)` and `this` should be `address(this)`).
- In the `getUint256Max` function of the `SafeMathUint256` library, consider changing the returned value from `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff` to `2 ** 256 - 1`, which should greatly favor readability, making the code more self-explanatory and less error-prone.
- In the `Time` contract, the return value for the `getTypeName` function is being implicitly type casted from a `string` to `bytes32`. Consider explicitly casting the `string` to `bytes32` before returning it for increased readability and to clarify the intentions of the code.
- The functions `isMultipleOf` and `div` of the `SafeMathUint256` library, and `div` of the `SafeMathInt256` library, allow the divisor `b` to be zero. Solidity automatically asserts when dividing by zero, but to favor explicitness, consider adding `require` statements that check the divisor is not zero. This pattern can be seen consistently applied in the `div` and `mod` functions of OpenZeppelin's `SafeMath` library.

Misc

- In the `Auction` contract, the `tradeCashForRep` function has an unnecessary `payable` modifier that must be removed to avoid locking Ether in the contract (as there is no functionality implemented to withdraw any Ether sent).
- To improve gas efficiency, consider modifying the functions `mul` in `SafeMathUint256` and `mul` in `SafeMathInt256` to simply return `0` when the first parameter `a` equals 0. For reference, check the latest OpenZeppelin's `SafeMath` library.

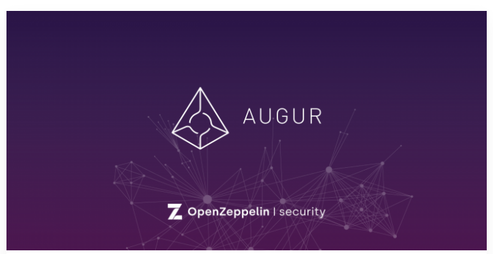
Conclusions

Originally, 8 critical and 5 high severity issues were found. After reviewing Augur team's fixes based on our initial report, 1 critical issue was disregarded and 1 additional High severity issue was added. Several changes were proposed to follow best practices and reduce the potential attack surface.

Security Audits

- If you are interested in smart contract security, you can continue the discussion in our forum, or even better, join the team 🚀
- If you are building a project of your own and would like to request a security audit, please do so here.

RELATED POSTS



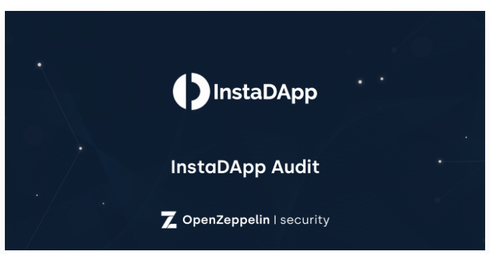
SECURITY AUDITS

Augur Core Audit

The Augur team asked us to review and audit their Augur Core project. We looked at the code and now...

READ MORE

by OpenZeppelin Security



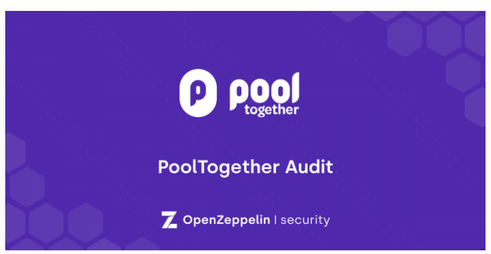
SECURITY AUDITS

InstaDApp Audit

The InstaDApp team asked us to audit their proxy wallet and wallet registry contracts. We examined...

READ MORE

by OpenZeppelin Security



SECURITY AUDITS

PoolTogether Audit

The PoolTogether team asked us to review and audit the system. We looked at the code and now...

READ MORE

by OpenZeppelin Security