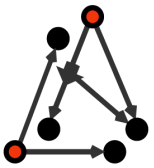

Least Authority
Freedom Matters

Security Audit Report

Augur Solidity Contracts

Report version: 12.01.2017

Overview

Augur has requested Least Authority perform a security audit of their Augur-Core trading Solidity contracts. Augur migrated their contracts to Solidity due to a vulnerability discovered in the Serpent Compiler by a different audit. For this audit, we will look at all of the Augur contracts: https://github.com/AugurProject/augur-core/tree/master/source/contracts

Coverage

Target Code and Revision

For this audit, we reviewed the latest stable releases of the Solidity contracts code found at: https://github.com/AugurProject/augur-core/tree/master/source/contracts

Specifically, we examined the Git revisions:

```
45e1afb7eb1a895d923c97fe01e068c772c583ef
```

All file references in this document use Unix-style paths relative to the project's root directory.

Manual Code Review

In manually reviewing all of the contract code, we looked for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also kept an eye out for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus was on the contract code, we examined dependency code and behavior when it was relevant to a particular line of investigation.

Our investigation focused on the following areas:

- Incentives around markets, reporting, and the appeal process
- Exploiting the 'escape hatches' in the controller contract
- Examining the rate adjustment code, and reasoning about Augur economics
- Reporting code, Market.sol, Reporting.sol, ReportingWindow.sol, StakeToken.sol
- And the trading folder

Findings

Code Quality

Overall, we found the code to be high quality. The code generally follows Ethereum best practices (i.e. Checks-effects-interactions pattern), with departures for legibility, where inconsistency in implementation hinders it. Consideration is given to preventing re-entrancy and other common issues.

Issues

We list the issues we found in the code in the order we found them.

Issue / Suggestion	Status
Issue A: Repetitive forking attack with disputes	Reported 21-11-17
Issue B: Settlement may error if perceived REP price falls	Reported 24-11-17
Issue C: Whitelist exploitation and Dev Mode reactivation	Reported 24-11-17
Issue D: Reporter bond adjustment can get stuck at low values	Reported 24-11-17
Issue E: Orderbook vulnerable to Miner frontrunning	Reported 26-11-17

Issue A: Repetitive forking attack with disputes

Reported: 21-11-17

Synopsis: An attacker with control of a large amount of REP can repetitively dispute results, regardless of their truth. This results in a repeatable system-wide denial of service.

Impact: Successful exploit leads to long (~120 day) settlement delays for all markets. The attack can be repeated as long as the attacker holds enough REP.

Preconditions: The attacker must hold and be willing to devalue a large amount of REP. The attacker needs to hold 137,500 REP and to repeat indefinitely, at most ~26.25% of the total supply, and at least, 7.5% with some additional effort.

Feasibility: Trivial to execute. The Augur UI will provide features for disputing markets.

Technical Details: Any user may call `disputeLastReporters()` in *Market.sol* to dispute the final report. To do so, the user must post a large REP bond. Disputing the final result creates a "fork." A child Universe is created for each possible market outcome. Users may then migrate REP to any child Universe.

The fork is active for 60 days (the "Fork Period"). While the fork is active, markets will not finalize. Any REP holder may migrate their REP to a child Universe. All users that migrate during the fork period receive a 5% bonus to their migrated REP. After the Fork Period, all un-migrated REP are migrated to the child Universe with the highest REP supply (the "Winning Universe").

An attacker holding sufficient funds may post the dispute bond, cause a fork, and then migrate the remainder of their REP to the child Universe they expect to win. They will receive a 5% bonus in that Universe, but likely forfeit their dispute bond. The attacker may then repeat the dispute process in the new child Universe.

If the attacker's REP holdings are large, the 5% bonus offsets the loss of the dispute bond. With enough starting REP, the bonus will be larger than the bond. If the bonus is at least the size of the dispute bond, then the attacker may repeat the dispute process indefinitely. This will cause permanent long delays in Market settlement.

Remediation: Resolving this issue requires careful consideration and modeling of incentives. It may be impossible to prevent this attack entirely. It may be mitigated by adjusting the amount of REP that must be bonded to make a fork.

Status: Unresolved. Research is ongoing.

Verification: Not done, yet.

Issue B: Settlement may error if perceived REP price falls

Reported: 24-11-17

Synopsis: Settlement calculation depends on external REP price information from an oracle. If the oracle reports a low REP price markets and reporting windows may enter a bad state and become unable to distribute ETH during settlement.

Impact: In the total-failure case markets may still be created and traded, but Complete Sets may not be settled, and markets may not finalize. ETH held by markets or reporting windows is inaccessible without developer-mode intervention. In the more benign case, ETH due to market participants may be incorrectly distributed to Reporters.

Preconditions: The owner of the oracle must report a low price of REP denominated in ETH before the first settlement event of any given Reporting Window.

Feasibility: Trivial if holding the contract owner keys. Otherwise: very expensive to deliberately engineer. May happen accidentally in certain market conditions.

Technical Details: *reporting/RepPriceOracle.sol* is responsible for maintaining an accurate record of the current REP price measured in `attoETH`. The price is set by the contract owner via the setRepPriceInAttoEth method. The oracle exposes the `getRepPriceInAttoEth` method to the public.

In *reporting/Universe.sol* the `getRepMarketCapInAttoeth` method calls `getRepPriceInAttoEth`, and returns the current REP market cap based on the REP token contract's reported supply. This is use by the `getReportingFeeDivisor` method. If the Oracle is manipulated, it may be used to pass bad values to this method. `getReportingFeeDivisor` is called the first time ETH settlement would occur in each reporting window. It compares the current REP market cap to a target market cap, determined by the amount of ETH currently invested in markets, to create the reporting fee divisor, which is then cached.

`calculateReportingFee` is called in the `sellCompleteSets` method in *trading/CompleteSets.sol*, as well as the `calculateReportingFee` method in *trading/ClaimTradingProceeds.sol*. These methods calculate ETH payouts for distribution to market participants.

The reporting fee is calculated in these methods by dividing the total potential payout by the reporting fee divisor. If `getReportingFeeDivisor` returns 0, then this will result in a divide-by-zero error. An actor with control of the Oracle may produce this reliably by causing the Oracle to report a REP price of 0.

The amount of ETH to be distributed to the participants is calculated by subtracting the reporting fee and the creator fee from the potential payout. If `getReportingFeeDivisor` returns 1, and the creator fee is non-zero, execution will error in the `SafeMath` library call when bounds are checked while calculating the participant share. If `getReportingFeeDivisor` returns 1 and the creator fee is zero, then execution will succeed, but reporters will receive all proceeds.

Remediation: An additional constant, `MINIMUM_REPORTING_FEE_DIVISOR`, should be added to *reporting/Reporting.sol*, as well an additional getter for that constant. This value should be checked during fee divisor calculation in `getReportingFeeDivisor`. Any value of 3 or greater is reasonable.

The behavior of `sellCompleteSets` method in *trading/CompleteSets.sol* is slightly different from the behavior of its counterpart in *trading/ClaimTradingProceeds.sol*. E.g. `sellCompleteSets` does not check that the payout is non-zero before attempting to transfer value. Consider updating to match.

Status: Under consideration by dev team.

Verification: Not done, yet.

Issue C: Whitelist exploitation and Dev Mode reactivation

Reported: 24-11-17

Synopsis: Malicious whitelist entries are easy to insert, hard to check for, and give broad control of deployed contracts. The whitelist is currently completely difficult to observe. As such, the whitelist is a high-value avenue of attack.

Impact: Whitelisted addresses are allowed to call many methods not intended to be externally accessible. A whitelisted attacker may disrupt or break the Market system, update contracts in place, or steal funds. Developer mode access gives sweeping control of all contracts. This includes the ability to update unilaterally sweep funds from all deployed contracts.

Preconditions: Requires compromising smart-contract code or compiler before deployment, the deployment process, or possession of the deployment key after the deployment process.

Feasibility: Dependent on Augur deployment process security. Any attacker that can compromise the deployment process may be able to insert a malicious whitelist entry and/or steal deployment keys. In this way, deployment is similar to a standard cryptographic ceremony. Malicious developers are particularly well-positioned to carry out this attack.

Technical Details: *Controller.sol* manages the whitelist and developer mode. Many methods throughout the project use the `onlyWhitielistedCallers` modifier to restrict access to whitelist members. The whitelist is intended to contain the deployment key (the owner of all contracts) as well as a definitive list of deployed Augur contracts.

Whitelisted callers may update the central Augur contract registry, which allows theft of funds in addition to more subtle modifications of Augur functionality. They can also add other addresses to the whitelist. Access is controlled by the `addToWhitelist` and `removeFromWhitelist` methods, which may be called by any Whitelisted callers. As such, a malicious whitelist member may add any number of other malicious entries.

The deployment key is whitelisted on contract instantiation. The deploy scripts then use the deployment key to whitelist most other Augur contracts. During this process, contracts are whitelisted if they are not Controller, Delegator, part of the legacy reputation system, or external libraries. There is no explicit list of contracts that should be whitelisted.

The `devModeOwnerOnly` modifier checks that the caller is the owner, and that the caller is whitelisted. To deactivate dev mode, the owner calls the `switchModeSoOnlyEmergencyStopsAndEscapeHatchesCanBeUsed` method. This removes the owner from the whitelist. If the owner is not whitelisted, all

`devModeOwnerOnly` checks will fail. As such, dev mode activations status is implicit, rather than explicit. This makes it difficult to observe.

Dev mode is intended to be permanently deactivated at some point in the future. A malicious actor who has compromised the whitelist can reactivate dev mode by adding the owner to the whitelist. This breaks system expectations, and makes theft of funds easier if the attacker has also compromised the deployment key.

Mitigation: Mitigation of this attack revolves around making whitelist and dev mode status and actions visible to users.

Before deployment, the team should take the following steps:

1. Add Events to all whitelist and DevMode functions.
2. Link the events to prominent notifications in the UI so that users are directly informed of all actions.
3. Modify the UI to make it clear when the system is running in DevMode.
4. Add a devMode flag to *Controller.sol*. This flag should be checked before taking any action that requires dev mode privileges. This flag should be set to false when devMode is deactivated, to prevent reactivation.
5. Create and publish an explicit list of contracts and addresses to be whitielisted.
6. Commit to this list in a public non-repudiable manner, e.g. by publishing a signed hash to a blockchain.
7. Prepare and publish a deployment process spec. Treat this as a standard cryptographic ceremony.
8. Prepare a secure management plan for the deployment key. Consider transferring ownership to a multi-sig contract.
9. Prepare an action plan for a compromised whitelist.

After deployment the team should take the following actions.

1. Publicly verify that the deployed whitelist matches the expected whitelist.
2. Publish instructions for independent verification.
3. Repeat this verification process regularly.

The team should consider using the same verification process for the smart contract registry.

Remediation: It is not possible to fully fix this vulnerability without extensive architectural changes. The whitelist and registry are critical to the interaction of many Augur contracts.

Status: Under consideration by dev team.

Verification: Not done, yet.

Issue D: Reporter bond adjustment can get stuck at low values

Reported: 24-11-2017

Synopsis: The reporter bond is automatically adjusted to manage the proportion of disputes, but this interacts with the economics of disputes, if it is adjusted too low, it could disincentivize making disputes at all.

Impact: This could undermine the economics of the Augur system, meaning untrue reports are not disputed, so augur would not be an effective prediction market.

Preconditions: This bug could occur “naturally” on it’s own without a malicious actor, if the rate of disputes happens to go low, and then the incentives have a runaway effect. It this tendency could also be encouraged by a sybil attack. An attacker would create many markets that didn’t do anything and maybe play both sides of them (to make them appear real and to guarantee they break even) This would then artificially increase the rate of non-disputed markets which would cause the rate to adjust downwards.

Feasibility: Assuming that human behaviour (in choosing to make disputes) is affected by the incentive model, then it certainly seems feasible. It’s possible that the expected rate of disputes is an over estimation, and that fixed numbers such as a 10% return on disputes is sufficiently attractive is incorrect.

Technical Details: If the designated reporter chooses a false report, any REP holder may file a dispute. The first dispute has a fixed cost of 1,100 REP. When a REP holder makes a dispute, their motivation is to gain a return, which assuming that the following reporting round agrees with the disputer, the return will be the designated report bond. The designated report bond is adjusted up and down according to the number of disputes in the previous reporting window. If the number of disputes is lower than 1% the REP bond is adjusted down, and if it is higher that 1% it is adjusted up. The reasoning here is that if barriers to creating markets (designated reporter bond) are lowered, it will mean more markets will be created, and that these cheap markets are also more likely to dispute. However, changing the rate also affects the expected rate of return from a dispute. In subsequent rounds of disputes, each round is 10 times bigger than the previous, meaning a 10% return (glossing over details). A 10% return in one month (the reporting window) is considered a sufficiently attractive return to motivate disputers. If the designated report bond is adjusted all the way down to the minimum, 2 REP, with the fixed dispute bond of 1,100 REP, the expected return is only 0.18%! It seems likely that rates somewhat higher than the minimum (say, around 1%) will not be considered worthwhile.

This bug could occur by itself, if the rate adjusts low enough, but this tendency could be encouraged by a malicious actor via a sybil attack, by creating many markets (the attacker would lose nothing except GAS and gain fees if people actually played those markets, they may play the market themselves, to make their sybils appeal real). If this increased the dispute rate downwards sufficiently, it could trigger runaway effects.

Mitigation: Users should monitor the rate, and possibly stop trading if the rate gets too low. Augur developers could then update the contracts.

Remediation: There are many ways this problem could be avoided:

- Have a fixed rate, instead of a floating rate, (but this would make the bond be ~$3000 USD at current prices, which ruins Augur's promise of a cheap prediction market).
- Have the dispute bonds to be 10 times the previous dispute, guaranteeing a 10% expected return on disputes, (but this could interfere with economics of forking).
- Let the dispute rates be set by the market creator, as are the various fees, market creators would theoretically need to set reasonable dispute rates to attract confidant traders. This would allow "the market to decide" what rates are best. Maybe this is the best setting to start with because it would provide the most to learn from.

Status: Active research.

Verification: Not done, yet.

Issue E: Orderbook vulnerable to Miner frontrunning

Reported: 26-11-17

Synopsis: Miners may reorder transactions while creating candidate blocks. This gives miners a privileged position in the on-chain exchange compared to other users.

Impact: Miners have much better expected value from trading Shares than other users. Miners purchases receive the best buy and sell prices in each trading batch. Miners decide which users get the worst buy and sell prices in each trading batch.

Preconditions: Only Miners and pool operators may perform this attack.

Feasibility: Moderate implementation costs. Once implemented, near-0 cost of execution.

Technical Details: Augur maintains on-chain orderbooks for trading shares in each Market. These are accessed via the public methods found in *trading/Trade.sol*, or via the public method `trading/FillOrder.sol:publicFillOrder`. Most users will access the orderbook via *trading/Trade.sol*, which gives the user the best price on the book when it is called.

A user calls these by creating and submitting an Ethereum transaction. A miner then orders these transactions within a candidate block, and mines on the block template until a valid block is found. The miner chooses the order these transactions are executed in.

The miner, when ordering transactions, may examine them and the orderbook. After doing so, they may choose to order them in any way they please. They may also create new transactions (for instance, a buy or sell), and insert them at any point in the order. Because trades are transactions, reordering transactions is equivalent to reordering trades.

In the naive version of the attack a miner can guarantee that their buys execute before all other buys and that their sells execute before all other sells. In advance of each user's buy order the Miner may take the best standing sell, and place a new one at a higher price. In

advance of each user's market sell order, the miner may sell to the best standing buy order, and place a new buy at a lower price. As such, miners may take a de-facto transaction fee from each user.

Mitigation: Users concerned with frontrunning may opt to use `trading/FillOrder.sol:publicFillOrder` to specify the standing order they would like to fill. This prevents miners from extracting a fee by manipulating transaction ordering, but means that the user's market order is not guaranteed to fill completely. It also makes it unlikely that the user will receive the best price.

Remediation: This problem may be impossible to fully remediate. No known model for an on-chain exchange prevents front-running by miners.

Status: Active research.

Verification: Not done, yet.