AugurLite Audit Report

May 14, 2019

Introduction

This document contains the results of the audit performed on the AugurLite project, developed by the Veil team. The goal of this audit was to verify that functionality works as intended, and that no critical issues exist.

Overview

Development Team	
Lead Developer(s)	Feridun Mert Çelebi
Point of Contact	Feridun Mert Çelebi
Audit Team	
Reviewers	Daniel Que, Nemil Dalal
Application	
Name	AugurLite
Audit Duration	May 10 to May 12, 2019
Commit Reviewed	12ef05931438086c075b3d9f5b4213ec10d072b8
Code Overview	Solidity: 2,180 lines, 61 files Typescript: 3,683 lines, 38 files Python: 2,017 lines, 25 files
Solidity Compiler Version	0.4.26

Disclaimer

This document reflects the understanding of security flaws and vulnerabilities as they are known to the audit team, and as they relate to the reviewed project. This document makes no statements on the viability of the project or the safety of its contracts. It's critical to recognize that there are inherent limitations in the smart contract auditing process.

Table of Contents

Summary

AugurLite is a slimmed-down fork of the Augur V1 smart contracts. AugurLite's goal is to provide users with stablecoin-denominated markets, which is important for supporting markets with a distant resolution time.

AugurLite is simplified by removing functionality that can be leveraged from existing projects. AugurLite has no reporting or dispute resolution functionality, no REP token, and no on-chain exchange. Market resolution is replaced by an indisputable oracle, REP token is no longer necessary (since its main purpose is for disputes), and exchanging tokens can be done with protocols such as 0x. The rest of the modifications to the original Augur contracts are thoroughly detailed in the CHANGELOG.

The original Augur contracts were audited by reputable firms prior to deployment in March 2018, and no critical issues have surfaced since then. AugurLite is a subset of Augur, and this audit was requested to verify no issues were introduced by the changes.

Our team assessed Veil's AugurLite Solidity smart contract code between May 10 and May 12, 2019. Our assessment consisted of the following:

- Separate manual audits of the codebase
- Inspection of the full diff between AugurLite and Augur Core
- Comparison of notes and pair auditing code with concerns raised
- Careful audits of a list of *flows* (important end-to-end contract interactions)
- Careful audits of a list of *checks* (intended invariants)
- Special focus on files flagged by Veil: `Universe.sol`, `Market.sol`, `ClaimTradingProceeds.sol`, `CompleteSets.sol`, and `Mailbox.sol`.

An assessment of the contracts with automated tooling and custom tests was not performed due to the timeline of the audit.

Upon conclusion of the audit, no critical vulnerabilities were found.

Flows

Flows are end-to-end contract interactions that are considered important for typical usage. We verified to the best of our ability that the following flows work as intended:

- Creating a market
- Buying a complete set
- Selling a complete set
- Resolving a market and claiming proceeds

Checks

Checks are statements about the contracts that are identified as important. We verified to the best of our ability that the following checks hold:

- Not possible to generate more ShareTokens than intended
- Not possible to steal another user's ShareTokens through AugurLite
- Not possible to steal another user's `denominationTokens` through AugurLite
- Not possible to lock a material amount of ShareTokens under normal circumstances (e.g. `denominationToken` is not frozen)
- Not possible to lock a material amount of `denominationTokens` under normal circumstances
- Contracts can't be initialized more than once
- Event logs can't be emitted with fake data
- No exploitable reentrancy vulnerabilities
- Callbacks (e.g. `onMint`, `onBurn`, `onTransfer`) can't be exploited for a denial of service (locked funds)
- No overflow vulnerabilities
- Not possible to create a malicious market that is accepted by the system
- Not possible to create a malicious ShareToken that is accepted by the system
- Only the `controller` can call privileged methods

Findings Summary

This section provides details about the issues found.

Severity Level Reference

Type	Description
CRITICAL	Exploitable issue leading to stolen or locked funds.
HIGH	Exploitable issue leading to unintended results, impacting funds.
MEDIUM	Exploitable issue leading to unintended results, not impacting funds.
LOW	Finding with low risk, or the development team has indicated it's not a concern.
INFORMATIONAL	Finding which doesn't pose a risk but may be relevant in the future.

Findings List

Finding	Description
Shareholder can avoid market creator fee	HIGH
Assumptions about `denominationToken` decimals causes unwanted misconfigurations	LOW
Dust amounts of tokens can be locked	LOW
`denominationToken` dependency can cause funds to get locked	INFORMATIONAL
Attacker can force another user to claim proceeds	INFORMATIONAL
Payout distribution not enforced for yes/no and categorical markets	INFORMATIONAL
Parasitic markets	INFORMATIONAL
ShareToken doesn't emit Transfer event on mint and burn	INFORMATIONAL

Findings

Shareholder can avoid market creator fee

Severity: HIGH

Summary
A shareholder can avoid creator fees by claiming shares in small batches. Note that this finding applies to the original Augur contracts as well.

Location
`source/contracts/trading/ClaimTradingProceeds.sol:18`

Technical Description
Consider the following:

- A market with `feeDivisor=100`
  - This represents a fee of 1%, achieved by initializing the market with `_feePerEthInAttoeth=10**16`.
- A shareholder, who distributes their arbitrarily large shareToken balances into accounts with 49 SHARES each.
- The market is resolved with `payoutNumerator=[0, 2]`, representing short and long payouts respectively.
- The shareholder holds long shares.
  - Note that 2 is the smallest possible value, given `_numTicks` must be a multiple of 2.

The rest follows by walking through an abridged and annotated version of the code:

`source/contracts/trading/ClaimTradingProceeds.sol:18:`

```
claimTradingProceeds:
  uint256 _numberOfShares = _shareToken.balanceOf(_shareHolder); // 49
  (_proceeds, _shareHolderShare, _creatorShare) = divideUpWinnings(_market,
_outcome, _numberOfShares); // see divideUpWinnings below (98, 0, 98)

divideUpWinningsdivideUpWinnings:
  _proceeds = calculateProceeds(_market, _outcome, _numberOfShares); // see
calculateProceeds below (98)
  _creatorShare = calculateCreatorFee(_market, _proceeds); // see
```

```
calculateCreatorFee below (0)
  _shareHolderShare = _proceeds.sub(_creatorShare); // 98 - 0
  return (_proceeds, _shareHolderShare, _creatorShare); // (98, 0, 98)

calculateProceeds:
  uint256 _payoutNumerator = _market.getPayoutNumerator(_outcome); // 2
  return _numberOfShares.mul(_payoutNumerator); // 49 * 2 = 98

calculateCreatorFee
  return _market.deriveMarketCreatorFeeAmount(_amount); // see
deriveMarketCreatorFeeAmount below (0)

deriveMarketCreatorFeeAmount:
  ...
  return _amount / feeDivisor; // 98 / 100 = 0
```

In the end, the shareholder has redeemed 49 SHARES with zero fees. They can rinse and repeat to redeem all shares.

Some notes:

- This works with any market type, but it's easier to illustrate with a scalar market since it doesn't impose high `numTicks` (10,000).
- A “winning long” outcome is not necessary. It also works with a “losing short” outcome (e.g. [2, 9998])
- Whether this can be performed with a real example depends on the parameters
- Performing this attack requires many transactions, and the gas costs probably make it impractical (simulations were not performed)

Recommendation
It would be possible to mitigate this issue by limiting the minimum claim size. However this approach would make a compromise in usability.

Update
The Veil team has decided to not make modifications since it has been deemed impractical to exploit with the expected configuration. We recommend revisiting this issue if those expectations change (e.g. using USDC, which has `decimals=6`, as the `denominationToken`).

Assumptions about `denominationToken` decimals causes unwanted misconfigurations

Severity: LOW

Summary
The original Augur contracts assume that `denominationToken` is wrapped ETH. This assumptions wasn't removed, and the code continues to use `1 ether` (Solidity keyword for 10^{18}) to compute fee-related values. This can lead to an incorrect configuration if using an ERC20 token with `decimals` other than 18 (e.g. USDC, which has `decimals=6`).

Locations
`source/contracts/reporting/Market.sol:23 (and other lines)`
`source/contracts/factories/MarketFactory.sol:10 (and other lines)`

Technical Description
When a market is initialized, `feeDivisor` is computed from `_feePerEthInAttoeth`. (Note that the prefix *atto* means 10^{-18}.) If deployed with a token with `decimals` other than 18, the resulting `feeDivisor` could be drastically different from intended.

`source/contracts/reporting/Market.sol:`

```
23: uint256 private constant MAX_FEE_PER_ETH_IN_ATTOETH = 1 ether / 2;
...
51: require(_feePerEthInAttoeth <= MAX_FEE_PER_ETH_IN_ATTOETH);
...
61: feeDivisor = _feePerEthInAttoeth == 0 ? 0 : 1 ether /
_feePerEthInAttoeth;
```

Also note that the `createYesNoMarket` and `createCategoricalMarket` methods use `1 ether` as the hardcoded value for `_maxPrice` in the `MarketCreated` event log, which may also be an assumption. However, max price isn't relevant for these markets, and it's unclear why these are hardcoded to `1 ether`.

Recommendation
Allow `denominationToken` decimals to be configurable in Universe.sol and Market.sol. Rename variables to avoid referencing ETH, atto, and wei.

Update
The Veil team opted to make `feeDivisor` to be directly configurable. This avoids the need to add a new parameter.

Dust amounts of tokens can be locked

Severity: LOW

Summary
Dust amounts of tokens can be locked in the contract due to truncating division.

Locations
`source/contracts/reporting/Market.sol:105`
`source/contracts/reporting/Market.sol:189`

Technical Description
This is best illustrated with an example. Calling `CompleteSets.sellCompleteSets` eventually executes the following:

`source/contracts/reporting/Market.sol:105`

```
return _amount / feeDivisor;
```

Since the quotient is rounded down, those dust amount are never transferred out of the contracts.

Another example is resolving a categorial market with 3 outcomes as invalid (this will evenly distribute the proceeds):

`source/contracts/reporting/Market.sol:189`

```
require(_previousValue == numTicks / numOutcomes);
```

Since `numTicks` is hardcoded as 10,000 for categorical markets, each option would get paid out 3,333 ticks.

Recommendation
None. These dust amounts are negligible unless the token has a small `decimals` value. Locked dust amounts are generally accepted by the community.

`denominationToken` dependency can cause funds to get locked

Severity: INFORMATIONAL

Summary
`denominationToken` is an external contract that AugurLite depends on. As with all external contract calls, it's important to understand the risks.

A malicious token could cause a plethora of problems, but that won't be explored since the Veil team has communicated that the candidates for `denominationToken` are USDC and DAI. While USDC and DAI are outside the scope of this audit, here are some high level details:

- USDC transfers can be frozen via an emergency pause function
- USDC can blacklist addresses, which prevents transfers in and out of them
- DAI transfers can potentially be frozen
    - Verifying this would require a deep dive. The functionality exists in DAI's token contract, but whether it can actually be invoked depends on the complex live deployment.

These pause and blacklist operations are *extremely rare* (none have ever happened with these tokens), but they're possible.

A failing call to `denominationToken`'s `transfer` or `transferFrom` can cause funds to be locked. For example, if a market's creator mailbox address was blacklisted, any attempt to claim proceeds with a non-zero creator share would fail.

Another nuance to make note of is DAI's global settlement. During global settlement, DAI's peg breaks and effectively takes on the price risk of ETH. Since the goal of AugurLite is to provide long-lived markets with a stable denomination, this should be considered.

Attacker can force another user to claim proceeds

Severity: INFORMATIONAL

Summary
If a user holds their ShareTokens in a wallet incompatible with `denominationToken`, an attacker can force the user to claim proceeds into that wallet, which would result in the payout being locked.

Location
`source/contracts/trading/ClaimTradingProceeds.sol:18`

Technical Description
`ClaimTradingProceeds.claimTradingProceeds` is an unrestricted function that can be called by any user. While it's probably intended as a feature to enable claiming proceeds to a cold storage address or to automate claiming, it can be exploited in a very specific scenario.

Consider a situation in which a shareholder has deposited their ShareTokens into a wallet contract that is incompatible with `denominationToken`. An attacker can call `claimTradingProceeds`, setting the victim's wallet as `_shareHolder`. This would redeem their shares for `denominationTokens`, sending them to the incompatible wallet. The tokens would thus be locked.

Recommendation
None. This is a compromise made for another feature.

Payout distribution not enforced for yes/no and categorical markets

Severity: INFORMATIONAL

Summary
Yes/no markets doesn't enforce outcomes of [10000, 0] or [0, 10000]. The same applies to categorical markets but with more outcomes (e.g. [10000, 0, 0, ...]).

For example, an oracle could report a yes/no market with a 60/40 split ([6000, 4000]).

This doesn’t need to be enforced.

Parasitic markets

Severity: INFORMATIONAL

Summary

The Augur whitepaper defines a *parasitic market* (aka. derivative markets) as “any market that does not pay reporting fees to Augur, but does resolve in accordance with the resolution of a native Augur market.”

Since Augur is unaware of the total value at stake, it can’t properly economically secure the network. With parasitic markets, it’s possible that the cost of launching an attack against REP is less than the profit to be made from such an attack.

It’s possible for AugurLite to be used as a parasitic market if it’s configured to use Augur’s resolutions as its own.

ShareToken doesn’t emit Transfer event on mint and burn

Severity: INFORMATIONAL

Summary
The ERC20 spec suggests that mint and burn should emit a Transfer event:

> “A token contract which creates new tokens SHOULD trigger a Transfer event with the `_from` address set to 0x0 when tokens are created.”

Location
source/contracts/libraries/token/VariableSupplyToken.sol:17

Recommendation
Emit Transfer event.

Update
The Veil team added Transfer events for mint and burn.

Nits and Notes

This section contains miscellaneous nits.

Unnecessary import

Description
Import of Initializable.sol in ITime.sol not needed.

Location
`source/contracts/ITime.sol:4`

Minor gas saving in SafeMath

Description
The `SafeMath` `div` function deviates slightly from OpenZeppelin's implementation, which uses `require` rather than an implicit assert to save gas in case of a failure. The same applies to `SafeMathInt256`.

```
require(b > 0);
```

Location
`source/contracts/libraries/math/SafeMathUint256.sol:16`

Update
The Veil team updated the `SafeMath` libraries to use `require`.

Unused functions

Description
Unused functions: fxpMul, fxpDiv. This also applies to SafeMathInt256

Location
`source/contracts/libraries/math/SafeMathUint256.sol:62`

Update
The Veil team updated the `SafeMath` libraries to remove these functions.