

SECURITY AUDIT REPORT

of Smart Contracts

December 20, 2017









Produced by



for



Table of Contents

Foreword.....	1
Introduction.....	2
About Augur	2
About the audit.....	2
Depth	2
Terminology and labels.....	3
System Risk Classification	4
Limitations	4
Findings.....	7
Issues.....	7
Issue 1: Reentrancy Analysis 	7
Issue 2: Potentially Dangerous Reentrancy in Market 	7
Issue 3: No Callstack Bugs 	8
Issue 4: Ether Transfers 	8
Issue 5: Any Whitelisted User Can Empty the Whitelist 	8
Issue 6: Compiler-dependent Computations 	8
Issue 7: Set Data Type 	9
Issue 8: Floating Value Formula 	9
Summary of system risk.....	10
Prior to audit	10

After audit 10

Recommendations..... 11

Conclusion..... 13

Foreword

We thank Augur for giving us the opportunity to audit their smart contract. This report outlines our methodology, limitations, and findings discovered during the security audit.

- ChainSecurity

Introduction

About Augur

Augur is a decentralized prediction market that runs on top of the Ethereum blockchain. The Augur platform estimates the probability of future events based on votes casted by users, thereby leveraging the wisdom of the crowd principle. Users are rewarded whenever they make correct predictions.

About the audit

Depth

The scope of the security audit conducted by ChainSecurity Ltd. was restricted to:

- Scanning the contracts listed above for generic security issues using automated systems and manually inspecting the results.
- 32-hours of manual audit of the contracts listed above for security issues.

Terminology and labels





For the purpose of this audit, ChainSecurity adopts the following terminology. To summarise security vulnerabilities, the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology¹) are specified.

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical- and business-related consequences of an exploit.

Severity is derived based on the likelihood and the impact determined previously, and is summarised in Table 1.

The severity of each finding is categorised into four ratings, depending on their criticality:

-  signifies a low-risk issue that can be considered as less important.
-  signifies a medium-risk issue that should be fixed.
-  signifies a high-risk issue that should be fixed very soon.
-  signifies a critical issue that needs to be fixed immediately.














LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

Table 1. Severity of an issue based on its impact and likelihood.

If no security impact is found for an issue, we label it as  (“no issue”).

If a security impact has been found for an issue, and it has been addressed technically during the auditing process, we also label it as  (“fixed”).

If the security impact for an issue has been addressed in another manner, we label it as  (“addressed”).

Findings that are labelled as either  fixed or  addressed are resolved and therefore pose no security threat. Their severity is still listed simply to give the reader a quick overview what kind of issues were found during the audit.

System Risk Classification

Points are allocated to the severity category (Table 2) of each issue found in the audit in order to determine the overall risk level (Table 3) of the smart contract system.





Severity Category of Finding	Points
	30 per finding
	10 per finding
	4 per finding
	1 per finding

Table 2. Allocation of points for each severity category

¹ https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology





System Risk Classification		Cumulative Points
	Critical risk	30 points or more
	High risk	17 – 29 points
	Medium risk	7 – 16 points
	Low risk	6 points or less

Table 3. Risk classification for systems based on the sum of all points

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing enables the discovery of vulnerabilities that were overlooked during development and areas where additional measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. ChainSecurity therefore carries out a source code review to determine all the locations that need to be fixed. ChainSecurity performs extensive auditing in order to discover as many vulnerabilities as possible.

Findings

Issues

The following are issues were investigated during the audit of the Augur source code.

Issue 1: Reentrancy Analysis

The following contracts make potentially unsafe calls using the `call.value()` function, which transfers all gas to the caller and may allow reentrant calls: `Controlled.sol`, `Extractable.sol`, `Mailbox.sol`, `Market.sol`, and `Cash.sol`.

The calls in `Controlled.sol` and `Extractable.sol` can only be made by the controller address. However, the target of these calls can be untrusted. They are considered safe since there are no critical state changes performed after the call.

The target of the call in `Mailbox.sol` is trusted (owner) and therefore we consider it as benign.

Two of the calls in `Market.sol` are considered benign:

1. The target of the call in function `withdrawInEmergency` is a potentially untrusted contract (`msg.sender`). However, there are no critical state changes that appear after the call. Furthermore, the function `withdrawInEmergency` is not called from another function.
2. The target of the call in function `initialize` is owner, which is trusted.

One of the calls in `Market.sol` is potentially dangerous (see Issue 2).

The calls in the functions `withdrawEther` and `withdrawEtherTo` of `Cash` are benign, since the amount is burned before the call.

Issue 2: Potentially Dangerous Reentrancy in Market

Likelihood: Low

Impact: Medium

One of the calls in `Market.sol` is public (any user can trigger it) and the target is untrusted (an arbitrary address provided by the untrusted transaction sender):

```
function firstReporterCompensationCheck (address _reporter) public onlyInGoodTimes
    returns (uint256) {
        ...
        require(_reporter.call .value(reporterGasCostsFeeAttoeth()));
        ...
    }
```


While there are no critical state changes that immediately follow the call in the function `firstReporterCompensationCheck`, there are relevant function calls in the function `StakeToken.buy()` (which calls `firstReporterCompensationCheck`). This means that an attacker may attempt to do nontrivial state changes before the execution returns to `StakeToken.buy()`. To avoid unforeseen security issues due to such state changes, we recommend using a safer call function that would transfer a small amount of gas that would not allow complex reentrant calls.

Issue 3: No Callstack Bugs

The Augur contracts do not contain any vulnerabilities that would allow attacks based on a callstack overflow. This is because all calls that do not explicitly propagate exceptions (such as `call.value()` and `send()`) are properly handled. In particular, all uses of `call.value()` are invoked within the `require()` statement.

Issue 4: Ether Transfers

We did not uncover any issues related to unexpected ether transfers.

Issue 5: Any Whitelisted User Can Empty the Whitelist

Likelihood: Low

Impact: Low

The `onlyWhitelistedCallers` modifier defined in `Controller` restricts access to users that are in the whitelist. Any whitelisted user can empty the whitelist using the function `removeFromWhitelist`:

```
function removeFromWhitelist(address _target) public onlyWhitelistedCallers
    returns (bool) {
        whitelist[_target] = false ;
        return true ;
    }
```

Note that the user can remove any other user from the whitelist, including the owner or his own address. This could potentially cause denial-of-service since all actions restricted to whitelisted users would be blocked and no user can be whitelisted since the `addToWhitelist` function uses the `onlyWhitelistedCallers` modifier.

Issue 6: Compiler-dependent Computations

Likelihood: Low

Impact: Low

The division and multiplication operators do not necessarily commute (due to rounding after division) and have equal precedence. Solidity does not specify an execution order of operators with equal precedence, which means that chained division and multiplication that are not explicitly ordered via brackets may depend on the order imposed by the compiler. The following

computation, defined in the `getOrCacheReportingFeeDivisor()` of the Universe contract depends on the compiler:

```
_currentFeeDivisor = _previousFeeDivisor * _repMarketCapInAttoeth /  
_targetRepMarketCapInAttoeth ;
```

Issue 7: Set Data Type



Likelihood: Low

Impact: Medium

Augur uses the Set data type, defined in `Set.sol`, but seems to provide no dedicated test cases for this data type. Set allows the insertion of `bytes32` or `address` objects. Consequently, additionally to manual inspection, ChainSecurity Ltd. performed extensive fuzzing to identify potential issues inside this data type. Our procedurally generated tests contained hundreds of assertions and uncovered the following behaviour:

Padding Issues If `bytes32` is used as an input, shorter inputs are padded to 32 bytes. Therefore, two inputs which the user perceives as different are padded to the same result. In particular the empty string is “contained” in the set as soon as an 0-string of arbitrary length is inserted.

Mixed Data Types The set allows to mix different data types, namely `address` and `bytes32`. This might have unexpected effects. In particular, an `address` might be added and a differently looking `bytes32` would be matched as contained.

In Appendix A, we provide a solidity test case which highlights the discussed issues along with the helper contract (`SetInstance`) used for testing. On the current implementation all four test cases fail, which might be unexpected for developers.

For the current use cases of the Set data type, the given implementation appears to be fine, as only addresses are inserted. However, this data type shouldn’t be seen as a traditional set and reuse for future use cases should only happen after careful consideration.

Issue 8: Floating Value Formula

The floating value formula in `Universe.sol` is used to compute the floating value for different bonds depending on the ratio of bad markets within the total number of markets. The resulting factor by which the value changes can be seen in Figure 1. This may be intended behaviour (requiring confirmation by Augur).

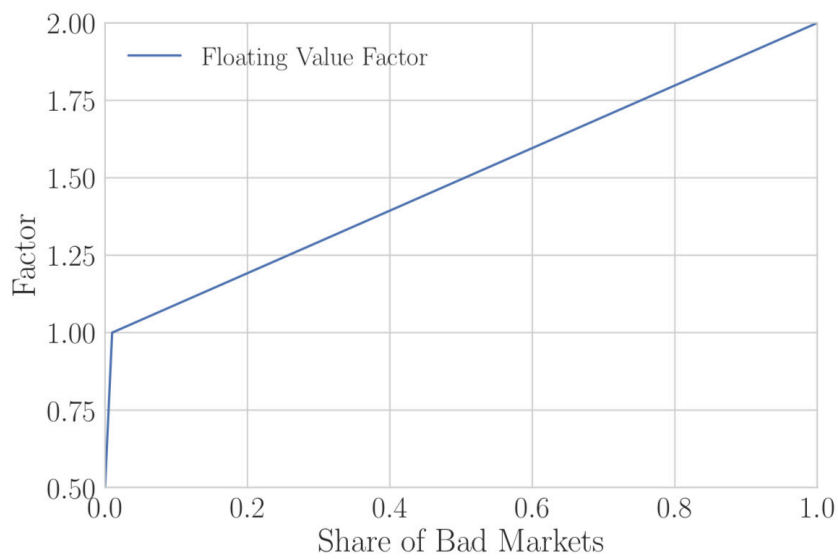






Figure1. Floating value computation

Summary of system risk

Prior to audit

Severity Category of Finding	Number of issues found	Subtotal
 C	0	0
 H	0	0
 M	0	0
 L	4	4

Total points: 4

System risk classification:



Our audit has revealed that the Augur smart contract system was at a **low** risk level prior to our audit and our subsequent amendments.

After audit

Augur's smart contract system continues to be at a **low** risk level.

Recommendations

The following amendments are recommended for the Augur smart contract system.

- The lookup function in Controller should validate that there is a contract entry for the provided _key.
- The registerContract function in Controller should validate that the name of the contract that is being registered (returned via _address.getTypeName) indeed matches the _key provided as argument.
- The address of the new controller should be validated in the following function (defined in Controlled.sol)

```
function setController(IController _controller) public onlyControllerCaller
    returns (bool) {
        controller = _controller ;
        return true ;
    }
```

- The name of the function switchModeSoOnlyEmergencyStopsAndEscapeHatchesCanBeUsed, defined in Controller.sol, is confusing. The function unregisterContract remains accessible to the owner even after the owner is removed from the whitelist.
- Multiple uses of call.value() for in the following functions:
Controlled.suicideFunds(),
Extractable.extractEther(),
Mailbox.withdrawEther(),
Market.initialize(),
Market.withdrawInEmergency(),
Market.firstReporterCompensationCheck(),
Cash.withdrawEther(), and
Cash.withdrawEtherTo().
Consider using the safer function that only sends a small constant amount gas to the caller.
- Unnecessary division in Universe.sol:

```
_newValue = _targetDivisor
    .mul(_previousValue
        .mul(_badMarkets)
        .div(_totalMarkets)
        .sub(_previousValue
            .div(_targetDivisor)))
    .div(_targetDivisor - 1) + _previousValue;
```

Regarding this formula:

$$\frac{\text{targetDivisor} \cdot \left(\frac{\text{previousValue} \cdot \text{badMarkets}}{\text{totalMarkets}} - \frac{\text{previousValue}}{\text{targetDivisor}} \right)}{(\text{targetDivisor} - 1)} + \text{previousValue} \quad (1)$$

$$\frac{\text{targetDivisor} \cdot \frac{\text{previousValue} \cdot \text{badMarkets}}{\text{totalMarkets}} - \text{previousValue}}{(\text{targetDivisor} - 1)} + \text{previousValue} \quad (2)$$

The second line requires a division less and therefore has a smaller numerical error. In the first line, due to the numerical error, the result might be $> 2 \cdot \text{previousValue}$.

- Typo inside the comment “infalitory bonuses” in `reporting / Universe.sol`.

Conclusion

The Augur smart contracts have been analyzed under different aspects, with different open-source tools as well as our fully fledged proprietary in-house tool. Overall, ChainSecurity found that Augur employs good coding practices and has clean, well-documented code.



CHAINSECURITY



www.chainsecurity.com



contact@chainsecurity.com



[@chain_security](https://twitter.com/chain_security)