STL

SOHO TOKEN

LABS

FOAM Protocol Public Report

PROJECT: foam.tcr/dapp/contracts

July 20th, 2018

Prepared For:

Kristoffer Josefsson | FOAM Protocol

Kristoffer@foam.space

Prepared By:

Jonathan Haas | SoHo Token Labs

Jonathan@sohotokenlabs.com

Elissa Shevinsky | SoHo Token Labs

Elissa@sohotokenlabs.com

Table of Contents

FOAM Protocol Audit

Executive Summary

Scope of Engagement

SoHo Token Labs, Inc. was engaged by the FOAM Project in mid July of 2018 in order to conduct a comprehensive security audit of the FOAM Protocol. SoHo Token Labs conducted this assessment over the course of 2 weeks.

The code reviewed represents a work in progress. While the official finalization of the FOAM protocol follows the public release of this report and official announcement by the FOAM Project team, the FOAM Protocol itself will be continually updated and modifications will be made after the completion of this report.

The FOAM protocol itself utilizes a number of third-party (non-proprietary) contracts. There are unique security considerations present (as detailed in our report) that relate to a codebase designed in this fashion. Where relevant, we recommend mitigation strategies that include updating to use the latest versions of these third party contracts.

The scope of SoHo Token Lab's engagement covered all relevant smart contracts contained within the dapp/contracts directory. Other elements within the foam.tcr repository, such as the FOAM web components, were not in scope for this audit. Certain interactions between the audited smart contracts and these external components are critical to support intended behavior, and data integrity as it relates to these external components is noted within our analysis.

SoHo Token Labs utilized static analysis, dynamic instrumentation and manual inspection in order to investigate potential security vulnerabilities. In the event that tooling determined areas of concern, manual inspection was performed to verify if the tooling flagged a potentially exploitable vulnerability or a false positive. SoHo Token Labs Inc. utilizes these toolsets at our discretion where deemed appropriate and cannot guarantee their accuracy.

Timeline

Audit Begun: July 10, 2018

Report Presentation: July 23, 2018

Verification of Remediation: August 7, 2018

Prepared for Public Release: August 7, 2018

Engagement Goals

The primary scope of the engagement was to determine answers to the following questions, alongside additional security verification:

- Do any aspects of the Foam protocol specification present inherent security concerns?
- Is it possible for an unauthorized thirty party to gain administrative access to the protocol?
- Is it possible to cause the contract to enter an unrecoverable state?
- Can the TCR infrastructure be manipulated to distort token balances?
- Are there any unique security concerns that the dapp poses upon the contract?

Protocol Specification

During the scope of the engagement, SoHo Token Labs assessed various facets of the specification against commit version a276aba. While the full specification is extensive and not included within this report for the sake of brevity, we include the description of the architecture breakdown below in order to establish important background: “The FOAM Token Curated Registry (TCR) is built with the Crypto-Spatial Coordinate (CSC) standard to form a registry that enables the blockchain to act as an index of spatial contracts and, by extension, allow spatial contracts to be queried and displayed on the Spatial Index Visualizer. CSCs and TCRs together with token economics make a powerful combination for a new form of mapping and maintaining what are known as Points of Interest (POI)”

Overall Assessment

SoHo Token Labs was engaged to evaluate and identify significant security concerns in the codebase of the FOAM Protocol. We have presented our findings in detail in this report, and made recommendations for best practices.

The FOAM protocol utilizes a system for token-weighted voting which enables a user to participate in multiple polls simultaneously with their tokens while preventing the double-voting of tokens within polls.

Where noted, the FOAM team has responded with their relevant mitigation strategy, or acceptance of risk.

Disclaimers

As of of the date of publication, the information provided in this report reflects the presently held, commercially reasonable understanding of SoHo Token Labs, Inc.'s knowledge of security patterns as they relate to the FOAM Protocol, with the understanding that distributed ledger technologies ("DLT") remain under frequent and continual development, and resultantly carry with them unknown technical risks and flaws. The scope of the review provided herein is limited solely to items denoted within "Scope of Engagement" and contained within "Directory Structure". The report does NOT cover, review, or opine upon security considerations unique to the Solidity compiler, tools used in the development of the protocol, or distributed ledger technologies themselves, or to any other matters not specifically covered in this report.
The contents of this report must NOT be construed as investment advice or advice of any other kind. This report does NOT have any bearing upon the potential economics of the FOAM protocol or any other relevant product, service or asset of FOAM or otherwise. This report is not and should not be relied upon by FOAM or any reader of this report as any form of financial, tax, legal, regulatory, or other advice.

To the full extent permissible by applicable law, SoHo Token Labs, Inc. disclaims all warranties, express or implied. The information in this report is provided "as is" without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. SoHo Token Labs, Inc. makes no warranties, representations, or guarantees about the FOAM Protocol. Use of this report and/or any of the information provided herein is at the users sole risk, and SoHo Token Labs, Inc. hereby disclaims, and each user of this report hereby waives, releases, and holds SoHo Token Labs, Inc. harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

Timeliness of Content

All content within this report is presented only as of the date published or indicated, to the commercially reasonable knowledge of SoHo Token Labs, Inc. as of such date, and may be superseded by subsequent events or for other reasons. The content contained within this report is subject to change without notice. SoHo Token Labs, Inc. does not guarantee or warrant the accuracy or timeliness of any of the content contained within this report, whether accessed through digital means or otherwise.
SoHo Token Labs, Inc. is not responsible for setting individual browser cache settings nor can it ensure any parties beyond those individuals directly listed within this report are receiving the most recent content as reasonably understood by SoHo Token Labs, Inc. as of the date this report is provided to Foam.

General Recommendations

Recommendations based on general best practices for Solidity smart contract development

Dependency on third-party contracts

The foam.tcr repository has dependencies upon a number of non-proprietary contracts (each assessed below). While itself not a security vulnerability, dependency on third-party code does increase the general attack surface of the platform. This practice requires that the FOAM protocol team stay up to date on potential security vulnerabilities that these third-party contracts may possess, and certain components themselves may have additional dependencies that the FOAM protocol team should be aware of.

As put best by OWASP, determining if you are vulnerable requires searching these databases, as well as keeping abreast of project mailing lists and announcements for anything that might be a vulnerability. If one of your components does have a vulnerability, you should carefully evaluate whether you are actually vulnerable by checking to see if your code uses the part of the component with the vulnerability and whether the flaw could result in an impact you care about.

Resolution: The FOAM team acknowledges the unique risks associated with dependencies upon third-party code and noted potential areas of security concern presented by SoHo Token Labs at the time of audit. SoHo Token Labs recommends continued vigilance with regards to any third-party components and their accompanying dependencies. In instances that require updating contract logic, FOAM protocol should ensure that updated components do not alter the intended business logic of the application.

Lack of “update” functionality

While remaining a hotly contested issue in regards to “true” decentralization, SoHo Token Labs recommends update or freeze functionality be included within the contract repository, at least for a finite period of time. As distributed ledger technology evolves and the threat landscape remains still largely undefined, SoHo Token Labs suggests at *least* a three month period in which centralized pause or update functionality is allocated to the FOAM protocol team.

Resolution: SoHo Token Labs received the following response from FOAM, and feel the mitigations placed adequately cover existing concerns regarding upgrade functionality: The Parameterizer.sol now inherits from Ownable. Parameterizer has a reserved parameter called _newRegistry which can only be proposed by the owner, but must be voted on by the community.

The _newRegistry address field is checked against the EIP165 interface introspection, so that the corresponding address is to be guaranteed to have an canReceiveListing(...) and a receiveListing(...) function.

When this parameter has been voted for, the Registry is no longer allowing apply and challenge to be called. It instead makes it possible to call migrateListing on non-challenged listings which calls into the contract given by _newRegistry through its migrate function and then deletes the listing.

Note that all open challenges are still valid in the old registry. The old registry never gets deleted. All the tokens held can still be withdrawn. All listings in challenge mode still work in the corresponding PLCRVoting contract that remains untouched.

Testing of Upgrade

Note that only the interface of the upgraded registry is specified. There are multiple possible implementations of future registries and we only provide one, RegistryUpgradeTest which presents a MVP in that it demonstrates an upgraded registry. More importantly, the new tests test the behaviour of the old registry before, during and after migration.

Specific Recommendations

Recommendations specific to the FOAM Project and those who implement the protocol

Tests, Code Coverage, and Lack of a Truffle Project

A majority of testing within the project is performed through the usage of smart contracts (typically suffixed with "Test"). While this practice does produce valid test results, SoHo Token Labs suggests the usage of the Truffe Framework or similar unit testing suites. Due to the testing nature of these contracts (and their lack of usage beyond local testing), SoHo Token Labs recommends these contracts ***not*** be published on the blockchain.

While usage of the Truffle framework and associated tooling is encouraged due to ease of development and testing, lack of a Truffle project is ***not*** an inherent vulnerability.

It is of note that sol-cover, the most popular line coverage utility is in active development and their self-stated accuracy is unknown. For this reason (in addition to their lack of support for projects not using the Truffle framework), SoHo Token Labs has only chosen to include sol-cover results generated by the contract authors themselves

Resolution: SoHo Token Labs received the following response from FOAM, and feel the mitigations placed adequately cover existing concerns regarding upgrade functionality: We have integrated the skmgoldin/tcr test-suite wholesale in the CI pipeline to demonstrate that the specification of the TCR is fulfilled. We have also added the PLCRVoting test-suite.

User Input and Controlling for Bad Actors

The foam.tcr is primarily interfaced with through the FOAM dApp. Understandably, this is considered to be the primary entry point of individuals interfacing with the contract. That noted, individuals may opt to interface with the controlling contracts *directly*, and these interactions may not be under the direct control of the FOAM team.

The FOAM team and SoHo Token Labs note that bad actors may attempt to perform actions that are outside the intended application functionality, and that current contract construction does not prevent such data inputs.

As these inputs may compromise the data integrity of the platform, it is our recommendation that the FOAM team ensure that relevant input of this nature is categorized as malicious and is excluded from use.

Resolution: The FOAM team recognizes the risk presented by malicious actors stemming from the ability to interacts with the contracts directly and is structuring mitigations within the non-smart contract infrastructure in order to handle these inputs before they are processed and presented to users. Our response from FOAM was as follows: We have sophisticated measures for dealing with invalid data entering the contracts on our backend, meaning we can validate the data for the user as long as they use our platform. Since the TCR allows any kind of data to be curated, it would defeat the purpose of the TCR to do anything beyond this.

Toolset Warnings & Manual Inspection

In addition to our manual review, our process involves utilizing automated toolsets in order to perform additional verification of the presence security vulnerabilities. Mythril and Oyente, two such toolkits, were used extensively during the course of our audit. Detection capabilities for each are listed in the appendix.

The following sections detail warnings generated by the automated tools and confirmation of false positives where applicable, in addition to findings generated through manual inspection.

Contracts Directory

AbstractTest.sol

No visibility specified explicitly for any functions within AbstractTest.sol#8,13,19:

When not specified explicitly, function visibility defaults to public. While this contract exists largely as a test suite as does not inherently pose any issues, compiler warnings such as these should be noted.

Resolution: We received the following response from FOAM: Any contract matching “*Test.sol” is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

Defining constructors as functions with the same name as the contract is deprecated:

Constructors should now be defined using `constructor(uint arg1, uint arg2) { ... }` to make them stand out and avoid bugs when contracts are renamed but not their constructors.

Resolution: We received the following response from FOAM: Any contract matching “*Test.sol” is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

AttributeStoreTest.sol

No visibility specified explicitly for any functions within AttributeStoreTest.sol#13,18,27:

When not specified explicitly, function visibility defaults to public. While this contract exists largely as a test suite as does not inherently pose any issues, compiler warnings such as these should be noted.

Resolution: We received the following response from FOAM: Any contract matching "*Test.sol" is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

Defining constructors as functions with the same name as the contract is deprecated:

Constructors should now be defined using `constructor(uint arg1, uint arg2) { ... }` to make them stand out and avoid bugs when contracts are renamed but not their constructors.

Resolution: We received the following response from FOAM: Any contract matching "*Test.sol" is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

Utilizing the variable argument mode of `keccak256/sha256/ripemd160` is deprecated:

As the variable arguments of `keccak256` and other hashing algorithms are being deprecated, such calls should now use `keccak256(abi.encodePacked(...))`.

Resolution: We received the following response from FOAM: Any contract matching "*Test.sol" is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

DLLTest.sol

No visibility specified explicitly for any functions within DLLTest.sol#14,20,26,35:

When not specified explicitly, function visibility defaults to public. While this contract exists largely as a test suite as does not inherently pose any issues, compiler warnings such as these should be noted.

Resolution: We received the following response from FOAM: Any contract matching "*Test.sol" is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

Defining constructors as functions with the same name as the contract is deprecated

Constructors should now be defined using `constructor(uint arg1, uint arg2) { ... }` to make them stand out and avoid bugs when contracts are renamed but not their constructors.

Resolution: We received the following response from FOAM: Any contract matching "*Test.sol" is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

Uninitialized storage pointer `d` within `testHalf` function in `DLLTest.sol#28`:

Within testHalf, variable `d` is declared as an uninitialized storage pointer. The explicit `storage` keyword should be used, and `d` should be properly initialized, as uninitialized storage pointers can have drastic consequences when exploited.

Resolution: We received the following response from FOAM: Any contract matching "*Test.sol" is not deployed in the real application and do not belong in the audit. We initially wanted to move these to a test directory but had trouble getting truffle to work, so we resorted to recognizing them as test in the filename.

FoamToken.sol

Race Condition via Allowance Double Spend Exploit:

As this vulnerability stems from flaws in the ERC20 specification, further description of the vulnerability exists below (in which the ERC20 specification is directly addressed for relevant security flaws.

Resolution: We received the following response from FOAM: We are aware of the possible front-running exploits but continue using ERC20 as it is an industry standard

SpatialUtils.sol

Utilizing the variable argument mode of keccak256/sha256/ripemd160 is deprecated:

As the variable arguments of keccak256 and other hashing algorithms are being deprecated, such calls should now use keccak256(abi.encodePacked(...)).

Resolution: The FOAM team has since resolved this issue in pull request 355.

Non-Proprietary Contracts

contracts/PLCRVoting.sol

PLCRVoting.sol is a contract initially developed by Aspyn Palatnick, Cem Ozer, and Yorke Rhodes at ConsenSys. The contract specifies a Solidity version of 0.4.22, but this has been updated from the previously published version of Solidity 0.4.8.

Potential for integer overflow within PLCRVoting.sol#51
Mythril, manual inspection, and prior audits do not raise concern for integer overflow. Oyente however denotes the potential for execution of overflow in the line mapping(uint => Poll) public pollMap -- however, given all mappings present are contained within uint256, this does not present issue.

contracts/Registry.sol

Registry.sol is a contract initially developed by Mike Goldin at ConsenSys. The contract specifies a Solidity version of 0.4.22, but this has been updated from the previously published version of Solidity 0.4.11.

Multiple message calls to external contracts within Registry.sol#92,119,134,136,137,174,190,248,279,340,440:

Throughout Registry.sol, multiple calls to external contracts are made. While such a practice is normally of concern, the called contracts are trusted (owned by the FOAM protocol team). Line 440 is unique in which it is a transfer to an external address, which could have repercussions if the recipient is in turn a smart contract -- however, correct function scoping (private) removes concern for this potential attack vector.

Resolution: N/A, as all contracts called are owned solely by the team or otherwise have their impact mitigated.

State change following external call within Registry.sol#136:

Within Registry.sol#134, the following call is made: require(listing.unstakedDeposit - _amount >= parameterizer.get("minDeposit"));. This external call ensures that the noted state change (listing.unstakedDeposit -= _amount;) does not underflow and as a trusted contract, the external contract call is not of concern.

Resolution: N/A, as all contracts called are owned solely by the team.

Potential for integer overflow within Registry.sol#118:

The potential for integer overflow exists within Registry.so#118, similar to that within Registry.sol#136. In this instance, the line listing.unstakedDeposit += _amount; grants the potential for overflow as no bounds checking is performed on either variable to ensure overflow will not occur. The usage of require or safeAdd is suggested here to avoid issue.

Resolution: The FOAM team has since resolved this issue in pull request 355.

contracts/Parameterizer.sol

Utilizing the variable argument mode of keccak256/sha256/ripemd160 is deprecated:

As the variable arguments of keccak256 and other hashing algorithms are being deprecated, such calls should now use keccak256(abi.encodePacked(...)).

Resolution: The FOAM team has since resolved this issue in pull request 355.

Potential division by zero on Parameterizer.sol#281:

As presently written, the line (voterTokens * rewardPool) / winningTokens can lead to potential division by 0. SafeDivision should be used in order to control for such a potential occurrence.

Resolution: The FOAM team has since resolved this issue in pull request 355.

Multiple message calls to external contracts within Parameterizer.sol#250,280,311,319:

Throughout Parameterizer.sol, multiple calls to external contracts are made. While such a practice is normally of concern, the called contracts are trusted (owned by the FOAM protocol team).

Resolution: N/A, as all contracts called are owned solely by the team.

erc20-tokens/contracts/eip20/EIP20.sol & zeppelin-solidity/contracts/token/ERC20/StandardToken.sol

Despite their correspondence to the same specification, two implementations of the ERC20 standard are present in the codebase. While both are utilized in differing capacities, each is vulnerable to the allowance double spend exploit:

Race Condition via Allowance Double Spend Exploit could lead to token theft:

A known race condition exists within the present implementation of the ERC20 standard. Due to the nature of this vulnerability being an inherent flaw in the ERC20 standard, considerations must be made for any divergence (as modifications made while no longer be ERC20 compliant).

The scenario for exploitation is as follows:

1. Alice calls approve(Bob, 1000), allocating 1000 tokens for Bob to spend
2. Alice opts to change the amount approved for Bob to spend to a lesser amount via approve (Bob, 500). Once mined, this decreases the number of tokens that Bob can spend to 500.
3. Bob sees the transaction and calls transferFrom(Alice, X, 1000) before
4. approve(Bob, 500)has been mined.
5. If Bob's transaction is mined prior to Alice's, 1000 tokens will be transferred by Bob.
6. However, once Alice's transaction is mined, Bob can call transfer From (Alice, X, 500), transferring a total of 1500 tokens despite Alice attempting to limit the total token allowance to 500.

The particular exploit requires the usage of both the transferFrom and approve functions. As demonstrated above, the race condition occurs when one calls approve a second time on a

spender that has already been allowed. If the spender sees the transaction containing the call before it has been mined, then the spender can call transferFrom to transfer the previous value and still receive the authorization to transfer the new value. While a multitude of approaches do exist to prevent this particular behavior from being exploited (one example of such is included below), they unfortunately may cause downstream functionality issues with those who implement such changes.

Resolution: A potential fix includes preventing a call to approve if all the previous tokens are not spent through adding a check that the allowed balance is 0:

```
require(allowed[msg.sender][_spender] == 0)
```

Directory Structure

At time of audit, the directory structure of the foam.tcr contract repository was as follows:

Directory Information:

```
├──── AbstractTest.sol
├──── AttributeStoreTest.sol
├──── DLLTest.sol
├──── FoamToken.sol
├──── PLCRVoting.sol
├──── Parameterizer.sol
├──── Registry.sol
└──── SpatialUtils.sol
```

0 directories, 8 files

Test Coverage

At time of audit, the test coverage of the foam.tcr contract repository was as follows:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
PLCRFVoting.sol	98.87	77.11	98.18	98.52	
Parameterizer.sol	96.2	55.17	98.45	95.24	102,103,105, 370
Registry.sol	100	84.09	100	100	100
All files	98.87	77.11	98.18	98.52	

Appendix

Mythril Detection Capabilities

Issue	Description	Mythril Detection Module(s)	References
Unprotected functions	Critical functions such as sends with non-zero value or suicide() calls are callable by anyone, or msg.sender is compared against an address in storage that can be written to. E.g. Parity wallet bugs.	Unchecked_suicide, Ether_send unchecked_retval	
Missing check on CALL return value		unchecked_retval	Handle errors in external calls
Re-entrancy	Contract state should never be relied on if untrusted contracts are called. State changes after external calls should be avoided.	external calls to untrusted contracts	Call external functions lastAvoid state changes after external calls
Multiple sends in a single transaction	External calls can fail accidentally or deliberately. Avoid combining multiple send() calls in a single transaction.		Favor pull over push for external calls

External call to untrusted contract		external calls to untrusted contracts	
Delegatecall or callcode to untrusted contract		delegatecall_forward	
Integer overflow/underflow		integer	Validate arithmetic
Timestamp dependence		Dependence on predictable variables	Miner time manipulation
Payable transaction does not revert in case of failure			
Use of tx.origin		tx_origin	Solidity documentation, Avoid using tx.origin
Type confusion			
Predictable RNG		Dependence on predictable variables	
Transaction order dependence		Transaction order dependence	Front Running
Information exposure			
Complex fallback function (uses more than 2,300 gas)	A too complex fallback function will cause send() and transfer() from other contracts to fail. To implement this we first need to fully implement gas simulation.		

Use require()instead of assert()	Use assert() only to check against states which should be completely unreachable.	[Exceptions]()	[Solidity docs]()
Use of depreciated functions	Use revert()instead of throw(), selfdestruct() instead of suicide(), keccak256() instead of sha3()		
Detect tautologies	Detect comparisons that always evaluate to 'true', see also [#54]()		
Call depth attack	Deprecated		

Oyente Detection Capabilities

Issue	**Description**
Re-entrancy	Contract state should never be relied on if untrusted contracts are called. State changes after external calls should be avoided.
Timestamp Dependence	The timestamp of the block can be manipulated by the miner, and so should not be used for critical components of the contract. Block numbers and average block time can be used to estimate time, but this is not future proof as block times may change (such as the changes expected during Casper).
Assertion Failure	An assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program.

	Assertion failures as such denote critical instances in which assumptions made by the developer no longer hold to be true.
Callstack Depth Attack	Deprecated
Transaction Order Dependence (TOD)	Since a transaction is in the mempool for a short while, one can know what actions will occur, before it is included in a block. This can be troublesome for things like decentralized markets, where a transaction to buy some tokens can be seen, and a market order implemented before the other transaction gets included.
Parity Multisig Bug 2	Unchecked kill/selfdestruct functions, such as those within the Parity Multisig Bug 2 can lead to destruction of the contract, sending funds to the given address provided.