PUBLIC

Security Audit

of DAOStack's Smart Contracts

January 15, 2019

Produced for

by

Table Of Content

Foreword 1

Executive Summary 1

Audit Overview 2

1. Scope of the Audit 2

2. Depth of Audit 3

3. Terminology 3

Limitations 5

System Overview 6

1. Reputation 6

2. Fixed Reputation Allocation 6

3. Auction for Reputation 6

4. Locking ETH for Reputation 6

5. Locking Token for Reputation 6

6. External Locking for Reputation 7

7. Forwarder 7

8. Making External Calls 7

Best Practices in DAOSTACK's project 8

1. Hard Requirements 8

2. Soft Requirements 8

Security Issues 9

1. Unlimited redemption of reputation H ✓ Fixed 9

2. Potential of locked tokens in `LockingToken4Reputation` due to unsafe math M ✓ Fixed . . 9

3. Non use of `SafeMath` results in theoretical overflows L ✓ Addressed 9

4. Enforcing continuation of ICO after cap has been reached H ✓ Fixed 11

5. Division Before Multiplication L ✓ Acknowledged . . . 12

6. Dependence on block.timestamp L ✓ Acknowledged . . . 12

Trust Model and Implications . . . 13

1. Trust in schemes and technical competence of agents M ✓ Acknowledged . . . 13

2. Owner of proposal may vote one behalf of users L ✓ Fixed . . . 13

3. Contract owner can obtain unearned reputation M ✓ Addressed . . . 13

Design Issues . . . 14

1. No benefit of using the `RealMath` library M ✓ Fixed . . . 14

2. Wrong unit denomination M ✓ Fixed . . . 15

3. Inconsistent interpretation of voting parameters M ✓ Fixed . . . 16

4. Minor code duplication in `Controller` L ✓ Fixed . . . 17

5. Unexpected results when a proposal does not exist L ✓ Fixed . . . 17

6. Missing check for `zero` address L ✓ Fixed . . . 17

7. Rounding issues in schemes M ✓ Fixed . . . 17

8. Uncatched exception when making a call proposal L ✓ Fixed . . . 18

9. Inefficient struct storage L ✓ Fixed . . . 19

10. Unchecked return value M ✓ Fixed . . . 19

11. Strong incentives for delayed bidding M ✓ Acknowledged . . . 20

12. Redundant operation when burning reputation L ✓ Fixed . . . 20

13. Compilation with experimental `pragma 0.5.0` fails L ✓ Fixed . . . 21

14. High complexity of execution L ✓ Acknowledged 21

15. Imprecise estimation of block numbers M ✓ Acknowledged 21

16. Unused event `RefreshReputation` L ✓ Fixed 22

17. Used `ERC20` instead of `IERC20` L ✓ Fixed 22

18. Unused imports L ✓ Fixed 22

19. Type of argument `reputationReward` unclear L ✓ Fixed 23

20. Wasteful conversion L ✓ Fixed 23

Recommendations / Suggestions 24

Disclaimer 27

Foreword

We first and foremost thank DAOSTACK for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The DAOSTACK smart contracts have been analyzed under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts and expert manual review.

Overall, we found that DAOSTACK employs good coding practices and has clean code. However, the system in its current state needs to be documented more exhaustively. CHAINSECURITY was able to uncover several issues in the newly introduced schemes which were successfully fixed by DAOSTACK before deployment.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on December 10, 2018 [1] and an updated version on January 14, 2019 [2]:

File	SHA-256 checksum
arc/../controller/Avatar.sol	d423c5a662a5d91e6b182a8ed54ab12e7ba08dcd8777c819774a1454a9ea50b8
arc/../controller/Controller.sol	81b79db8103c92495f561e3edea2566bcd5e52c27b45069d9a406d50b62a3582
arc/../controller/ControllerInterface.sol	97ba611b8b99bf5fee8cf5b4856e957c32ebf3deddd4edda86aa04de103ff6ed
arc/../controller/DAOToken.sol	bb959c7ee7e119bd49b0731d55cb707fb26ef0979b309d6eeefd812ef9ef9e77
arc/../controller/UController.sol	211a96cce968930502c9275533e4b0e62228f0f0e74719bead45eabea977b33d
arc/../globalConstraints/GlobalConstraintInterface.sol	ff163ce46cc520151f2cad1829aa5389c6327a7d20afb7d2a9ebb0d510b3fb6c
arc/../globalConstraints/TokenCapGC.sol	a311ec70483485231637f96baa2561537dad3ec5cbb5ff4eaded5a77227dc56e
arc/../libs/SafeERC20.sol	28257a72747aad58edbfea41de7d97f27e5657fc438906984348634791ed6739
arc/../schemes/Auction4Reputation.sol	31073aa879fdcd6ee40be7e4c3b28d8014ae6c89c41e2a7eb26cbca91aacc268
arc/../schemes/ExternalLocking4Reputation.sol	4c59484d8ed8a85705b05e37bb69ea2603d594552c3b78368b65683647988124
arc/../schemes/FixedReputationAllocation.sol	3c1814b16b62945c64764bc1e9a4cce383d7b98175464a4b4cac230aa5a4d946
arc/../schemes/Forwarder.sol	618aed64d5a300f51124a46c2d55bbe38739ef08fb671efcfd5c66db21d340e6
arc/../schemes/Locking4Reputation.sol	9b4faa92197dd2a6bd0ca37c00c7b1d88ef850dcd2a89134721ccf52051a3e1a
arc/../schemes/LockingEth4Reputation.sol	2232c799bbd4d508dc559a6162e8a33fa3f5c390e5409accec935c9ec9749d65
arc/../schemes/LockingToken4Reputation.sol	64a726c5fc057246cafb1f668398afc93d7a79397369242322a4a8cbdb09bd8e
arc/../schemes/PriceOracleInterface.sol	a3ac76b863791feca2b89c3e5d59e35d4eabb953a1350f4e6268f45f02f5757c
arc/../universalSchemes/ContributionReward.sol	dd30309a56d2c9405d2cf513acddefc54521c5312d3297409a9552dccdbd0720
arc/../universalSchemes/DaoCreator.sol	7a9a491aa7d668fbddea189b49a0cd82603a072ad27251e13baeadcc9bcf348a
arc/../universalSchemes/GenericScheme.sol	57a769e8552a7187a2998d4d391717e25bd35ee1229aff292a55217f0be16e4c
arc/../universalSchemes/GlobalConstraintRegistrar.sol	b4cf0293abd033083d432afd9d1d9cbf414fd3d8ce0d3d9a7fe5990188c97fef
arc/../universalSchemes/OrganizationRegister.sol	85017d44551e46e4b1129b9f5b29148aa198c7ee4276e6b4f31d3c8d1184ed6e
arc/../universalSchemes/SchemeRegistrar.sol	d549a6fb390c092397af756a50087cd396c064b01bb231b02e094f827324533e
arc/../universalSchemes/UniversalScheme.sol	2d6f9d46ce0aad1c274b6674d0366009a6f20ccb2f74de405bedf46ee64ff749
arc/../universalSchemes/UniversalSchemeInterface.sol	162081708d9574031abd655117d9e7c632dfcb14d2da0a090c5d483511abdba6
arc/../universalSchemes/UpgradeScheme.sol	bcf3e4672de8aafc9177e3eb28b3310eca204803944d3984b7f4a2ccc695cb8d
arc/../universalSchemes/VestingScheme.sol	2c62d692caa92e38aa0290c337724bf9007a4b5d2edfd4bafe154cb00fb9ee33
arc/../universalSchemes/VoteInOrganizationScheme.sol	f8331766e2bd22c5d1c38027d908affb76cb96df59deabed4bf40e4317dd55e3
arc/../utils/Redeemer.sol	9c19551002194bc9dfcc8b56d8c7b4216b0b2a09f6779dc78728c4f09789b09a
arc/../votingMachines/VotingMachineCallbacks.sol	33913ae6738240e5da7fe141a3b91c7ff7926197a8e6e0d885eb0918fc935a8f

[1] https://github.com/daostack/arc/tree/c75e9b8f9647e2f876f2205cdbc00077a418cc91, https://github.com/daostack/infra/tree/823194f3ba89139d82b6709bf4aace53f3c9634c

[2] https://github.com/daostack/arc/tree/bcb7d5c868d5a53fc73bd86ee001de447df61643, https://github.com/daostack/infra/tree/2d74f5e8862617a3c3f18c3dfbf301edc507acae

File	SHA-256 checksum
infra/../Reputation.sol	44b4830278c014ecc0083d54373d063daa48feef0c87db55e2e496e1495200be
infra/../libs/RealMath.sol	172f01f498882e3254af53824f06617396ac1ae80ee60664975e2b8335e90f50
infra/../votingMachines/AbsoluteVote.sol	fde68e53790d88ce3b0247e5d21d588398780610416fd82defd62321c99ea746
infra/../votingMachines/GenesisProtocol.sol	892c0a4db3ccc3abdf7a6f36d1e6952324d72609ab3f2028271619fda1027cc3
infra/../votingMachines/GenesisProtocolLogic.sol	57d932b943ec478a20eaa9e58b367f430639bf1da9cf13755da1e9675c431912
infra/../votingMachines/IntVoteInterface.sol	fc2b493a5f3a9ed6e714882144d97149b7c1a3c478c80ae41e620557128833db
infra/../votingMachines/ProposalExecuteInterface.sol	28cba373af75109b255aac4a6adde9ebab64cd8d78e7c35861203c662f57e5e1
infra/../votingMachines/QuorumVote.sol	77e0debf6e03ae1997d0b326e24d5abe7532fbe09ecd502295c68920322cc84c
infra/../votingMachines/VotingMachineCallbacksInterface.sol	cd40dc8fa1b337bb34674ccbe17dc5a8f669cf6fefaf7c7f001892f950273d4d

Please note that only the updated versions of `GenesisProtocol.sol` and `GenesisProtocolLogic.sol` have been audited.

Depth of Audit

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

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[3]).

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 calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

- L Low: can be considered as less important
- M Medium: should be fixed
- H High: we strongly suggest to fix it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	High	Medium	Low
High	C	H	M
Medium	H	M	L
Low	M	L	L

[3] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

During the audit concerns might arise or tools might flag certain security issues. After careful inspection of the potential security impact, we assign the following labels:

- ✓ No Issue : no security impact
- ✓ Fixed : during the course of the audit process, the issue has been addressed technically
- ✓ Addressed : issue addressed otherwise by improving documentation or further specification
- ✓ Acknowledged : issue is meant to be fixed in the future without immediate changes to the code

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

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 allows to discover vulnerabilities that were overlooked during development and areas where additional security 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. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

System Overview

Reputation

Agent's votes are weighted by their reputation. User reputation is similar to a user's token balance with the difference that reputation is not transferable. Reputation can however be minted and burned by the contract `owner`. A user's reputation is tracked over time to form reputation history in the form of checkpoints. Every time the reputation amount changes either through minting or burning, a new checkpoint which stores the current block number and the new reputation amount, is added to the user history. This enables the system to know exactly how much reputation a user had at any block number.

There are multiple schemes in the DAO implementing different distributions of reputation.

Fixed Reputation Allocation

Contract `FixedReputationAllocation` implements a scheme that allocates a predefined amount of reputation to whitelisted beneficiaries. The owner sets the amount of reputation reward to be distributed, adds all beneficiaries to the whitelist and enables the redemption. Beneficiaries have to call `redeem()` to collect their reputation.

Auction for Reputation

Contract `Auction4Reputation` implements a scheme to auction off a certain amount of reputation in exchange for a given ERC20 token. The auction runs within a certain timeframe and may be divided into multiple sub-auctions. All parameters, the reputation reward to be distributed, the start time of the auction, the end time of the auction, the amount of auctions during this timeframe and the time from which on users can claim their reputation are set by the owner upon initialization. Furthermore the ERC20 token accepted as payment and the address which can claim these must be set.

The total amount of reputation to be distributed is split equally among all sub-auctions. All participants during an auction pay the amount of tokens they contribute using the `bid` function. The amount of reputation one receives is the percentage of one's contribution during the sub-auction multiplied by the reward to be distributed per auction, rounded down.

Only after all auctions are completed and the redemption is enabled can users claim their reputation using the `redeem` function. The owner can transfer the collected tokens to the wallet address defined during the initialization, once the auction has reached its end.

Locking ETH for Reputation

Contract `LockingEther4Reputation` implements a scheme to distribute reputation in exchange for locking up ETH for a certain amount of time.

Parameters are: total reputation reward to be distributed, the locking start time, the locking end time, the redemption start and the maximum locking period. These are set by the owner upon initialization.

Users can participate by calling `lock` with the intended parameters and `msg.value`. Furthermore, calls to `release` to retrieve the funds and to `redeem` to claim the rewards are necessary interactions after the respective times elapsed.

Locking Token for Reputation

Contract `LockingToken4Reputation` implements a scheme to distribute reputation in exchange for locking up tokens for a certain amount of time.

A price oracle returns the exchange rate for the supported tokens.

Parameters are: total reputation reward to be distributed, the start time of the lockings, the end time of the lockings, the start time of the redemption, the maximum locking period and the address of the price oracle. These are set by the owner upon initialization.

Users can participate by calling `lock` with the desired parameters. Furthermore calls to `release` to retrieve the funds and `redeem` to claim the reward are necessary interactions after the respective timeframes have elapsed.

External Locking for Reputation

Contract `ExternalLocking4Reputation` implements a scheme to distribute reputation in exchange for locking up tokens in an external contract.

The external locking contract address and all other parameters, like the total reputation reward to be distributed, the claiming start time, the claiming end time and the time users are able to redeem their reputation is set by the owner upon initialization.

Users interact with the `claim` function which handles the interaction with the external locking contract. If the call to the external contract was successful it locks the returned amount of tokens.

Users must call `redeem()` once the time has advanced past the `redeemEnableTime` to claim their reputation.

Note that the locking and retrieval of the locked funds happens in the external contract.

Forwarder

Contract `Forwarder` implements a scheme which forwards all calls via the fallback function to a defined contract. This contract is intended to be an avatar. The defined contract and the expiration time, after which any call will revert are set by the owner upon initialization. The only exception is an `unregisterSelf` functionality, allowing anyone to unregister the scheme after the expiration time has passed. This scheme is intended to forward calls to a dao.

Making External Calls

The `GenericScheme` contract implements a new `universal scheme` that enables proposing and executing calls to arbitrary functions. Any user is able to propose that a given `avatar` executes a call to an external function and a set of parameters as provided by the proposal maker. The voting options are either for or against the execution of the call.

Should the proposal be accepted, the call will be executed through the `controller` contract which now supports making generic contracts using the organization's `avatar`.

Best Practices in DAOSTACK's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when DAOSTACK's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the DAOSTACK's project can be audited by CHAINSECURITY.

- [x] The code is provided as a Git repository to allow the review of future code changes.
- [x] Code duplication is minimal, or justified and documented.
- [x] Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with DAOSTACK's project. No library file is mixed with DAOSTACK's own files.
- [x] The code compiles with the latest Solidity compiler version. If DAOSTACK uses an older version, the reasons are documented.
- [x] There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to DAOSTACK.

- [x] There are migration scripts.
- [x] There are tests.
- [x] The tests are related to the migration scripts and a clear separation is made between the two.
- [x] The tests are easy to run for CHAINSECURITY, using the documentation provided by DAOSTACK.
- [ ] The test coverage is available or can be obtained easily.
- [x] The output of the build process (including possible flattened files) is not committed to the Git repository.
- [x] The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- [x] There is no dead code.
- [x] The code is well documented.
- [x] The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- [ ] Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- [x] There are no getter functions for public variables, or the reason why these getters are in the code is given.
- [x] Function are grouped together according either to the Solidity guidelines[4], or to their functionality.

[4] https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions

Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Unlimited redemption of reputation ✓ Fixed

The `FixedReputationAllocation` scheme is intended to allocate a predefined reputation amount to whitelisted beneficiaries. A beneficiary can call `redeem()` repeatedly and each time his reputation gets increased by the value of the `beneficiaryReward`. Consequently, the total reputation to be distributed is not limited to the value of the state variable `reputation reward`, which is described by “the total reputation this contract will reward”.

Likelihood: High
Impact: High

Fixed: A beneficiary is now removed from the `beneficiaries` map after redeeming their reputation. This prevents multiple successful calls to the `redeem` function by the same beneficiary.

Potential of locked tokens in `LockingToken4Reputation` due to unsafe math ✓ Fixed

A `LockingToken4Reputation` contract allows users to lock up tokens and earn reputation. This contract may support multiple tokens, depending on the `priceOracle`.

Irregardless of the token to be locked up, the `lock` function is called which calls the internal `_lock` function with the respective parameters. The contract keeps track of the amount of tokens currently locked up. The new amount is calculated with `totalLocked += _amount;` using the standard addition operation. Note that there is one counter only for all different tokens. Next `totalLockedLeft` is updated with the new amount of `totalLocked`: `totalLockedLeft = totalLocked;`. Thus if the previous addition overflowed, the variable `totalLocked` will be significantly smaller than the actual amount of locked tokens.

When releasing tokens, the contract keeps track of how many tokens are left locked up, notably this is done with a `SafeMath` subtraction: `totalLockedLeft = totalLockedLeft.sub(amount);` This transaction cannot underflow as `SafeMath` would make it revert, thus the release of tokens would not be possible anymore if the amount of tokens to be released is > `totalLockedLeft`.

Likelihood: Low
Impact: High

Fixed: DAOSTACK solved the problem by using `SafeMath` instead of the standard addition operation, this prevents any overflow and thus removes the risk of locked tokens.

Non use of `SafeMath` results in theoretical overflows ✓ Addressed

The `SafeMath` library is not consistently used throughout the code base resulting in possible overflows during arithmetic operations. A separate issue has been opened for an overflow with an actual and direct impact on the system, while this issue summarizes overflows which are less probable.

Unauthorized token withdrawals

Only possible in theory, as overflowing a `uint256` by incrementing it by one inside the large and gas-heavy `lock` function is not feasible for any attacker. Nonetheless CHAINSECURITY wants to raise awareness for the following scenario: The vulnerability itself is in the `Locking4Reputation` contract but will be exploited through `LockingToken4Reputation` which inherits from `Locking4Reputation`. We assume the attacker controls two addresses, `0xA` and `0xB`.

- The attacker, using `0xA`, locks 10000 cheap tokens and we assume the token is deployed at address `0xLO`. This token must be known to the `priceOracleContract` s.t. the locking is successful.

- A lockingId will be created: keccak256(abi.encodePacket(this, lockingsCounter). Note that no address is not included. This will result in that both 0xA and 0xB will be able withdraw "their" tokens in the end. The lockingId will be used to create a locking in the storage at the place lockers[0xA][lockingId].
- lockingsCounter is a counter which gets incremented with lockingsCounter++.
- Finally lockedTokens[lockingId] is set to the address of the worthless token at 0xLO. Now we need to wait until the counter overflows or make this happen, since we want to create a new locking with the same lockingId.
- After the counter overflowed, the attacker, using 0xB, locks one unit of a high value token deployed at 0xHI with the same lockingId.
- Again, the lockingId will be created with keccak256(abi.encodePacket(this, lockingsCounter). Note that this will create a locking in the storage at the place lockers[0xB][lockingId]
- The attacker uses 0xA and 0xB so he can withdraw everything. Otherwise he would overwrite it and could only withdraw the latter. At this point lockedTokens[lockingId] is overwritten by the address of the high value token at 0xHI.
- Now, using 0xA, the attacker releases his tokens by calling release(). Everything will pass and the amount of locked up tokens will be transferred with the transfer function of the address stored in the lockedTokens[lockingId]. Note that this was initially the address of the cheap token at 0xLO but has just been overwritten by address the high value token at 0xHI.
- Hence, despite 0xA locking up 1000 worthless tokens at 0xLO, the attacker now can get a transfer of 1000 high value tokens at 0xHI.We remark that for this to work the contract must have enough funds of the higher value token, but this will likely be the case, e.g. tokens of other users which are currently locked up. More so, legitimate users with locked up tokens will not be able to release their tokens anymore if the contract doesn't have enough tokens anymore.

Fixed: In the updated code this is not possible anymore as SafeMath is now used which prevents an overflow of lockingsCounter.

Execution of proposal with few votes

In infra contract AbsoluteVote: Proposals can be executed when following condition holds:

```
if (proposal.votes[cnt] > totalReputation*precReq/100)
```

Note that precReq can only be between 1 and 100. The totalReputation is not necessarily bounded and may as high as $2^{256}-1$. The total reputation is set by the owner when setting up the distribution of the reputation. It is noteworthy that if the proposal has multiple choices, the first choice which fulfill this condition is executed.

Addressed: This issue was address by moving the division forward which leads to a loss of precision. This is described in a separate issue, namely Division before Multiplication. ChainSecurity recommends to consider another fix or use the original code.

Overflowing reputation through ContributionReward

The redeemReputation function defined in the universal scheme ContributionReward calculates the reputation using:

```
229 reputation = int(periodsToPay) * _proposal.reputationChange;
```

ContributionReward.sol

In the highly unlikely event that the multiplication parameters are sufficiently high the operation may overflow.

Fixed: DAOStack added validation of the proposal parameters which ensures that the maximum amount of periods times the reputationChange does not overflow.

Overflow in GenesisProtocolLogic possible

The GenesisProtocolLogic._execute function makes an untrusted call to getTotalReputationSupply() function of the contract VotingMachineCallbacksInterface which returns a uint256 number and gets stored in totalReputation. This is then multiplied by the queuedVoteRequiredPercentage which is between 50 to 100. Hence the value of executionBar can be manipulated by the proposer to some extent.

```
491 uint totalReputation = VotingMachineCallbacksInterface(proposal.callbacks).
      getTotalReputationSupply(_proposalId);
492 uint executionBar = totalReputation * params.queuedVoteRequiredPercentage/100;
```

GenesisProtocolLogic.sol

In function redeem of GenesisProtocolLogic contract, the operation below would result in an integer underflow if expirationCallBountyPercentage is 100. This can be set in the executeBoosted function.

```
271 uint _totalStakes = ((totalStakes*(100 - proposal.
      expirationCallBountyPercentage))/100) - proposal.daoBounty;
```

GenesisProtocolLogic.sol

A further theoretical integer overflow, although practically mostly impossible to exploit is:

```
287 uint preBoostedVotes = proposal.preBoostedVotes[YES] + proposal.
      preBoostedVotes[NO];
```

GenesisProtocolLogic.sol

ChainSecurity recommends to fix both integer overflow issue by using the SafeMath library.

Addressed: DAOStack has acknowledged that the totalReputation can have max value for uint256 type. DAOStack has fixed or addressed other reported overflow issues. Due to a corner case where the value of expirationCallBountyPercentage is 100, totalStakes can still underflow.

Likelihood: Low
Impact: High

Enforcing continuation of ICO after cap has been reached ✓ Fixed

start() of SimpleICO is public and may be invoked by anyone for any existing avatar on the condition that this avatar has no currently running ICO in this contract and getParametersFromController() returns a hash were the parameter cap is non-zero. Parameters for the ICO will be provided by the Controller and are retrieved by a call to getParametersFromController(). Clear specifications of the expected behavior of this function are missing, implicitly it is assumed that it only returns a hash of parameters when it's ready to open an ICO which is not documented however and may not hold.

If the ICO reaches its max. funding before the endBlock, anyone may "restart" it immediately by calling start(). This works under the assumption that getParametersfromController() returns the same hash again which hold in the current implementation.

start() can be called successfully because as the funding goal was reached isActive() will return false . A new organization struct will be initialized with the paramsHash, the new avatarContractICO and overwrite the old organizationsICOInfo[_avatar].

Calls to the mintToken() function will still go to the same contract as before, the outcome depends on this function.

Likelihood: Low
Impact: High

Fixed: DAOSTACK removed the SimpleICO contract.

Division Before Multiplication ✓ Acknowledged

The division operation may be performed before the multiplication as the order of evaluation of the expression is not guaranteed for same precedence operators. This could result in possible fractional errors in the result.

```
494         uint256 executionBar = (totalReputation/100) * params.
                queuedVoteRequiredPercentage;
```

GenesisProtocolLogic.sol

```
253             if (proposal.votes[cnt] > (totalReputation/100)*precReq) {
```

AbsoluteVote.sol

```
27          if (proposal.totalVotes > totalReputation*precReq/100) {
```

QuorumVote.sol

CHAINSECURITY recommends to explicitly perform multiplication before division operation.

Likelihood: Low
Impact: Medium

Acknowledged: DAOSTACK acknowledged that the totalReputation can be max of uint256 type. DAOSTACK is aware of the fractional errors and they are bearable.

Dependence on block.timestamp ✓ Acknowledged

Time critical events such as the start/end of an auction or the redemption period use now, an alias for block .timestamp for determining the time.CHAINSECURITY wants to make DAOSTACK aware that miners are able to manipulate this value. According to the Ethereum Yellowpaper, the only condition for a block's timestamp is that it is bigger than the timestamp of it's parent's block. Popular clients as Geth and Parity accept blocks with timestamps up to 15 seconds in the future. CHAINSECURITY recommends to consider the best practices regarding the use of timestamps in solidity[5].

If the time dependent event can vary by 15 seconds, the usage of timestamps is acceptable.

Likelihood: Low
Impact: Medium

Acknowledged: DAOSTACK is aware of the best practices and confirms this is no issue for their set of contracts.

[5] https://consensys.github.io/smart-contract-best-practices/recommendations/#timestamp-dependence

Trust Model and Implications

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into DAOSTACK, including in DAOSTACK's ability to deal with such powers appropriately.

CHAINSECURITY raises these issues to increase awareness about what roles have which powers and what consequences can be encountered in case of collusion, corruption or malicious behaviour, as this is important to consider for both DAOSTACK and its users.

Trust in schemes and technical competence of agents ✓ Acknowledged

Upon creation of a new proposal, the callback address is set to `msg.sender`. This callback contract is supposed to provide all required functionality for a `VotingMachineCallbacksInterface`, namely functionality to execute the proposal `executeProposal()` and related to the reputation `getTotalReputationSupply()` and `reputationOf()`. It is important to note that participating users need to trust the implementation of these functions. Proposals to be voted on are created by registered schemes. A malicious scheme may not execute the supposed choice as expected.

Initially schemes are added when an organization is forged. `DaoCreator` features a `setSchemes()` function where the avatar decides which schemes are added. Users either need to trust the organization or need a high level of technical competence to fully understand what the bytecode of the registered schemes is doing.

Contract `SchemeRegistrar` allows an organization to vote on the registering of new schemes while the organization is already operating. Users need to be very careful to really understand what the bytecode of the scheme at the given address is actually doing so that a malicious scheme does not get added to the organization.

Acknowledged: DAOSTACK acknowledges that malicious schemes can do harm to the system but assumes that such schemes will not be registered to the organization in the first place. Additionally, schemes can be verified on platforms such as `Etherscan` before they are registered through voting.

Owner of proposal may vote one behalf of users ✓ Fixed

The `AbsoluteVote` contract features an `ownerVote` function which allows the owner of a proposal to vote on behalf of anyone with reputation if the `allowOwner` flag of the proposal is true. This creates a trust issue for users, especially as the owner can not only vote but also override any already casted vote. Note that only registered schemes may be able to exploit this in a meaningful way. Please refer to issue `Trust in Schemes and technical competence of agents` regarding trust in schemes.

Fixed: DAOSTACK replaced the `allowOwner` with the `voteOnBehalf` functionality which resolves the problem of overwriting casted votes of unsuspecting users while providing a proxy for votes functionality. This approach gives a clean separation between the two modes of operations, either no `voteOnBehalf` address is set or the address set for `voteOnBehalf` needs to vote as proxy on behalf of every participating address.

Contract owner can obtain unearned reputation ✓ Addressed

`Auction4Reputation` auctions of reputation for tokens, enabling users to buy voting power according to their token contribution. Users participating in this scheme must understand the underlying assumption that the owner is trusted and this should be clearly documented.

As all collected tokens will be transferred to the wallet address of the owner, the owner may simply participate in the auction. However instead of paying for the reputation with his own tokens, the owner just locks up the other contributor's tokens in the wallet address for a certain amount of time. Consequently the owner has voting power without a matching token contribution, which is misleading and unfair to the users participating in this scheme.

Addressed: DAOSTACK notes that the wallet address should be a trusted account and claims that in most cases the wallet will be the `avatar` address.

Design Issues

The points listed here are general recommendations about the design and style of DAOSTACK's project. They highlight possible ways for DAOSTACK to further improve the code.

No benefit of using the `RealMath` library ✓ Fixed

The `RealMath` library used actually provides no benefit regarding precision but significantly increases the gas consumption of the execution.

For doing a multiplication, using `RealMath` provides no benefit if both values are converted just prior using `toReal()` as they won't have a fractional part. Performing `RealMath.div` keeps fractions decimals, however just after doing the division the result is converted using `fromReal()` which truncates the fractional part. In sum this behaves just like `SafeMath.div` or `a/b` but is significantly more expensive in terms of gas.

This can be observed in the following parts of the implementation:

- `Locking4Reputation.sol`

```
int256 repRelation = int216(score).toReal().mul(int216(reputationReward)
        .toReal());
reputation = uint256(repRelation.div(int216(totalScore).toReal())
        .fromReal());
```

- `FixedReputationAllocation.sol`

```
beneficiaryReward = uint256(int216(reputationReward)
        .div(int256(numberOfBeneficiaries)).fromReal());
```

- `Auction4Reputation.sol`

```
  int256 repRelation = int216(bid).toReal().mul(int216(
      auctionReputationReward)
        .toReal());
  reputation = uint256(repRelation.div(int216(auction.totalBid).toReal())
        .fromReal());
```

- `GenesisProtocolLogic.sol`

```
  uint256(int256(averageDownstakesOfBoosted) +
        ((int216(proposal.stakes[NO])-int216(averageDownstakesOfBoosted))
        .toReal()
        .div(int216(orgBoostedProposalsCnt[proposal.organizationId])
        .toReal())).fromReal());
```

```
uint256(int216(averageDownstakesOfBoosted
        .mul(boostedProposals+1)
        .sub(proposal.stakes[NO])).toReal()
        .div(int216(boostedProposals).toReal()).fromReal());
```

```
uint((int216(proposal.stakes[YES]).toReal()
        .div(int216(proposal.stakes[NO]).toReal())).fromReal());
```

There are only two further uses of functions of the `RealMath` library:

- `GenesisProtocolLogic.sol`

```
int alpha = int216(_params[4]).fraction(int216(1000));
        //set a limit for power for a given alpha to prevent overflow
        uint256 limitExponent = 172;//for alpha less or equal 2
        uint256 j = 2;
        for (uint256 i = 2; i < 16; i = i*2) {
            if ((uint(alpha.fromReal()) > i) && (uint(alpha.fromReal()) <= i
                *2)) {
                limitExponent = limitExponent/j;
                break;
            }
            j++;
```

Note that this can easily be implemented without `RealMath`. Furthermore please refer to issue `Wasteful conversion`.

```
    function threshold(bytes32 _paramsHash, bytes32 _organizationId) public
        view returns(uint256) {
        int256 power = int216(orgBoostedProposalsCnt[_organizationId]).toReal
            ();
        Parameters storage params = parameters[_paramsHash];

        if (power.fromReal() > int(params.limitExponentValue)) {
            power = int216(params.limitExponentValue).toReal();
        }

        return uint(params.thresholdConst.pow(power).fromReal());
    }
```

CHAINSECURITY advises DAOSTACK to remove the unnecessary and complex `RealMath` library. The required `pow()` functionality may be implemented directly.

Fixed: DAOSTACK removed the redundant `RealMath` usage and reduced the library functionality to only the exponentiation operation.

Wrong unit denomination M ✓ Fixed

The `SimpleICO` contract features following variables to store ETH related values:

- `totalEthRaised`
- `uint cap`, described as `Cap in Eth`
- `uint price`, described as `Price represents Tokens per 1 Eth`
- `uint incomingEther`
- `uint change`

These variables are implicitly assumed to store values in ether units. However, they interact directly with `msg.value` without any prior conversion and `msg.value` is denominated in wei[6]. Consequently, there is a mismatch in orders of magnitude.

Note, that while DAOSTACK has tests, these test were done with hard-coded numerical values assumed to be ETHs while actually all test operations handled some small wei amounts only. Any real ETH transaction to the `donate()` function would have uncovered the mismatch immediately.

Fixed: DAOSTACK removed the `SimpleICO` contract.

[6] https://solidity.readthedocs.io/en/v0.4.25/units-and-global-variables.html#block-and-transaction-properties

Inconsistent interpretation of voting parameters

The implementation of the voting machines is not consistent on whether 0 or 1 describes the first `choice` of a proposal. 0 is the default value for an uninitialized `uint` and inside the `AbsoluteVote` contract a vote for `choice 0` is described as "abstain". The implementation however treats `choice 0` no different than any other choice which can be voted for and executed if the required percentage of votes has been reached.

This has the following implications:

- `AbsoluteVote` starts counting choices at 0. Hence the maximal number of choices resulting is `MAX_NUM_OF_CHOICES` + 1 which might be unexpected.
- `QuorumVote`, which inherits from `AbsoluteVote` and only overrides `_execute`, starts checking choices from 1 instead of 0, so the 0 proposal can never win or be executed. Note, that through the inherited `_vote()` function from `AbsoluteVote` the 0 choice can still get votes.
- `getAllowedRangeOfChoices()` returns `(1, MAX_NUM_OF_CHOICES)` which is again inconsistent as `AbsoluteVote` also treats 0 as a valid choice.
- `getNumberOfChoices()` returns `numOfChoices` which is currently `numOfChoices` - 1, as the 0 option is not included.
- `isAbstainAllow()` is hardcoded to return `true`. While one can vote 0 in the current implementation, this does not necessarily mean to abstain as there may be a valid choice at position 0.
- Depending on what `ProposalExecuteInterface(tmpProposal.callbacks).executeProposal(_proposalId,int(cnt))` does in case `choice 0` wins, e.g. if there is a choice 0, the call completes successfully or not.
- A generic scheme cannot execute a choice 0 if it won the proposal.

Additionally, contract `AbsoluteVote` has a state variable `MAX_NUM_OF_CHOICES` which seems to represent the maximum number of choices a vote can have.

- In `propose()` and `internalVote()` some `require()` clauses imply that the count is made up to and including zero, meaning 0 is also already a valid proposal:

```
require(_numOfChoices > 0 && _numOfChoices <= MAX_NUM_OF_CHOICES);
...
proposal.numOfChoices = _numOfChoices;
...
require(_vote <= proposal.numOfChoices);
```

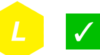
- In `_execute()` we can observe execution from zero up to and including the number of choices:

```
for(uint cnt = 0; cnt <= proposal.numOfChoices; cnt++) {
```

- However in `QuorumVote`, which inherits from `AbsoluteVote` we see in `_execute()` that there are only checks for proposals starting from 1, but `_vote()` allows one to also vote for choice 0:

```
for (uint cnt = 1; cnt<=proposal.numOfChoices; cnt++) {
```

Fixed: DAOStack fixed the issue by consistently implementing 0 as a valid proposal choice. To further aid consistency, the code was documented to make explicit the assumption that the abstain vote is excluded from `MAX_NUM_OF_CHOICES`.

Minor code duplication in Controller 

Function unregisterScheme defined in the Controller contract implements its own logic to check if the scheme to be unregistered has been previously registered.

```
function unregisterScheme( address _scheme,address _avatar)
    ...
    {
        //check if the scheme is registered
        if (schemes[_scheme].permissions&bytes4(1) == bytes4(0)) {
            return false;
        }
        ...
    }
```

However, this logic is already implemented in the internal _isSchemeRegistered function. Thus unregisterScheme can make an internal function call instead of duplicating code.

Fixed: DAOSTACK removed the code duplication and instead calls the internal function.

Unexpected results when a proposal does not exist

Function balanceOfStakingToken defined in contract VotingMachineCallbacks takes a StandardToken address and a proposalId and returns the balance of the proposal's avatar for that token.

However, there is no check that the proposal exists in the proposalsInfo mapping. If that is the case then the following line:

```
return _stakingToken.balanceOf(address(avatar));
```

will return the balance of the ZERO address, which is the number of burned tokens. This can give a false information to the callee of balanceOfStakingToken.

Fixed: DAOSTACK added a check if the proposal exists. For non-existing proposals this function will now return 0.

Missing check for zero address

Function setParameters in contract SimpleICO creates a new ICO configuration based on the parameters it receives as input. Two of those parameters are the addresses of the ICO admin and beneficiary. Since these are critical roles DAOSTACK could consider adding a require statement checking that the addresses are not mistakenly set to the zero address.

Fixed: DAOSTACK removed the SimpleICO contract.

Rounding issues in schemes

The Auction4Reputation contract suffers from two rounding issues:

- I: Rewards really distributed are smaller or equal to the rewards initially intended to be distributed
- II: More auctions can be created than the _numberOfAuctions

I: Rewards really distributed are smaller or equal to the rewards initially intended to be distributed
Checks in initialize() are insufficient to ensure correct execution:

```
require(_redeemEnableTime >= _auctionsEndTime)
require((_auctionsEndTime.sub(_auctionsStartTime)).div(_numberOfAuctions) > 0)
```

However, the division in `auctionReputationReward = _reputationReward / _numberOfAuctions;` truncates the result and rounds it down to the nearest integer. Resulting in the fact that: `actual rewards distributed is <= total rewards to be distributed`. There will be a 0 `reputationReward` to be distributed per auction in case `reputationReward < numberOfAuctions`.

II: More auctions can be created than the `_numberOfAuctions`

While the number of auctions is set upon creation by the owner, there may be more auctions due to rounding issues. During the execution of the constructor the `auctionPeriod` is set to: `auctionPeriod = (_auctionsEndTime.sub(_auctionsStartTime)).div(_numberOfAuctions);` While `SafeMath` is used, the operation is performed on `uint` types and the result of the division will be truncated if necessary. Rounding errors will be encountered when the values `_auctionsStartTime`, `_auctionsEndTime` and `_numberOfAuctions` have not been carefully chosen, as there are no checks to enforce a specific value range.

When bidding for an auction by calling `bid()`, the `auctionId` is calculated as follows: `auctionId = (now - auctionsStartTime)/ auctionPeriod;`. As `auctionPeriod` may have been truncated when it was calculated, it may be smaller than expected as it lacks the fractional part. The result of this division is consequently higher than expected, which might result in: `auctionId` > `numberOfAuction`.

A minimal example demonstrates the vulnerability:

```
_reputationReward = 1000
_auctionsStartTime = 1225
_auctionsEndTime = 2123
numberOfAuctions = 100
```

After initialization, we have:

```
auctionPeriod = (2123-1125)/100 = 9.98 = 9
auctionReputationReward = (1000/100) = 10
reputationRewardLeft = 1000
```

Edge cases for bids:

```
bidding at the start:
1225-1225 / 9 = 0 -> Period 0
bidding at the very end:
2123-1225 / 9 = 898  / 9 = 110.88 -> Period 110
```

Resulting in a total of 110 possible periods. Thus the `auctionReputationReward` can be claimed 110 times, however only until the 100th time the `auctionReputationReward` has been payed out as later transactions will `revert` due to insufficient reputation remaining. Consequently legitimate contributors can't redeem their reputation.

Further rounding risk is depending on the correct functionality of `RealMath`:

```
int256 repRelation = int216(bid).toReal().mul(int216(auctionReputationReward).
    toReal());
reputation = uint256(repRelation.div(int216(auction.totalBid).toReal()).
    fromReal());
```

Fixed: DAOSTACK resolved this by removing all instances where the result of a division may be truncated. The duration of a period is not calculated anymore but passed as an argument, which removes the loss of precision due to truncation. Argument `_reputationReward`, the total reputation to be distributed, was replaced by `_auctionReputationReward`, the reputation to be distributed per auction which removes another source of rounding errors.

Uncatched exception when making a call proposal ✓ Fixed

The universal scheme `GenericScheme` allows users to propose a call on external contracts. The `proposeCall` function first fetches the scheme parameters which are registered at the organization's `controller` via the interface function `getParametersFromController(_avatar)`.

However, if the organization of the specified `avatar` has not registered the `GenericCall` scheme then the following call will return a Parameters `struct` with default values for the fields:

```
Parameters memory params = parameters[getParametersFromController(_avatar)];
```

This will cause the execution flow to throw an exception when calling a function on the zero address:

```
    IntVoteInterface intVote = params.intVote;
    bytes32 proposalId = intVote.propose(2, params.voteParams,msg.sender,
        _avatar);
```

CHAINSECURITY recommends adding a check whether the organization has registered the scheme so that the transaction fails gracefully if the condition is not met.

Fixed: DAOSTACK added a `require` statement to the `getParametersFromController` function which ensures that the scheme is registered or the transaction reverts.

Inefficient struct storage

There are several cases throughout the codebase where reordering the fields in a `struct` can reduce gas costs. Tightly packing smaller data types into one word allows to save significant gas costs.

`struct Proposal` can save one full storage slot by placing `bool open` right after `address owner`. Similar reorderings can be done in the `SchemeProposal struct` of the `SchemeRegistrar` contract and others.

```
struct Proposal {
    address owner; // the proposal's owner
    bytes32 organizationId; // the organization Id
    address callbacks;
    uint numOfChoices;
    bytes32 paramsHash; // the hash of the parameters of the proposal
    uint totalVotes;
    mapping(uint=>uint) votes;
    mapping(address=>Voter) voters;
    bool open; // voting open flag
}
```

Another way of reducing gas consumption is using optimal datatypes for `struct` fields. An example instance is `struct Parameters`:

```
struct Parameters {
    uint precReq; // how many percentages required for the proposal to be
        passed
    bool allowOwner; // does this proposal has an owner who has owner rights?
}
```

Percentage required can only be values between 1 and 100, so `uint256` is unnecessary. Usage of a smaller datatype may be considered so that the field `bool allowOwner` fits into the same storage slot.

The field `proposalType` of the `struct GCProposal` indicates whether the proposal wants to add or remove a global constraint, however takes a full storage slot to do so. This information can be easily stored in a `bool` field. The boolean field `proposalType` can then be placed after `address gc` in the struct, saving one full storage slot by allowing the compiler to place them together and hence reducing overall gas costs. Similar optimizations can be done in the `UpgradeProposal` struct of the `UpgradeScheme` contract and in the `SchemeProposal` struct of the `SchemeRegistrar` contract.

Fixed: DAOSTACK optimized the inefficient `struct`s.

Unchecked return value ✓ Fixed

There are several examples in the codebase where an external function is invoked but its return value is not checked before proceeding with the execution.

Firstly, if a choice of a proposal in `AbsoluteVote` reaches the threshold, `executeProposal` of the callback address is invoked. Given the interface this function is supposed to return a `boolean`. In the current implementation of `AbsoluteVote` this return value is not checked and the function returns `true` hardcoded. DAOSTACK

is advised to consider the implications and handling of a situation where the execution of the choice fails.

Another instance is when a vote on a vesting proposal is successful and the corresponding voting machine calls function executeProposal on the VestingScheme contract. As a part of its implementation the function has to mint the number of vested tokens so that the VestingScheme contract can transfer them to the beneficiary of the vesting when the conditions for that are met. This is done through a call to the controller's mintTokens function:

```
controller.mintTokens(tokensToMint, this,avatar);
```

This function returns a boolean value which indicates the success of the transaction. However there is no check that the return value is true and execution of executeProposal continues regardless. Therefore even if the operation fails the vesting will be created bu the beneficiary will not be able to collect any tokens since they have not been minted to the VestingScheme contract. Even worse, the beneficiary of the failed vesting creation may collect tokens allocated for a different user since all recipients get their tokens transferred from the contract's balance.

Fixed: DAOSTACK solved the problem in the executeProposal function defined in the AbsoluteVote contract by removing the hardcoded **return** true and replacing it with the actual return value of the call to the ProposalExecuteInterface().

A **require** statement has been added to the call to controller.mintTokens() in the VestingScheme contract to ensure the call was successful.

Strong incentives for delayed bidding ✓ Acknowledged

By design, a reputation auction may be divided into multiple periods, each auctioning off an equal part of the total reputation. The reputation to be distributed is divided proportionally to the overall contribution during the period. Consequently users do not know in advance how much reputation they will obtain, as this largely depends on the contribution of other parties. Each period will have a different price.

Hence, there is no incentive to contribute early within a period. All rational actors would only commit very close to the end of a phase, only when favorable conditions exist and more information is available as there is a strong incentive to maximize ones reputation in the system. This might be done by crafting transactions which first check that the current block's timestamp is close the end of the period and the rate is acceptable before omitting.

On a related note, a malicious miner may choose not to include any related transactions after his own even though there might be transactions with higher fees trying to get a bid in. Note that a miner may increase the **block**.timestamp up to 15 seconds into the future[7].

Acknowledged: DAOSTACK is aware that there is an incentive for delayed bidding. A minimum auction period of 15 seconds is now enforced.

Redundant operation when burning reputation ✓ Fixed

The owner of the Reputation contract is able to burn a certain value of a user's reputation through the burn function. The function is responsible for reducing both the user's balance and the total supply.

```
    if (curTotalSupply < amountBurned) {
        amountBurned = curTotalSupply;
    }
    uint previousBalanceFrom = balanceOf(_owner);
    if (previousBalanceFrom < amountBurned) {
        amountBurned = previousBalanceFrom;
    }
```

In case the value to be burned is higher than the total supply or the particular user's balance, then the value gets truncated and leaves the user with no reputation. The first operation is checking if the value is larger than the total supply. This is redundant since in that case the burn amount will get truncated to the user's balance in the next step.

[7]https://consensys.github.io/smart-contract-best-practices/recommendations/#timestamp-dependence

Fixed: DAOSTACK removed the redundant check if the value to be burned is higher than the current total supply.

Compilation with experimental `pragma 0.5.0` fails ✓ Fixed

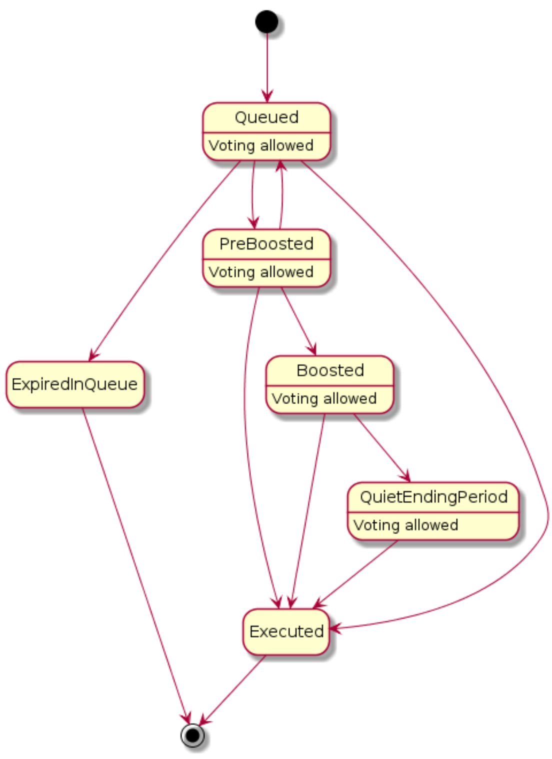
Due to missing external visibility modifiers, compilation with experimental `pragma 0.5.0` fails. Fallback functions in contracts `Avatar` and `SimpleICO` should be defined as `external`.

Fixed: All contracts now compile with `solc 0.5.2`.

High complexity of execution ✓ Acknowledged

The `GenesisProtocolLogic._execute()` function has a high cyclomatic complexity which is 17. The complex methods might have some unintended paths in the code, which could result in security issues. More complex methods tend to have less readability and more maintenance resources required.

Regarding the code flow, below is the state chart diagram for a proposal.



Acknowledged: DAOSTACK acknowledges the issue.

Imprecise estimation of block numbers ✓ Acknowledged

The `GenesisProtocolLogic.executeBoosted()` function ensures that the `expirationCallBounty` is given to the transaction initiator who executes a boosted proposal.

As per specification, a transaction initiator will receive $(1+t)\%$ of the upstake held on that proposal, where t is the number of blocks passing since the boosting-qualification block.

In the `executeBoosted` function this is implemented as below:

```
225 uint256 expirationCallBountyPercentage =
226 // solhint-disable-next-line not-rely-on-time
227 (1 + now.sub(proposal.currentBoostedVotePeriodLimit + proposal.times[1]).div
        (15));
```

GenesisProtocolLogic.sol

This code assumes that all blocks are generated within exact 15 second intervals. However, this varies[8]. Hence, the calculated `expirationCallBountyPercentage` is not guaranteed, which, if tolerated, should be documented clearly.

Acknowledged: DAOSTACK acknowledged that this will be documented clearly to let the users know that the calculation is not guaranteed.

Unused event `RefreshReputation` ✓ Fixed

The `RefreshReputation` event definition is present in the `AbsoluteVote` contract. However, it is not being used anywhere in the code.

Fixed: DAOSTACK has fixed the code and removed the unused event.

Used `ERC20` instead of `IERC20` ✓ Fixed

DAOSTACK is using `Open Zeppelin 2.1.0-rc.2`, which has different implementation for `ERC20` from the version `1.12.0`. The `ERC20` is full implementation of `ERC20` standard and `IERC20` is an interface. There are some contracts which are using `ERC20` to only use interface definition, it does not require full `ERC20` implementation. Hence, it would cost more gas in the deployment of contract and contains dead code. Following are the contracts using `ERC20` only for the interface purposes:

- infra/votingMachines/GenesisProtocol.sol
- infra/votingMachines/GenesisProtocolLogic.sol
- infra/votingMachines/VotingMachineCallbacksInterface.sol
- arc/controller/Avatar.sol
- arc/controller/Controller.sol
- arc/controller/ControllerInterface.sol
- arc/controller/UController.sol
- arc/globalConstraints/TokenCapGC.sol
- arc/schemes/Auction4Reputation.sol
- arc/schemes/LockingToken4Reputation.sol
- arc/universalSchemes/ContributionReward.sol
- arc/universalSchemes/OrganizationRegister.sol
- arc/universalSchemes/VestingScheme.sol
- arc/utils/Redeemer.sol
- arc/votingMachines/VotingMachineCallbacks.sol

CHAINSECURITY recommends using `IERC20` interface, instead of `ERC20` to all places where only method interface is required.

Fixed: DAOSTACK has updated the project to use `Open Zeppelin 2.1.1` version. DAOSTACK has fixed all the above listed files and uses functions using `IERC20` interface. This reduces the contract deployment cost.

Unused imports ✓ Fixed

There are some unused `ERC20` import statements present in the following contracts:

- `UniversalScheme.sol`
- `UniversalSchemeInterface.sol`

[8] https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule

Fixed: DAOSTACK has fixed the issue by removing the unused import statements from these contracts.

Type of argument `reputationReward` unclear L ✓ Fixed

The `initialize` function of the `FixedReputationAllocation` contract takes argument `_reputationReward` of type `uint256` which sets the total reputation this contract will reward. Once all beneficiaries have been added, the reward per beneficiary is calculated as below:

```
97 beneficiaryReward = uint256(int216(reputationReward).div(int256(
      numberOfBeneficiaries)).fromReal());
```

FixedReputationAllocation.sol

The calculation implicitly assumes the variable `reputationReward` is already in the `RealMath` format. This may be fine however should be clearly documented in the `initialize` function, as this will easily lead to unintentional mistakes and lead to unexpected behavior. A more fail-safe version regarding user interaction with the contract may be to take the normal value as argument and convert it using the `toReal()` function.

Fixed: DAOSTACK is not converting `reputationReward` variable using `RealMath` library. Hence normal values for `_reputationReward` argument will be allowed in `initialize` function.

Wasteful conversion

In the `GenesisProtocolLogic` contract, the `setParameters` function contains the following loop:

```
418     int alpha = int216(_params[4]).fraction(int216(1000));
419     //set a limit for power for a given alpha to prevent overflow
420     uint256 limitExponent = 172;//for alpha less or equal 2
421     uint256 j = 2;
422     for (uint256 i = 2; i < 16; i = i*2) {
423         if ((uint(alpha.fromReal()) > i) && (uint(alpha.fromReal()) <= i*2)) {
424             limitExponent = limitExponent/j;
425             break;
426         }
427         j++;
428     }
429
```

GenesisProtocolLogic.sol

The variable `alpha` is never written to throughout the loop, however it is converted twice in each iteration in the if clause. Additionally, the conversion of `alpha` to a `RealMath int` before the loop does not make sense as it is then only accessed as `alpha.fromReal()`. At the very least, it looks like the conversions in every single loop iteration can be avoided.

Fixed: DAOSTACK has removed `alpha` variable and not converting to other datatypes. This removes unnecessary conversion and consumes less gas.

Recommendations / Suggestions

The contracts often use the `uint` type as it defaults to a full-word unsigned integer. A declaration could be made such that it explicitly highlights the size e.g. `uint256`

Parameter @param for `proposeChangeUpgradingScheme()` in the `UpgradeScheme` contract is described with `@param _params ???`.

`event BeneficiaryAddressAdded(address _beneficiary);` in the `FixedReputationAllocation` contract contains an address parameter which is not indexed. Indexing the parameter would enable to easily search logs for a given address.

Two contracts of the `universalSchemes`, `ContributionReward` and `GlobalConstraintRegistrar` contain TODO comments. DAOSTACK is advised to resolve remaining TODOs.

The test suite contains tests to ensure that calls to a few hardcoded `internal` function revert. By design of the Solidity this is impossible. Even if this would turn out to be possible due to an unknown major security flaw in the compiler, it will not occur through normal function calls.

Under the assumption of a correct and trusted truffle and solc setup, a more robust way to test this would be to verify that the `ABI` only includes the expected external/public functions.

Contract `Reputation` defines a `public` state variable `creationBlock` which stores the block number when the contract was created. However, its value is neither referred to internally, nor accessed by any external contract through the automatically generated `getter` function.

The `SimpleICO` contract has a `donate` function, which is actually used to purchase tokens as a obvious comment within the function states:

```
// Compute how much tokens to buy:
```

Hence, such naming is misleading users and should be revisited.

Parameter `address owner` in the `burn` function of the `Reputation` contract may cause confusion, although it is described in the parameter description comment just above. In an `onlyOwner` function, `owner` is usually associated with the owner/administrator of the contract, not the owner of the tokens to be burned as it is the case here. DAOSTACK may consider to rename it to a more appropriate name like `address user`.

`MirrorContractICO` inherits `Destructible`, however for `MirrorContractICO` contracts having been deployed in `start()` of `SimpleICO` the owner(`SimpleICO` contract) features no functionality to invoke the `destruct()` function nor any other functionality inherited from `Destructible`. Consequently the inheritance of `Destructible` is not used and the unnecessary part of the code just wastes gas.

Note that it is a good thing that MirrorContract cannot simply be destructed during an ongoing ICO as this would lead to multiple problems, including ETH sent to this address after the contract has been selfdestructed will be locked forever.

In the `GenesisProtocolLogic` contract there are some places where `require()` error messages are used inconsistently:

```
233     require(stakingToken.transfer(msg.sender, expirationCallBounty),"
            transfer␣to␣msg.sender␣failed");
```

GenesisProtocolLogic.sol

```
304     require(stakingToken.transfer(_beneficiary, rewards[0]));
```

GenesisProtocolLogic.sol

CHAINSECURITY recommends to use `require()` with error messages at required places only. However, for the same kind of calls, their usage should be consistent.

✓ There is a function which has its visibility set to `public` and accepts an array as an argument.

```
402         function setParameters(
403                 uint[11] memory _params, //use array here due to stack too
                        deep issue.
404                 address _voteOnBehalf
405         )
406         public
407         returns(bytes32)
```

GenesisProtocolLogic.sol

ChainSecurity recommends changing the visibility of these functions to **external**[9]. This makes the function more efficient in terms of gas costs.

✓ Code indentation is not consistent in `AbsoluteVote` contract.

```
43   mapping(bytes32=>Parameters) public parameters;  // A mapping from
         hashes to parameters
44   mapping(bytes32=>Proposal) public proposals; // Mapping from the ID of
         the proposal to the proposal itself.
45   mapping(bytes32        => address     ) public organizations;
```

AbsoluteVote.sol

ChainSecurity recommends fixing the code indentation to have consistency.

✓ The `internalVote()` function present in `GenesisProtocolLogic` contract has a `if` statement which is bit complex and not readable easily.

```
686 if (((proposal.state == ProposalState.Boosted) &&
687 // solhint-disable-next-line not-rely-on-time
688     ((now - proposal.times[1]) >= (params.boostedVotePeriodLimit - params.
            quietEndingPeriod))||
689     (proposal.state == ProposalState.QuietEndingPeriod))) {
```

GenesisProtocolLogic.sol

When the `if` statement is normalized, it is checking the condition below: As the && and || short-circuiting operator is used in conjunction readability of code is decreased. A reader has to take short-circuiting operator properties into consideration to understand the actual logic of the `if` statement as the logic is not straight forward.

```
if (
    proposal.state == ProposalState.Boosted &&
    (now - proposal.times[1]) >= (params.boostedVotePeriodLimit - params.
        quietEndingPeriod) ||
    proposal.state == ProposalState.QuietEndingPeriod
)
```

✓ The `GenesisProtocol` contract defines the `ETH_SIGN_PREFIX` constant, which is used to convert the signature to Ethereum signature.

```
25     string public constant ETH_SIGN_PREFIX= "\x19Ethereum␣Signed␣Message:\
           n32";
```

GenesisProtocol.sol

However, the OpenZeppelin also has the `ECDSA` library which supports this functionality and could be imported by DAOStack.

[9] https://solidity.readthedocs.io/en/v0.4.24/contracts.html#visibility-and-getters

- ☑ `ContributionRewards` features a new function `validateProposalParams(uint[5] memory _rewards, int256 _reputationChange)`. Contrary to all other functions and despite the complex input parameter `_rewards`, this one does not feature a docstring documenting the expected input parameters and return values. ChainSecurity recommends to add documentation for better readability and to keep consistency throughout the codebase.
- ☑ `DAOToken` is commented with `@dev ERC20 compatible token`. It is a mintable, destructible, burnable token. Note that destructible is ambiguous, there is no reachable `SELFDESTRUCT`.

Post-audit comment: DAOStack has fixed some of the issues above and is aware of all the implications of those points which were not addressed. Given this awareness, DAOStack has to perform no more code changes with regards to these recommendations.

Disclaimer

UPON REQUEST BY DAOSTACK, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..