

Security Audit

of DIGIX's Smart Contracts

December 10, 2018









Produced for



by



Table Of Content

Foreword	1
Executive Summary	1
Scope	2
1. Included in the scope	2
2. Out of scope	2
Audit Overview	3
1. Scope of the Audit	3
2. Depth of Audit	4
3. Terminology	5
Limitations	6
System Overview	7
1. Voting overview	7
2. Extra voting features	7
Best Practices in DIGIX's project	10
1. Hard Requirements	10
2. Soft Requirements	10
Security Issues	11
1. Roles	11
1.1 Usage of extcodesize to verify EOAs  	12
1.2 Use of tx.origin for authorization  	13
1.3 Unrestricted calls to readProposalDraftVotingTime  	13
2. Timing and Quarters	13
2.1 Start of first quarter can be in the past  	14
3. Quarter: Lockup and Main phase	14

4.	Proposal phases			15
4.1	Preproposals can be overwritten	M	✓ Fixed	21
4.2	Draft voting passes with minority	L	✓ Acknowledged	21
5.	Special Proposals			21
5.1	Unchecked arithmetic operations	L	✓ Fixed	23
5.2	Special proposals details can be silently updated	M	✓ Fixed	23
5.3	Array underflows	M	✓ Fixed	24
6.	Voting			24
6.1	PRL can unpause stopped proposal	M	✓ Fixed	27
7.	Reputation			27
8.	Rewards			28
8.1	Old DAO can be funded	L	✓ Fixed	29
8.2	Wrong parameter description	L	✓ Fixed	29
	Trust Issues			31
1.	Remarks on the migration process	H	✓ Acknowledged	31
1.1	Malicious fund transfer		✓ Acknowledged	31
1.2	Information migration		✓ Acknowledged	31
1.3	Technical competence when migrating		✓ Acknowledged	31
	Design Issues			32
1.	Special Proposals may fail unintentionally	M	✓ Fixed	32
2.	Preliminary iteration	L	✓ Fixed	32
3.	DaoFundingManager can receive arbitrary funds	L	✓ Fixed	33

4.	Deprecated <code>constant</code> keyword			33
5.	Assigning to function arguments			33
6.	Duplicate code			33
7.	Fallback function is public			33
8.	Inefficient fund tracking			34
9.	Non-indexed events			34
10.	Unnecessary loop iterations			34
11.	Duplicate funding checks for preproposals			34
12.	Unnecessary calculations			35
13.	Broad function visibility			35
14.	Closed proposals can be reclosed			35
15.	Failing test cases in <code>DaoRewardsManager</code>			36
16.	Inefficient DAO storage			37
17.	Missing input validation			38
18.	Old compiler version			38
19.	Underspecified ownership structure			38
20.	Suboptimal <code>struct Proposal</code>			39
21.	TODOs in code			39
	Recommendations / Suggestions			40

Disclaimer	41
----------------------	----

Foreword

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

– ChainSecurity

Executive Summary

The DIGIX smart contracts have been analyzed under different aspects, with a variety of customized and publicly available tools for automated security analysis of Ethereum smart contracts, as well as with expert manual review. The audit was performed within a clearly defined scope and strived to verify a functional specification developed by DIGIX and CHAINSECURITY.

Overall, we found that DIGIX employs good coding practices and succeeded in building a complex DAO system with clean code and modularity. However, CHAINSECURITY managed to uncover several high and medium severity security issues which need to be addressed and a wide range of design issues that when fixed can help improve the system. More so, DIGIX needs to significantly extend the system's overall documentation for future users and developers.

Scope

DIGIX requested a precisely scoped audit, meant to assess the technical foundation of DIGIX's project in its current state. To define this scope, CHAINSECURITY listed potential points of failure and agreed with DIGIX upon them.

Issues that have been encountered while verifying this specification have been listed, even when they were not explicitly mentioned in the specification. However, this list should not be considered exhaustive with respect to the security of DIGIX's smart contracts. CHAINSECURITY strove to verify the points listed here, to provide a report whose contents could serve as potential guidelines in the future.

The main specification sections are listed below and a detailed description of the reviewed properties can be found in the issues section.

Included in the scope

- 1. Roles
- 2. Timing and quarters
- 3. Quarter: Lockup and Main phase
- 4. Proposal phases
- 5. Special proposals
- 6. Voting
- 7. Reputation
- 8. Rewards

Out of scope

- All non-smart contract components (e.g. user interfaces of dApps)
- All smart contracts whose hashes are not included in the audit overview

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 August 10, 2018¹ and updated versions on December 6, 2018²:

¹<https://github.com/DigixGlobal/dao-contracts/tree/81ca6846d8965effc0c652274401808bb13f5869>, <https://github.com/DigixGlobal/cacp-contracts/tree/2a3cd3d2d1cfd6c6d97572b426d096a86d23961c>

²<https://github.com/DigixGlobal/dao-contracts/tree/84ac4a5f3271925f8ad9586c5987f4283eb92ab0>, <https://github.com/DigixGlobal/cacp-contracts/tree/3ec2e84d23a6e6ff178110eaa50652f168032115>

File	SHA-256 checksum
cacp/ResolverClient.sol	d240b477582501f4c8b198ed91267758ddc8b20fc0c237e99492de0d9678876f
cacp/ContractResolver.sol	73bf758f0c52b5c1e5fc22bb02db8f443f095b91e662d0d380a12372b331a484
dao/lib/MathHelper.sol	73d8dad82cd8ef760b17ef85ab9d1366be723039414627fbc5b3886e895f8820
dao/lib/DaoStructs.sol	4c93d3d8aadf8e15c1cfce3951c53cfadb5740b770cabddf5500a3785b39d538
dao/lib/DaoIntermediateStructs.sol	af28f90014778ddd6b54c45aff51ff63d08dc67911c208fc682d2df8012a38bb
dao/interactive/DaoInformation.sol	f91deb1f991ad08be69dd3a682d40f4bc61d6a1ee2cb367b68d7c49a1fd6d3ec
dao/interactive/DaoSpecialVotingClaims.sol	b5448c19561e3d29a85b3fdbb4f0462412b9a9fe82c7a4ea7e08ec3852b1a76e
dao/interactive/DaoSpecialProposal.sol	ff4586d1bbe4084b8cf81de7613b265eacd192c82d173963d9342f56431ea3fe
dao/interactive/DaoVoting.sol	d0a093c7908878e2b2829b123feb5206481052f3cc9b99b122db40fb50ef1e04
dao/interactive/DaoRewardsManager.sol	e825a446ae1b16cbab565f5edb3f4321250407ae5cec2a5cfd4ad9df10a8fe8b
dao/interactive/DaoVotingClaims.sol	8a1cd5fb2c2b41f99f2834441de635f3e25f352136225cf68f582a7a6a9e2bba
dao/interactive/DaoIdentity.sol	0469ebae87d50a42a0d72129cf1afae7e180a2c6697cad1c75718ab025ade489
dao/interactive/Dao.sol	f4f3d1047c8dd926f53df924234fd829f9e30230c78162728916b12ed0335756
dao/interactive/DaoStakeLocking.sol	d47cd893112350145578e0ba4245ad192506287806ceadf973c0e6bb28be6613
dao/interactive/DaoWhitelisting.sol	cac952d3bcce904637e1793545c1a6b9d66f2ec237249aeab1553dbcc848787
dao/interactive/DaoRewardsManagerExtras.sol	7e86a40664d06432731b455ed2647294af9ef88fe960815c0fcba430a9e9cf96
dao/interactive/DaoFundingManager.sol	ad9273b4fdd8e6520565a08a7505ed50ce2b55483362daf0f13ab91caa219ecc
dao/storage/DaoConfigsStorage.sol	092c82af12523c39f3cfaa82b45c72dc06c45f8424b5ba55816a29fdb8099994
dao/storage/DaoUpgradeStorage.sol	c4e9718e0703a15adb675f34abf3aece77939b34627ee81233abe86f1af1529b
dao/storage/DaoSpecialStorage.sol	3a9bf33e2e1d51cac42580e6f1c84b45ac8ffe9aee7697e32565c779c63041df
dao/storage/DaoStakeStorage.sol	6101e0ee69aa077cf434c9710485144230e9c0b650908ddcacfe491ce416c481
dao/storage/DaoStorage.sol	43e58ebe259c3460c5b03f9909fe378a0b6cff455590610773c64c6169be5e7f
dao/storage/DaoWhitelistingStorage.sol	9dfe80966489e3c9804a34737458fab887230b5bb72769dbbf4c0c28f587e6b5
dao/storage/DaoPointsStorage.sol	f2c6479b74d0cb1c403488908c00a26a8bbfb2693e805b625761a17b1797321e
dao/storage/DaoIdentityStorage.sol	f1122ecca3608a70cb62a6550c265e960ec2bd1db84842e45e7f92860c4896bb
dao/storage/DaoRewardsStorage.sol	f205e1de0ec41f2638db9e5cb9a6a0fbb29366dfd93ac867e7507bde7baf24ad
dao/storage/IntermediateResultsStorage.sol	c620f6cb9c604e451abdcb6a8995e6d561689c42bfd8940433c20a5f7eed476
dao/storage/DaoProposalCounterStorage.sol	a3e629878d4c271da7ba3e32d64efec33e3e4059e5f6e68eed1a034cc9098a5d
dao/service/DaoCalculatorService.sol	20303f14f2074675d7b709df1709876a02ba96ca3641211b237435a70af51bf6
dao/service/DaoListingService.sol	ee3f8c79ff35235b9c2283a417e7d8a9425f6604873a611d197f8b790b3a714f
dao/common/DaoServiceCommon.sol	2c2e66e667be2836e32fab6168da66085578719e8347baa809ffb1c12ffd928f
dao/common/DaoCommonMini.sol	50bd26b443967a39e08e658d40b11b89aff3010f5815a6b7dcb49bb059f2c4ba
dao/common/DaoRewardsManagerCommon.sol	70d560aefd0dd408347e556a542838365cf0aa9e51e7d1528ad6eee2254ea8f7
dao/common/DaoWhitelistingCommon.sol	db697f645b6ff3d4ff61095c14b3550554cf98efd176b7060ab52d50d082a6a2
dao/common/DaoConstants.sol	8544adbbba75e36d484499f837d3902c15328e4e4a9866d60edcd003aadb59369
dao/common/IdentityCommon.sol	75ec4ab634877e5f339a094d097c9c21a6d851c6c4978905528f00763622e81a
dao/common/DaoCommon.sol	e63bbd557eeb8dd446b006ec2169913275e8387f154ab5a8347ead6a6edd8ff4

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 methodology³).

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:


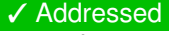
-  Low: can be considered as less important
-  Medium: should be fixed
-  High: we strongly suggest to fix it before release
-  Critical: needs to be fixed before release

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

LIKELIHOOD	IMPACT		
	High	Medium	Low
High			
Medium			
Low			

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.

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

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

DIGIX realized a feature rich DAO to vote on user submitted proposals, including the management of corresponding funds in ETH. The DIGIX DAO is linked to the gold-backed DIGIX (DGX) and the DIGIX Dao (DGD) tokens. These token contracts were not included in the scope of the audit.

Voting overview

The core functionality provided by the DIGIX Dao are the voting procedures. The participants can vote on two kinds of proposals. These are

- Common proposals, initiated by other participants
- Special proposals, initiated by founders to change DAO configuration parameters

A proposal can consist of multiple milestones and an initial collateral which is required to open a proposal. Each time the proposer accomplishes a milestone and passes an interim voting round, a new part of the initially proposed funding is released. If the last milestone and voting round on the proposal is successfully accomplished the proposer receives his initial collateral and a predefined bonus known as the final reward.

Extra voting features

Active participants receive rewards for contributing to the DAO. Inactivity gets punished with either less or no financial rewards and a loss of reputation or quarter points. To measure the voter's activity DIGIX introduces quarter points and reputation points as measures of quarterly and overall participation.

Each proposer needs to go through a KYC process, validated by a special KYC administrator role. A special Policy-Regulatory-Department (PRL) administrator is able to at will to pause, unpaue or stop a proposal. Proposals that were not finalized by its corresponding proposer in a certain time frame can be closed by founder accounts. When initiating a proposal, the proposer needs to lock up collateral and specify the previously mentioned milestone funding and final reward. The root account which is supposed to be controlled by a multi-signature wallet has the possibility to migrate the DAO and therefore update and change the DAO contract without going through the full voting process.

As already mentioned earlier the DIGIX DAO knows different roles. CHAINSECURITY provides a concise overview in the table below

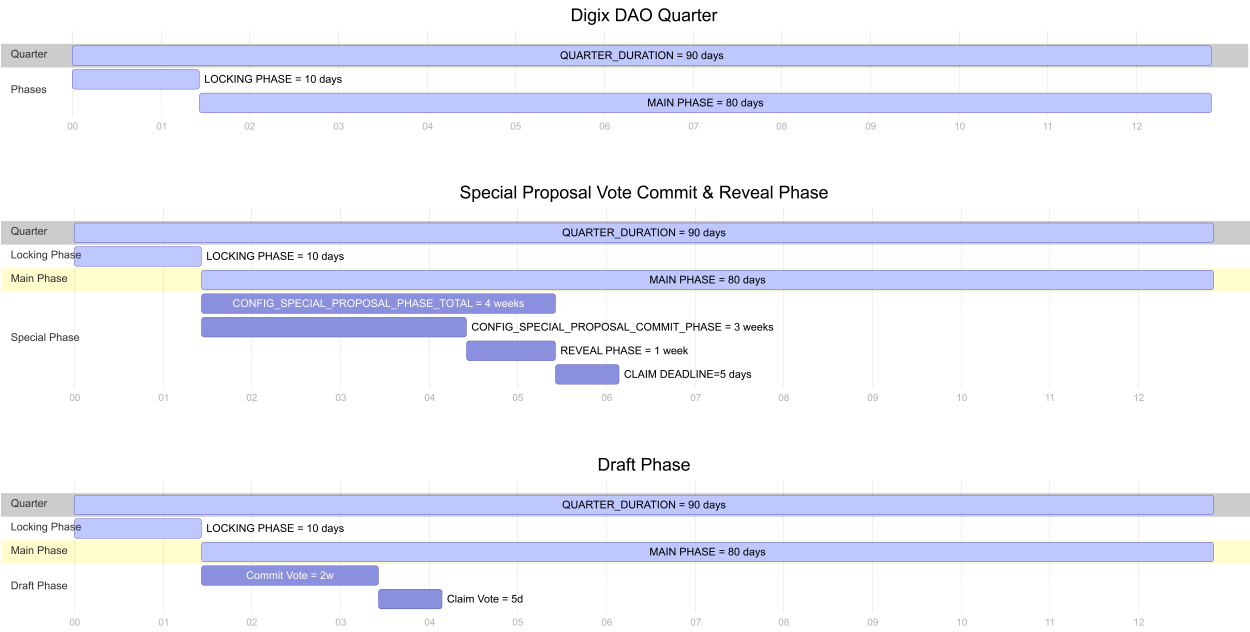
Role	Obtained by	Who	Additionally	Rights
Root	The only predefined role(root) is set as the account which deploys the contracts. After deployment, the account is supposed to transfer its rights to a multi-signature wallet	A multi-signature wallet controlled by DIGIX founders	Can set itself to any role	<ul style="list-style-type: none">• Add and remove accounts from specific roles• Migrate the DAO
Founders	Root granting privileges	DIGIX founders	Can be any other role	<ul style="list-style-type: none">• Close inactive proposals• Start special proposals• Claim special proposal results• Trigger the calculation of global rewards
PRL	Root granting privileges	Independent law firm/DIGIX legal counselor	Can be any other role	<ul style="list-style-type: none">• Update the PRL status of a proposal• Can whitelist a contract address

Role	Obtained by	Who	Additionally	Rights
KYC Admin	Root granting privileges	Independent law firm/DIGIX compliance head	Can be any other role	<ul style="list-style-type: none"> Update the KYC data of a user
Moderator	<ul style="list-style-type: none"> Locking more DGD than <code>CONFIG_MINIMUM_DGD_FOR_MODERATOR</code> Have at least <code>CONFIG_MINIMUM_REPUTATION_FOR_MODERATOR</code> locked reputation stake 	Arbitrary individuals	Is also Participant, can be any other role	<ul style="list-style-type: none"> Endorse preproposals Vote on preproposals
Proposer	<ul style="list-style-type: none"> Be at least participant Pass KYC Submit a proposal 	Arbitrary individuals	Is also Participant, can be any other role	<ul style="list-style-type: none"> Submit preproposal Modify proposals Change fundings Finalize proposals Finish milestones Add proposal documents Claim voting result Claim funding Claim final reward Claim milestone funds
Participant	Locking more DGD than <code>CONFIG_MINIMUM_LOCKED_DGD</code>	Arbitrary individuals	Is Account, can be any other role	<ul style="list-style-type: none"> Submit proposals if KYC approved Vote on proposals Claim DGX Continue participation
Account	Controlling an address	Arbitrary individuals	Can be other any role	<ul style="list-style-type: none"> Lock DGD Withdraw DGD Claim voting result Claim badge

Note, that:

- It is not enforced that the main roles are held by different addresses.
- Accounts can vote on their own proposals.
- Voting power and rewards are proportional to the amount of locked tokens.
- As initially there are no accounts with enough reputation points, accounts can obtain badges. These can be exchanged for enough reputation points to become a Moderator.

The diagrams below provide an overview of some phases in the voting procedure. Light-blue time frames are fixed, dark-blue ones are sliding.



Best Practices in DIGIX'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 DIGIX's project fitted the criterion when the audit started.

Hard Requirements

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

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

- ✓ There are migration scripts.
- ✓ There are tests.
- ✓ The tests are related to the migration scripts and a clear separation is made between the two.
- ✓ The tests are easy to run for CHAINSECURITY, using the documentation provided by DIGIX.
- ✓ The test coverage is available or can be obtained easily.
- ✗ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ✗ 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.
- ✗ There is no dead code.
- ✓ The code is well documented.
- ✗ 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.
- ✓ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ✓ Function are grouped together according either to the Solidity guidelines⁴, 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.

Roles

- ✓ There are six roles: participants, moderators, founders and Policy-Regulatory-Department (PRL), root, and KYC admin

The roles and groups are defined in the constructor of `DaoIdentity` (except for the root). The only preset group is "admins" but the code implements access control(modifiers) for "root" `if_root()`, founder `if_founder()`, PRL `if_prl()` and kyc admin `if_kyc_admin()`. The remaining two role checks, for participants and moderators, only query the current locked DGD and reputation points to check whether an account has the corresponding role.

- ✓ A participant must call either `lockDGD`, `withdrawDGD` or `confirmContinuedParticipation` to participate in a quarter

`DaoCommon.isParticipant(user)== true` means that:

```
(daoRewardsStorage().lastParticipatedQuarter(_user) ==
    currentQuarterIndex())
&& (daoStakeStorage().lockedDGDStake(_user) >= getUintConfig(
    CONFIG_MINIMUM_LOCKED_DGD));
```

To set `daoRewardsStorage().lastParticipatedQuarter(_user)` to `currentQuarterIndex()` one needs to call `updateLastParticipatedQuarter`. This method is only called from the two functions `lockDGDInternal` and `withdrawDGD`, whereas `confirmContinuedParticipation` just calls `lockDGDInternal` with zero DGD. Thus, the specification holds.

- ✓ The minimum lockup amount is more than or equal to `minimumDgdToParticipate` in quarter

This is checked in both functions (`lockDGDInternal`, `withdrawDGD`) by the if-clause `_newInfo.userLockedDGDStake>=getUintConfig(...)` before calling `updateLastParticipatedQuarter`.

- ✓ Moderator: participant (lock-up stake might be different `CONFIG_MINIMUM_DGD_FOR_MODERATOR`) with minimum amount of `ReputationPoints`

To be a moderator (`isModerator`) these conditions need to hold:

```
(daoRewardsStorage().lastParticipatedQuarter(_user) ==
    currentQuarterIndex())
&& (daoStakeStorage().lockedDGDStake(_user) >= getUintConfig(
    CONFIG_MINIMUM_DGD_FOR_MODERATOR))
&& (daoPointsStorage().getReputation(_user) >= getUintConfig(
    CONFIG_MINIMUM_REPUTATION_FOR_MODERATOR));
```

`CONFIG_MINIMUM_DGD_FOR_MODERATOR`, `CONFIG_MINIMUM_REPUTATION_FOR_MODERATOR` are included in both.

- ✓ Founders: addresses representing DIGIX, are set by root

To add an account to any group the `addGroupUser` function needs to be called. The modifier `if_root()` restricts access to

```
require(identity_storage().read_user_role_id(msg.sender)==ROLES_ROOT);
```

and the root is set when initializing the system.

- ✓ PRL: addresses set by root, can pause or stop proposals and whitelist addresses

Setting group members works as described in the previous property verification. A proposal is paused if the boolean below is true:

```
(,,,,,,,,_isPausedOrStopped,)=daoStorage().readProposal(_proposalId);
```

To set the variable, `updateProposalPRL()` needs to be called. This call happens from `updatePRL()` and the whole call chain is guarded by corresponding **requires** and modifiers. Specifically, `if_prl()` checks

```
require(identity_storage().read_user_role_id(msg.sender)==ROLES_PRLS);
```

Thus, this call is limited to accounts in the PRLS role. The PRL can whitelist contracts by calling `setWhitelisted`. The modifier `if_prl` protects the function from being called by any other group than `prl`. `DaoWhiteListing.setWhitelisted` calls `daoWhitelistingStorage().setWhitelisted`. However, to check if a contract is whitelisted `isWhitelisted(address)` is called, which is broken and can be circumvented as described in issue 1.1 below. This issue has been fixed during the audit process.

- ✓ Root: is assigned to the deployer of the `DaoIdentityStorage` contract. Can add/remove users to founder, `prl`, `kyc` admin and root roles

The root role and thus the root group is created in the constructor of the `ContractResolver`. There a call to `init_ac_groups()` in `ACGroups` is made. As shown previously it is only this role that can call `addGroupUser/removeGroupUser` and therefore add/remove other roles. Nonetheless, CHAINSECURITY remarks that the ownership setup is heavily underspecified as raised in design issue 19.

1.1 Usage of `extcodesize` to verify EOAs

DIGIX makes use of the `extcodesize` assembly instruction to check for contract size in several modifiers which are supposed to restrict access to certain information for contract accounts and only allow externally owned accounts (EOAs) to read it. However, such checks are not secure and can be easily circumvented by calling from the constructor of a contract account⁵.

Such checks are exhaustively used in the DIGIX code base and the following parts are affected:

- In CACP: the modifiers `if_contract` and `unless_contract` are broken, although so far these are only used in mocks
- In DAO:
 - `DaoListingService.sol`, `DaoWhitelistingCommon.sol`: the function `isWhitelisted` is broken
 - `DaoCommonMini.sol`: modifier `ifNotContract` is broken

CHAINSECURITY notes that `isWhitelisted` is used in many places:

- `DaoCalculatorService.sol`: in `minimumVotingQuorum`
- `DaoListingService.sol`: in `listProposalsInState` and `listProposalsInStateFrom`
- `DaoSpecialStorage.sol`: in six functions
- `DaoStorage.sol`: in nineteen functions

Additionally `ifNotContract` is used in the `DaoStakeLocking` contract. This means that bots can now stake and participate in the DAO by calling `lockDGD`. Given that all of these checks can be fully circumvented, they can be removed and the restriction policy and its enforcement needs to be reconsidered.

Likelihood: High

Impact: Medium

Fixed: DIGIX now checks `if(msg.sender==tx.origin)` and explicitly passes on this `msg.sender` as an argument to the function they call. Note that if the current sender would not be passed in such manner, then the property would be circumventable.

⁵<https://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-extcodesize-to-check-for-externally-owned-accounts>

1.2 Use of `tx.origin` for authorization

The `ContractResolver` contract makes use of the `origin` statement for authorization in its modifier:

```
modifier if_owner_origin() {
    require(tx.origin == owner);
    -;
}
```

However, the transaction origin can be manipulated easily and should not be used for access control⁶. In this contract, the vulnerable `if_owner_origin` modifier is one of the checks on the execution of important functions such as `init_register_contract`, `register_contract` and `unregister_contract`. Given the previous, this modifier should be changed.

In an iterated review DIGIX removed the `register_contract` and `unregister_contract` functions. However the modifier using `tx.origin` is still present and `tx.origin` itself is still used for authentication in the `DaoStakeLocking` and `DaoWhitelistingCommon` contracts.

Likelihood: Medium

Impact: High

Addressed: DIGIX clarifies the following:

- `if_owner_origin()` is only used in `init_register_contract` which will only be called upon deployment after which `lock_resolver_forever` will be called, locking `ContractResolver` and disabling any more calls to `init_register_contract`.
- DIGIX acknowledges that before `lock_resolver_forever` is called, the deployer might be tricked to call malicious contracts that call `init_register_contract` to register contracts to `ContractResolver`. However, DIGIX states that the migration process will make sure to just use the deployer account to deploy the contracts, and `lock_resolver_forever` right away.
- `tx.origin` in `DaoStakeLocking` and `DaoWhitelistingCommon` is only used to detect whether `msg.sender` is a contract or an EOA.

1.3 Unrestricted calls to `readProposalDraftVotingTime`

Most of the `constant` methods present in `DaoStorage` are protected with the `isWhitelisted` modifier and cannot be called by any other contract.

However, the `readProposalDraftVotingTime` method is allowed to be called from any deployed contract and has no restrictions in place.

Likelihood: Medium

Impact: Low

Fixed: DIGIX added a `require` statement `require(isWhitelisted(msg.sender))` that prevents unauthorized access by non-whitelisted callers.

Timing and Quarters

- ✓ The `conf(A_CONSTANT)` is the value returned by `getUintConfig(A_CONSTANT)` in `DaoCommonMini` (and any contract that inherits it) where the values of `A_CONSTANT` have been defined in the `DaoConstants` contract

The corresponding functions are `getUintConfig(bytes32 _configKey)` and `daoConfigsStorage()`. The `DaoStorage` contract defines the mappings with explicit configuration values

```
mapping (bytes32 => uint256) public uintConfigs;
```

- ✓ No block timestamp issues can lead to unintended behaviour (e.g. no time slips)

⁶<https://solidity.readthedocs.io/en/latest/security-considerations.html#tx-origin>

A lot of the DIGIX logic needs to rely on `block.timestamps`, and `now` is often used. While blocks timestamps can be manipulated, the system can tolerate 15 second drifts^a as the calculations rely on daily accuracy. Hence, CHAINSECURITY does not foresee any issues.

^a<https://consensys.github.io/smart-contract-best-practices/recommendations/#timestamp-manipulation>

- ✓ A period (quarter) is always exactly 90 days = `conf(CONFIG_QUARTER_DURATION)`
 - The variable is set in the `DaoConstants` contract by the field `uint256` `QUARTER_DURATION = 90 days`, guarded by access control. This is used by the functions `getTimeLeftInQuarter`, `timeInQuarter` and `getQuarterIndex` in `DaoCommonMini`, as well as in `calculateAdditionalLockedDGDStake` in the contract `DaoCalculatorService`.
- ✓ The lockup Period is always exactly `conf(CONFIG_LOCKING_PHASE_DURATION)`
 - The function `isLockingPhase` checks if:

```
currentTimeInQuarter () < getUintConfig(CONFIG_LOCKING_PHASE_DURATION)
```

and is used to enforce this property, e.g. in `withdraw`, `lockDGD` and `confirmContinuedParticipation` with corresponding `requires`.

2.1 Start of first quarter can be in the past

The Dao contract's founder role can set the start of the first quarter using `setStartOfFirstQuarter` and can set any `uint256` value to start the quarter from that epoch timestamp. If `_start` is set as timestamp representing some past time, this can cause issues with calculations of the quarter. `setStartOfFirstQuarter` should allow only future timestamp values as arguments for `_start`.

Likelihood: Low

Impact: Medium

DIGIX added `require(_start > 0)`, however this only resolves the issue raised in the suggestions regarding that this function can be called multiple times, if called with `_start = 0`. For this issue, the `require` statement should ensure the argument is a timestamp in the future, for this `_start` must be greater than the current time, `block.timestamp`.

Fixed: DIGIX now correctly checks the start of the quarter.

Quarter: Lockup and Main phase

The Lockup DGD phase are the first `conf (CONFIG_LOCKING_PHASE_DURATION)` seconds of the quarter. It shall hold, that:

- ✓ No accounts other than the participants of the previous quarter or accounts that locked less than `minimumDgdToParticipate` can choose to call `DaoStakeLocking.withdrawDGD` to withdraw the funds (partially or completely) or or leave them untouched
 - Leaving them untouched refers to a call to `DaoStakeLocking.confirmContinuedParticipation`, in case a user locked at least `minimumDgdToParticipate`. This is enforced by:

```
require(_info.userActualLockedDGD > 0);
```
- ✓ `lockedDGDStake` equals sum of DGD left untouched, withdrawn or added in `conf(CONFIG_LOCKING_PHASE_DURATION)` seconds
 - This holds true under the assumption that `lockedDGDStake` does not count stakes that are less than the required minimum.
- ✓ No accounts without DGD become a participant
 - This holds true as the participant status relies on the amount of DGD locked and hence is implied by the previous property.

The main phase lasts from `conf(CONFIG_LOCKING_PHASE_DURATION)` seconds to 90 days. It shall hold, that:

- ✓ No account (including participants, moderators, ...) can withdraw locked DGD in this period

`withdrawDGD` checks for `require(isLockingPhase())`. `isLockingPhase` in turn checks that `currentTimeInQuarter()` is smaller than `getUintConfig(CONFIG_LOCKING_PHASE_DURATION)`. Thus, withdrawals can not be called outside the locking phase.

- ✓ No governance activity (`submitPreproposal`, `modifyProposal`, `changeFundings`, `finalizeProposal`, `finishMilestone`, `addProposalDoc`, `endorseProposal`, `closeProposal`, `voteOnDraft`, `commitVoteOnSpecialProposal`, `revealVoteOnSpecialProposal`, `commitVoteOnProposal`, `revealVoteOnProposal`) is possible outside of this period

This holds true as all of the above functions are guarded by corresponding timing modifiers, specifically `isMainPhase` `ifCommitPhaseSpecial`, `ifRevealPhaseSpecial`, `ifCommitPhase` and `ifRevealPhase`.

- ✓ Participants can top up the locked DGD. But the new `lockedDGDStake` is the sum of the DGD stake during the lockup period and the additional DGD times $(90 \text{ days} - t \text{ seconds}) / (90 \text{ days} - \text{conf}(C_LOCK_PHASE_DUR) \text{ seconds})$

The function `calculateAdditionalLockedDGDStake` performs the correct calculation.

Proposal phases

Endorsement phase:

- ✓ No participant without successful KYC can open pre-proposals

`submitPreproposal` checks `senderCanDoProposerOperations`:

- `require` check: `isMainPhase()` (in `DaoMini`)
- `require` check: `isParticipant(msg.sender)`
- `require` check: `is_kyc_approved` in `DaoIdentityStorage`, which in turn calls `read_kyc_info`. The `kyc` fields can only be written to by the `DaoIdentityStorage` contract.

Hence, in order to submit a proposal the KYC information must have been set by the authorized `DaoIdentityStorage` contract. The property holds.

- ✓ No participant can open a preproposal without a sufficient collateral = `conf(CON_PREPROPOSAL_DEPOSIT)` wei

`submitPreproposal` checks `senderCanDoProposerOperations`:

- `require` check: `isParticipant(msg.sender)`, which checks that `lockedDGDStake(_user)` is bigger or equal to `getUintConfig(CONFIG_MINIMUM_LOCKED_DGD)`.
- Also an explicit check is made whether the deposit was sent along with the actual call to submit the pre-proposal through another `require`.

Hence, it is not possible to successfully call `submitPreproposal` without sending along the right `msg.value`.

- ✓ No malformed proposals can be opened (needs to have the fields: an ipfs hash, milestones + funding requirements, proposer final reward)

Three parameters need to be passed to `submitPreproposal`: `bytes32 _docIpfsHash`, `uint256 [] _milestonesFundings`, and `uint256 _finalReward`

- `ifFundingPossiblechecks` checks if the milestones are fundable
- `checkNonDigixFundings` does the same but for non founders
- `daoStorage().addProposal` adds the `docIpfsHash` by calling `append`. `append` returns a boolean value on whether the appending process was successful or not, however the return value is not checked.

A new check was introduced in the `addProposal` function, which overlaps with checks already in-place in the `append` method but resolves the issue. It is best practice to check return values from interactions with data structures, hence DIGIX can consider to check for the return values of the `append` method instead.

- ✓ No other than one of the following states (phases) shall apply to a proposal: Endorsement, Draft, Draft Voting, Voting, milestone Delivery, Interim Voting, Closed

If at all, these are only implicitly present through the corresponding time phases and action allowed to be taken during that time and do not have an explicit representation in the code base. Only the below states are explicitly encoded:

```
bytes32 PROPOSAL_STATE_PREPROPOSAL = "proposal_state_preproposal";
bytes32 PROPOSAL_STATE_DRAFT = "proposal_state_draft";
bytes32 PROPOSAL_STATE_MODERATED = "proposal_state_moderated";
bytes32 PROPOSAL_STATE_ONGOING = "proposal_state_ongoing";
bytes32 PROPOSAL_STATE_CLOSED = "proposal_state_closed";
bytes32 PROPOSAL_STATE_ARCHIVED = "proposal_state_archived";
```

DIGIX remarks that originally there was a confusion in terminology between "allowed phases for a proposal" and "possible states of proposal". The last state in the list above was added in an updated code version.

Draft proposal phase: It shall hold, that:

- ✓ Only moderators can move pre-proposals to drafts
 - | `endorseProposal` checks `require(isModerator(msg.sender))`.
- ✓ In Draft phase,
 - ✓ Proposer can update its draft details
 - | By calling `modifyProposal`, which can only be successfully done if called by proposer, since there is a check `require(isFromProposer(_proposalId))`.
 - ✓ Proposer can finalize and move to Draft Voting phase
 - | `finalizeProposal` is only callable by the proposer, enforced by `require(isFromProposer(_proposalId))`.
- ✓ A proposal in the Draft phase or Endorsement phase that is older than `conf(CONF_PROPOSAL_DEAD_DURATION)` can be optionally closed by the founders using a `founderCloseProposals` transaction
 - | `founderCloseProposals` correctly enforces these restrictions with `require` clauses. However, an additional check could be introduced to explicitly verify for the proposal to have the allowed state.
- ✓ Proposals will need ETH collateral to be locked before creation
 - | Enforced by the following checks in `submitPreproposal()`:

```
require(msg.value == getUintConfig(CONFIG_PREPROPOSAL_DEPOSIT))
require(address(daoFundingManager()).call.value(msg.value)())
```
- ✓ The amount of collateral can be voted upon/set
 - | Can be done by calling `updateUintConfigs` from the `DaoSpecialVotingClaims` contract, through the `claimSpecialProposalVotingResult` function.
- ✓ Proposers can receive back their collateral in either of the only three cases:
 - ✓ Proposer cancelled his proposal before any voting activity took place
 - | A proposal can be cancelled by calling `closeProposal`, if not yet finalized. Because `closeProposal` checks that `require(_finalVersion == EMPTY_BYTES)`, a proposal can only go into the voting phase once it is finalized.
 - ✓ Proposal did not pass draft voting or first voting phase, meaning that no funds have been released to the proposer

* `claimDraftVotingResult` refunds the collateral in the following case when the time exceeds:

```
if (now > daoStorage().readProposalDraftVotingTime(_proposalId)
    .add(getUintConfig(CONFIG_DRAFT_VOTING_PHASE))
    .add(getUintConfig(CONFIG_VOTE_CLAIMING_DEADLINE))
    || !isNonDigixProposalsWithinLimit(_proposalId))
```

* `claimDraftVotingResult` refunds the collateral when draft voting is failed. It internally calls `processDraftVotingClaim` which in turn calls `processCollateralRefund` when voting failed.

* `claimProposalVotingResult` does so when `index=0` (first voting round) calls the function `processCollateralRefund`, when voting result fails.

✓ The proposal completes all milestones and gets past the final voting round

| `processSuccessfulVotingClaim` is correctly called from `claimProposalVotingResult` if the final round is successfully passed.

Draft voting phase: It shall hold, that:

✓ Draft Voting Phase lasts `conf(CONFIG_DRAFT_VOTING_PHASE)` seconds

| It is specified as `uintConfigs[CONFIG_DRAFT_VOTING_PHASE] = 2 weeks`. Draft voting shall not be possible after this time. Thus `voteOnDraft()` can not be called anymore. This is enforced by the modifier `ifDraftVotingPhase`, which calls

```
requireInPhase(
    daoStorage().readProposalDraftVotingTime(_proposalId),
    0,
    getUintConfig(CONFIG_DRAFT_VOTING_PHASE)
);
```

✓ Draft Voting Phase allows no modifications of proposals

| To modify a proposal `modifyProposal` needs to be called. This is not possible because of `isEditable(_proposalId)` which checks for `_finalVersion == EMPTY_BYTES`. This field is set when calling `daoStorage().finalizeProposal(_proposalId)` in `finalizeProposal()`.

✓ Draft Voting Phase allows moderators to openly vote (y/n) and change their vote

| The function `voteOnDraft` (this is the only way to call `addDraftVote()`) checks if `isModerator(msg.sender)` and directly adds the vote. So the vote is open and can only be true or false due to `bool _voteYes`. The function can be called multiple times in between `ifDraftVotingPhase(_proposalId)` and just overrides the previous vote.

✓ A Draft can move to a ModeratedProposal

| This state transition is possible.

✓ A Draft proposal can move to a Moderated Proposal if:

```
(quorum || #DGDVotedStake geq MinDraftQuorum)
&& (quota || #yes/quorum geq MinDraftQuota)}
```

| To move a draft to a moderated proposal the participant who initiated the proposal needs to call `claimDraftVotingResult` and this calls `processDraftVotingClaim` which checks:

```
(_currentResults.currentForCount.add(_currentResults.
    currentAgainstCount)
    > daoCalculatorService().minimumDraftQuorum(_proposalId))
&& (daoCalculatorService().draftQuotaPass(_currentResults.
    currentForCount,
    _currentResults.currentAgainstCount))
```


With `minimumDraftQuorum` which calls `calculateMinQuorum` calculating:

```
uint256 _ethInDao = get_contract(CONTRACT_DAO_FUNDING_MANAGER).balance;
// add the fixed portion of the quorum
_minimumQuorum = (_totalStake.mul(_fixedQuorumPortionNumerator))
.div(_fixedQuorumPortionDenominator);
// add the dynamic portion of the quorum
_minimumQuorum = _minimumQuorum.add(_totalStake.mul(_ethAsked.
mul(_scalingFactorNumerator)).div(_ethInDao.mul(
_scalingFactorDenominator)));
```

with the corresponding configuration values in accordance with the specification and `draftQuotaPass` computing:

```
_passed = _for.mul(getUintConfig(CONFIG_DRAFT_QUOTA_DENOMINATOR))
> getUintConfig(CONFIG_DRAFT_QUOTA_NUMERATOR).mul(_for.add(
_against));
```

These calculations conform to the specification requirements.

- ✓ The draft proposal voting result can be claimed only after the draft voting phase, and within a vote claiming deadline(`conf(CONFIG_VOTE_CLAIMING_DEADLINE)`). If the vote claiming period is over, the draft voting result MUST be failed.

`DaoVotingClaims.claimDraftVotingResult()` checks

```
ifDraftNotClaimed(_proposalId)
ifAfterDraftVotingPhase(_proposalId)
```

as well as the corresponding timing boundaries.

Moderated proposal voting phase: It shall hold, that:

- ✓ It lasts `conf(CONFIG_VOTING_PHASE_TOTAL)` seconds

The total time frame consists of the `CommitPhase` and `RevealPhase`. The former lasts from zero to `CONFIG_VOTING_COMMIT_PHASE`. The latter starts after until `CONFIG_VOTING_PHASE_TOTAL`.

- ✓ It allows all participants to vote y/n following a commit/reveal scheme
 - ✓ Votes can be submitted during the previously described time frames.

- ✓ Moderated Proposal is passed if:

```
(quorum OR #DGDVotedStake geq MinVotingQuorum)
&& (quota OR #yes/quorum geq MinVoteQuota)
```

`setProposalPass` is changing a proposal's state from `PROPOSAL_STATE_MODERATED` to the next state `PROPOSAL_STATE_ONGOING`. In the process `claimProposalVotingResult` correctly checks the vote and claim, where `calculateMinQuorum` is calculated as defined in the governance whitepaper formula in section 6.3. and validates for the `MinVotingQuorum` and `votingQuotaPass` checks for `MinVoteQuota`.

- ✓ If Voting passed AND PRL approves (which is by default, but can be changed to paused/stopped), then first milestone funds can be claimed by the proposer
 - ✓ If PRL calls `updatePRL` to stop/pause a proposal before the proposer gets funding, the action will be successful. When voting passes by calling `claimProposalVotingResult`, the proposer can claim the funding calling `claimFunding` after a successful vote.
- ✓ The voting round result can be claimed only after the voting phase, and within a vote claiming deadline. If the vote claiming period is over, the voting result MUST be failed

Voting results can be claimed only after the voting phase, as the method is protected with the modifier `ifAfterProposalRevealPhase`. If the milestone is already claimed, another request for the same milestone will be rejected. The milestone voting result can be claimed only before the `CONFIG_VOTE_CLAIMING_DEADLINE` deadline. This is checked with:

```
if (now < startOfmilestone(_proposalId, _index)
    .add(getUintConfig(CONFIG_VOTE_CLAIMING_DEADLINE)))
```

Milestone delivery phase: It shall hold, that:

- ✓ The proposal is funded before each milestone to achieve its goal
 - A Proposal voting claim is done first which sets the voting result, i.e. calling `claimProposalVotingResult`. After that `claimFunding` needs to be called by the proposer to get the milestone funding. This call sets the funded status of that milestone. Hence before starting any milestone, if the vote is passed then the proposer can claim the corresponding milestone funding.
- ✓ After a phase a vote decides if the proposal moves on to the next milestone
 - After getting the first milestone (`index=0`) funding, the proposer will work on the milestone and once he finishes it, he will call `finishMilestone`, starting the next `InterimVotingPhase`.
 - Voting time is checked with `getTimelineForNextVote` to ensure that for `InterimVotingPhase` enough time is left in the quarter. Otherwise, the next quarter will start.
 - Once `InterimVotingPhase` starts, participants will vote during the `CommitPhase`, using the `DaoVoting` method `commitVoteOnProposal`.
 - Once the `CommitPhase` is over, participants will reveal their votes in the `RevealPhase`, using the `DaoVoting` method `revealVoteOnProposal`.
 - Afterwards, the proposer claims his proposal voting results by `claimProposalVotingResult`.
 - If the voting result is passed, the proposer will call `claimFunding` to claim the funds for the next milestone.

Hence, a proposal moves to the next milestone after successful voting.
- ✓ The proposer is supposed to end a milestone delivery phase, if he thinks the milestone is already achieved (this is the only way to end a milestone delivery phase)
 - A proposer can call `finishMilestone` during the main phase to notify that a Milestone is completed.
 - A proposer can only do this operation after the actual start time of the Milestone delivery phase.
 - A proposer cannot call `finishMilestone` again for the same milestone.
 - By calling only `finishMilestone` method, a proposer can start the interim voting phase for the next milestone phase if there are any more milestones left. Hence this is the only way to stop the milestone delivery phase and move to the next voting and milestone phase.

Interim voting phase: It shall hold, that:

- ✓ The interim voting phase lasts for `conf (CONFIG_INTERIM_PHASE_TOTAL)` seconds, starting from when the proposer calls `finishMilestone` to explicitly end the previous milestone delivery phase
 - `ifCommitPhase` ensures that for the first milestone voting, lasts for `CONFIG_VOTING_COMMIT_PHASE` and for the rest interim voting phases it lasts for `CONFIG_INTERIM_COMMIT_PHASE`. Calling `finishMilestone` marks a milestone completed and starts the next round of interim voting.
- ✓ All participants can vote (y/n) on next funding release
 - In Voting round each participant can call `commitVoteOnProposal` with a `_commitHash` which consists of `SHA3(address(pub_address), bool(vote), bytes32(random string))`.
- ✓ Voting is passed if:

```
(#DGDStakeVoted geq MinInterimVotingQuorum)
AND (#DGDYesStake/quorum geq #MinInterimVotingQuota)
```


When calling `claimProposalVotingResult` from `DaoVotingClaims` the return value `_passed` is true, when:

- call to `countProposalVote` returns `_passed` as true
- which in turn checks `isVoteCountPassed`
- combined it is evaluated that:

```
(#DGDDStakeVoted geq MinInterimVotingQuorum)
AND (#DGDDYesStake/quorum geq #MinInterimVotingQuota)
```

- ✓ If the vote passed and the PRL has approved the proposal, the funding for the next milestone funding can be claimed by the proposer

For a proposer to successfully call `claimFunding` of the `DaoFundingManager` contract this must hold that the voting must have passed and the proposal was not stopped by PRL. Hence, property holds (under the assumption that "PRL approved" is interpreted as "PRL did not pause").

- ✓ After the last milestone, voting on whether the proposer receives `FinalReward` happens

Voting proceeds just as in the third property of this section and payout just as in the fourth with the minor difference that when `claimFunding` calls `readProposalMilestone` with the last index, it returns `_proposal.proposalVersions[_finalVersion].finalReward`.

- ✓ The voting round result can be claimed only after the voting phase, and within a vote claiming deadline. If the vote claiming period is over, the voting result MUST be failed

This is enforced by `ifAfterProposalRevealPhase` modifier in combination with `require(isMainPhase())` and setting `_passed` to false right away and only change it if within limit.

Limits on non-Digix proposals: If not initiated by the founders, proposals are bound to these limits:

- ✓ Total funding for the proposal cannot exceed `conf(CONFIG_MAX_FUNDING_FOR_NON_DIGIX)` wei

In `DaoConfigStorage` the mapping `uintConfigs` is set to 2 ETH as maximum total funding. The function `DaoCommon.checkNonDigixFundings` implements corresponding checks:

```
require(MathHelper.sumNumbers(_milestonesFundings).add(_finalReward)
    <= getUintConfig(CONFIG_MAX_FUNDING_FOR_NON_DIGIX));
require(_milestonesFundings.length
    <= getUintConfig(CONFIG_MAX_MILESTONES_FOR_NON_DIGIX));
```

The function is called if the proposer calls `submitPreproposal`, `modifyProposal` and `changeFundings`. These are the only functions affecting the total funding of a proposal.

- ✓ Total number of milestones in a proposal cannot exceed `conf(CONF_MAX_MILESTONES_FOR_NON_DIGIX)` limit

In `DaoConfigStorage` the mapping `uintConfigs` is set to 2 ETH as maximum total funding per milestone. These are checked by the same requires as in the previous property. The verifying function is called if the proposer calls `submitPreproposal`, `modifyProposal` and `changeFundings`. These are the only functions affecting the milestone funding of a proposal.

- ✓ Total number of non-Digix proposals that get pass the first Voting Round in a quarter must not exceed `conf(CONFIG_NON_DIGIX_PROPOSAL_CAP_PER_QUARTER)`

In `DaoConfigStorage` the mapping `uintConfigs` is setting the number to 10. The variable is only used in the `DaoCommon` function `isNonDigixProposalsWithinLimit`. The `claimProposalVotingResult` function needs to be called to increment the proposal counter. To pass the first voting round this function needs to be called. Therefore the the specification holds.

4.1 Preproposals can be overwritten M ✓ Fixed

A malicious participant can simply hijack existing preproposals by calling the `submitPreproposal` function and replaying a previously submitted `_docIpfsHash` in the DAO contract.

While this still requires for the attacker to fulfill the criteria of being a participant and to submit the collateral for a preproposal, an attacker can benefit by claiming a high-quality proposal to be his own and modify the funding and reward schemes.

Likelihood: High

Impact: Low

Fixed: DIGIX now ensures that existing preproposals cannot be overwritten by checking that the proposal ID of the `struct` `proposalById` at this storage location is empty.

4.2 Draft voting passes with minority L ✓ Acknowledged

The DIGIX's specifications define that moving a draft proposal to a moderated proposal requires at least:

```
(quorum OR #DGDVotedStake geq MinDraftQuorum) AND  
(quota OR #yes/quorum geq MinDraftQuota)
```

Given the quorum requirement is fulfilled, the initial values to calculate the quota are:

```
uintConfigs[CONFIG_DRAFT_QUOTA_NUMERATOR] = 30;  
uintConfigs[CONFIG_DRAFT_QUOTA_DENOMINATOR] = 100;
```

and the function implemented in `daoCalculatorService().draftQuotaPass()` is:

$Passed := n_{yesVotes} * DENOMINATOR > Total * NUMERATOR$

Thus, a vote can pass even though it has a minority of "yes" voters. Let us assume a total of three votes, one "yes" and two "no". Hence:

$Passed := 1 * 100 > 30 * 3$

Therefore, the vote can pass with any results over 30%. Here it passed with 33.33% which is a minority.

Likelihood: Low

Impact: Medium

Fixed: DIGIX states that the numbers in the `DaoConfigsStorage`'s constructor are dummy values for now. They will be set to proper configuration values upon deployment of the DAO.

Special Proposals

It shall hold that

- ✓ No account other than the founder can initiate a special proposal
 - To create a special proposal `daoSpecialStorage().addSpecialProposal()` needs to be called. This is exclusively (due to `require(sender_is(CONTRACT_DAO_SPECIAL_PROPOSAL))`) possible by calling `createSpecialProposal()`. This function has the modifier `if_founder()`, which checks `identity_storage().read_user_role_id(msg.sender) == ROLES_FOUNDERS`. Thus only accounts belonging to the group founders can initiate a special proposal.
- ✓ Only one kind of special proposals are possible (change parameters in the governance model)
 - When claiming the voting results by calling `claimSpecialProposalVotingResult()` the function `setConfigs()` only changes the parameters of the governance model.
- ✓ Consists of one voting phase lasting (`conf(CONFIG_SPECIAL_PROPOSAL_PHASE_TOTAL)` seconds)
 - As mentioned a special proposal can only be initialized by calling `createSpecialProposal()`. To start the voting `startSpecialProposalVoting()` needs to be called. This function registers the start time and checks if enough time is left in this quarter to start the proposal. Also covered in the following properties.
- ✓ The voting phase has `conf(CONFIG_SPECIAL_PROPOSAL_COMMIT_PHASE)` seconds for submitting votes

To submit a vote `commitVoteOnSpecialProposal()` needs to be called. The implemented modifier `ifCommitPhaseSpecial()` checks if the vote is in the submitting phase by calling `requireInPhase()` which checks

```
require(_startingPoint > 0);
require(now < _startingPoint.add(_relativePhaseEnd));
require(now >= _startingPoint.add(_relativePhaseStart));
```

with

```
_startingPoint = requireInPhase(daoSpecialStorage().readVotingTime(
    _proposalId)
// the time startSpecialProposalVoting() was called
_relativePhaseStart=0;
_relativePhaseEnd=getUintConfig(CONFIG_SPECIAL_PROPOSAL_COMMIT_PHASE));
```

and thus a participant has `CONFIG_SPECIAL_PROPOSAL_COMMIT_PHASE` seconds to submit a vote.

✓ After the voting process

```
conf(CONFIG_SPECIAL_PROPOSAL_PHASE_TOTAL)} –
conf(CONFIG_SPECIAL_PROPOSAL_COMMIT_PHASE)}
```

seconds remain for revealing votes

To reveal a vote `revealVoteOnSpecialProposal()` needs to be called. The function's modifier (`ifRevealPhaseSpecial`) checks for

```
requireInPhase(
    daoSpecialStorage().readVotingTime(_proposalId),
    getUintConfig(CONFIG_SPECIAL_PROPOSAL_COMMIT_PHASE),
    getUintConfig(CONFIG_SPECIAL_PROPOSAL_PHASE_TOTAL));
```

As mentioned before this is:

```
require(proposalsById[_proposalId].voting.startTime > 0);
require(now < proposalsById[_proposalId].voting.startTime.add(
    CONFIG_SPECIAL_PROPOSAL_PHASE_TOTAL));
require(now >= proposalsById[_proposalId].voting.startTime.add(
    CONFIG_SPECIAL_PROPOSAL_COMMIT_PHASE));
```

Thus, it needs to be within the time window T where:

```
CONFIG_SPECIAL_PROPOSAL_COMMIT_PHASE < T
< CONFIG_SPECIAL_PROPOSAL_PHASE_TOTAL
```

✓ A successful vote needs at least a quorum of `conf(CONFIG_SPECIAL_PROPOSAL_QUORUM_NUMERATOR)` / `conf(CONFIG_SPECIAL_PROPOSAL_QUORUM_DENOMINATOR)` of the `totalLockedDGDStake`

For a special proposal to pass, the votes need to be claimed by the proper `claimSpecialProposalVR` before `CONFIG_VOTE_CLAIMING_DEADLINE` ends. If not, it fails. The function checks if the proposal passed by:

```
if (( _currentResults.currentForCount.add(_currentResults.
    currentAgainstCount) > daoCalculatorService().
    minimumVotingQuorumForSpecial())
    && (daoCalculatorService().votingQuotaForSpecialPass(currentResults.
        currentForCount, _currentResults.currentAgainstCount))) {
    _passed = true;
    setConfigs(_proposalId);
}
daoSpecialStorage().setPass(_proposalId, _passed);
```

```
daoSpecialStorage().setVotingClaim(_proposalId, true);
```

The function `minimumVotingQuorumForSpecial()` checks if enough participants voted by:

```
_minQuorum = getUintConfig(CONFIG_SPECIAL_PROPOSAL_QUORUM_NUMERATOR).  
mul(  
    daoStakeStorage().totalLockedDGDStake()).div(  
    getUintConfig(CONFIG_SPECIAL_PROPOSAL_QUORUM_DENOMINATOR));
```

with `uintConfigs[CONFIG_SPECIAL_PROPOSAL_QUORUM_NUMERATOR] = 70`; and correspondingly `uintConfigs [CONFIG_SPECIAL_PROPOSAL_QUORUM_DENOMINATOR] = 100`;

- ✓ Also a min quota of `conf(CONFIG_SPECIAL_QUOTA_NUMERATOR)/conf(CONFIG_SPECIAL_QUOTA_DENOMINATOR)` is needed

When calling the voting results by calling `claimSpecialProposalVotingResult()` the function `setConfigs()` changes the parameters of the governance model.

If this is true it also checks if the vote passed or not by calling `votingQuotaForSpecialPass`.

```
function votingQuotaForSpecialPass(uint256 _for, uint256 _against)  
    public  
    constant  
    returns (bool _passed)  
{  
    if ((_for.mul(getUintConfig(CONFIG_SPECIAL_QUOTA_DENOMINATOR)))  
        >  
        (getUintConfig(CONFIG_SPECIAL_QUOTA_NUMERATOR)  
        .mul(_for.add(_against)))) {  
        _passed = true;  
    }  
}
```

with `uintConfigs[CONFIG_SPECIAL_QUOTA_DENOMINATOR] = 100`; and the parameter `uintConfigs [CONFIG_SPECIAL_QUOTA_NUMERATOR] = 51`; Note: a tie and the vote does not pass.

5.1 Unchecked arithmetic operations

In the `DaoCommon` contract, the modifier `ifAfterDraftVotingPhase` adds two timestamps directly by using the `+` operator, without checking for overflow or using a library such as `SafeMath`.

```
uint256 _start = daoStorage().readProposalDraftVotingTime(_proposalId);  
require(_start > 0);  
require(now >= _start + getUintConfig(CONFIG_DRAFT_VOTING_PHASE));
```

Given that only the `DaoSpecialVotingClaims` contract can set the configuration values (by calling the function `updateUintConfigs` in the `DaoConfigStorage` contract) a overflow might seem unlikely, however no checks on writing the corresponding fields exist.

Likelihood: Low

Impact: Low

Fixed: DIGIX now uses the `SafeMath` function `add()` for the addition.

5.2 Special proposals details can be silently updated

The Founder can call `createSpecialProposal` method multiple times replaying the same hash `_doc` and update the configuration of a special proposal silently, as no events are emitted.

This would allow the founder to change critical values and settings that affect the whole user base without them noticing, either by accident or on purpose. A proposal creation should be allowed only once and a separation between creation and updates can be made as is done with normal proposals.

Likelihood: Low

Impact: High

Fixed: Overwriting an already existing special proposal is now prevented, a check has been introduced which ensures that no previous special proposal with the same hash already exists or the call will revert.

5.3 Array underflows

In the following functions the listed arrays can underflow and read from unknown memory locations if called with `_operations = 0`, given that all other provided arguments are valid.

- `DaoSpecialVotingClaims.claimSpecialProposalVotingResult()`, concretely `address _lastVoter = _voters[_voters.length - 1]`
- `DaoVotingClaims.claimDraftVotingResult()`, concretely `moderators[_moderators.length-1]`
- `DaoVotingClaims.claimProposalVotingResult()`, concretely `address _lastVoter = _voters[_voters.length - 1]`

The same pattern is used in the functions `calculateVoterBonus()` and `sumEffectiveBalance()`. However these take the correct precautionary measures, checking whether `_operations = 0` and returning accordingly.

Likelihood: Low

Impact: High

Fixed: DIGIX implemented the proposed checks.

Voting

Voting on proposals:

- ✓ No accounts with less than `minimumDgdToParticipate` can participate (vote on any proposal in the DAO)

Voting methods present in `DaoVoting`:

- `voteOnDraft` (only Moderator allowed)
- `commitVoteOnProposal` (only Participant allowed)
- `commitVoteOnSpecialProposal` (only Participant allowed)

The requirements for moderators and participants are already checked in the roles section. There is no other place in the code to vote and call these methods. Hence, the property holds true.

- ✓ No participant with less than `minimumDgdToModerate` and `minimumRpToModerate` can become a Moderator
 - ▮ See section roles - Moderator.
- ✓ Stopped proposals (by PRL) cannot be unpaused and will be deemed as over
 - ▮ This holds true and corresponding checks were introduced.

Voting Mechanics:

- ✓ Voting power is equal to `lockedDGDStake` by the participant or moderator

When voting (on a draft or draft proposal) the participant or moderator needs to call one of these functions to reveal the vote:

`voteOnDraft()` which calls:

```
...
uint256 _moderatorStake = daoStakeStorage().lockedDGDStake(_moderator);
...
daoStorage().addDraftVote(_proposalId, _moderator, _voteYes,
    _moderatorStake);
...
```

Adding this to the proposal (draftVoting) struct through the `addDraftVote` function. When counting the draft votes `claimDraftVotingResult` needs to be called which ultimately calls `countVotes`

in `DaoStructs.sol`. This contract directly accesses the stored votes (which are weights), that were set before and sums them up. Otherwise the participants and moderators call `revealVoteOnProposal()` and `revealVoteOnSpecialProposal()` which both ultimately call `revealVote()` in `DaoStructs.sol` with

```
function revealVote(
    Voting storage _voting,
    address _voter,
    bool _vote,
    uint256 _weight
)
    public
{
    if (_vote) {
        _voting.yesVotes[_voter] = _weight;
    } else {
        _voting.noVotes[_voter] = _weight;
    }
}
```

Because the call is made with:

```
daoStorage().revealVote(_proposalId, msg.sender, _vote,
    daoStakeStorage().lockedDGDStake(msg.sender), _index);
daoStorage().revealVote(_proposalId, msg.sender, _vote,
    daoStakeStorage().lockedDGDStake(msg.sender));
```

This directly sets the votes as the stake. The voting results are determined by calling the functions `claimProposalVotingResult()` in `DaoVotingClaims.sol` or `claimSpecialProposalVotingResult()` in `DaoSpecialVotingClaims.sol`. These ultimately call to check if passed:

```
( _currentResults.currentForCount.add(_currentResults.
    currentAgainstCount) > daoCalculatorService().
    minimumVotingQuorumForSpecial()) &&
    (daoCalculatorService().votingQuotaForSpecialPass(
        _currentResults.currentForCount, _currentResults.
        currentAgainstCount))
```

and

```
_passed = ( _currentResults.currentForCount.add(_currentResults.
    currentAgainstCount) > daoCalculatorService().minimumVotingQuorum(
    _proposalId, _index))
    && (daoCalculatorService().votingQuotaPass(
        _currentResults.currentForCount, _currentResults.
        currentAgainstCount));
```

Thus, it counts the weights set previously.

- ✓ It is not possible to recover the vote without knowing the `CommitSecret`

This depends on the hashing performed by `DIGIX`, i.e. `keccak256(abi.encodePacked(msg.sender, _vote, _salt))`. As the generation of the random salt is not provided and out of scope, no hard guarantees can be given. Hence, this holds true under the assumption that no hash collisions can be performed and that a truly random salt is used.

- ✓ A voter can change his vote in the commit period by committing again

The function `commitVote()` would just override the committed hash `proposalsById[_proposalId].votingRounds[_index].commits[_voter] = _hash;` and `proposalsById[_proposalId].voting.commits[_voter] = _hash;` and for drafts `addDraftVote()` would override the votes directly with:

```

DaoStructs.Proposal storage _proposal = proposalsById[_proposalId];
    if (_vote) {
        _proposal.draftVoting.yesVotes[_voter] = _weight;
        if (_proposal.draftVoting.noVotes[_voter] > 0) { //
            minimize number of writes to storage, since EIP-1087 is
            not implemented yet
            _proposal.draftVoting.noVotes[_voter] = 0;
        }
    } else {
        _proposal.draftVoting.noVotes[_voter] = _weight;
        if (_proposal.draftVoting.yesVotes[_voter] > 0) {
            _proposal.draftVoting.yesVotes[_voter] = 0;
        }
    }
}

```

Therefore this property holds.

- ✓ Only the last committed vote counts in when the vote is revealed in the reveal period
 - See previous specification item. New calls override the committed hash. Thus, it always is the most current vote.
- ✓ The commitSecret needs to be revealed by the voter in the reveal period
 - See description above. The hash would be set, but if the secret is not revealed, the variables yesVotes[_voter] and noVotes[_voter], which are the ones that are counted in the end, are just not set for this specific user on the corresponding proposal.
- ✓ Only successfully revealed votes are counted
 - This is given implicitly when following the previously confirmed specification properties.
- ✓ The min. quora shall equal the specification in 6.3⁷

The code defines different quorum calculations, such as minVotingQuorumForSpecial, the function calculateMinQuorum, and minimumDraftQuorum. However the underlying formula stays the same as just the input parameters vary.

```

minQuorum = totalStake * (fixedMinDGDDStakeNeeded + ETHAsked / ETHInDao
    * scalingFactor)

```

The code calculates:

```

function calculateMinQuorum(
    uint256 _totalStake,
    uint256 _fixedQuorumPortionNumerator,
    uint256 _fixedQuorumPortionDenominator,
    uint256 _scalingFactorNumerator,
    uint256 _scalingFactorDenominator,
    uint256 _ethAsked
)
    internal
    constant
    returns (uint256 _minimumQuorum)
{
    uint256 _ethInDao = get_contract(CONTRACT_DAO_FUNDING_MANAGER).
        balance;
    // add the fixed portion of the quorum
    _minimumQuorum = (_totalStake.mul(_fixedQuorumPortionNumerator)
        ).div(_fixedQuorumPortionDenominator);
    // add the dynamic portion of the quorum

```

⁷<https://github.com/DigixGlobal/dao-contracts/blob/master/doc/GovernanceModel.pdf>


```

        _minimumQuorum = _minimumQuorum.add(_totalStake.mul(_ethAsked.
            mul(_scalingFactorNumerator)).div(_ethInDao.mul(
                _scalingFactorDenominator)));
    }

```

Thus this holds true.

6.1 PRL can unpause stopped proposal

The PRL role can easily pause or unpause a stopped proposal by proceeding as follows:

- PRL calls `Dao.updatePRL()` with the STOP action on a proposal. This internally closes the proposal by setting it to `PROPOSAL_STATE_CLOSED`.
- PRL now calls `Dao.updatePRL()`, passing the same `proposalId` and the UNPAUSE action. This successfully resets the previously stopped proposal state.

Such behavior is possible as there are no checks to ensure whether the proposal was already stopped or not and directly violates the property stated in the whitepaper, section 4.3: Stopped proposals cannot be unpaused⁸.

Likelihood: Low

Impact: High

Fixed: DIGIX introduced a **require** and now no action can be done on a stopped proposal.

Reputation

- ✓ One DigixDao Badge can be redeemed for `conf(CONFIG_REPUTATION_POINT_BOOST_FOR_BADGE)` reputation points (still DGD needs to be locked up to to become a moderator)
 - `DaoStakeLocking.redeemBadge()` method will be called by anyone to redeem his badge and get `CONFIG_REPUTATION_POINT_BOOST_FOR_BADGE` reputation points. Also, `refreshModeratorStatus()` checks that the moderator has sufficient DGDs locked.
- ✓ Only one DigixDAO Badge can be redeemed for a specific address
 - This holds true as the method is protected with `check require (!daoStakeStorage().redeemedBadge(msg.sender));` and does not allow same address to redeem several badges.

- ✓ The Reputations points at the end of each quarter as defined in 7.3⁹

Methods reviewed

- `DaoPointsStorage.addReputation`
- `DaoPointsStorage.subtractReputation` seems to be handling a case when more points need to be reduced from the current balance. That is why special handling is present in the code. Balance 0 is kept even if more points are to be reduced and returns a success result.
- `DaoRewardsManager.updateRPfromQP` method calculates `changeInRP` value and adds/subtracts from user's reputation points. It only calculates the first two formulas mentioned in the specification document.
- `updateUserReputationUntilPreviousQuarter` checks and calculates reputation points to reduce when a user has not participated in the quarter. The code listed below code is in accordance with the specification and multiplies with every quarter with no participation.

```

_reputationDeduction =
    (currentQuarterIndex().sub(1).sub(
        _lastQuarterThatReputationWasUpdated))
    .mul(getUintConfig(CONFIG_MAXIMUM_REPUTATION_DEDUCTION)
        .add(getUintConfig(CONFIG_PUNISHMENT_FOR_NOT_LOCKING)));

```

⁸<https://github.com/DigixGlobal/dao-contracts/blob/160e456a9683c3e477ff707c469a8e96c25acac4/doc/GovernanceModel.pdf>

⁹<https://github.com/DigixGlobal/dao-contracts/blob/master/doc/GovernanceModel.pdf>

- `updateRewardsAndReputationBeforeNewQuarter` is called from `withdrawDGD` and `lockDGDInternal`, so ReputationPoints calculations are done every upon lock/withdraw operation.

Hence, all holds true for the first part of the 7.3 section in the governance whitepaper.

- ✓ The Reputation bonus for "consistent votes" follows what is detailed in 7.3, with an additional details that was not mentioned: The participant who made the "consistent vote" must also be a participant in the current quarter to receive the reputation bonus.

The calculation is correct and as defined in section 7.3 of the document.

```
uint256 _bonus = _qp.mul(_rate).mul(
    getConfig(CONFIG_REPUTATION_PER_EXTRA_QP_NUM))
    .div(_base
        .mul(getConfig(CONFIG_REPUTATION_PER_EXTRA_QP_DEN)));
```

And:

- `calculateVoterBonus` is filtering out the yes/no votes of participants and distributing them with bonus reputation points.
- `claimProposalVotingResult` calls `calculateVoterBonus` only during interim voting rounds.
- Bonus reputation points are only given to voters who are participants of the current quarter as well. This is checked in `addBonusReputation`

```
if (isParticipant(_voters[i])) {
    // only give bonus reputation to current participants
    daoPointsStorage().addReputation(_voters[i], _bonus);
}
```

All specification properties hold.

Rewards

It shall hold, that

- ✓ The DGX rewards are distributed to participants and moderators accordingly to the formula in 8¹⁰.

The specs defines the DGX rewards for participants and moderators as follows:

$$base * \left(1 + \frac{(QP - minQP)}{QPS}\right) * \left(1 + \frac{RP}{RPS}\right)$$

The code implementation is:

```
uint256 _baseDGDBalance = MathHelper.min(_quarterPoint,
    _minimalParticipationPoint).mul(_lockedDGDStake).div(
    _minimalParticipationPoint);
_effectiveDGDBalance =
    _baseDGDBalance
    .mul(_quarterPointScalingFactor.add(_quarterPoint).sub(
        _minimalParticipationPoint))
    .mul(_reputationPointScalingFactor.add(_reputationPoint))
    .div(_quarterPointScalingFactor.mul(
        _reputationPointScalingFactor));
```

The result is then distributed by fraction of the staked DGD, as well as the moderators and participants shares.

When rewriting the formula above we end up with the code implementation by:

¹⁰<https://github.com/DigixGlobal/dao-contracts/blob/master/doc/GovernanceModel.pdf>

$$\begin{aligned}
&= base * \left(1 + \frac{RP}{RPS} + \frac{(QP - minQP)}{QPS} + \frac{(QP - minQP)}{QPS} * \frac{RP}{RPS} \right) \\
&= base * \left(\frac{QPS * RPS}{QPS * RPS} + \frac{RP * QPS}{QPS * RPS} + \frac{(QP - minQP) * RPS}{QPS * RPS} + \frac{(QP - minQP) * RP}{QPS * RPS} \right) \\
&= base * \left(\frac{QPS * RPS + RP * QPS + QP * RPS - minQP * PRS + QP * RP - minQP * RP}{QPS * RPS} \right) \\
&= base * \left(\frac{(QPS + QP - minQP) * RPS + RP * (QPS + QP - minQP)}{QPS * RPS} \right) \\
&= base * \left(\frac{(RPS + RP) * (QPS + QP - minQP)}{QPS * RPS} \right) \\
&= base * \left(\frac{(QPS + QP - minQP) * (RPS + RP)}{QPS * RPS} \right)
\end{aligned}$$

- ✓ Any participant who keeps their DGX rewards in the DAO is supposed to pay for demurrage fees for the duration that the DGX rewards stays in the DAO, from the corresponding dgxDistributionDay. The demurrage fees are calculated using calculateDemurrage function of a DgxDemurrageCalculator contract (like in here), which is in similar fashion to how MockDgxDemurrageCalculate contract is implemented in dao-contracts repository.

It is not possible to check the final implementation. The demurrage fee is deducted from the claimable DGX. By:

```
_claimableDGX = _claimableDGX.sub(
    daoCalculatorService().calculateDemurrage(_claimableDGX,
        _days_elapsed));
```

But calculateDemurrage() is only implemented in a mock contract in the project. But it shall in the future be linked to the contract(DgxDemurrageCalculator) provided here^a. The calculation is in a two step process, with two different time frames. First, the demurrage fee is calculated for the corresponding quarter in the daoRewardsManager function updateUserRewardsForLastParticipatingQuarter. Then, the demurrage fee up until the user claims his rewards is calculated when the user calls daoRewardsManager.claimRewards().

^a<https://etherscan.io/address/0xcd76744cd377707279cd500e40a08d707147c871>

8.1 Old DAO can be funded

The DaoFundingManager has a fallback function which can receive ETH, funding the DAO. However, once the migration process happens and all funds are migrated to a new DaoFundingManager, the old one is still fundable and can receive money by users who did not keep track of the address change. This issue highlights why events notifying users about migrations and other major events are important.

Likelihood: Low

Impact: Low

Fixed: DIGIX solved the problem by only allow deposits by a given funding source and the DAO contract. This prevents users from accidentally deposit ETH to the contract.

8.2 Wrong parameter description

The DaoFundingStorage contract defines two functions taking a uint argument named _ethAmount, addEth and withdrawEth. However such naming is misleading insofar as at least addEth is called with the msg.value argument, which is denominated in Wei and not ETH. Hence, CHAINSECURITY recommends to review the

naming and reconsider whether the original mismatch has any further implications for the balance calculations or any other assumptions (e.g. the `ifFundingPossible` modifier).

Likelihood: Low

Impact: Medium

Fixed: DIGIX solved this by removing the `DaoFundingStorage` contract whose sole purpose was the tracking of the `DaoFundingManagers` balance. Tracking the balance of the `DaoFundingManager` contract is now done directly by reading it from state. There are no functions `addEth` and `withdrawEth` anymore, a new function `weiInDao` returns the contract's balance.

Trust Issues

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 DIGIX, including in DIGIX's ability to deal with such powers appropriately.

Remarks on the migration process

The DIGIX DAO system can be migrated to a new set of contracts. Contracts to be deployed are a main Dao, a FundingManager and a RewardsManager. These carry functionality central to the proposal creation, voting and rewards payoff. Given these powers, malicious behaviour of the contracts can have grave consequences and trust into the code needs to be highlighted.

Acknowledged: DIGIX acknowledges that the scope of this audit only concerns this current set of contracts, migration to a new set of contract is out of scope. An event has been added to signal the migration of DigixDao.

1.1 Malicious fund transfer

A malicious root can call `migrateToNewDao` during any locking phase after setting addresses controlled by him through `setNewDaoContracts`. This would transfer all ETH and DGXs the `DaoFundingManager` contract holds through `moveFundsToNewDao` and `moveDGXsToNewDao` to any address provided by such a malicious root. More so, the whole process emits absolutely no events which makes it harder for users to timely observe such behavior.

Acknowledged: DIGIX acknowledges that DigixDAO participants will have to trust this DIGIX-controlled root address to be securely managed by DIGIX. The root will be a multisig wallet that is controlled by DIGIX founders.

1.2 Information migration

Information like the on-going proposals, including voting rounds results, reputation points and pending changes such as the `ClaimableDGXs`, need to be copied over to the new contracts. This process is not fully documented and again emits no on-chain events, leaving the user without the possibility to audit these important migrations and possible modifications.

Acknowledged: DIGIX acknowledges that DigixDAO participants will need to trust that DIGIX will migrate the data (proposal, dgx rewards, etc) to the new set of contracts

1.3 Technical competence when migrating

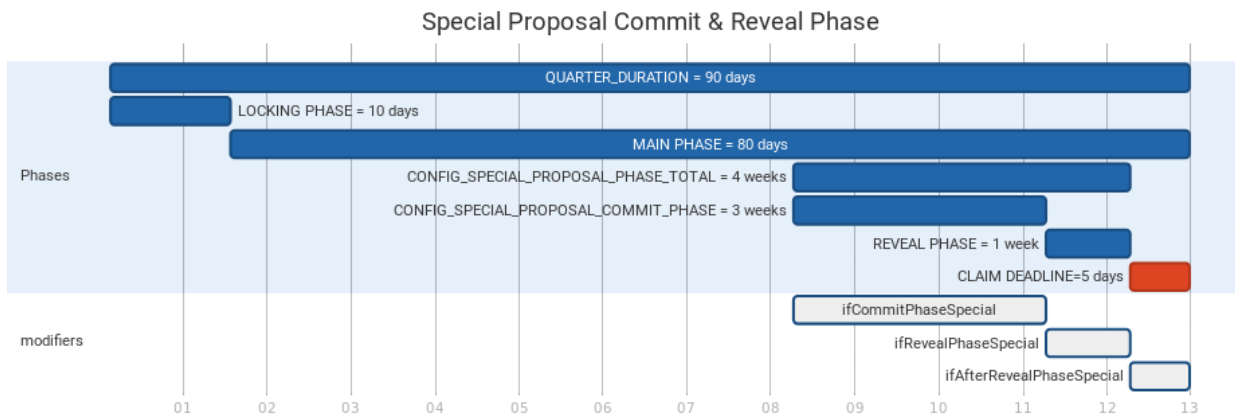
The design of DIGIX allows multiple central components, such as voting and reward procedures, to be migrated. In case of a migration the new smart contracts have to be carefully evaluated to make sure that no functionality breaks, no vulnerabilities are contained and no additional permissions are granted. In practice, every user should have the technical skills to review the proposed upgrades independently, without relying on third-party explanations. Given the migration features of DIGIX's project, there is no real bound on how complex future migrations will be, which in turn implies that the minimum required technical skills of voters is unknown as of now.

Acknowledged: DIGIX acknowledges that DigixDAO participants will have to trust that DIGIX will conduct proper security audit on the new set of contracts, as well as inform DigixDAO participants, before migrating DigixDAO to the new set of contracts.

Design Issues

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

Special Proposals may fail unintentionally M ✓ Fixed



`DaoSpecialVotingClaims.claimSpecialProposalVotingResult` checks that the current time must be less than `startVotingTime + CONFIG_SPECIAL_PROPOSAL_PHASE_TOTAL + CONFIG_VOTE_CLAIMING_DEADLINE` otherwise it will fail.

However, when starting the special proposal by calling `startSpecialProposalVoting`, it checks that the time left in quarter must be greater than 4 weeks.

`CONFIG_VOTE_CLAIMING_DEADLINE` highlighted in red in the above image would be adjusted according to the day when the special proposal was started. For example if the proposal is started on the 61st day and hour 23:00:00, only one hour would be available for the proposer to claim his voting results.

Hence, if `startSpecialProposalVoting` is called between 57th - 61st day of a quarter, then there would be less than the designated 5 days for the proposer to claim the special voting, because the `require(isMainPhase())` condition would stop the proposer from claiming his voting result in the next quarter (locking phase of the next quarter).

Note that the time 23:00:00 is proportional to the start day and hour of the quarter.

Fixed: DIGIX explicitly added the vote claiming deadline to the requirement for time left in a quarter.

Preliminary iteration L ✓ Fixed

In `DaoCommon.checkNonDigixFundings()` DIGIX encodes the following checks:

```
require(MathHelper.sumNumbers(_milestonesFundings).add(_finalReward) <=
    getUIntConfig(CONFIG_MAX_FUNDING_FOR_NON_DIGIX));
require(_milestonesFundings.length <= getUIntConfig(
    CONFIG_MAX_MILESTONES_FOR_NON_DIGIX));
```

However, as `sumNumbers` iterates over `_milestonesFundings` we suggest to reorder the checks to first validate the length and then loop to avoid potentially useless computation.

Fixed: The statements have been reordered.

DaoFundingManager can receive arbitrary funds

The fallback function of the DaoFundingManager states the intention "to receive ETH funds from DigixDAO crowdsale contract". However, transferred funds are not checked for their origin and anyone can successfully send ETH to the contract. If this is not intended, a mechanism to prevent this should be implemented.

```
/**
 * @notice Payable function to receive ETH funds from DigixDAO crowdsale
 * contract
 */
function () payable public {
    daoFundingStorage().addEth(msg.value);
}
```

Fixed: DIGIX added require statements that only a special funding address or the Dao contract may deposit ETH. There is no risk for users accidentally depositing ETH to the contract anymore.

Deprecated constant keyword

Many functions in DIGIX's code base have their visibility declared as `constant` which is a deprecated keyword and should not be used. Instead, the visibility can be described with `view`¹¹. Numerous examples of this can be found in the DAO and CACP repositories. Even after initial review some occurrences can be still found in the code base. Specifically,

- ResolverClient.is_locked method
- ResolverClient.get_contract method
- ContractResolver.get_contract method

Fixed: DIGIX updated the implementation and exchanged the deprecated `constant` keyword with `view` where applicable.

Assigning to function arguments

Assigning to function arguments is considered bad practice and should be avoided. A concrete example of this can be found in the `calculateGlobalRewardsBeforeNewQuarter` function of the DaoRewardsManager contract.

Fixed: DIGIX introduced local variables instead of reusing the function arguments.

Duplicate code

The `isWhitelisted` and `daoWhitelistingStorage` functions declared in the DaoWhitelistingCommon contract are directly inherited by the daoListingService contract, where they are overwritten with an identical implementation. This has no use and should be avoided.

Fixed: DIGIX removed the duplicate implementations in the DaoListingService contract.

Fallback function is public

The fallback function is used in the DaoFundingManager contract. CHAINSECURITY recommends to update its visibility to `external`, given that this will be enforced with the upcoming breaking compiler release and is recommended in the Solidity documentation¹².

¹¹<https://solidity.readthedocs.io/en/latest/contracts.html#view-functions>

¹²<https://solidity.readthedocs.io/en/latest/050-breaking-changes.html#explicitness-requirements>

Fixed: DIGIX changed the visibility of the fallback function in the `DaoFundingManager` contract to external.

Inefficient fund tracking

The `DaoFundingStorage` contract has one public state variable `ethInDao` and two functions to increment and decrement it. Both methods can only be called by the `DaoFundingManger` contract, which is done in that contract whenever ETH is received through its fallback function or withdrawn through calls to `claimFunding`, `refundCollateral` or `moveFundsToNewDao`. Hence, it seems that the only purpose of the `DaoFundingStorage` is to expose and track the amount of funds in `DaoFundingManager`.

If this is indeed the case, such a design is highly inefficient given that a simple public getter function returning `address(this).balance` could provide the same functionality. This way significant gas costs will be saved by avoiding external function calls on each ETH transfer and mitigating the deployment of another contract.

Fixed: DIGIX removed the `DaoFundingStorage` contract and uses the balance of `DaoFundingManager`. `address` in the new `weiInDao` function.

Non-indexed events

No parameters are indexed in the events of the `DaoStakeLocking` contract. CHAINSECURITY recommends to index the relevant event parameters to allow DIGIX and its dApps to quickly search for these and simplify retrospective auditing.

```
event RedeemBadge(address _user);
event LockDGD(address _user, uint256 _amount, uint256 _currentLockedDGDStake);
event WithdrawDGD(address _user, uint256 _amount);
```

DaoStakeLocking.sol

Fixed: DIGIX indexed parameters of events as required, additionally more events were introduced.

Unnecessary loop iterations

The `ResolverClient.sender_is_from` method uses a loop to iterate over three addresses that are allowed to call certain methods. Currently this loop is implementing a logical OR. Hence, if an address is valid and once `_isFrom` is set to true, there is no need to continue iterating.

CHAINSECURITY recommends using a `break` when `_isFrom` is set to true to avoid unnecessary computations and allow the caller to save some gas costs.

Fixed: DIGIX introduced a `break`; after the `msg.sender` has been found, this avoids unnecessary iteration over the remainder of the loop.

Duplicate funding checks for preproposals

Preproposals can be distinguished by their initiator, who can be a normal user or a authorized DIGIX role. When a call to `submitPreproposal` in the `Dao` contract is made, the modifier `ifFundingPossible` takes funding details as arguments and checks whether the DAO can fund such a proposal. Later in the function body of `submitPreproposal` a call with the same parameters is made to `checkNonDigixFundings`, where depending on the check `is_founder` different funding requirements are verified.

To avoid separate checks and the entailed function call overhead, the modifier and function can be combined to a single modifier verifying the funding by differentiating on `is_founder`.

In an iterated review DIGIX stated that both checks are necessary, which is definitely true. Nonetheless, the two checks can be performed by the same modifier or function receiving different arguments.

Fixed: DIGIX removed the `isFundingPossible` modifier and moved the logic to a single `require` in the `submitPreproposal` function, since this is the only place where this check needs to happen.

Unnecessary calculations

The `calculateAdditionalLockedDGDStake` function of the `DaoCalculatorService` contract sometimes is called with the value `0` as an argument for `_additionalDgd`, e.g. from the `confirmContinuedParticipation` function of the `DaoStakeLocking` contract. When `_additionalDgd` is zero, it is expected that the returned result will always be zero as well.

CHAINSECURITY recommends to add a check for passed `0` values and in such cases return `0` immediately without doing any further calculations, which will reduce gas consumption.

Fixed: DIGIX's updated implementation only calls `calculateAdditionalLockedDGDStake()` if `_amount > 0`.

Broad function visibility

Many contract interactions happen in the DIGIX system. In order to allow cross-contracts calls but restrict them to allowed roles, `require(sender_is (SOMECONTRACT))` is often used.

However, for functions that are expected to be called from other contracts only, the visibility can be restricted to `external` instead of `public`. This allows to save gas costs, as `public` functions copy array function arguments to memory. A list of functions where such savings would be possible can be found below.

- `DaoStorage`
 - `readVotingCount`
 - `readVotingRoundVotes`
 - `readDraftVotingCount`
 - `readVotingRoundVotes`
 - `changeFundings`
 - `addProposal`
 - `editProposal`
- `Dao`
 - `submitPreproposal`
 - `modifyProposal`
 - `changeFundings`
 - `founderCloseProposals`
- `DaoSpecialStorage.readVotingCount`
- `DaoSpecialProposal.createSpecialProposal`
- `DaoConfigsStorage.updateUintConfigs`

Fixed: DIGIX implemented the recommendation as suggested wherever required.

Closed proposals can be reclosed

The founders are allowed to close an already closed proposal by calling `founderCloseProposals` method and providing the same `proposalId` again. This happens as no checks on the proposal status are present and the return value of `remove_item` which is called from `closeProposalInternal` is not checked. This return value would indicate whether the proposal was in the list of current proposals or not, before adding it to the closed list again.

Fixed: `founderCloseProposals()` now checks the status of the proposal first and closes only proposals which have not yet been finalized or closed, this implies proposals with a current state of either `PROPOSAL_STATE_PREPROPOSAL` or `PROPOSAL_STATE_DRAFT`.

CHAINSECURITY wants to make DIGIX aware that there are several non-deterministic test cases that eventually fail, specifically in `DaoRewardsManager.js`. Most notably these are the following tests:

`Q[2]/DaoRewardsManager.js` sometimes fails due to the assertion: `assert.deepEqual(pointsAfter[i], bN(calculatedReputation [i]))` and sometimes throws.

Note that the behavior is not deterministic, and the test may need to be run multiple times for it to actually throw. CHAINSECURITY was unable to establish if it is an error in the contract suite or in the test set. We remark that the failure below indicates a higher rounding error than $1e-9$. Example of a failure:

```
1) Contract: DaoRewardsManager
   updateRewardsAndReputationBeforeNewQuarter
     [Q2]:

  AssertionError: expected { Object (s, e, ...) } to deeply equal { Object
    (s, e, ...) }
  + expected - actual

    {
      [
        -      614080
        +      614060
      ]
      e: 5
      s: 1
    }
  }
```

Another case of non-deterministic behavior has been identified in: `[Q3 and Q4]/DaoRewardsManager.js`. This test case throws rarely, but eventually does.

```
1) Contract: DaoRewardsManager
   updateRewardsAndReputationBeforeNewQuarter
     [Q3 and Q4]:

  AssertionError: expected 44 to deeply equal 73
  + expected - actual

  -44
  +73
```

And:

```
1) Contract: DaoRewardsManager
   claimRewards
     [claimable dgx > 0]: success:
  Error: VM Exception while processing transaction: revert
```

Also the test case "daoRewardsManager already has some dgx unclaimed form previous quarter":

```
3) Contract: DaoRewardsManager
   calculateGlobalRewardsBeforeNewQuarter
     [daoRewardsManager already has some dgx unclaimed from previous
       quarter]: verify quarter info:

  AssertionError: expected { Object (s, e, ...) } to deeply equal { Object
    (s, e, ...) }
  + expected - actual

    {
      "c": [
        -      1626
```

```

+   3
-   ]
+   "e": 3
+   "e": 0
+   "s": 1
-   }

```

Furthermore the test script `DaoRewardsManager-140participants.js` fails to run for CHAINSECURITY since the updated code was received. While a full and in-depth investigation of the true reasons of the failures are out of scope of the audit review and cannot be performed within these time frames, CHAINSECURITY raises awareness for these as they may point to further underlying issues.

Fixed: DIGIX acknowledges that some test cases are failing and states this is most likely because the tests run slower than expected and hence some assumptions about the phase timings are wrong. CHAINSECURITY has remaining doubts about these failing test cases. Test values used in `DaoRewardsManager.js` are randomly initialized in `setMockValues()`. A failing test case can be reproduced by setting the values to:

```

const mockStakes =
  [8720663076,13404752094,2497495611,40609592851,38351005242,29758583424];
const mockModeratorStakes =
  [34796176485,2758928230884,507613499037,946687842902];
const mockQPs = [8,7,9,9,6,9,3,5,1,4];
const mockModeratorQPs = [2,5,2,0];
const mockRPs = [8300,10700,12000,10100,9800,6000];
const mockModeratorRPs = [1654100,310100,574100,1021100];

```

Note that the times for the phase durations do not matter, even if they are increased as suggested to resolve the problem of slow running tests, the output always remains the same:

```

1) Contract: DaoRewardsManager
   updateRewardsAndReputationBeforeNewQuarter
     [Q2]:

AssertionError: expected { Object (s, e, ...) } to deeply equal { Object
  (s, e, ...) }
+ expected - actual

{
  "c": [
-   1654205
+   1654199
  ]
  "e": 6
  "s": 1
}

at a.map (test/interactive/DaoRewardsManager.js:260:18)

```

Upon a final review DIGIX identified the issues in the random initialization of moderators' stakes in the mock files used for testing and could fully resolve the issue, concluding it was inherent to the test case, not the contracts.

Inefficient DAO storage Acknowledged

The `DaoConfigsStorage` contract updates extremely big lists of configuration flags one at a time, which is very costly and happens in the constructor and the function `updateUintConfigs`.

While this operation is below the block gas limit, as is reading the configuration with `readUintConfigs`, there is a possibility of receiving out-of-gas exceptions when executing `updateUintConfigs()` if this method is being called from other contracts during iterative operations.

The new contract creation in itself is a costly operation and makes for a long constructor. Hence, DIGIX can consider to move the configuration settings in to an `init` method, which should be called only once.

Acknowledged: DIGIX is aware of this and states: "We have chosen to keep the long constructor, as it still fits the gas limit and will not grow any bigger for this set of DIGIXDAO contracts. In case of `updateUintConfigs`, it is called only once when Special Proposals pass, and this transaction is still within the gas limit, so we will keep it as it is."

Missing input validation

It is possible to call the functions

- `DaoSpecialVotingClaims.claimSpecialProposalVotingResult()`
- `DaoVotingClaims.claimDraftVotingResult()`
- `DaoVotingClaims.claimProposalVotingResult()`

with the argument `_operations = 0`. Even though only the proposer can successfully perform such a call and there is no reason for him to do so, an accidental mistake can be avoided. Such a call would lead to unintended behavior like the underflow mentioned 5.3. CHAINSECURITY suggest to introduce appropriate checks.

Fixed: DIGIX introduced the recommended checks.

Old compiler version

DIGIX uses older compiler versions in its code, mostly 0.4.24 but some contracts. e.g. `ContractResolver`, use the even older version 0.4.19. Without a documented reason, the latest version (0.4.25) should be used homogeneously in all contracts. Given that the latest release was a bug fix only release¹³ and DIGIX might be exposed to previous compiler flaws through the examples below, a upgrade is strongly recommended.

```
uintConfigs[CONFIG_MINIMUM_LOCKED_DGD] = 10 ** 9;  
uintConfigs[CONFIG_MINIMUM_DGD_FOR_MODERATOR] = 100 * (10 ** 9);
```

DaoConstants.sol

Fixed: DIGIX updated all files to compiler version 0.4.25.

Underspecified ownership structure

In the DIGIX system there are the roles given in the specification (e.g. Participants, Moderators, Founders, PRL, Root, and KYCadmin). However, there are also so-called "groups".

The difference between "roles" and "groups" is not well-defined, neither by code comments nor in accompanying material. More so, the exact powers of the root of a group are not clear. Seemingly even more groups are present, as the contracts differentiate between the groups `nsadmins`, `uladmins` and `admins`.

The different access control roles/groups are not specified to allow a clear understanding to potential users of the DAO, who have to rely on these roles and their powers to maintain the system and need to trust those to not be malicious. This trust is especially high in the current case, where documentation is not available.

Addressed: DIGIX specifies: "The `cdap` library allows for creating multiple groups in a certain role. However, our DAO contracts only needs to keep track of the roles, hence we just create one dummy group for each role and hence the group doesn't have much significance." Further the `ContractResolver` in `cacp-contracts` has been simplified to only have one owner, whose role is only to deploy the DAO contracts before locking it. CHAINSECURITY recommends to additionally review the table in the system overview.

¹³ <https://blog.ethereum.org/2018/09/13/solidity-bugfix-release/>

Suboptimal struct Proposal

The DIGIX system needs access to a lot of information at different stages in time and makes heavy use of structs. As the Solidity compiler does not perform optimizations as one might expect¹⁴ it is necessary to ensure that structs are tightly packed by ordering them to align with 32 byte words. CHAINSECURITY identified an opportunity to do so and recommends DIGIX to adopt these to save significant gas costs.

Reordering below saves 5209 gas on transaction costs and also the same on execution costs:

```
struct Proposal {
    bytes32 proposalId;
    bytes32 currentState;
    uint256 timeCreated;
    DoublyLinkedList.Bytes proposalVersionDocs;
    mapping (bytes32 => ProposalVersion) proposalVersions;
    Voting draftVoting;
    mapping (uint256 => Voting) votingRounds;
    uint256 collateralStatus;
    uint256 collateralAmount;
    bytes32 finalVersion;
    PrlAction[] prlActions;
    address proposer;
    address endorser;
    bool isPausedOrStopped;
    bool isDigix;
}
```

DaoStructs.sol

Fixed: DIGIX adapted the proposed changes.

TODOs in code

Several contracts, concretely DaoConfigsStorage, DaoVotingClaims and DaoCommonMini, still contain TODO code comments indicating that certain functionality is not fixed. CHAINSECURITY remarks that no TODO clauses should be contained within a contract to be deployed and recommends to carefully review these cases before deployment.

Fixed: DIGIX resolved outstanding TODOs wherever required.

¹⁴ <https://solidity.readthedocs.io/en/latest/miscellaneous.html#layout-of-state-variables-in-storage>

Recommendations / Suggestions

- ✓ Dao.sol uses

```
require(address(daoFundingManager()).call.value (msg.value)())
```

to transfer ETH submitted with a proposal. While the DaoFundingManager contract is a trusted role, the call still forwards all gas. In this concrete case, the funding manager contract's fallback function is triggered, which does nothing more than increase the DaoFundingStorage's balance. Hence, as an additional security measure, the gas stipend to that call could be limited¹⁵.

- ✓ DaoPointsStorage.subtractReputation() is reducing the ReputationPoints of a user. The method is designed in such a way that if a user has X ReputationPoints and subtractReputation is called with Y points, such that the latter is bigger than the former, then ReputationPoints is set to zero, otherwise $X = X - Y$ is calculated. CHAINSECURITY recommends to rename the method to reduceReputation and addReputation() to increaseReputation().
- ✓ DIGIX has implemented a complex voting processes, which consist of several stages on the proposal object. At each stage different functions can be called by different users of the ecosystem. Given the complexity of these procedures a lifecycle/state diagram should be made to complement the documentation. This will help future developers, users and further clarify the design.
- ✓ As per the definition of Dao.setStartOfFirstQuarter() method, it should be called only once. However, when the value for the _start argument is set to zero this method is allowed to be called multiple times. CHAINSECURITY recommends adding a check to ensure that _start is always greater than zero. This would actually enforce the defined specification.
- Several structs can be further improved by reordering their fields to introduce minor gas savings. These are SpecialProposal, IntermediateResults and QuarterRewardsInfo.
- ✓ The ACGroups contract implements the two functions add_user_to_group and delete_user_from_group, which take as a parameter the argument bytes32 _group. A simple sanity check for inequality to zero bytes can be introduced.
- ✓ The DaoVoting contract uses the variable bool _voteYes to consider yes and no votes for the voteOnDraft function. This is confusing and the variable should be renamed to vote.
- ✓ CHAINSECURITY notes that DIGIX should document that calculateGlobalRewardsBeforeNewQuarter must be called in the DaoRewardsManager first during the LockingPhase of each quarter. Without calling this function successfully, many other activities like locking/withdraw of DIGIX TOKENS cannot be done.
- ✓ In DaoContracts.register_contract() an additional check could be introduced to verify if accidentally a zero address or a already registered contract is registered.
- ✓ The visibility modifier for a function should come before any custom modifiers¹⁶. This is violated in:
 - DaoFundingManager contract, fallback function
 - DaoRewardsManager contract, calculateGlobalRewardsBeforeNewQuarter
- ✓ quarterIndex is used to represent the number of the current quarter. Since it is called a index, it should start with a zero, however quarterIndex starts with a one. CHAINSECURITY recommends to either change the naming to quarterNumber or the quarterIndex should start with a zero.
- ✓ The Dao contract mentions that the proposer has to send in a collateral to open a new pre-proposal. However, the corresponding variable is referred to by DEPOSIT. Given that there is no documentation clarifying that the collateral and deposit refer to the same value, the naming should be revisited.

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

¹⁵ <https://solidity.readthedocs.io/en/latest/security-considerations.html#sending-and-receiving-ether>

¹⁶ <https://solidity.readthedocs.io/en/latest/style-guide.html#function-declaration>

Disclaimer

UPON REQUEST BY DIGIX, 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..