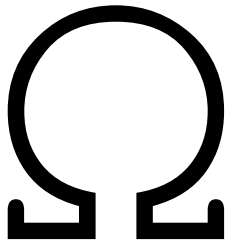
DXdao dxGovernance Audit

Final Audit Report

May 30, 2022



Team Omega

Summary

DXdao has asked Team Omega to audit the contracts that define the behavior of the DXdao dxGovernance contracts.

We found 2 high severity issues - these are issues that can lead to a loss of funds, and are essential to fix.

We classified 5 issues as “medium” - these are issues we believe you should definitely address.

In addition, 19 issues were classified as “low”, and 28 issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Scope of the Audit

The audit was requested for the contracts committed here:

`https://github.com/DXgovernance/dxdao-contracts/commit/b7eb732c0c3c81ea53ed928a3077d71781941b50`

And concerns the following contracts:

```
./contracts/dxvote/PermissionRegistry.sol
./contracts/dxvote/WalletScheme.sol
./contracts/dxvote/DXDVotingMachine.sol
./contracts/dxvote/utils/DXDVestingFactory.sol
./contracts/dxvote/utils/DXdaoNFT.sol
./contracts/daostack/controller/Avatar.sol
./contracts/daostack/controller/Controller.sol
```

Resolution

DXdao subsequently addressed a number of issues in the following commit:

`https://github.com/DXgovernance/dxdao-contracts/commit/54410e27dc25bb580d5fb99b60509b0d38889098`

We reviewed the changes, and described the resolution of each issue below.

Severity	Number of issues	Number of resolved issues
High	2	2
Medium	6	3
Low	19	14
Info	28	11

Methods Used

Code Review

We manually inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment.

Automatic analysis

We have used static analysis tools to detect common potential vulnerabilities. No high severity issues were identified with the automated processes. Some low severity issues, concerning mostly the solidity version setting, arithmetic operations, and functions visibility, were found and we have included them below in the appropriate parts of the report.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc
Info	Matters of opinion

Findings

General

G1. Observations about the general state of the repository and code [info] [partially resolved]

On the whole, we found the repository and the code in a relatively unfinished state, for example:

- The in-line documentation contains many errors - this concerns typos but also wrong or misleading comments.
- The compiler and the linter report warnings that can easily be fixed.
- There are duplicated contracts, for example the PermissionRegistry.sol and GlobalPermissionRegistry.sol, which are very similar.
- Continuous integration errors are ignored.
- Test coverage is incomplete

Although none of these observations directly concern the bytecode that is eventually deployed, addressing these concerns will help to avoid mistakes, and it will make it easier to guarantee and maintain the quality of the code.
Recommendation: Address the issues listed.
Severity: Info
Resolution: This issue was partially resolved: documentation was improved, most linter warnings (and all that concerns the contracts audited) were handled, duplicated contracts were removed, and test coverage is improved but still incomplete.

G2. Contracts are used from different sources [low] [partially resolved]

Some contracts and interfaces throughout the codebase are imported from different sources:

- SafeERC20.sol
- VotingMachineCallbacksInterface.sol
- ProposalExecuteInterface.sol

Duplicate code can lead to inconsistencies and unclarity in the code used.
Recommendation: Use only a single source file through the whole codebase for each contract.
Severity: Low
Resolution: This issue was partially resolved. SafeERC20.sol is still used from 2 different sources, but the other files now come from a single source.

G3. Continuous Integration tests fail [low] [partially resolved]

The installation step in github actions fails, and so tests are not run on github - cf. https://github.com/DXgovernance/dxdao-contracts/actions
Recommendation: Fix the continuous integration. Configure github so that no PRs can be merged unless all tests pass.
Severity: Low
Resolution: This issue was partially addressed but not resolved - the linter process fails but github actions pass all the same

G4. Mark functions as external where possible [low] [partially resolved]

There are a number of functions in the contracts that are marked as public, but are never called internally. You can improve readability and save some gas by marking these functions as external instead of public.

Examples are:

- `transferOwnership` and `setTimeDelay` in PermissionRegistry.sol
- create in DXDVestingFactory.sol
- `getBoostedVoteRequiredPercentage` in DXDVotingMachine.sol

Recommendation: Mark functions that are not used by the contract internally as external.
Severity: Low

Resolution: This issue was partially resolved. The mentioned functions, with the exception of `setTimeDelay` (now called `setPermissionDelay`), are now marked external.

G5. Do not use deprecated compiler versions [info] [not resolved]

The hardhat configuration file installs the following solc compiler versions: 0.4.25, 0.5.17, 0.6.8, 0.7.6 and 0.8.8. The contracts under review are using the 0.5.17 version, which is the latest in the 0.5.* series, and so is recommended to be used. However, you should be aware that the version 0.8.8 (which is used by some of the contracts in the repository) is deprecated, and should be upgraded.
Severity: Info - these compiler versions are not used for the contracts under review
Recommendation: Use the latest 0.8.* compiler version. And remove the compiler versions that are not used from the hardhat configuration file.
Resolution: This issue was not resolved, although now more contracts are using the 0.8.8 version.

G6. Deploy script fails on local network [info] [not resolved]

Running the deploy command as written in the README:

```
yarn hardhat run --network hardhat scripts/deploy-dxvote.js
```

on the local hardhat network fails, which makes testing the script more difficult.
Recommendation: Fix the deployment script for the local hardhat network.
Severity: The above-mentioned command now works, but the README has been changed and now instructs the user to run a non-existent script.

G7. Deploy script is not tested [info] [partially resolved]

The correct functioning of the deployment script “scripts/deploy-dxvote.js” is essential for the security and functioning of the system. As mentioned in issue G7, the deployment script is currently not working on the local hardhat network, which would probably have been caught if the script was tested.
Recommendation: Add tests for the deploy script and run it as part of the continuous integration process.
Severity: Info
Resolution: This issue was partially resolved. There’s now a test running the deploy script but with no verification of the results which significantly limits the utility of the test, and the test is not run as part of the Continuous Integration process.

G8. Use a locked pragma [info] [not resolved]

The Solidity pragma version used in some of the contracts is set as floating: pragma solidity ^0.x.x instead of being a fixed version. This will make the compilation step more deterministic, and make it easier for third parties, or yourself, to verify the deployed bytecode. E.g. Avatar.sol declares:

```
pragma solidity ^0.5.4;
```

But is actually compiled, with the current hardhat settings, by solc version 0.5.11
Recommendation: Remove the ^ symbol in the pragma definition to make the solidity version fixed and ensure bytecode consistency.
Severity: Info
Resolution: This issue was not resolved.

G9. Command for running gas report does not work [info] [resolved]

Running, as the README instructs, the "`ENABLE_GAS_REPORTER=true yarn test`" command does not output a gas report.
Recommendation: Fix the command or the instructions.
Severity: Info
Resolution: This issue was resolved as recommended.

G10. Coverage is not part of the continuous integration flow [info] [resolved]

The continuous integration does not run the coverage check. This is recommended to ensure that no new code is being added without proper testing.
Recommendation: Add coverage check to the continuous integration flow to check coverage level of new PRs is not lower than the existing level.
Severity: Info
Resolution: The coverage command is now run in github actions

G11. Coverage is not complete [info] [not resolved]

The contracts in the "daostack" directory are excluded from coverage by the settings.

The contracts in the dxvote directory are not fully tested – specifically, the coverage of DXDVotingMachine.sol is incomplete.

all files dxvote/

95.38% Statements 248/260 **81.18%** Branches 138/170 **96.77%** Functions 30/31 **95.75%** Lines 248/259

File		Statements		Branches		Functions		Lines	
DXDVotingMachine.sol		89.38%	101/113	68.92%	51/74	92.86%	13/14	90.27%	102/113
PermissionRegistry.sol		100%	47/47	88.89%	32/36	100%	6/6	100%	43/43
WalletScheme.sol		100%	100/100	91.67%	55/60	100%	11/11	100%	103/103

Specifically, what is not tested in DXDVotingMachine.sol is:

- Many checks on input variables
- The `voteOnBehalf` logic in the vote() function
- The `proposalStatusWithVotes` function
- The `else` case in the `payable` function
- Many transitions in the state machine defined by the `_execute()` function

Recommendation: Try to have a complete test coverage - i.e. include the contracts in the daostack directory in the coverage report, and add tests for the DXDVotingMachine, particularly for the _execute() function, which is complex, crucial for the correct operation of the system, and has been changed wrt the original Daostack contracts.
Severity: Info
Resolution: This issue was not resolved.

G12. Compilation warnings [info] [partially resolved]

The compiler reports a number of warnings:

```
contracts/daostack/votingMachines/DXDVotingMachineCallbacks.sol:22:68:
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
    function mintReputation(uint256 _amount, address _beneficiary,
bytes32 _proposalId)

^-----------------^
contracts/daostack/votingMachines/DXDVotingMachineCallbacks.sol:30:68:
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
```

```
    function burnReputation(uint256 _amount, address _beneficiary,
bytes32 _proposalId)

^-----------------^
contracts/daostack/votingMachines/DXDVotingMachineCallbacks.sol:42:9:
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
        bytes32 _proposalId)
        ^-----------------^

contracts/daostack/votingMachines/DXDVotingMachineCallbacks.sol:50:58:
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
    function balanceOfStakingToken(IERC20 _stakingToken, bytes32
_proposalId) external view returns(uint256) {
contracts/dxvote/WalletScheme.sol:286:35: Warning: Unused local variable.
                    (address _to, uint256 _) =
                                  ^-------^

Warning: SPDX license identifier not provided in source file. Before
publishing, consider adding a comment containing
"SPDX-License-Identifier: <SPDX-License>" to each source file. Use
"SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please
see https://spdx.org for more information.
--> contracts/dxvote/utils/DXdaoNFT.sol
```

Recommendation: Fix the warnings that concern unused function parameters by removing the parameter names (but not the function argument) and add an SPDX license identifier to the DXdaoNFT.sol file.
Severity: Info
Resolution: This issue was resolved in all contracts except for DXDVotingMachineCallbacks.sol.

G13. Eslint configuration is invalid [info] [partially resolved]

Running “yarn lint” throws an error:

```
~/omega/dxdao/dxdao-contracts (develop)> yarn lint
yarn run v1.22.17
$ eslint .
Error: /Users/jelle/omega/dxdao/dxdao-contracts/.eslintrc.json:
      Configuration for rule "max-len" is invalid:
      Severity should be one of the following: 0 = off, 1 = warn, 2 = error
(you passed '"warning"').
```

Recommendation: Fix the eslint configuration file.
Severity: Info

Resolution: This issue was partially resolved. The lint command now works, but it still seems to be incorrectly configured for the project as the linter returns 2335 errors and warnings.

G14. Add SPDX license identifier to all files [info] [partially resolved]

Some of the solidity files in the repository contain the following line which helps to automatically identify the license under which the file was published:

```
// SPDX-License-Identifier: AGPL-3.0
```

None of the files under review contain an SPDX license identifier.
Recommendation: Add the license identifier to all solidity files.
Severity: Info
Resolution: This issue was partially resolved. Some files, but not all, now include the SPDX license identifier.

DXDVotingMachine.sol

The DXDVotingMachine is an adaptation of the DAOstack contracts GenesisProtocol.sol and GenesisProtocolLogic.sol, that define the decision logic. The most important changes with respect to the DAOstack contracts are:

- Logic to refund users for the transaction costs incurred on voting
- Logic that allows users to sign a vote off-chain and have the vote submitted by a third party
- Logic that allows users to signal a vote on-chain without actually casting the vote (the vote can then subsequently trustlessly cast by a third party)
- Changes in the logic of the state machine that defines the various state transitions that a proposal goes through before being accepted or rejected. This logic is defined in the _execute()) function. The main changes in the execution procedure are:
    - A new setting, “boostedExecutionBar”, is added, that defines the minimal amount of votes that is needed to decide if a proposal that is in the “Boosted” or “QuietEndingPeriod” is considered to be accepted (previously, a simple relative majority sufficed for a boosted proposal to be accepted)
    - The time during which a proposal is in a “Boosted” state is slightly shortened, as the time is not being counted from the moment that the proposal changes its state on-chain, but from the time that the PreBoosted period expired.
    - A third change is described below in issue V2 - we believe it to be a bug

V1. Anyone can steal and manipulate signaled votes [high] [resolved]

The `executeSignaledVote` function accepts a `voteDecision` and `amount` parameters and uses these values to register the vote. The vote decision and amount that were signaled by the original reputation holder are ignored. The function is callable by anyone, so if a voter signals a vote, an attacker can call `executeSignalVote` to override the values for `voteDecision` and `amount`, and so effectively hijack the vote.
Recommendation: Instead of accepting `voteDecision` and `amount` as function parameters in the `executeSignalVote` function, use the values saved in the `votesSignaled` array for counting the vote.
Severity: High
Resolution: This issue was resolved as recommended.

V2. The counter for boosted proposals is not updated when a boosted proposal times out [high] [resolved]

In lines 508ff, the value of `orgBoostedProposalsCnt` is decreased in case the execution state of the proposal is `BoostedBarCrossed`. However, the counter is not diminished in case a Boosted proposal timed out before reaching the required number of votes (i.e. when the execution state is equal to `BoostedTimeOut`). This means that the counter is not a correct representation of the actual state of the state machine - the actual number of boosted proposals will be smaller than what the counter indicates. As there are no other ways to diminish the counter, this error is irrecoverable.
Recommendation: Also decrease the counter in case of `BoostedTimeOut`, as it done in the original code https://github.com/daostack/infra/blob/master/contracts/votingMachines/GenesisProtocolLogic.sol#L597
Severity: High. The `orgBoostedProposalsCnt` regulates the amount of stakes needed to reach the boosted state, and is used to determine when the ceiling of `MAX_BOOSTED_PROPOSALS` is hit, making it ever more difficult to boost a proposal, conceivably halting the operation of the DAO.
Resolution: This issue was resolved as recommended.

V3. ETH sent from a caller with unset organizationRefunds parameters will be stuck [low] [resolved]

If an account sends ETH to the contract, but is not registered in the `organizationRefunds` mapping, their ETH will be stuck in the contract.
Recommendation: Add an else clause to the if condition in the fallback function that reverts the transaction to ensure no ETH gets stuck in the contract by mistake.
Severity: Low

Resolution: This issue was resolved as recommended.

V4. Do state changes before external calls [low] [resolved]

On line 270 there is a transfer call to the `msg.sender` before the state change of decreasing the organization balance occurs (line 271). This is not a vulnerability by itself as the transfer call limits the amount of gas that the transferee can use and so protects from re-entrancy, but it is against the best practice, which is to do state changes before external calls, and could potentially cause issues if the transfer call is substituted with a call value in the future.
Recommendation: Do state changes before doing external calls, which in this case would mean moving line 270 to be after line 275 where the state change ends.
Severity: Low
Resolution: This issue was resolved as recommended.

V5. No option for organizations to withdraw their refund balance [low] [resolved]

It is not possible for organizations to withdraw their unused ETH from their refund balance. The only way for them to use the ETH after depositing is paying it to voters as refunds. This could mean ETH that is not used for voting would be locked in the contract forever, which could cause a loss for an organization that, for example, wants to migrate to a different voting machine.
Recommendation: Add a withdraw refund balance function to allow organizations to withdraw their existing balance from the contract.
Severity: Low
Resolution: This issue was resolved as recommended.

V6. Best practices regarding SafeMath and multiply before dividing [low] [resolved]

One line 376, it reads:

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

This goes against best practices in two ways: the recommendation to do multiplication before division to limit the chance of rounding errors, and to use `SafeMath` for any multiplication to avoid overflow errors. This is true also for the `boostedExecutionBar` variable calculation in line 378.
Recommendation: Regarding the order of the arithmetic operators - it is important that in this case best practices are ignored and the division happens before the multiplication. This is because the amount of

reputation in the reputation contract is uncapped - and so multiplying `totalReputation` by a value greater than one could lead to an overflow. Throwing an error in case of overflow errors, which is what SafeMath does, is not an option here, because this would effectively make it impossible to execute any proposals at all. We recommend documenting this deviation from best practice here with a comment near line 376 and near line 378, so that this will not be accidentally "fixed" in future iterations of the contract.
Severity: Low
Resolution: This issue was resolved as recommended.

V7. refundVote signature can be simplified [low] [resolved]

The internal method `_refundVote` is always called with `msg.sender` as its second argument. The code can be simplified somewhat by removing the parameter and just reading `msg.sender` directly.
Recommendation: Remove the toAddress parameter as described.
Severity: Low (some gas can be saved)
Resolution: This issue was resolved as recommended.

V8. Unnecessary require statement in executeSignedVote and executeSignaledVote [low] [resolved]

The implementations of `executeSignedVote` and `executeSignaledVote` require that `voteDecision > 0`. However, the value of `voteDecision` is already checked in the `internalVote` implementation, so these statements can be omitted
Recommendation. Remove these unnecessary `require` statements.
Severity: Low (some gas can be saved)
Resolution: This issue was resolved as recommended.

V9. Incomplete check in input value voteDecision [low] [resolved]

On line 163 in `shareSignedVote`, it is required that `voteDecision > 0`. However, the actual requirement defined in `GenesisProtocol.internalVote` is

```
require(_vote <= NUM_OF_CHOICES && _vote > 0, "0 < _vote <= 2");
```

i.e. the vote value must be either 1 or 2.
Recommendation: Check also the maximum value of the `voteDecision`.
Severity: Low
Resolution: This issue was resolved as recommended.

V10. Refund for voters is not guaranteed [info] [not resolved]

If there is a refund configured in the organization, but the organization does not have enough ETH deposited in the DXDVotingMachine contract to cover the refund, then no refund will be given to a voter. This might be even worse if for example two users are voting at the same time but there is only enough ETH to refund one. In which case both will vote expecting a refund but only one will eventually get it.
Recommendation: One option would be to allow users to specify if they want to cancel the vote if there is no refund, and revert early in case they do and there is not enough ETH to refund them. Another option is to keep a record of all the votes that were not refunded and let them withdraw a refund when there is enough ETH in the contract.
Severity: Info
Resolution: This issue was not resolved.

V11. Amount of gas refund can be estimated instead of hard coded [info] [not resolved]

The amount of gas that is used by voting depends on the context and is highly variable. The amount that is refunded to a voter (or an executor of a vote) is fixed, and depends on the value of `voteGas`. So the amount of ETH that is refunded to the user will almost always be lower (or higher) than the actual amount of gas used. This limits the useability of the refund logic - user's will indeed pay less, but the cost of a voting transaction remains variable.
Recommendation: Consider estimating the actual amount of gas used in the transaction by comparing the value of `msg.gas` at the start of the transaction to the value at the end of the transaction instead of using a fixed value to determine the amount of gas that is refunded.
Severity: Info
Resolution: This issue was not resolved.

V12. Allow anyone to share a signed vote in shareSignedVote [info] [resolved]

The function `shareSignedVote` is used to throw an event with a signed message that represents a vote (from the signer of the message). This function is implemented in such a way that only the voter herself can share a signed vote. It seems more useful if a signed vote can be shared by any user and not just by the voter itself.
Recommendation: Consider refactoring the function and allow any account to share a signed vote.
Severity: Info
Resolution: This issue was resolved as recommended.

V13. Do state changes before external calls [low] [new]

In the function `withdrawRefundBalance`, ether is sent to the caller's address before the internal balance is updated. This is a typical pattern that can be exploited by reentry.
In this case, the external call uses solidity's native `transfer` function, which can consume only a very limited amount of gas and so makes re-entrency infeasible, so this is not does not represent a vulnerability with the current gas budgets.
Severity: Low
Recommendation: Although this does not represent a vulnerability at the moment, gas costs may change in future versions of the EVM, and we recommend to follow the best practices and update the balance before sending ether (i.e. invert the last two lines of the function).

V14. Code duplication from GenesisProtocolLogic [info] [new]

There are a number of functions that are copied from `GenesisProtocolLogic.sol` and slightly adapted. This duplication concerns the logic of certain essential functions such as `propose`, `stake` and `internalVote`.
Severity: Info
Recommendation: We strongly recommend to not duplicate the core code of the protocol, as it the code will become increasingly hard to maintain.

V15. Unnecessary require in withdrawRefundBalance [info] [new]

The `withdrawRefundBalance` function allows organizations that have funded the contact with ether to refund user's transaction costs, to withdraw their funds. It contains the following requirement:

```
require(organizationRefunds[msg.sender].voteGas > 0,
            "DXDVotingMachine: Address not registered in
organizationRefounds"
```

It is not clear why this requirement was added: the `msg.sender`'s balance can be positive also when the value of `voteGas` can be set to 0, and in any case the `msg.sender` can easily circumvent this requirement by setting `voteGas` to a non-zero value before calling `withdrawRefundBalance`.
Severity: Info
Recommendation: Remove the `require` statement. If instead you decide to keep it, fix the typo in the error message.

V16. Getter function for numOfChoices can be removed [info] [new]

The code contains a function `getNumberOfChoices` that is a simple getter function for the `numOfChoices` mapping. As `numOfChoices` is a public variable, the compiler will automatically generate a getter function, so defining a new function is not necessary.
Severity: Info
Recommendation: Remove the function, as it will save some gas on deployment.

PermissionRegistry.sol

P1. Disallowed permissions are identical to unset permissions in getPermission [medium] [partially resolved]

The `getPermission` function returns (0, 0) both in case the permission was never set and in case it was set as disallowed (where `isSet` would be true but `fromTime` and `valueAllowed` will both be 0). This means contracts calling this function cannot distinguish between a permission that was explicitly disallowed (i.e. set using `setPermission` with the `allowed` flag set to false) and a permission that was never set.
For example, if the permission for sending token X was set with `allowed` is false in the PermissionScheme, the WalletScheme will now allow unlimited transfers of token X, instead of disallowing all token X transfers, because of the way the requirement check on line 267 is implemented, while the expected behavior would be that these token transfers are not allowed at all.
Severity: Medium
Recommendation: There are different ways of resolving this problem - one way is to also return the value of `isSet` in the `getPermission` function. But, in the light of our observations in P2, we recommend overhauling the permission model altogether.
Resolution: This issue was partially resolved. The `getPermission` function still returns the same result for both unset and disallowed permissions, but at least with regards to the WalletScheme, it is now clear and consistent through the code that unset permissions are practically disallowed.

P2. Permission data structure is overly complex [low] [not resolved]

The PermissionRegistry is a contract that stores a set of permissions that are used by the WalletScheme contract. These permissions are interpreted (a) as a kind of whitelist of contracts and function signatures that can be called in individual transactions within a proposal and (b) as constraints on the amount of ether and tokens that can be transferred in a proposal.

The permissions are stored in a mapping that maps three addresses and a function signature *(asset, from, to, signature)* to an `amount`, *a* `startTime` and a boolean `isSet` value.

The permission system is not well documented, and we believe the implementation contains some bugs, which we have noted below, but as far as we can see the intention is more or less as follows.

- If `asset` is the zero address, and `from` is either the address of the wallet scheme itself, or of of the avatar, then a set permission allows the WalletScheme to execute the function given by `signature` on the contract found at the `to` address, but only if it does not send more than a combined `amount` of Eth in the entire proposal
- If `asset` is not the zero address, then `asset` is expected to be an ERC20 contract, and the `signature` must be equal to `ANY_SIGNATURE` (all other values are ignored). This signals that the WalletScheme is not allowed to send more than `amount` of tokens using calls to `transfer` or `approve`
- Variables `ANY_CONTRACT` and `ANY_SIGNATURE` can be used to set more generic permissions. There are precedence rules that define how permissions that use these special values interact with other permissions

There are some exceptions to these rules:

- The permission *(asset, avatar, walletscheme_address, ANY_SIGNATURE)* is used to limit the total amount of `asset` tokens that can be sent by the proposal to any address. These are however interpreted differently from the other permissions - see issue G2.
- Permissions of the form *(0x0, …, …, transfer)* or *(0x0, …, …, approve)* are ignored (i.e. it is not possible to limit the amount of ETH send with transfer or approve calls)
- Permissions of the form *(asset, .., .., signature)* are ignored if asset is different from 0x0 and signature is different from `ANY_SIGNATURE`

This system is complex, which makes it hard to predict the effect of setting certain permissions. For example, one may think that setting *(ERC20Token, avatar, to, ANY_SIGNATURE)* to `amount` is a guarantee that no more than `amount` tokens can be sent to or approved for the *to*Address. But if another permission of the form *(0x0, avatar, ERC20Token, increaseAllowance)* is set as well, we can send any amount of tokens using a call to `increaseAllowance` (which is implemented by many newer ERC20 tokens). Similarly, setting (*0x0, avatar, to, ANY_SIGNATURE)* to `amount` suggests that not more than *amount* Ether can be sent to the *to*Address. But there is no such guarantee - if another permission of the form (*to, avatar, someotheraddress, ANY_SIGNATURE)* is set, then the scheme can send any amount of Ether it wants to the *to* address with a call to `to.transfer(someotheraddress, 0)` transaction.

Recommendation: We recommend changing the data structure of the permission system and make a clear distinction between *constraints* (on the amount of ERC20 token and Ether that can be transferred)

and *permissions* (to call certain functions and/or certain contracts). For example, consider defining two separate mappings: constraints that define transfer limits that can be encoded as *(asset_address, to_address, max_amount)* and permissions can be encoded as *(contract_address, function_signature)*. We also recommend to clearly document any permission system that you implement.
Severity: Low
Resolution: This issue was not resolved.

P3. Missing check new owner address is not zero on transferOwnership [info] [resolved]

The `transferOwnership` function lacks a check that the address of the new owner of the contract is not the zero address. This check exists in the constructor, but is lacking in the `transferOwnership` function.
Recommendation: Add a check to the `transferOwnership` function that the address the ownership is being transferred to is not zero. Or inherit from OpenZeppelin's Ownable contract.
Severity: Info
Resolution: This issue was resolved as recommended. The contract now inherits OpenZeppelin's OwnableUpgradeable contract.

P4. Make an onlyOwner modifier instead of duplicating the require call [info] [resolved]

Throughout the code, there are 3 checks that the `msg.sender` is the owner, this could be converted to a modifier to save gas, avoid code duplication, and improve readability.
Recommendation: Make an `onlyOwner` modifier and use that in the relevant functions instead of duplicating the require call. Or, alternatively, use the OpenZeppelin's Ownable contract.
Severity: Info
Resolution: This issue was resolved as recommended. The contract now inherits OpenZeppelin's OwnableUpgradeable contract, and multiple checks that the caller is the owner have been removed.

P5. Unnecessary code duplication in setPermission and setAdminPermission [info] [resolved]

The `setPermission` and `setAdminPermission` are nearly the same, and could be united into a single function to avoid the code duplication that currently exists between them. This will make the code clearer, and will make it harder to make mistakes.
Recommendation: Unite the functions into a single function, with a proper handling of just the differences.
Severity: Info

Resolution: This issue was resolved as recommended.

P6. TimeDelay in setAdminPermission is not enforceable [info] [not resolved]

Because the owner of the contract can at any time set the timeDelay to 0 (and can do this without a time delay), setting the fromTime to a value timeDelay seconds after the current time should be taken as just a default value, as the owner can manipulate the timeDelay at will.
Recommendation: The behavior would be more explicit (and therefore safer) if the timeDelay was added as an explicit parameter to the setAdminPermission function
Severity: Info
Resolution: This issue was not resolved. In addition, the code was changed so that each address can set its own delay, which means it is practically unenforceable for anyone now, and not just the admin.

P7. Wrong require statement on setPermissions [info] [resolved]

On line 165ff, in `setPermissions`, it reads that

```
require(
      msg.sender != owner && to != address(this),
      "PermissionRegistry: Cant set owner permissions"
);
```

This effectively means that the owner can not use `setPermission` to set her own permissions on any address.
Recommendation: We believe that what was intended is a requirement that the owner does not set permissions on the `PermissionRegistry` itself, i.e. that the requirement should be the same as in `setAdminPermissions`:

```
require(!(msg.sender == owner && to == address(this)))
```

Severity: Info
Resolution: This issue was resolved as recommended. It is now explicitly disabled to set permissions on the PermissionRegistry address.

P8. Unenforced requirement that that timeDelay > 0 [info] [resolved]

In the constructor, it is required that the value of timeDelay is bigger than 0. However, in `setTimeDelay`, this is not required for the new value, so this requirement can easily be circumvented

Recommendation: Decide if the `timeDelay` is allowed to be 0. If so, enforce that in setTimeDelay as well. if not, remove the `require` statement from the constructor.
Severity: Info
Resolution: This issue was resolved as recommended. There is no longer a requirement that the timeDelay must be greater than 0.

P9. Token approve and transfer permissions are checked incorrectly [medium] [new]

The functions `getPermission` and `setPermissionUsed` ignore the case where `asset` is not equal to 0 and the `functionSignature` argument is a specific function signature. This means that if permissions are set that are specific to the `transfer` or `approval` of tokens are ignored, which could allow transfers or approvals of higher values than intended.
Severity: Medium
Recommendation: Add a function specific check when the asset is a token just like those in the checks of the ETH asset permissions.

P10. Users can prevent the owner from overriding their permissions [low] [new]

The owner has the ability to set and edit users permissions in the registry. However, since only the user can control their time delay, they could set their time delay high enough that calling the `setPermission` function, which does addition of the block timestamp with the time delay the user set, will always overflow and fail.
The admin will still be able to disable the permission, but not edit its value.
Severity: Low
Recommendation: Allow specifying the time delay while setting permission instead of saving a time delay beforehand. This will save gas and reduce code complexity while fixing this issue without changing the security assumptions.

P11. Avoid unnecessary checks in setPermissionUsed [info] [new]

In `setPermissionUsed`, if the asset is set to ETH, the permission will execute 4 if checks reading from storage. This could be optimized by re-writing the checks as if/ else clauses, just like in the `getPermission` function.
Severity: Info
Recommendation: Use the same checks as written in the `getPermission` function. We also recommend avoiding code duplication and instead extract the common code into a helper function - cf P12.

P12. Avoid code duplication in getPermission and setPermissionUsed functions [info] [new]

The `getPermission` and `setPermissionUsed` functions share a significant part of their code, which could be avoided by moving the shared code to a utility function.
Severity: Info
Recommendation: Avoid code duplication and instead extract the common code into a helper function.

P13. getPermissionDelay can be removed [low] [new]

The function `getPermissionDelay` is not necessary, as the `permissionDelay` mapping is public, and the compiler will create a getter function automatically.
Severity: Low (some gas can be saved)
Recommendation: Remove the function.

P14. Forgotten "TO DO" statement [info] [new]

The following line should either be removed, or a removePermission function should be implemented.

```
 // TO DO: Add removePermission function that will set the value isSet in the permissions to false and trigger PermissionRemoved event
```

Severity: Info
Recommendation: Handle the commend and remove it.

WalletScheme.sol

W1. Limits on ERC20 token transfers are not enforceable [medium] [not resolved]

Some permissions are intended to define a limit on the total amount of ERC20 tokens that can be transferred in a single proposal. The implementation enforces these limits by extracting the function signature from the `calldata`, and if it is either a "`transfer`" or an "`approve`" call, it will extract the amount of tokens from the `calldata` and count those.
This implementation is problematic, as there are ways of `sending` and `approving` tokens that are different from these two methods. Such methods live within the ERC20 standard (such as `transferFrom`), but also other methods are commonly used (such as `safeTransfer`,

`increaseAllowance`). In addition, calls to transfer tokens could be triggered by calling other contracts. These kinds of transfers or approvals are not counted towards the maximum amount of tokens that are allowed to be transferred.
Recommendation: Instead of trying to determine from the function signature and its arguments how many tokens were sent (a task that is impossible as the ERC20 contract can define any number of functions that do token transfer), check the difference between the token balance and approvals before and after the transaction is executed.
Severity: Medium.
Resolution: This issue was not resolved.

W2. Inconsistent handling of permissions where fromTime is 0 [low] [resolved]

In line 267, permissions where `fromTime` is `0` are treated as unlimited permissions. However, in line 328, such permissions are treated as disallowed.
Recommendation: Distinguish between disallowed calls and unset calls as described in issue P2 and treat each case properly.
Severity: Low, but see P1 and P2.
Resolution: This issue was resolved. Permissions which have `fromTime` set to 0 are now strictly disallowed.

W3. assetsUsed array confuses ETH with address(0) [low] [resolved]

In the `assetsUsed` array that is used in the execute proposal function, `address(0)` is being used to represent ETH. Also when a token is used in the proposal more than once, the second occurrence would not be recorded in the array and so the value stays as the default which is the `address(0)`. This triggers unnecessary duplicated permission checks of ETH max value transfer in the permission registry in the loop on line 257ff.
Recommendation: Instead of using a fix-length array, use a dynamic sized array for `assetsUsed`, and only push addresses into that array if they are of assets that are actually used.
Severity: Low - some gas can be saved
Resolution: This issue was resolved. The array was removed from the code.

W4. Avoid using assembly [low] [resolved]

There are some places in the code where assembly is used for functionality that can just as well be done in solidity.
Recommendation: It is best practice to avoid assembly if equivalent solidity code can be used.

Replace the lines at line 549ff:

```
assembly {
            to := mload(add(_data, 36))
            value := mload(add(_data, 68))
      }
```

With:

```
(to, value) = abi.decode(_data[4:], "address,uint256")
```

Which will have the same effect, but will also do some type checking on the input values and the length of the input, which may help to avoid accidental errors by the caller of the function.

Similarly, the code:

```
bytes32 functionSignature = bytes32(0);
      assembly {
            functionSignature := mload(add(data, 32))
      }
return bytes4(functionSignature);
```

Can be replaced with:

```
return bytes4(data[:4])
```

Severity: Low - although the assembly calls used here are straightforward, it is still better to rely on the type checks from the solidity implementation.
Resolution: This issue was resolved as recommended.

W5. Unnecessary setting of empty value in an empty array [info] [resolved]

In line 231, an entry in an address array is being set to its default value of `address(0)`. Since it is certain that the array entry was already empty, there is no reason to set it to the default value like this.
Recommendation: Remove the assignment of empty value to an empty array entry in line 231.
Severity: Info
Resolution: This issue was resolved. The array is no longer used.

W6. Use of functionCall with obfuscate original error message [info] [new]

In several places in the Walletscheme contracts, direct calls to functions are replaced with openZeppelins `functionCall`, for example:

```
controller.mintReputation(_amount, _beneficiary, _address(avatar))
```

Is replaced with

```
controller.functionCall(
    abi.encodeWithSignature(
                  "mintReputation(uint256,address,address)",
                  _amount,
                  _beneficiary,
                  address(avatar)
    ),
    "WalletScheme: DXDVotingMachine callback mintReputation error"
),
```

This pattern will obfuscate any errors that may be thrown in the `mintReputation` code, and replace those with a generic error message. This will make it harder to debug the code or retrace any errors that users may report.
Recommendation: Call the functions directly when possible
Severity: Info - there will be some difference in gas costs

ERC20Factory.sol

This file was previously called DXDVestingFactory.sol

F1. Do not use “draft” contracts in production [medium] [not resolved]

The vesting factory deploys instances of OpenZeppelin’s TokenVesting contract:

```
import "openzeppelin-solidity/contracts/drafts/TokenVesting.sol";
```

This import is from version 2.4.0 of openzeppelin, which is an older version (the current version is 4.4.2). It was included as a draft contract there, and was subsequently removed from the Openzeppelin repository and was not included in future releases of the library.
Recommendation: Version 4.4.1 of OpenZeppelin introduced `VestingWallet.sol` contract, and we recommend you use that. See also issue G5, which discusses the OpenZeppelin versions used in the code.
Severity: Medium
Resolution: This issue was not resolved.

F2. Setting revocable to true in the VestingFactory does not have any effect [medium] [resolved]

In the creation of a TokenVesting contract instance, the flag for making the contract revocable is set to true. The effect of setting this flag to true is that the owner of the vesting contract can at any moment retrieve the unclaimed tokens from the contract. However, because the owner of the vesting contract is set to the account that deployed it, the owner will be the vesting factory, which has no functionality to call the revoke function. So no unclaimed token can be removed, whether the revoke flag is set to true or not.
Recommendation: Set the flag to "false" for clarity. Or avoid the issue altogether by using the newer OpenZeppelin VestingWallet contract, as recommended in F1.
Severity: Medium
Resolution: This issue was resolved. The ownership of the new vesting contract is transferred to an address that is set in the constructor of the Factory contract

F3. Use SafeERC20 for token transfers [low] [not resolved]

The ERC20 standard does not require that a call to transfer throws an error when the token transfer is unsuccessful. As a result - depending on the implementation of the ERC20 token - the "create" transaction might succeed, even if the token transfer in line 23 fails, but returns false instead of throwing an error. It is best practice to use the SafeERC20 library (as do other contracts in the repository) to ensure the transaction will revert if the token transfer fails.
Recommendation: Use the SafeERC20 library and its safeTransferFrom function for the token transfer in line 23.
Severity: Low
Resolution: This issue was not resolved. `SafeERC20` has been properly imported, but it is not used. Instead, the regular `transferFrom` function is still being used, instead of `safeTransferFrom`

ERC721Factory.sol

This file was previously called DXDdaoNFT.sol

N1. Use "_safeMint" instead of "_mint" [medium] [resolved]

On line 21, in the `mint` function, the `_mint` function of Openzeppelin's ERC721 implementation is called. As recommended by the ERC721 standard, it is better to use `_safemint`, which will check, if the recipient is a contract, if the recipient is able to handle ERC721 tokens. See

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L256
Recommendation: Use `_safemint` instead of `_mint`
Severity: Medium - the minted NFTs will be lost if sent to a contract that can not handle ERC721 tokens.
Resolution: This issue was resolved as recommended.

Avatar.sol

This Avatar.sol file is an almost verbatim copy of the Avatar.sol file the @daostack/arc repository, version 0.0.1-rc.57 (which was the latest official release of @daoastack/arc contracts). The differences are that the compiler version in the daostack release is set to 0.5.11, and that the dxDAO repository uses a local, adapted, version Reputation.sol rather than importing the file from the @daostack/infra repository. These differences are almost trivial and can safely be ignored (although see G9).

This contract was part of the audit by ChainSecurity in January 2019 https://github.com/ChainSecurity/audits/blob/master/ChainSecurity_DAOstack_v2.pdf . The version of Arc that was audited was subsequently released as 0.0.1-rc.7-2.
The contract was mostly unchanged since that date - changes were the addition of the `_value` parameter in `genericCall` and the addition of a `metadata()` function that can be used to generate an Event.

A1. Do not use “transfer” [info] [not resolved]

On line 69, in `sendEther`, the ether is sent with a transfer call:

```
_to.transfer(_amountInWei);
```

Transfer calls use a fixed amount of 2300 gas. It is possible that the costs of EVM instructions change in the future, as has happened in the past, and in that case, this operation may fail. See also: https://swcregistry.io/docs/SWC-134#hardcoded-gas-limitssol
Recommendation: Instead, use `_to.call.value(_amountInWei).("")` which is more future-proof
Severity: Info
Resolution: This issue was not resolved.

Controller.sol

This file is a verbatim copy of the Controller.sol file the @daostack/arc repository at version 0.0.1-rc.23.

This contract was part of the audit by ChainSecurity in January 2019 https://github.com/ChainSecurity/audits/blob/master/ChainSecurity_DAOstack_v2.pdf . The version of Arc that was audited was subsequently released as 0.0.1-rc.7-2.

The contract was mostly unchanged since that date - the changes regard those mentioned under Avatar.sol - the controller allows for passing a `value` argument in the wrapper function that calls `avatar.genericCall`, and adds a new `metadata` wrapper function that calls `avatar.metadata`. A new modifier, `onlyMetaDataScheme`, was added as well, which checks the same permission bit as `onlyGenericCallScheme` does.

C1. Update Controller.sol to the latest version of arc [low] [not resolved]

The latest “non-experimental” version of the @daoastack/arc contracts is 0.0.1-rc.57. The Controller.sol contract has been left mostly unchanged between version 23 and 57 - the differences are that the compiler version is updated, the separate ControllerInterface contract was removed, and a small bug was fixed by emitting a `RegisterScheme` event when a scheme is registered in the constructor.
Recommendation: If possible, update the contract to be (equal to) the latest version 0.0.1-rc.57 of @daostack/arc. In any case, fix the bug and raise the `RegisterScheme` event in the constructor, and also use a fixed solidity version as we recommend in G9.
Resolution: This issue was not resolved.