DXDAO

ERC20Guild

Smart Contract Security Review

Version: 2.2

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of a set of DXdao smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the DXdao smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the DXdao smart contracts.

Overview

DXdao is a decentralized organization powered by DAOstack's alchemy framework and composed of over 450 unique Ethereum addresses. DXdao develops, governs, and grows various DeFi protocols.

DXdao also developed a set of basic DAO and governance contracts called `guild`. A Guild contract converts its users' assets (tokens) into voting power for casting vote on one or more proposals. The vote is cast directly or through a proxy by providing a valid signature. A user needs to have a certain percentage of the total voting power in order to create a proposal. A proposal also needs a predefined percentage of voting power to pass.

A guild needs a `TokenVault` contract to store assets and a `PermissionRegistry` contract to manage callable functions during proposal execution and to control the quantity of assets that can be transferred during a function call.

Security Assessment Summary

This review was conducted on the files hosted on the DXdao repository and were assessed at commit 54410e2 (tagged v1.0.0). This commit contains an essential update from the original target commit of 27ae128.

The list of assessed contracts is as follows.

1. `BaseERC20Guild.sol`
2. `ERC20Guild.sol`
3. `ERC20GuildUpgradeable.sol`
4. `IERC20Guild.sol`
5. `DXDGuild.sol`
6. `EnforcedBinaryGuild.sol`
7. `EnforcedBinarySnapshotERC20Guild.sol`
8. `SnapshotERC20Guild.sol`
9. `SnapshotRepERC20Guild.sol`
10. `ERC20GuildWithERC1271.sol`
11. `PermissionRegistry.sol`
12. `TokenVault.sol`

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The fixes of the raised issues were re-assessed at commit 048aafe. In this new version, `EnforcedBinaryGuild.sol` and `EnforcedBinarySnapshotERC20Guild.sol` were removed and some contracts were heavily modified. Therefore, all issues related to removed codes are no longer applicable. It is also worth noting that the re-assessment was done exclusively on the part of the codes where the issues were raised and did not include refactored codes. Commit c142818 was assessed in relation to mitigating DXD-12.

The manual code review section of the report, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 27 issues during this assessment. Categorised by their severity:

- Critical: 6 issues.
- High: 5 issues.
- Medium: 6 issues.

- Low: 5 issues.
- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the DXdao smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
DXD-01	Denial of Service on `withdrawTokens()`	Critical	Resolved
DXD-02	Double-voting in `BaseERC20Guild`	Critical	Resolved
DXD-03	Double-voting in `SnapshotERC20Guild`	Critical	Resolved
DXD-04	Snapshot Malfunction	Critical	Resolved
DXD-05	Incorrect `votingPower` Accounting	Critical	Resolved
DXD-06	Voting Tally Does Not Count the Zero Action	Critical	Resolved
DXD-07	Denial-of-Service Using Failed Proposals	High	Closed
DXD-08	Low `totalLocked` May Result in Guild Takeover	High	Resolved
DXD-09	Duplicate Actions Unaccounted for During Voting Tally	High	Closed
DXD-10	Zero `totalLocked` Allows Everyone to Create Proposals	High	Resolved
DXD-11	Gas Refund on `setVote()` May Be Subject to Large Scale Manipulation	High	Resolved
DXD-12	Frontrunning Proposal Execution	Medium	Resolved
DXD-13	*Fee-on-transfer* Tokens Are Not Supported	Medium	Closed
DXD-14	Denial-of-Service using Proposals with Failed Calls	Medium	Closed
DXD-15	Denial-of-Service using Junk Proposals	Medium	Resolved
DXD-16	`PermissionRegistry` Owner Changes Permissions Set By Others	Medium	Closed
DXD-17	`totalMembers` Can Be Manipulated	Medium	Resolved
DXD-18	Voting Tally Results in Unexpected Winner	Low	Resolved
DXD-19	Early Withdrawal To Manipulate `getVotingPowerForProposalExecution()`	Low	Resolved
DXD-20	Cannot Receive ETH From `send()` or `transfer()`	Low	Resolved
DXD-21	Failing Proposal through `setPermissionUsed()`	Low	Closed
DXD-22	Incompatible Permissioning on `ERC20` Assets	Low	Closed
DXD-23	Potentially Confusing value on ERC20 Operation	Informational	Closed
DXD-24	Duplicated Checks on `createProposal()` and `endProposal()`	Informational	Closed
DXD-25	Input Length Not Checked in `votingPowerOfMultipleAt()`	Informational	Resolved
DXD-26	Non-upgradeable TokenVault	Informational	Resolved

DXD-27	Miscellaneous General Comments	Informational	Resolved

DXD-01	Denial of Service on `withdrawTokens()`		
Asset	`BaseERC20Guild.sol, SnapshotERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `withdrawTokens()` function withdraws locked user tokens from allocated guild contracts. This function fails to check whether a user tries to withdraw zero `tokenAmount`.

If a user has no balance and withdraws zero tokens, `totalMembers` will reduce by one. This can be seen in the following excerpt from `BaseERC20Guild.sol` as indicated in the following snippet[1]:

```
if (tokensLocked[msg.sender].amount == 0) totalMembers = totalMembers.sub(1);
```

A malicious user could launch a DoS attack against the system through repeatedly withdrawing zero tokens until `totalMembers == 0`. This would prevent other users from withdrawing tokens, due to the aforementioned check causing an `Integer Overflow` error.

Recommendations

Add an extra check to make sure users withdraw non-zero `tokenAmount`. Also, if `totalMembers` is not essential to the protocol, it can be removed from to minimize complexity and avoid pitfalls.

Resolution

The issue has been fixed on PR#179. A user cannot withdraw zero amount.

[1] The same contract logic persists in the `SnapshotERC20Guild.sol` for the same function `withdrawTokens()` on line [**105**].

DXD-02	Double-voting in `BaseERC20Guild`		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The contract `BaseERC20Guild` enables users to cast votes on proposals using their voting powers. Voting power is acquired by locking tokens through the `lockTokens()` function, where the tokens will be locked until `lockTime` period passes.

When a proposal is first submitted, the user can vote on this proposal by calling the `setVote()` function. Then, after casting their vote, the user can withdraw their tokens by calling `withdrawTokens()`. The token withdrawal does not revoke their proposal vote.

Having their tokens back, the user can transfer the tokens to another account and vote again under the guise of a new user. This allows for double-voting.

The scenario above is possible if the user locks tokens before a proposal is proposed, and therefore the `lockTime` expires before `proposalTime` passes. In this case, the system configuration where `_lockTime >= _proposalTime` does not prevent the user from withdrawing tokens before the `proposalTime` ends.

Recommendations

Make sure this behaviour is intended. The testing team acknowledges that some implementation contracts have snapshot mechanism to mitigate this issue. However, other contracts such as `DXDGuild`, `ERC20GuildWithERC1271`, and `EnforcedBinaryGuild` do not have similar protection.

Resolution

The issue has been fixed on PR#187. The locked tokens' `lockTime` will be extended when the users vote.

DXD-03	Double-voting in `SnapshotERC20Guild`		
Asset	`SnapshotERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The contract `SnapshotERC20Guild` enables users to cast votes on proposals using their voting powers. Unlike `BaseERC20Guild`, voting power is managed by a snapshot mechanism. One of the purposes behind most snapshot mechanisms is to prevent people flooding the voting power, after a proposal is created. In `SnapshotERC20Guild`, the snapshot process has unexpected behaviour:

1. `lockTokens()`: Calling this function does not increase proposal votingPower unless the user then calls `lockTokens()` again, or `withdrawTokens()`. This is due to calling `_updateAccountSnapshot(msg.sender)` on line [**83**], before updating the `tokensLocked[msg.sender].amount` on line [**86**].
2. `withdrawTokens`: Calling this function unexpectedly increases the `votingPower`. This is due to the function calling `_updateAccountSnapshot(msg.sender)` on line [**100**] before updating the `tokensLocked[msg.sender].amount` on line [**102**].

As a result, a malicious user is able to lock tokens, wait the allocated `block.timestamp`, and create a new proposal to increment the `snapshotId`. After this setup, they are able to call `withdrawTokens()`, which will withdraw the tokens from the guild contract, however the snapshot will maintain voting power as the pre-withdrawal `tokensLocked[msg.sender]`.

After withdrawal, the malicious adversary is able to transfer the tokens to another account and repeat the process, gaining `votingPower` across multiple accounts that do not hold any tokens. This effectively allows a single user to double vote repeatedly on any future proposal.

Recommendations

The testing team recommends that all tokens are appropriately locked during proposal voting and to ensure that `votingPower` at snapshot intervals indicate the correct amount actually held by that account. This could be achieved by placing the `_updateAccountSnapshot(msg.sender)` after `tokensLocked[msg.sender].amount` has been updated.

Resolution

The development team indicated that the issue has been fixed on PR#198.

DXD-04	Snapshot Malfunction		
Asset	`SnapshotERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In the contract `SnapshotERC20Guild`, function `setVote()` and `setSignedVote()` manage the users's votes by first checking the snapshot and then calling the parent contract's `setVote()` or `setSignedVote()` function.

The contract `SnapshotERC20Guild` derives from `ERC20GuildUpgradeable` (or `BaseERC20Guild`). While snapshots are used in `SnapshotERC20Guild`, the parent contract `ERC20GuildUpgradeable` does not have access to the snapshots. For example, `SnapshotERC20Guild.setVote()` calls the function `BaseERC20Guild.setVote()` which directly uses the current `votingPower` before processing further. The same error occurs with `SnapshotERC20Guild.setVote()` relying on `BaseERC20Guild.setSignedVote()`. These checks against the parent contract's `votingPower` do not take into account applied snapshots in the child contract.

If the voters withdraw their tokens after proposal creation and before voting, the parent contract will revert with `Invalid votingPower` message.

Recommendations

The testing team recommends refactoring the `votingPower` checks across parent and child contracts to allow `votingPower` checks to be performed only on child contracts. This will preserve `votingPower` data across different implementations, for example, through snapshots. Alternatively, the voting logic could be moved to the child contract to avoid calling the parent contract's functions.

Resolution

The issue has been fixed on PR#196. Function `setVote()` no longer calls `super.setVote()` and all checks are done on the `SnapshotERC20Guild` contract.

DXD-05	Incorrect `votingPower` Accounting		
Asset	`SnapshotRepERC20GUild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Contract `SnapshotRepERC20Guild` replaces token locking with a snapshot mechanism implemented in the `ERC20SnapshotRep` token. The guild contract derives from `ERC20GuildUpgradeable` or `BaseERC20Guild` and overrides some important functions to set votes and create proposals.

The contract however, does not override the `votingPower` accounting algorithm, for example `getTotalLocked()` function. This function is used to query the total amount of tokens locked by the contract. `totalLocked` is then used to determine the minimum amount of tokens required to submit a proposal and to pass a proposal as defined in `votingPowerForProposalCreation` and `votingPowerForProposalExecution` respectively.

As a result, anyone with an insufficient amount of tokens can propose and pass a proposal, because `getVotingPowerForProposalCreation()` and `getVotingPowerForProposalExecution()` will always produce zero.

Recommendations

The testing team recommends refactoring the `votingPower` accounting and making sure that `createProposal()` and `endProposal()` satisfy the required conditions for snapshot data.

Resolution

The issue has been fixed on PR#197. Function `getTotalLocked()` and `getSnapshotVotingPowerForProposalExecution()` are implemented in `SnapshotRepERC20Guild.sol`

DXD-06	Voting Tally Does Not Count the Zero Action		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The function `endProposal()` allows users to tally votes, and determine the winning action on a proposal after `block.timestamp` exceeds the proposal `endTime`. Voting is tallied using the following logic:

```
    uint256 winningAction = 0;
    uint256 i = 1;
  for (i = 1; i < proposals[proposalId].totalVotes.length; i++) {
    if (
      proposals[proposalId].totalVotes[i] >= getVotingPowerForProposalExecution() &&
      proposals[proposalId].totalVotes[i] > proposals[proposalId].totalVotes[winningAction]
    ) winningAction = i;
  }
```

This logic assumes that if enough votes are placed on `proposals[proposalId].totalVotes[0]` (no action) then the other votes will not meet the required `votingPowerForProposalExecution`. Therefore, the default `winningAction=0` will result in `ProposalState.Rejected`.

This assumption by `endProposal` is not enforced in `setConfig()` and can be deemed invalid under likely conditions. As a result, if `votingPowerForProposalExecution < 50` it is possible that the majority voted for `action=0`, but some minority (say 30 percent) vote for `action=1`. The tally logic may result in `action=1` being the winning action.

This could lead to a malicious user being able to vote for a proposal that the majority rejected. Results could lead to alterations in sensitive guild configurations using `setConfig`.

Recommendations

The testing team recommends tallying votes from `i=0`. Alternatively, if intended, consider adding the following condition in the function `setConfig()`

```
require(_votingPowerForProposalExecution >= 50, "required assumption by endProposal default voting tally");
```

Resolution

The issue has been fixed on PR#192. The `winningAction` of zero is now recognised, which indicates a rejected proposal.

DXD-07	Denial-of-Service Using Failed Proposals		
Asset	`EnforcedBinaryGuild.sol, EnforcedBinarySnapshotERC20Guild.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Contracts `EnforcedBinaryGuild` and `EnforcedBinarySnapshotERC20Guild` add a new action as a **NO** option on each submitted proposal, in order to indicate no action is preferred. When function `endProposal()` is called to end a proposal's lifecycle, if **NO** option wins the vote, then the function does not call `super.endProposal()`, or the derived `BaseERC20Guild.endProposal()`

As a result, the `activeProposalsNow` counter is not decreased, which should be done on line [**389**] of function `BaseERC20Guild.endProposal()`.

This behaviour can be weaponised by a malicious user with enough voting power to win a proposal and perform a DoS attack against the protocol. The malicious user spams the system with proposals and votes on the proposals with **NO** option, so the proposals will fill up the `activeProposalsNow` which will not allow anyone to create a new proposal.

Recommendations

The testing team recommends implementing a `BaseERC20Guild.endProposal()` feature in contract `EnforcedBinaryGuild` and `EnforcedBinarySnapshotERC20Guild` to mitigate this issue. This would also remove the redundant computation to determine the `winningAction` which is currently done twice if `winningAction != proposals{[}proposalId{]}.totalVotes.length - 1`.

Resolution

The issue is no longer applicable with the removal of the two contracts, `EnforcedBinaryGuild.sol` and `EnforcedBinarySnapshotERC20Guild.sol` on PR#192.

DXD-08	Low `totalLocked` May Result in Guild Takeover		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The contract `BaseERC20Guild` enables users to cast votes on proposals using their voting powers. This contract is inherited by all alternative implementations (ie `SnapshotERC20Guild`, `EnforcedBinaryGuild` etc) and the voting powers dictate critical decisions that impact the guild. A malicious user that can control proposal voting and tallying mechanisms could steal funds, prevent withdrawal of funds, or prevent proposals being submitted.

`BaseERC20Guild` relies on function `getTotalLocked()` in order to calculate `getVotingPowerForProposalCreation()` and `getVotingPowerForProposalExecution()`. Both these functions are critical for determining whether proposals can be created and executed.

If `totalLocked` tokens falls below a certain threshold, a malicious adversary may become the observed voting majority. This may allow them to quickly submit a proposal and have it executed before other users can respond.

This could be used to call `setConfig`, adjusting the `_proposalTime` and `_lockTime` to `1` in order to spam further proposals. Alternatively, this issue could also be leveraged to cause a proposal denial of service by setting `_votingPowerForProposalCreation` to unobtainable values.

Recommendations

Make sure this behaviour is understood. The testing team recommends setting a configurable, lowerbound threshold for `totalLocked` to prevent guilds that have just started from getting taken over, or cases where `totalLocked` suddenly drops.

Additionally, consider setting maximum conditions for the input parameters in `setConfig` (such as `_votingPowerForProposalCreation, _votingPowerForProposalCreation, _lockTime`). Doing this may prevent large token holders from causing Denial of Service.

Resolution

The issue has been (partially) fixed on PR#182.

New configuration settings, namely `minimumMembersForProposalCreation` and `minimumTokensLockedForProposalCreation` were added to mitigate the issue.

The following comment was added by the development team:

Added configurable minimum amounts preventing this issue and anything beyond those protections is intended and a human issue that documentation will cover explaining that if someone gains complete majority even if it is a low number of tokens then they have control. Users must be aware of this.

DXD-09	Duplicate Actions Unaccounted for During Voting Tally		
Asset	EnforcedBinaryGuild.sol, EnforcedBinarySnapshotERC20Guild.sol		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

`EnforcedBinaryGuild` allows for proposal creation using static call data. An additional feature of this contract, is that "No" actions are enforced. When a user creates a proposal, they input the relevant actions into the function `EnforcedBinaryGuild.createProposal`, which assumes the user is not factoring in for 'no action'. The following logic then appends the 'no action' as folows;

```
58  totalActions = totalActions.add(1);
    return super.createProposal(_to, _data, _value, totalActions, title, contentHash);
```

On line line [**60**] there is one extra `totalActions` to account for the enforced 'no action' and a call to `ERC20GuildUpgradeable` or `BaseERC20Guild`. The parent contract already assumes that `action=0` is a no action in the `endProposal()` function voting tally mechanism. `BaseERC20Guild` then appends another action to the `totalActions`. This can be seen on line [**299**] in `BaseERC20Guild.sol`:

```
299  newProposal.totalVotes = new uint256[](totalActions.add(1));
```

At this state `newProposal.totalVotes` has 2 additional actions that can be voted on. If some users vote for `action=0` they may be voting for no action or `ProposalState.Rejected`. Likewise if they vote for `action=totalActions.length` they may again be voting for a rejected proposal.

The `endProposal` makes no reconciliation between votes across these no actions. There is no process to tally votes for no action across the `action=0` and `action=totalActions.length`. Thus a malicious user could take advantage of proposals where users have split there votes across multiple no action positions.

Recommendations

Secure coding practices promote the use of lean parent or top level contracts that stand as baseline implementation only, making duplicate behaviour across inherited contracts less likely. The testing team recommends avoiding repeated behaviour such as adding additional elements to `totalVotes`, and deciding on whether this should be implemented in the parent, or in the child contract.

Resolution

The issue is no longer applicable with the removal of the two contracts, `EnforcedBinaryGuild.sol` and `EnforcedBinarySnapshotERC20Guild.sol` on PR#192.

DXD-10	Zero `totalLocked` Allows Everyone to Create Proposals		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The function `getVotingPowerForProposalCreation()` relies on `getTotalLocked()` to compute the minimum voting power for creating a proposal. In a case where no tokens are locked, then `getVotingPowerForProposalCreation()` will return zero because `getTotalLocked()` is zero. If this occurs, anyone can successfully submit a proposal through function `createProposal()`, allowing malicious users to flood the system with junk proposals that nobody is motivated to remove from the system.

Recommendations

The testing team recommends setting a lower bound on `getVotingPowerForProposalCreation` to prevent zero `totalLocked` from allowing anyone to create proposals.

Resolution

The issue has been (partially) fixed on PR#182.

New configuration settings, namely `minimumMembersForProposalCreation` and `minimumTokensLockedForProposalCreation` were added to mitigate the issue.

DXD-11	Gas Refund on `setVote()` May Be Subject to Large Scale Manipulation		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `setVote()` function tries to refund the gas used by the user to vote on a proposal if the contract has enough ETH. The amount of gas refunded depends on `voteGas` and the lower price between `tx.gasprice` and `maxGasPrice`. It is important to note that both these variables could be subject to manipulation by a malicious adversary.

The value of `tx.gasprice` could be arbitrarily increased by a user who is either a miner or in collusion with one. In both cases excess gas amounts may be refunded back to the user after a successful exploit, prompting the user to set arbitrarily high `tx.gasprice` and therefore only be limited by `maxGasPrice`

Furthermore, `maxGasPrice` is controlled by a `setConfig` which can be called from within the guild proposals itself. A single user is able to wait until sufficiently low `totalLocked` tokens exist in a guild before creating a proposal that manipulates `maxGasPrice` arbitrarily high.

Finally, a user can benefit from the gas refund if `voteGas` is not set properly. This happens if the number in `voteGas` is higher than the amount of gas used to execute `setVote()`. The user can optimise their profits by using as many accounts as possible to vote and receive refunds.

Variations of these three manipulations may be leveraged to drain all funds from a guild that holds ETH.

Recommendations

Make sure this behaviour is intended. The testing team recommends adding an extra check to the code to ensure the `voteGas` does not deviate too much from the average gas usage on `setVote()`. Additionally, an oracle service could be employed to have a more reliable measure of gas prices based on average gas prices observed.

Resolution

The issue has been (partially) fixed on PR#185. A hard limit of 117,000 gas was implemented.

DXD-12	Frontrunning Proposal Execution		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The contracts `PermissionRegistry.sol` and `ERC20Guild` (`BaseERC20Guild` or any related implementation contracts) complement each other. These contracts provide a checking mechanism to ensure that each function call is permitted and the amount of assets transferred is within specified limits. When a proposal is executed, the `ERC20Guild` contract calls `PermissionRegistry.setPermissionUsed()` to register the asset transfer.

This mechanism can be disrupted by a malicious user who aims to prevent the proposal from executing. This is done by frontrunning transaction on `ERC20Guild.endProposal()` by calling `ERC20Guild.setPermissionUsed()` with `valueTransferred` equals to `valueAllowed` and other information that corresponds to the proposal.

As a result of this frontrunning, `ERC20Guild.endProposal()` will fail if the blockchain tries to execute the transaction within the same block as the malicious `ERC20Guild.setPermissionUsed()`.

The frontrunning action may keep going until `timeForExecution` passes. In this case, the proposal will execute with `ProposalState.Failed` result.

Recommendations

The issue can be mitigated by adding access control to `PermissionRegistry.setPermissionUsed()` such that it is only callable by the respective permissioned owner.

Resolution

The issue has been fixed on commit c142818. An access control check was added to the `setETHPermissionUsed()` function.

It is worth noting that the project has undergone a substantial code refactor, and that the permissioning mechanism and function names have changed significantly.

DXD-13	*Fee-on-transfer* Tokens Are Not Supported		
Asset	`BaseERC20Guild.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`BaseERC20Guild` assumes that all ERC20 contracts send all token amounts to the destination. However, fee-on-transfer tokens (for example those with transfer taxes) behave differently. The amount of tokens sent by the sender is not the same as the amount of tokens received by the destination. If this type of token is used, the contract will not work as intended because there will be discrepancies between user balances and the tokens stored on `TokenVault`.

Recommendations

The testing team recommends using before and after snapshots when receiving tokens. The difference between the two snapshots should be the `tokenAmount` added to the user's locked balance.

Resolution

The development team indicated that the issue has been marked as "wont fix" as shown in PR#191.

DXD-14	Denial-of-Service using Proposals with Failed Calls		
Asset	`BaseERC20Guild.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `endProposal()` function is called to finalise a proposal. If the proposal is successful, then the function executes the calls in the proposal. However, it is possible that one of the calls fails. In this case, the function call to `endProposal()` also fails. As such, the proposal takes up one slot in `activeProposalNow` and is unremovable.

Malicious users with enough voting power for proposal execution can launch a DoS attack against the system by proposing malformed proposals with revert calls and vote on them. When the number of malicious proposals equals `activeProposalNow`, the system cannot take any new proposals.

Recommendations

The testing team recommends adding a mechanism to remove proposals that fail to execute to free-up `activeProposalsNow` and allow new proposal submissions.

Resolution

The issue was marked as "Non issue" with the following comment from the development team:

The guilds has a `timeForExecution` variable that is the amount of time a proposal has to be executed, in case it fails and time has passed it would be marked as failed.

This means that the proposals that cannot be executed will be cleared when the time expires by calling function `endProposal()`.

DXD-15	Denial-of-Service using Junk Proposals		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

A malicious user with enough tokens can flood the system with junk proposals that are expensive to execute. This is done by adding as many actions as possible on each proposal (note that the function `endProposal()` iterates through all actions to determine the `winningAction`).

Since there is no reward or gas refund to executing these proposals, it is possible that no one will call function `endProposal()` to dispose of the junk proposals. If the maximum number of active proposals is reached, the system will not accept any new proposals.

This attack however, is expensive for the attacker to launch. A proposal with 100 actions consume nearly 11 million gas, while ending the proposal takes less than 600,000 gas.

A simpler variation of this attack is to keep sending proposals to the system such that `activeProposalsNow` is always at its maximum so that no other users can submit legitimate proposals.

Recommendations

The testing team recommends freezing the proposer's tokens for each proposal they submit until the proposal is cleared/executed. This will deter the proposer from submitting more proposals than what is necessary because the attack will be more expensive for them.

Resolution

The issue has been fixed on PR#199. A max action limit of 10 per proposal was introduced.

DXD-16	PermissionRegistry Owner Changes Permissions Set By Others		
Asset	PermissionRegistry.sol		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The function `setPermisssion()` has a condition where the contract owner can modify permission entries that belong to others, namely the permissions with different `from` addresses than the contract owner's. A malicious owner can call `setPermission()` to set new permissions or modify existing permissions for other users. This can disrupt the permission system of any users that depend on the `PermissionRegistry` contract to store their permissions.

Recommendations

Make sure this behaviour is expected. The testing team recognises that the development team intends to either renounce the ownership or transfer the ownership to a DAO contract. However, the testing team recommends that necessary steps are taken to ensure that no `EOA` (Externally Owned Account) holds the ownership of `PermissionRegistry` contract at the end of the deployment process.

Resolution

The issue has been marked as "Intended behaviour" by the development team with the following comments:

Adding a check to not allow EOA to be permissionRegistry owners stills allow the PermissionRegsitry to be owned by a smart wallet contract and there fore owned by a single address.

We will add comments to make sure the permission registry ownership is well understood.

DXD-17	`totalMembers` Can Be Manipulated		
Asset	`SnapshotERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `lockTokens()` function is used by a user to lock tokens and gain voting power. This function however, can be used to manipulate the `totalMembers` variable because it accepts zero amount as an input. A malicious user can call `lockTokens()` as many times as they want to increase the `totalMembers` variable.

Similarly, function `withdrawTokens()` can be used to decrease the `totalMembers` by withdrawing zero amount (see DXD-01)

Recommendations

The testing team recommends preventing zero amount as an input on `lockTokens()` and `withdrawTokens()`.

Resolution

The issue has been fixed on PR#180. A user can no longer lock zero tokens.

DXD-18	Voting Tally Results in Unexpected Winner		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The function `endProposal()` allows users to tally votes, and determine the winning action on a proposal after `block.timestamp` exceeds the proposal `endTime`. Voting is tallied using the following logic:

```
uint256 winningAction = 0;
uint256 i = 1;
for (i = 1; i < proposals[proposalId].totalVotes.length; i++) {
  if (
    proposals[proposalId].totalVotes[i] >= getVotingPowerForProposalExecution() &&
    proposals[proposalId].totalVotes[i] > proposals[proposalId].totalVotes[winningAction]
  ) winningAction = i;
}
```

In the event that two or more actions have the same number of votes, the one positioned earlier in the array will unexpectedly become the winning action despite another option being equally as favoured.

Recommendations

Make sure this behaviour is understood. If this behaviour is not intended, the testing team recommends exploring alternatives, such as rejecting a proposal and forcing a revote.

Resolution

The issue has been fixed on PR#183. The proposal is now rejected if two or more actions get the same amount of voting.

DXD-19	Early Withdrawal To Manipulate getVotingPowerForProposalExecution()		
Asset	BaseERC20Guild.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Users who vote on a proposal are incentivised to withdraw their locked tokens before the `proposalTime` expires. The aim of this behaviour is to reduce the amount of `getVotingPowerForProposalExecution()` which is a specific percentage of the total locked tokens stored in variable `totalLocked`. If `totalLocked` is reduced, then `getVotingPowerForProposalExecution()` will also be reduced and the users' voted proposal will have a greater chance to pass.

Recommendations

The testing team recommends adding a snapshot mechanism to the relevant section of the protocol to mitigate this issue.

Resolution

The issue has been fixed on PR#187. The locked tokens' `lockTime` will be extended when the users vote, preventing early withdrawal.

DXD-20	Cannot Receive ETH From `send()` or `transfer()`		
Asset	`BaseERC20Guild.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The contract `BaseERC20Guild` allows any account to send ETH to it. However, since this contract intends to use a proxy contract (for upgradeability purposes), ETH transfers through function `send()` and `transfer()` may fail due to insufficient gas allowance.

According to the Solidity Documentation, both functions, only have 2300 gas allowance to conduct the operation. Traditionally, this amount of gas is sufficient for sending ETH to a receiver address. However, it is not enough for additional operations on a proxy contract, such as a `DELEGATECALL` to execute the logic on the contract implementation on the proxy contract's context.

This means, ETH transfers to `BaseERC20Guild` are only possible through low level `call`s which forward 63/64th of the gas allowance.

Recommendations

Make sure this behaviour is understood. The testing team recommends removing `receive()` to avoid misleading the user for transferring ETH to the contract using `send()` and `transfer()`. Instead, `fallback() external payable {}` can be used.

Resolution

The fixes on PR#181 replaced `receive()` with `fallback()`.

The testing team further recommends documenting this behaviour to prevent unwanted events in the future.

DXD-21	Failing Proposal through `setPermissionUsed()`		
Asset	`PermissionRegistry.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `setPermissionUsed()` function helps identify whether a transaction is a permitted transaction or not. Additionally, it also checks whether the value transferred in the transaction exceeds a certain threshold.

A permission accepts `ANY_ADDRESS` and `ANY_SIGNATURE` as an indication that any transaction from a sender for an asset is accepted. However, the current implementation has an edge case which may cause a proposal to revert if the proposal has any of the following conditions:

- `to == 0xaAaAaAaaAaAaAaaAaAAAAAAAAaaaAaAaAaaAaaAa`
- `functionSignature == 0xaaaaaaaa`

where `0xaAaAaAaaAaAaAaaAaAAAAAAAAaaaAaAaAaaAaaAa` is constant `ANY_ADDRESS` and `0xaaaaaaaa` is constant `ANY_SIGNATURE`.

If the conditions above are met, then `_setValueTransferred()` will be called multiple times in function `setPermissionUsed()`, which increases `permission.valueTransferred` on each call. Then, if `permission.valueTransferred` becomes greater than `permission.valueAllowed` after the multiple calls to `_setValueTransferred()`, the transaction (or proposal execution) reverts.

Recommendations

The testing team recommends adding the following checks on either `setPermissionUsed()` or `setPermission()` to avoid this issue:

```
require(to != ANY_ADDRESS, "Cannot use reserved address");
require(functionSignature != ANY_SIGNATURE, "Cannot use reserved signature");
```

Resolution

The issue was marked as "Expected behaviour" by the development team with the following comment:

Not a bug, this is the intended behavior to be able to set "global" transfer limits.

DXD-22	Incompatible Permissioning on `ERC20` Assets		
Asset	`PermissionRegistry.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The functions `getPermission()` and `setPermissionUsed()` indicate that when `ERC20` is used in a proposal, the asset requires `ANY_SIGNATURE` permission in order for the proposal to be executed successfully. This permission indicates that `PermissionRegistry` requires that all functions in the `ERC20` contracts are callable by the proposal.

However, this requirement is not enforced during proposal creation (in any guild contracts) or in `setPermission()`. As a result, if a guild only permits specific function calls in the `ERC20` contract, then the proposal will revert due to the lack of permission.

This indicates incompatibility between the permission setting process and the permission checking for `ERC20` assets.

Recommendations

Make sure this behaviour is understood. The testing team recommends enforcing strict permissions on `getPermission()` and `setPermissionUsed()` such that `ERC20` assets are treated the same way as the native token. This can be done by removing the use of wildcard `ANY_ADDRESS` and `ANY_SIGNATURE`.

Resolution

The issue may no longer applicable with the refactoring of `PermissionRegistry` contract on PR#191.

DXD-23	Potentially Confusing value on ERC20 Operation
Asset	`BaseERC20Guild.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `BaseERC20Guild` contract supports two different asset types: ETH and ERC20 tokens. Both types are managed differently during proposal execution, especially in the `endProposal()` function. If the asset is an ERC20 token, the `endProposal()` function extracts the asset value from the transaction data, while on the other hand, if the asset is ETH, the function takes the asset value directly from the proposal's `value` variable.

Since there are two values here (in the proposal and in the transaction data), there is potential for confusion. If the user thinks that the proposal's value must reflect the amount of ERC20 tokens transferred, then the proposal fails to execute because it tries to transfer ETH along with the ERC20 operation, which should be rejected by the ERC20 contract. To make it successful, the proposal's value must be zero if the user intends to transfer or approve ERC20 tokens, or otherwise the proposal execution fails.

Recommendations

The testing team recommends adding a mention of this behaviour in the documentation. Alternatively, an extra check can be introduced to ensure the proposal's `value` is zero when dealing with ERC20 asset transfers.

Resolution

The issue was marked as "Documentation" by the development team with the following comment:

This will be addresses in frontend and coming documentation.

DXD-24	Duplicated Checks on `createProposal()` and `endProposal()`
Asset	`EnforcedBinaryGuild.sol, EnforcedBinarySnapshotERC20Guild.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `createProposal()` function conducts sanity checks on input parameters, i.e., making sure `totalActions > 0` and the inputs have the same length. However, the same action is also done on `super.createProposal()` which is called at the end of the function. Therefore, the same checks are performed twice.

Similarly, the `endProposal()` function checks whether a certain condition holds before determining the proposal status and executing actions, while the same checks are also conducted on `super.endProposal()`.

Recommendations

The testing team recommends implementing `createProposal()` and `endProposal()` on the child contract to remove the duplicate checks.

Resolution

The issue is no longer applicable with the removal of the two contracts, `EnforcedBinaryGuild.sol` and `EnforcedBinarySnapshotERC20Guild.sol` on PR#192.

DXD-25	Input Length Not Checked in `votingPowerOfMultipleAt()`
Asset	`SnapshotERC20Guild.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `votingPowerOfMultipleAt()` function is used to query the voting power of multiple users at given `snapshotIds`. The logic requires that the number of `snapshotIds` be at least equal to the number of accounts. However, there is no check to ensure the fulfillment of this requirement. If `snapshotIds.length < accounts.length`, then the function call reverts.

Recommendations

The testing team recommends adding a check to ensure `accounts.length` is equal to `snapshotIds.length`.

Resolution

The issue has been fixed on PR#190. An input length check was added to function `votingPowerOfMultipleAt()`.

DXD-26	Non-upgradeable TokenVault
Asset	`TokenVault.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `TokenVault` contract implementation indicates that this contract is prepared to be upgradeable. However, the way the contract is instantiated in the `initialize()` function of the `ERC20Guild` or `ERC20GuildUpgradeable` contract does not allow for upgradeability. The lack of proxy mechanism in the contract instantiation will prevent the contract from being upgraded while preserving the contract storage.

`TokenVault` also implements its own initialisation scheme while deriving OpenZeppelin's `Initializable` library, which is redundant. This can be seen in the excerpts below:

```
modifier isInitialized() { // This is a redundant modifier
    require(initialized, "TokenVault: Not initilized");
    _;
}
```

```
function initialize(address _token, address _admin) external initializer {
    token = IERC20Upgradeable(_token);
    admin = _admin;
    initialized = true; // This is a redundant variable
}
```

Recommendations

The testing team recommends implementing a proxy contract while instantiating the `TokenVault` contract and necessary functions to upgrade the `TokenVault` on the guild contract.

The testing team also suggests removing modifier `isInitialized()` and variable `initialized` because initialisation is already covered by OpenZeppelin's `Initializable` library.

Resolution

The issue has been fixed on PR#200. Function `initialize()` was removed from the contract along with modifier `isInitialized` and other upgradeable-related components.

The testing team further recommends replacing OpenZeppelin's `contracts-upgradeable` library with the non-upgradeable version.

DXD-27	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **BaseERC20Guild.sol**

   1a) Typo:

   - line [**179**]: more tha -> more than
   - line [**23,231, 240, 249, 286, 459, 466**]: cant -> can't or cannot

   1b) Programming style on `if..else`.

   Code on line [**323-329**] use `if..else` logic but they try to check different conditions. The `if` statement checks `winningAction`, but the `else if` statement checks whether the proposal exceeds `timeForExecution`. This reduces code readability.

   The testing team recommends separating the condition check on line [**323-329**].

   1c) `totalActions` check when creating a proposal.

   All actions in the proposal need to have the same number of calls, as indicated by `callsPerAction` calculation in line [**332-336**]. However, there is no check to enforce this.

   The testing team recommends adding a check to validate the number of calls during proposal creation, for example by using modulus operation. The need to have identical number of calls for each action also exposes the system's weakness, because each action needs to adopt the highest number of calls on the proposal's actions, where the unused calls must be set to have zero destination or zero data.

   1d) Incorrectly worded require on line [**178**]

   `"ERC20Guild: Only callable by ERC20guild itself when initialized"` suggests that this function should **only** be called during initialisation, whereas dev comments suggest it `can be called only executing a proposal or when it is initialized`. Consider modifying the require statement as they appear to be conflicting.

2. **ERC20GuildUpgradeable.sol**

   2a) Typo:

   line [**63**]: more tha -> more than

3. **ERC20GuildWithERC1271.sol**

   3a) Typo:

   line [**649**]: more tha -> more than

   3b) `isValidSignature()` depends on `votingPower`

   Function `isValidSignature()`'s return value depends on the signer's `votingPower`. If the signer's `votingPower` is zero, regardless of whether the signature is valid, the function returns zero.

   Make sure this behaviour is intended.

4. **SnapshotERC20Guild.sol**

   4a) Incorrect variable naming
   Function `getVotingPowerForProposalExecution()` requires `snapshotId` as input, but the current variable name is `proposalId`.

   4b) Incorrect `@title` comment
   line [**10**] `@title SnapshotERC20Guild` -> `@title EnforcedBinarySnapshotERC20Guild`

   4c) Typo:
   line [**45**]: amd -> and

5. **SnapshotRepERC20Guild.sol**

   5a) Typo:
   line [**25**]: Initilizer -> Initializer

   5b) Incorrect revert message
   The word `SnapshotERC20Guild` in revert messages in `SnapshotRepERC20Guild.sol` can be replaced with `SnapshotRepERC20Guild` to avoid confusion with another contract with the same name.

6. **PermissionRegistry.sol**

   6a) Constant `emptyPermission` never used
   If constant `emptyPermission` is never used in the contract, it can be safely removed.

   6b) Typo:
   - line [**91**]: Cant -> Can't
   - line [**147, 151**]: Check is there an allowance -> Check if there is an allowance

7. **TokenVault.sol**

   7a) No revert message
   The keyword `require` on line [**40**] can use a message to provide details, e.g. `Deposit must be sent through admin`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The issues were addressed in PR#188.

1. **BaseERC20Guild.sol**

   1a) Fixed
   1b) Not fixed with the following notes: *We won't change code style since readability is ok.*
   1c) Not fixed with the following notes: *Line 280 is already verifying that all calls are balanced (same length for to, data, value). But added a new check to validate that totalActions make sense with the calls length we are getting.*
   1d) Fixed.

2. **ERC20GuildUpgradeable.sol**

2a) Fixed

3. **ERC20GuildWithERC1271.sol**

3a) Fixed

3b) Not fixed with the following notes: *After discussing with Augusto we won't change the function name.*

4. **SnapshotERC20Guild.sol**

4a) Fixed.

4b) *The raised issue was for EnforcedBinarySnapshotERC20Guild which was removed.*

4c) *The raised issue was for EnforcedBinarySnapshotERC20Guild which was removed.*

5. **SnapshotRepERC20Guild.sol**

5a) Fixed.

5b) Fixed.

6. **PermissionRegistry.sol**

6a) Fixed.

6b) Typo on line [91] still exists. The other typos were no longer applicable.

7. **TokenVault.sol**

7a) Fixed.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_init                                   PASSED   [1%]
test_init                                   PASSED   [2%]
test_init                                   PASSED   [3%]
test_receive                                PASSED   [4%]
test_receive_transferSend                   XFAIL    [5%]
test_setConfig                              PASSED   [6%]
test_setPermission                          PASSED   [7%]
test_setPermissionDelay_pbt                 SKIPPED  [8%]
test_setPermissionDelay                     PASSED   [9%]
test_createProposal                         PASSED   [10%]
test_createProposal_noVotingPower           PASSED   [12%]
test_createProposal_exceedActiveProposal    PASSED   [13%]
test_createProposal_empty                   PASSED   [14%]
test_setVote                                PASSED   [15%]
test_setVote_zero                           PASSED   [16%]
test_setVote_gas                            PASSED   [17%]
test_setSignedVote                          PASSED   [18%]
test_endProposal_noVote                     PASSED   [19%]
test_endProposal_error                      PASSED   [20%]
test_proposal_setConfig                     PASSED   [21%]
test_proposal_actionZero                    PASSED   [23%]
test_proposal_setPermission                 PASSED   [24%]
test_proposal_actionTest                    PASSED   [25%]
test_proposal_actionTest_permissionOwner    PASSED   [26%]
test_proposal_actionTest_double             PASSED   [27%]
test_proposal_actionTest_setPermissionUsed  PASSED   [28%]
test_proposal_actionTest_frontrunAttack     PASSED   [29%]
test_proposal_actionTest_ERC20              PASSED   [30%]
test_proposal_actionTest_reentrancy         PASSED   [31%]
test_proposal_multiActions                  PASSED   [32%]
test_proposal_actionRevert                  PASSED   [34%]
test_proposal_actionMany                    PASSED   [35%]
test_proposal_actionTooMany                 PASSED   [36%]
test_proposal_actionAny                     PASSED   [37%]
test_proposal_actionAny2                    PASSED   [38%]
test_proposal_sendETH_EOA                   PASSED   [39%]
test_withdrawTokens_fail                    PASSED   [40%]
test_lockTokens_withdrawTokens              PASSED   [41%]
test_withdrawTokens_withdrawZero            PASSED   [42%]
test_setConfig_fail                         PASSED   [43%]
test_proposal_timeForExecution              PASSED   [45%]
test_proposal_vote_withdraw                 PASSED   [46%]
test_proposal_doubleVote                    PASSED   [47%]
test_proposal_actionRevert_dos              PASSED   [48%]
test_init                                   PASSED   [49%]
test_setEIP1271SignedHash                   PASSED   [50%]
test_init                                   PASSED   [51%]
test_createProposal_executed                PASSED   [52%]
test_createProposal_actionZero              PASSED   [53%]
test_createProposal_actionLast              PASSED   [54%]
test_createProposal_DOS                     PASSED   [56%]
test_init                                   PASSED   [57%]
test_createProposal_mock                    PASSED   [58%]
test_createProposal_executed                PASSED   [59%]
test_createProposal_failed                  PASSED   [60%]
test_createProposal_rejected                PASSED   [61%]
test_createProposal_DOS                     PASSED   [62%]
test_pay                                    PASSED   [63%]
test_init                                   PASSED   [64%]
test_setPermissionDelay                     PASSED   [65%]
test_setPermission                          PASSED   [67%]
```

```
test_setPermission_fail                     PASSED   [68%]
test_setPermissionUsed_any                  XFAIL    (...)[
test_setPermissionUsed_test                 PASSED   [70%]
test_init                                   PASSED   [71%]
test_createProposal_mock                    PASSED   [72%]
test_snapshots                              PASSED   [73%]
test_createProposal_executed                PASSED   [74%]
test_createProposal_votingPowerChange       XFAIL    [75%]
test_createProposal_rejected                PASSED   [76%]
test_createProposal_newVoter                PASSED   [78%]
test_withdrawTokens_withdrawZero            PASSED   [79%]
test_setSignedVote                          PASSED   [80%]
test_setSignedVote_votingPowerChange        XFAIL    [81%]
test_totalMembers                           PASSED   [82%]
test_init                                   PASSED   [83%]
test_lockTokens_withdrawTokens              PASSED   [84%]
test_createProposal_mock                    PASSED   [85%]
test_createProposal_mock_zeroPower          PASSED   [86%]
test_proposal_vote_zeroPower                PASSED   [87%]
test_proposal_executed                      PASSED   [89%]
test_proposal_votingPowerChange             PASSED   [90%]
test_proposal_rejected                      XFAIL    (T...)[
test_proposal_noVote                        PASSED   [92%]
test_proposal_oneVote                       PASSED   [93%]
test_proposal_newVoter                      PASSED   [94%]
test_setSignedVote                          PASSED   [95%]
test_proposal_snapshots                     PASSED   [96%]
test_deploy                                 PASSED   [97%]
test_init                                   PASSED   [98%]
test_deposit_withdraw                       PASSED   [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

		Likelihood		
		Low	Medium	High
Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].