Basis

Security Assessment

Basis Smart Contracts
October 5, 2018

Prepared For:
Brian Freyburger | *Intangible Labs*
b@intangiblelabs.co

Prepared By:
Robert Tonic | *Trail of Bits*
robert.tonic@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Benjamin Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

Dominik Czarnota | *Trail of Bits*
dominik.czarnota@trailofbits.com

Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Executive Summary

From August 20th to October 5th, Trail of Bits assessed the Basis Solidity smart contracts. Five engineers conducted this assessment over the course of 12 person-weeks. The assessment focused primarily on the ERC20-compatible tokens used by the Basis contracts, including Basis, its shares, and bond tokens. Additionally, the contracts related to the election committee were reviewed.

The first week of the assessment was spent learning the Basis smart contracts, as well as searching for common Solidity vulnerabilities. Subsequently, the second week was spent analyzing the election committee contract for errors which would allow an attacker to manipulate or block votes, or compromise a proposal's execution. During the third week focus shifted to the bond token contracts, searching for minting, transfer, and management issues. The following two weeks were spent reviewing the Basis and share tokens, with a focus on minting, transfer, management, and distribution. The final week of the assessment was spent analyzing the multi-sig wallet, and reviewing the patches provided for issues identified during the assessment.

This assessment identified a variety of findings, ranging from high- to informational-severity, across multiple smart contracts. High-severity findings included incorrect authorization schemas between contracts, lack of access controls, improper whitelist initialization, invalid proposal execution, vote tally disruption, blockable bond distribution, and internal state modification while a contract is paused. Medium-severity findings included an improper implementation of allowance, a race condition in the whitelist contracts, and incorrect parameter validation during bond creation. Finally, low- and informational-severity findings included incorrect event emission, incorrect return values, incorrect documentation, allowance modification during a paused state, and an inability for the election commission to control the Basis mint address or a provided whitelist contract.

See the Findings Summary for an index of all findings. Due to multiple versions of code being delivered for review, each finding details the version in which it was found; either v0 or v1. Discovered issues were generally addressed immediately by the development team, and the Fix Log tracks their remediation status.

In addition to the security findings, Appendix B details code quality issues not related to any particular vulnerability. Appendix C and Appendix D detail extracted security properties of the Basis system and efforts to verify them with Echidna and Manticore.

Overall, the code reviewed is indicative of a typical work in progress. A significant portion of functionality is still in an experimental phase. In some areas, bugs prevent code from running. The general consequence is confusion regarding correct system operation. Additionally, the codebase consists of numerous Solidity smart contracts, most of which have a significant amount of interaction either through inheritance or normal operation. The combination of these complex interactions with the aforementioned confusion regarding correct system operation and non-executable code made the Basis system very difficult to review, and, we believe, is the root cause of many of the identified issues.

Basis requires further development before it is mature enough for production deployment. Documentation of expected system operation and environment is necessary. The system's testing suite needs to be expanded. Once this is completed, another assessment is recommended to re-evaluate the security posture of the Basis system.

Engagement Goals & Scope

The goal of the engagement was to evaluate the security of the Basis smart contracts with a focus on answering the following questions:

- Is it possible for contracts to transition into an invalid states, either intentionally or by accident?
- Is it possible to cause the contracts to enter an unrecoverable state?
- Can a state be unreachable due a programing error or to gas limitations?
- Is it possible to manipulate or block the balances of the basis, share, or bond tokens?
- Is it possible to manipulate the payments of shares or bonds?
- Is it possible to manipulate, block, delay or disturb the election committee functionality, including voting, tallying and proposal execution?
- Can delegates increase their vote weight or avoid penalties?

Issues caused by code off-chain or requiring the collusion of a majority of delegates were out-of-scope for this engagement.

Coverage

During the review of contracts defining ERC20-compatible tokens, we looked for common Solidity flaws such as integer overflows, re-entrancy vulnerabilities, and unprotected functions. We also looked for more nuanced flaws, such as logical errors and race conditions. Additionally, we looked into whether the tokens could be trapped without Basis' intervention.

Token management contracts were reviewed for logical flaws and race conditions as well as unauthorized access to administrative methods which allow the creation, modification or burning of tokens, bonds and shares. Additionally, scenarios where an attacker could manipulate tokens, bonds or shares and avoid penalties were explored.

Whitelist contracts were reviewed for logical flaws, race conditions, and unauthorized access to administrative methods which the allow addition, modification or removal of whitelisted addresses. Further analysis was performed to evaluate the impact of race conditions on the effectiveness of whitelists used in the Basis system.

The election commission contracts were reviewed for flaws which would allow an attacker to manipulate or block votes, or could compromise proposal execution.

Finally, contracts defining common data structures were checked for correctness.

Project Dashboard

Application Summary

Name	Basis
Type	Ethereum smart contract
Platform	Solidity

Engagement Summary

Dates	August 20th, 2018 - October 5th, 2018
Method	Whitebox
Consultants Engaged	5
Level of Effort	12 person-weeks

Vulnerability Summary

Total High Severity Issues	11	■■■■■■■■■■■
Total Medium Severity Issues	3	■■■
Total Low Severity Issues	6	■■■■■■
Total Informational Severity Issues	2	■■
Total Undetermined Severity Issues	0	
Total	22	

Category Breakdown

Data Validation	7	■■■■■■■
Access Controls	8	■■■■■■■■
Logging	1	■
Timing	2	■■
Numerics	1	■
Error Reporting	1	■
Denial of Service	2	■■
Total	22	

Recommendations Summary

This section aggregates all the recommendations made during the engagement. They are split into short-term recommendations, which address issue's immediate causes, and long-term recommendations, which include adjustments in the development process.

Short Term

❑ **Ensure appropriate return values.** Named return values should be verified for appropriate reassignment with regards to function branches.

❑ **Simplify contract inheritance.** The current "mix-in" approach to inheritance is complex and could lead to human error. Functionality should be consolidated to improve readability.

❑ **Adopt consistent naming conventions.** Review variable naming for consistency and descriptiveness. Ensure consistency across all contracts, and rename non-descriptive variables. Consider adopting the Solidity style-guide naming conventions.

❑ **Improve unit test coverage across the codebase.** We recommend testing to ensure appropriate event emission and return types.

❑ **Improve code documentation.** Documentation of system components, access controls, and expected user interaction should be expanded.

Long Term

❑ **Consider static analysis, fuzzing, and symbolic testing of critical components.** Use Slither, Echidna, and Manticore to find unhandled edge cases through property testing and symbolic execution.

❑ **Review access controls.** The current state of the desired access controls of the contracts is unclear. Review and document implemented access controls to ensure appropriate restrictions are applied for all users within the system.

❑ **Review pausing architecture.** Review the pausing architecture to ensure dependent contracts handle states before, during, and after a pause when invoking functions.

❑ **Document third-party whitelist functionality.** Document situations in which third-party control may be malicious. Define a procedure to follow to handle such situations.

❑ **Monitor on-chain timestamp transactions.** On-chain timestamps may be manipulated by a miner. Careful monitor the transactions heavily dependant on timestamps can help to foresee this kind of attacks.

❑ **Reduce complexity through the use of revert.** Avoiding revert increases the complexity of computations, since undo functionality must be implemented. This complexity can be reduced through the use of revert.

Findings Summary

#	Title	Type	Severity
1	Incorrect allowance check permits an attacker to perform undesired transactions to an allowed account	Access Controls	Medium
2	BasisPolicy.executeOracleVote always returns false	Data validation	Low
3	Incorrect event information emission in Ownable.acceptOwnership	Logging	Low
4	Incorrect authorization schema between BondTokenManager and the policies will prevent calls to createBondToken	Access Controls	High
5	Re-use of a delegate address allows invalid proposals to execute	Data Validation	High
6	Previously executed proposals can be re-executed	Data Validation	High
7	Non-reinitialization of daily pledges will block the tally	Data Validation	High
8	A multiplication overflow allows an attacker to block the tally	Numerics	High
9	block.timestamp can be manipulated	Timing	Low
10	The approve function can be called on paused contracts	Access Controls	Low
11	A race condition in the Whitelist contract's functions makes them ineffective	Timing	Medium
12	An attacker can block the bond distribution	Denial of Service	High
13	Creating bonds with certain parameters can block the bond distribution	Denial of Service	Medium
14	ERC20 Incorrect return values and documentation	Error Reporting	Informational

15	SimpleShares allows infinite token creation	Data Validation	High
16	ShareToken whitelist cannot be initialized	Access Controls	High
17	User can be penalized even if no tokens are transferred	Data Validation	Informational
18	<removed after discussion with Basis>	Access Controls	Informational
19	SimpleShares contract allows execution during pause	Access Controls	High
20	setMinimumDistribution is not protected	Access Controls	High
21	<removed after discussion with Basis>	Data Validation	Low
22	Basis owner is the only mint-able address	Access Controls	Low
23	The Whitelist contract is not controlled by the election commission	Access Controls	Low
24	pendingDistribution and minDistribution arrays are not properly used	Data Validation	High

1. Incorrect allowance check permits an attacker to perform undesired transactions to an allowed account

Severity: Medium
Type: Access Controls
Target: ERC20.sol (v0)

Difficulty: Low
Finding ID: TOB-Basis-001

Description

The ERC20 standard has an allowance system that permits one user to transfer tokens on behalf of another user. Due to an incorrect allowance check, an attacker can transfer funds of another user without consent. An undesired transfer may lead to unexpected behavior and is particularly risky if the destination is a smart contract.

The `transferFrom` method uses the `allowance` mapping to track the amount of tokens users are able to transfer on behalf of other users.. However, `ERC20.transferFrom`, does not check the `allowance` of the caller, but rather checks the `allowance` of the destination instead.

```
function transferFrom(address _from, address _to, uint256 _amount) public
returns (bool success) {
   allowance[_from][_to] = allowance[_from][_to].sub(_amount);
```

Figure 1: `ERC20.transferFrom` *function*

As a result, any user may transfer funds on behalf of another. If the destination is a smart contract, this may lead to a situation where the smart contract prematurely receives the funds and fails to perform the transfer when it is expected.

Exploit Scenario

Bob's smart contract is an exchange. Bob's smart contract uses `transferFrom` as proof of a deposit. Alice wants to deposit 1,000 Basis tokens in the exchange, and call approve. Eve sees the transaction, and transfers the tokens to the exchange before Alice is able to perform the call to the deposit function. As a result, 1,000 Basis tokens are trapped in the exchange and Alice needs to contact Bob to undo the transfer.

Recommendation

In the short term, change all `allowance[_from][_to]` to `allowance[_from][msg.sender]` in `ERC20.transferFrom`.

In the long term, consider Improving unit test coverage. Issues like this one can be found with more thorough testing against documented standards.

2. BasisPolicy.executeOracleVote always returns false

Severity: Low
Type: Data validation
Target: BasisPolicy.sol (v0)

Difficulty: Low
Finding ID: TOB-Basis-002

Description

The `BasisPolicy.executeOracleVote` method returns a boolean value, which is always false, even in cases of success. Returning false on success may lead to unexpected behavior for the caller, especially if the caller is a smart contract that checks the return value.

`executeOracleVote` should return the boolean `finished` (BasisPolicy.sol#L35):

```
function executeOracleVote() public returns (bool finished) {
```

Figure 1: `executeOracleVote` *declaration*

Within the definition, `finished` is never assigned and the default value for booleans in Solidity is false, thus all return statements in `executeOracleVote` return false.
The caller may assume that the execution wasn't successful, causing them to resubmit funds or duplicate orders. A smart contract calling this function may enter a locked state.

Exploit Scenario

Bob's smart contract is responsible for calling `executeOracleVote`. Bob's smart contract waits for `executeOracleVote` to succeed to change its internal state. As a result of never returning true, Bob's smart contract is trapped.

Recommendation

Short term, assign `finished` and return true when the function execution is a success.

Long term, avoid the declaration of variables within the return statement, and use only variable declarations in the function body. Ensure that all the paths of each function end with a return statement.

3. Incorrect event information emission in Ownable.acceptOwnership

Severity: Low
Type: Logging
Target: Ownable.sol (v0)

Difficulty: Low
Finding ID: TOB-Basis-003

Description

`Ownable.acceptOwnership` emits an event with incorrect values, which may mislead third parties watching the event and lead to an incorrect audit log.

The `OwnershipTransferred` event intends to return the previous owner address as the first argument, and the new owner as the second :

```
event OwnershipTransferred(address indexed previousOwner, address indexed
newOwner);
```

Figure 1: `OwnershipTransferred` *declaration*

The `Ownable.acceptOwnership` method is responsible for emitting the `OwnershipTransferred` event.

```
function acceptOwnership() public onlyPendingOwner {
   owner = pendingOwner;
   delete pendingOwner;

   emit OwnershipTransferred(owner, pendingOwner);
}
```

Figure 2: `acceptOwnership` *function*

However, `pendingOwner` is deleted before the event is emitted and therefore is always zero upon emission.

The event emitted contains the new owner as first argument and zero as the second argument, instead of the previous owner address as first argument and the new owner address as second argument.

As a result, the event emits incorrect information that will mislead actors watching the events.

Exploit Scenario

Bob watches the events emitted to detect bugs or incorrect contract uses. Bob sees an `OwnershipTransferred` event, thinks that an incorrect ownership change occured and contacts the Basis team. As a result, the Basis team wastes time checking the ownership change.

Recommendation
Short term, emit the event before the `owner` assignation, so that `owner` contains the previous owner address and `pendingOwner` contains the new one.

Long term, improve the event testing in unit tests.

4. Incorrect authorization schema between BondTokenManager and the policies will prevent calls to createBondToken

Severity: High
Difficulty: Low
Type: Access Controls
Finding ID: TOB-Basis-004
Target: BasisPolicy.sol (v0), BondTokenManager.sol (v0)

Description

The `BondTokenManager` contract has functions only callable by its controller. `BasisPolicy` and `ModifiablePolicy` call some of these functions. However, only one contract can be the controller. Therefore, either `BasisPolicy` or `ModifiablePolicy` will not be able to call the `BondTokenManager` functions.

The `BondTokenManager.distribute` and `BondTokenManager.createBondToken` methods are protected by the `onlyController` modifier.

```
function distribute() public onlyController returns (bool complete) {
```

Figure 1: `distribute` *function*

```
function createBondToken(bool senior, uint256 faceValue, uint256 term,
uint256 expireTerm, uint256 payoutPercent) public
   onlyController
```

Figure 2: `createBondToken` *function*

`BondTokenManager.distribute` is called by `BasisPolicy.executeOracleVote` .

```
function executeOracleVote() public returns (bool finished) {
    ...
    if (!bondTokenManager.distribute()) {
```

Figure 3: `executeOracleVote` *function*

Even though `ModifiablePolicy` is not yet fully implemented, the documentation provided by the Basis team indicates that it will call `BondTokenManager.createBondToken` during the execution of `ModifiablePolicy.executeContraction`.

Only one of `BasisPolicy` and `ModifiablePolicy` can be the controller of `BondTokenManager`. As a result, either `BasisPolicy` or `ModifiablePolicy` will not be able to call the `BondTokenManager` functions.

Exploit Scenario

`BondTokenManager` is deployed and `BasisPolicy` is its controller. `ModifiablePolicy` cannot call `createBondToken` so the creation of bonds fails. As a result, Basis supply can only increase and the coin price is unstable.

Recommendation
In the short term, fix the authorization schema; one solution is to set `ModifiablePolicy` as the controller of `BondTokenManager` and make `ModifiablePolicy` call `distribution`. Another is to change the modifier of `createBondToken` to authorize only the `ModifiablePolicy` to call `createBondToken`, instead of the controller.

In the long term, create documentation for the authorization schema to highlight intended capabilities for each user. For example, highlighting which functions are expected to be called by other contracts and users.

5. Re-use of a delegate address allows invalid proposals to execute

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-Basis-005
Target: ElectionCommission.sol (v0)

Description
If a proposal obtains half of the delegate weights during the tally, it can be executed. Pairing this logic with a re-use of the same delegate address allows any proposal to pass if only one delegate votes for it.

The `tallyProposalResult(Proposal proposal, address[] delegates)` checks if the proposal has enough votes to be executed.

```
function tallyProposalResult(Proposal proposal, address[] delegates) public
returns (bool success) {
    …
        // verify that this delegate did indeed vote for the proposal
        if (proposalDelegateVote[proposal][delegates[i]] > 0) {
            tallyCount =
tallyCount.add(shares.delegateEffectiveWeight(delegates[i]));
        …
    if (tallyCount > (tallyActivePledges.add(1)) / 2) {
       proposalState[proposal] = ProposalState.PASSED;
    }
    …
```

Figure 1 : `tallyProposalResult` function

Within this implementation, there is no check to ensure that a delegate voter is unique in the `delegates` list. If the same delegate is repeated in that list, its weight will be counted multiple times.

As a result, an attacker can pass any proposal by submitting the same address multiple times, creating a false consensus.

Exploit Scenario
Bob is a malicious delegate who submits a proposal and votes for it. Bob calls `tallyProposalResult` with its own address multiple times to make the proposal pass. As a result, Bob can execute the malicious proposal.

Recommendation

In the short term, prevent the tallying of repeated addresses in `tallyProposalResult`. A simple solution is to require the `delegates` list to be ordered and to check at each iteration that the next delegate has an address strictly greater than the previous one.

`ElectionCommission` lacks unit tests and documentation despite its complexity. In the long term, consider adding tests covering all potential proposal-counting scenarios. Consider adding documentation detailing the expected behavior of each function. Document the difference between the two `tallyProposalResult` functions.

6. Previously executed proposals can be re-executed

Severity: High
Type: Data Validation
Target: ElectionCommission.sol (v0)

Difficulty: Low
Finding ID: TOB-Basis-006

Description

Once a proposal is passed, it can be executed. However, it is possible to repeat the vote counting on a previously-executed proposal in order to re-execute the proposal multiple times.

A proposal has three states: `PROPOSED`, `PASSED`, `EXECUTED` (ElectionCommission.sol#L227):

```
enum ProposalState { PROPOSED, PASSED, EXECUTED }
```

Figure 1: Proposal states

The `tallyProposalResult` functions are used to pass a proposal. If the proposal has enough votes, its state is set to `PASSED`.

Once a proposal is passed, it can be executed through `execute`:

```
function execute(Proposal proposal) external notDuringVoteTally {
   require(proposalState[proposal] == ProposalState.PASSED);
   require(now < proposal.expiration());
   proposalState[proposal] = ProposalState.EXECUTED;

   require(_currentProposal == Proposal(0));
   _currentProposal = proposal;
   _currentProposal.execute();
   delete _currentProposal;
}
```

Figure 1: `ElectionCommission.execute` *function*

The `execute` function checks that the proposal has the `PASSED` state. Upon a proposal's execution, its state is changed to `EXECUTED`.

`tallyProposalResult(Proposal proposal, address[] delegates)` (ElectionCommission.sol#L283) does not check that the proposal state is `PROPOSED`. Therefore, once a proposal is executed, an attacker can trigger a vote tally again in order to change its state from `EXECUTED` to `PASSED`.

As a result, an attacker can re-execute a proposal.

Exploit Scenario
A proposal is passed and executed. The proposal pauses the system for one hour. Bob forces a voting tally, and then re-executes the proposal, causing the system to be paused a second time. As a result, Basis holders lose trust in the system.

Recommendation
In the short term, verify the proposal is in the `PASSED` state withinin `tallyProposalResult` before continuing with the tally.

`ElectionComission` lacks unit tests despite its complexity. In the long term, consider adding test coverage for all election scenarios.

7. Non-reinitialization of daily pledges will block the tally

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-Basis-007
Target: ElectionCommission.sol (v0)

Description

Everyday, ElectionCommission computes the number of pledges that voted during the day, through `tallyPledgesToday`. However, `tallyPledgesToday` is never re-initialized, and thus, it will have an incorrect value starting on the second election day. As a result, the pledges' median computation will be broken and the tally will be blocked.

The value of `tallyPledgesToday` is computed through `tallyOracleSupplyChange`.

```
function tallyOracleSupplyChange() public onlyDuringVoteTally returns (bool
complete) {
    …

    uint256 currPledgesToday = tallyPledgesToday;
    …

    if (weight > 0 && lastVoteDate == date) {
        currPledgesToday = currPledgesToday.add(weight);
        …

    tallyPledgesToday = currPledgesToday;
    …
```

Figure 1: `tallyOracleSupplyChange` *function*

The variable `tallyPledgesToday` is not reinitialized on a daily basis. As a result, it will contain every day the sum of the pledges computed on previous days; Once the number of daily pledges is greater than the number of pledges from active delegates, the median computation will never terminate. As a result, a call to `tallyOracleSupplyChange` will never return success. No modification to the Basis total supply will be possible and it will not be possible to vote for a new proposal.

After a few days of execution, Basis Token will have a fixed total supply of tokens, which will prevent its price from stabilizing.

Exploit Scenario

Intangible Labs launches the Basis token. After a few days, `ElectionCommission` is blocked. As a result, Basis doesn't have a stable price.

Recommendation
In the short term, reinitialize `tallyPledgesToday` to 0 during the daily tally reset (ElectionCommission.sol#L116-L123).

`ElectionComission` lacks unit tests despite its complexity. In the long term, consider adding test coverage of multi-day simulations.

8. A multiplication overflow allows an attacker to block the tally

Severity: High
Difficulty: Low
Type: Numerics
Finding ID: TOB-Basis-008
Target: ElectionCommission.sol (v0)

Description

Every day, delegates can vote for a new Basis total supply. If a malicious delegate votes for an arbitrarily large supply increase, they can block the tally by triggering a multiplication overflow.

A delegate can vote for a new Basis total supply through `postOracleSupplyChange`.

```
function postOracleSupplyChange(int256 supplyChange) public
notDuringVoteTally {
   shares.noteDelegateVoted(msg.sender);
   oracleSupplyChange[msg.sender] = supplyChange;
```

Figure 1: `postOracleSupplyChange` *function*

The `tallyOracleSupplyChange` function will call `convertSupplyChangeToPercentileChange` to convert the total supply to a percentage:

```
function tallyOracleSupplyChange() public onlyDuringVoteTally returns (bool
complete) {
 …
  int16 idx =
convertSupplyChangeToPercentileChange(oracleSupplyChange[address(currDelega
te)]);
```

Figure 2: `tallyOracleSupplyChange` *function*

The `convertSupplyChangeToPercentileChange` function uses `SafeMath` to multiply the total supply by 10,000.

```
function convertSupplyChangeToPercentileChange(int256 delta) internal view
returns (int16) {
   delta = delta.mul(int256(10000)) / basis.totalSupply().signed();
```

Figure 3: `convertSupplyChangeToPercentileChange` *function*

However, `SafeMath.mul` throws an error in case of overflow.

```
function mul(int256 a, int256 b) internal pure returns (int256 c) {
   if (a == 0) return 0;
   c = a * b;
   require(b == c / a);
}
```

Figure 4: `mul(int256 a, int256 b)` *function*

An attacker can prevent a successful call to `tallyOracleSupplyChange` by voting for a very large supply number that will trigger an overflow in `SafeMath.mul`. This will always cause `tallyOracleSupplyChange` to throw.

As a result, a malicious delegate can block the tally.

Exploit Scenario

The Basis token price decreases. A proposal is made to decrease the total supply. The majority of the delegates vote in favor. Bob is a delegate who wants to prevent the burn of tokens, and therefore votes for a new total supply of `2**255-1`. As a result, the tally is blocked and the Basis total supply does not decrease.

Recommendation

In the short term, prevent unrealistic values in `postOracleSupplyChange`. Additionally, be aware that SafeMath will throw if an overflow is triggered.

`ElectionComission` lacks unit tests despite its complexity. In the long term, consider testing all arithmetic operations though a fuzzer, such as Echidna, or symbolic execution engine, such as Manticore.

9. block.timestamp can be manipulated

Severity: Low
Type: Timing
Target: ElectionCommission.sol (v0)

Difficulty: High
Finding ID: TOB-Basis-009

Description
The Ethereum specification does not provide a guarantee that the time of EVM is correct. A malicious miner could manipulate the timestamp to perform actions such as invalidating votes in the `ElectionComission`, especially when they arrive near the tally period.

The ElectionComission contract depends on a precise mechanism to determine the current time. It is implemented using `block.timestamp`:

```
// Vote Tally is at EOD.
function duringVoteTally() public view returns (bool) {
   uint256 start =
currentDate().add(1).mul(tmVoteTallyPeriod).sub(tmVoteTallyLength);
   return now >= start;
}
```

Figure 1: `duringVoteTally` *function*

However, the Ethereum Virtual Machine only requires that the timestamp of a block is greater than the previous referenced block's timestamp:

H_{s} is the timestamp (in Unix's time()) of block H and must fulfil the relation:

$$H_{\mathrm{s}} > P(H)_{H_{\mathrm{s}}} \tag{48}$$

Figure 2: Timestamp specification in the Yellow Paper

There is a reference in the original Ethereum whitepaper to constraining timestamps to 15 minutes, but that information is out of date. Recently, the Ethereum 'Yellow Paper' has dropped that 15-minute limit. Even though some popular Ethereum clients like geth implement validations on the timestamps, there is no agreement on a particular mitigation.

Exploit Scenario
The Ethereum clients remove any timestamp constraint that is not in the Ethereum specification. Alice creates a proposal. Bob is a malicious miner, and wants to block Alice's proposal. He increases the timestamp of the current block and prevents the tally from occuring.

Recommendation
While a contract should not rely on `timestamp`, Ethereum does not provide any reliable alternative. In some cases, `block.number` can be used, but estimating the time from it is not precise enough for the purposes of `ElectionComission`.

The best mitigation is to be aware of this limitation and carefully monitor changes in the Ethereum specification and the client's implementation. Additionally, the blockchain's timestamps can be monitored to try to catch a miner performing this attack.

10. The approve function can be called on paused contracts

Severity: Low
Type: Access Controls
Target: ERC20.sol (v0)

Difficulty: Low
Finding ID: TOB-Basis-010

Description
The token included functionality to "pause" any further interactions. Specifically, once a token is paused, no further internal state changes should be possible until such time as the token is "un-paused". However, this assertion is not respected for the token's allowance within the approve function.

If a contract is in the paused state, internal state modifications should be forbidden. However, `ERC20.approve` does not revert when the contract is paused:

```
function approve(address _spender, uint256 _amount) public returns (bool
success) {
  require(_spender != address(0));

  allowance[msg.sender][_spender] = _amount;

  emit Approval(msg.sender, _spender, _amount);

  return true;
}
```

Figure 1: `approve` *function*

Exploit Scenario
A proposal to pause the Basis contract is approved in order to snapshot the token's data and migrate the contract. While the contract is paused, some clients can still successfully call `approve`. If the contract is migrated, some clients will incorrectly assume their allowances causing potential undefined behaviours.

Recommendation
In the short term,dd the `whenNotPaused` modifier to the `approve` function. Improve the `ERC20` unit tests to ensure `approve` cannot be called when the contract is paused.

In the long term, consider using the [Echidna](https://github.com/trailofbits/echidna) fuzzer or [Manticore](https://github.com/trailofbits/manticore) symbolic executor to ensure any function changing the state of the contract will revert if the contract is paused.

11. A race condition in the Whitelist contract's functions makes them ineffective

Severity: Medium
Type: Timing
Target: Whitelist.sol (v1)

Difficulty: High
Finding ID: TOB-Basis-011

Description

The `Whitelist` contract provides a set of functions to add, remove, or update addresses in a whitelist. Once the owner calls the `remove` or `update` functions with a user's address, that user should no longer be able to perform any operation with tokens.

This schema is vulnerable to a race condition if the user removed from the whitelist is monitoring unconfirmed transactions on the blockchain. If this user sees the transaction containing the call before it has been mined, they can call `transfer` to move their tokens to another address, effectively circumventing the restrictions imposed by the whitelist.

```
contract Whitelist is WhitelistInterface, Ownable {
  …
  function remove(address addr) public onlyOwner returns (bool success) {
  …
  }

  function update(address addr, bytes32 hash, uint expTime) public
onlyOwner returns (bool success) {
  …
  }
  …
}
```

Figure 1: the prototypes of `remove` *and* `update` *functions of the* `Whitelist` *contract.*

Exploit Scenario

1. Alice calls `remove(Bob)`. This will forbid Bob from transferring his tokens.
2. Bob sees the unconfirmed transaction and calls transfer to move all his tokens to another (whitelisted) account before Alice's transaction has been mined. He pays a higher fee to ensure that the transfer call will be mined before the `remove` call.
3. If Bob's transaction is mined before Alice's, the removal of Bob from the whitelist will be ineffective since he can still spend his tokens.

Recommendation

There is no straightforward solution to this issue. One possible mitigation is to pause the contract before any operation performed by `Whitelist`. So, if Alice calls `pause` with a high gas price and waits until the transaction is confirmed, a subsequent call to `remove(Bob)` will succeed in freezing Bob's balance.

12. An attacker can block the bond distribution

Severity: High
Type: Denial of Service
Target: BondTokenManager.sol (v1)

Difficulty: High
Finding ID: TOB-Basis-012

Description

An attacker can block the distribution of bonds through large transfers of Basis tokens to the bond token manager. This issue is caused by SafeMath's handling of overflows using revert.

A special user, the `controller` is the only user authorized to create new bonds and distribute tokens from them through invocation of `distributeBonds`:

```
function distributeToBond(bool senior) public onlyController returns (bool)
{
   …
   if (distributionType == DistributionType.NONE) {
       distributed = 0;
       originalBalance = basis.balanceOf(this);
       …
   }
   bool success = distributeToBond();
   …
}
```

Figure 1: `distributeToBond` *function*

This function calculates the amount of tokens to pay per bond using the `distributeToBond` internal function:

```
function distributeToBond() internal returns (bool) {
  …
  uint256 maxToDistribute = token.payoutPercent().mul(originalBalance) /
100;
  …
}
```

Figure 2: Relevant excerpt of the internal `distributeToBond` *implementation.*

An attacker can't call `distributeBonds` directly, but he can control the amount of tokens in `originalBalance` by performing a transfer to the `BondTokenManager` contract. The transfer of a large amount of tokens will trigger a revert in execution of `distributeToBond`.

```
function mul(int256 a, int256 b) internal pure returns (int256 c) {
   if (a == 0) return 0;
   c = a * b;
   require(b == c / a);
}
```

Figure 3: SafeMath's `mul` implementation

Exploit Scenario

Alice is the `controller` of the `BondTokenManager` and creates a token with a large `payoutPercent`. Bob transfers a significant amount of Basis to the `BondTokenManager`, with the intention to overflow during distribution. Alice subsequently invokes the `distibuteToBond` function, which reverts due to an integer overflow when calculating the maximum amount of Basis to distribute.

Recommendation

Ensure the `originalBalance` will never reach a value which could cause an integer overflow when multiplied by a token's `payoutPercent.`

13. Creating bonds with certain parameters can block the bond distribution

Severity: Medium
Type: Denial of Service
Target: BondTokenManager.sol (v1), BondToken.sol (v1)

Difficulty: High
Finding ID: TOB-Basis-013

Description

Creating bonds with special parameters can cause an unexpected revert that will block bond distribution.

A special user, the `controller`, is responsible for creating new bonds by calling `createBondToken`. This user is also able to distribute tokens from them by calling the `distributeBonds` function.

```
function distributeToBond(bool senior) public onlyController returns (bool)
{
   ...
   if (distributionType == DistributionType.NONE) {
       distributed = 0;
       originalBalance = basis.balanceOf(this);
       ...
   }
   bool success = distributeToBond();
   ...
}
```

Figure 1: `distributeToBond` *function*

This function performs some computations to calculate the number of tokens to pay per bound using the `distributeToBond` internal function:

```
function distributeToBond() internal returns (bool) {
  ...
  uint256 rights = token.faceValueRemaining();
  ...
}
```

Figure 2: Relevant excerpt of the internal `distributeToBond` *implementation.*

This function calls `faceValueRemaining` for each created bond.

```
function faceValueRemaining() public view returns (uint256) {
   uint256 date = electionCommission.currentDate();
```

```
    if (!expires || date < expireDate.sub(decayTerm)) return
_faceValueRemaining;
    if (date >= expireDate) return 0;
    // multiplying by frac via: a * (n/d) = a / d * n + (a % d) * n / d
    uint256 tm = expireDate.sub(date);
    uint256 maxPayout = (originalFaceValue / decayTerm).mul(tm) +
(originalFaceValue % decayTerm).mul(tm) / decayTerm;
    if (maxPayout <= distributed) return 0;
    return maxPayout.sub(distributed);
}
```

Figure 3: The `faceValueRemaining` function

However, if the bonds are created with invalid parameters, a call to `faceValueRemaining` will cause a revert, blocking the entire bond distribution procedure. For instance, creating a bond using the following parameters--where `now = 1` and `tmVoteTallyStartTm = 0`--will produce a revert when calling `faceValueRemaining`:

- idx = 0
- faceValue = 72355920835041424055675830507646146103459978360064439941438696660210395643904
- term = 43422033679676465764398625258674719296277291483336614717638829803138782855168
- decayTerm = 43422033679676465764398625258674719296277291483336614717638829803138782855168
- payoutPercent = 1

These values are just an example, as there are numerous parameters that can trigger this issue.

Exploit Scenario
Alice is the `controller` of the `BondTokenManager`. She creates a new bond with certain error-inducing parameters. The creation of bond tokens succeeds. However, if she calls `distributeBonds`, it will always revert causing the `BondTokenManager` contract to be trapped.

Recommendation

In the short term, carefully validate the parameters of the bond tokens during their creation: revert if they are not valid.

In the long term, consider using the Echidna fuzzer or the Manticore symbolic executor to check that no revert can happen during the call to `faceValueRemaining`.

14. ERC20 Incorrect return values and documentation

Severity: Informational
Difficulty: High
Type: Error Reporting
Finding ID: TOB-Basis-014
Target: ERC20.sol (v1)

Description

Inline documentation for the `doTransferFrom` and `doTranserTo` functions indicates that these functions should return `false` when no tokens are transferred. However, the implementations are incorrect and do not comply with the ERC20 standard. When transfering an `amount` of `0`, `false` is not returned. Furthermore, the `doTransferTo` function will never return `false`.

The `ERC20` contract is where the `transfer` function is defined, which invokes the `doTransfer` function.

```
function transfer(address _to, uint256 _amount) public returns (bool
success) {
    return doTransfer(msg.sender, _to, _amount);
}
```

Figure 1: Implementation of `transfer`

The `doTransfer` function subsequently executes `doTransferFrom` and `doTransferTo` functions which perform the appropriate modifications to account balances.

```
// note that transfers of 0 are used to invoke the
doTransferFrom/doTransferTo hooks.
function doTransfer(address _from, address _to, uint256 _amount) internal
returns (bool success) {
   require(_to != address(0));
   if (!doTransferFrom(_from, _amount)) {
       return false;
   }
   if (!doTransferTo(_to, _amount)) {
       revertTransferFrom(_from, _amount);
       return false;
   }
   …
}
```

Figure 2: Implementation of `doTransfer`

Within the `doTransferFrom` function, the amount to reduce from the specified address is checked against the address's balance to ensure a user is unable to transfer more funds than are available. However, a check to ensure that funds are removed is not performed.

```
// returns false when NO funds are removed, otherwise all funds were
removed
function doTransferFrom(address _from, uint256 _amount) internal
whenNotPaused returns (bool success) {
   if (_balanceOf[_from] < _amount) {
       emit Reason("Source does not have sufficient funds.");
       return false;
   }
   _balanceOf[_from] = _balanceOf[_from].sub(_amount);
   return true;
}
```

Figure 3: Implementation of `doTransferFrom`

In the `doTransfer` function, no checks are performed to ensure that funds are moved. This forces the function into returning only `true` or `revert` due to SafeMath addition.

```
// returns false when NO funds are moved, otherwise all funds were moved
function doTransferTo(address _to, uint256 _amount) internal whenNotPaused
returns (bool success) {
    _balanceOf[_to] = _balanceOf[_to].add(_amount);
    return true;
}
```

Figure 4: Implementation of `doTransferTo`

Exploit Scenario

Alice performs a transfer where `amount = 0`, expecting the transfer to fail. Due to a lack of checks to ensure an amount greater than 0 is being transferred, the transfer succeeds.

Recommendation

Ensure documentation of expected function behavior matches the implementation logic. Add checks to ensure movement of funds occurs as expected.

15. SimpleShares allows infinite token creation

Severity: Critical
Type: Data Validation
Target: SimpleShares.sol (v1)

Difficulty: Low
Finding ID: TOB-Basis-015

Description

The `SimpleShares` implementation of the `doTransferFrom` function uses `super.doTransferTo` instead of `super.doTransferFrom`. This would allow an attacker to create an infinite number of tokens through executing the `SimpleShares.transfer` function:

```
function transfer(address _to, uint256 _amount) public returns (bool
success) {
    return doTransfer(msg.sender, _to, _amount);
}
```

Figure 1: `SimpleShares` *Implementation of* `transfer`

The inherited `ERC20` contract's `doTransfer` function is then executed through the inheritance chain, which invokes `doTransferFrom`.

```
// note that transfers of 0 are used to invoke the
doTransferFrom/doTransferTo hooks.
function doTransfer(address _from, address _to, uint256 _amount) internal
returns (bool success) {
   require(_to != address(0));
   if (!doTransferFrom(_from, _amount)) {
       return false;
   }
   if (!doTransferTo(_to, _amount)) {
       revertTransferFrom(_from, _amount);
       return false;
   }
   …
}
```

Figure 2: `ERC20` *Implementation of* `doTransfer`

The invocation of `doTransferFrom` causes the execution of the `SimpleShares` implementation, which incorrectly executes `doTransferTo` instead of `doTransferFrom`.

```
function doTransferFrom(address _from, uint256 _amount) internal returns
```

```
(bool) {
    if (!applyDistributions(_from)) {
        return false;
    }
    return super.doTransferTo(_from, _amount);
}
```

Figure 3: `SimpleShares` *Implementation of* `doTransferFrom`

Exploit Scenario
Bob has 100 tokens. He transfers the tokens he has to his own address to obtain 200 tokens.

Recommendation
In the short term, invoke `super.doTransferFrom` instead of `super.doTransferTo` within the `SimpleShares` implementation of `doTransferFrom`.

`SimpleShares` has no unit tests. In the long term, ensure all contracts have unit tests to cover the expected functionality.

16. ShareToken whitelist cannot be initialized

Severity: High
Type: Access Controls
Target: ShareToken.sol (v1), WhiteListToken.sol (v1)

Difficulty: Low
Finding ID: TOB-Basis-016

Description

`ShareToken` inherits from the `WhitelistToken` contract, but there is currently no function to set the `whitelist`. This renders whitelist protection useless, as there is no code to add users to the whitelist.

```
contract WhitelistToken is SafeTransferToken {
    WhitelistInterface public whitelist;
```

Figure 1: The definition of the `whitelist` *function in the* `WhitelistToken` *contract*

When no `whitelist` is set, the `isWhitelisted` function always returns true. This results in any caller being able to execute any whitelist-protected function.

```
function isWhitelisted(address _to) public view returns (bool success) {
    return WhitelistInterface(0) == whitelist || whitelist.contains(_to);
}
```

Figure 2: The definition of the `isWhitelisted` *function in the* `WhitelistToken` *contract*

An example can be seen within the `WhitelistToken` implementation, where a `transfer` requires being added to the whitelist, but the `isWhitelisted` call will be useless if no `whitelist` is set.

```
function doTransferTo(address _to, uint256 _amount) internal returns (bool
success) {
    if (!isWhitelisted(_to)) {
        emit Reason("Destination address is not on the whitelist.");
        return false;
    }
    return super.doTransferTo(_to, _amount);
}
```

Figure 3: The definition of the `doTransfer` *function in the* `WhitelistToken` *contract*

Exploit Scenario

Alice deploys the `ShareToken`, and attempts to perform a `transfer` without setting the `whitelist`. Because no `whitelist` is specified, a call from any user to transfer passes the `whitelist` verification.

Recommendation
Implement a method of setting or modifying the `whitelist` variable in the `WhitelistToken` contract.

17. User can be penalized even if no tokens are transferred

Severity: Informational
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-Basis-017
Target: ShareToken.sol (v1)

Description

The `ShareToken` functions that transfer, mint or burn tokens apply user penalizations when tokens are moved using `realizeUserPenalty`.

```
function doTransferFrom(address _user, uint256 _amount) internal
whenNotPaused returns (bool success) {
    // Before sending any tokens we want to apply the penalty so as to not
allow gaming.
    if (!ShareTokenManagerInterface(controller).realizeUserPenalty(_user))
{
        return false;
    }
    …

function doTransferTo(address _user, uint256 _amount) internal
whenNotPaused returns (bool success) {
   // When receiving any tokens we want to apply the penalty so as to not
penalize the new tokens.
   if (!ShareTokenManagerInterface(controller).realizeUserPenalty(_user)) {
       return false;
   }
    …
```

Figure 1: Transfer related function implementations in the `ShareToken` *contract*

Under two special circumstances, users can be penalized despite no tokens being transferred:

(1) If a user performs a zero-tokens transfer or a transfer to his own address, the user will be penalized despite no tokens being moved.
(2) If a user performs a transfer to another address, but it fails, he can still be penalized. This circumstance requires the transfer failing when the call to `realizeUserPenalty` returns `false`:

```
function realizeUserPenalty(address _user) public returns (bool success) {
    // all methods of token transfers go through realizeUserPenalty
```

```
    // restrict realizing penalty (and therefore all transfers) during
voting periods
    if (electionCommission.duringVoteTally()) {
        emit Reason("Cannot realize user penalty during voting period.");
        return false;
    }
    if (!realizeDelegatePenalty(uDelegateSelected[_user])) return false;
    …
```

Figure 2: the `realizeUserPenaly` *function*

Exploit Scenario

Alice starts a token transfer of her Share tokens but uses a lower amount of gas than necessary. Instead of reverting as expected, the transaction returns an error and Alice gets penalized, despite no tokens being transferred.

Recommendation

Ensure tokens are successfully transferred before user penalization occurs, and enough gas is provided to execute the `realizeDelegatePenalty` function during a transfer.

18. <removed after discussion with Basis>

Note: this issue related to penalties not being applied when token movement occurred through the use of escrow functions. Upon further investigation and a discussion with Basis, it was determined that this movement of tokens worked as expected.

19. SimpleShares contract allows execution during pause

Severity: High
Difficulty: Low
Type: Access Controls
Finding ID: TOB-Basis-019
Target: SimpleShares.sol (v1), MultiDistributor.sol (v1)

Description

Some functions defined in the `SimpleShares` contract are not restricted to prevent execution during a paused contract state. This leads to successful function invocation during a paused state.

The `SimpleShares` implementation of `distribute` allows for execution during a paused state, invoking the `MultiDistributor` implementation of `distribute`.

```
// Distribute all unallocated balance of token to current token holders.
function distribute(address token) public returns (bool) {
    return super.distribute(token, totalSupply);
}
```

Figure 1: `SimpleShares` *implementation of* `distribute`

Since neither the `SimpleShares` or `MultiDistributor` implementations are modified to restrict execution during a paused state, successful execution can occur.

```
function distribute(address token, uint256 totalSupply) internal returns
(bool)
```

Figure 2: `MultiDistributor` *function definition of* `distribute`

Exploit Scenario

The `SimpleShares` contract enters a paused state. Bob invokes the `SimpleShares` `distribute` function, which executes successfully despite the contract being paused.

Recommendation

In the short term, ensure appropriate modifiers are applied to the `SimpleShares` contract functions to prevent successful invocations during a paused state.

In the long term, modify the `MultiDistributor` contract to take into consideration if the base contract is paused.

20. setMinimumDistribution is not protected

Severity: High
Difficulty: Low
Type: Access Controls
Finding ID: TOB-Basis-020
Target: MultiDistributor.sol (v1)

Description
An attacker can call the `setMinimumDistribution` function in `MultiDistributor` to add useless values to the tokens list and increase the gas cost of all related functions.

The `setMinimumDistribution` function allows a trusted participant to add a distribution into the tokens list.

```
// The minimum distribution requirement is to prevent DOS attack via many
negligible distributions. Thus, this
// method should ONLY be used by trusted participants. A minimum
distribution of zero prevents any and all
// distributions of this token.
function setMinimumDistribution(address token, uint256
tokenMinDistribution) public {
   uint256 idx = tokenIdx[token];
   if (idx == 0) {
     tokenIdx[token] = idx = tokens.length;
     tokens.push(token);
   }

   minDistribution[idx] = tokenMinDistribution;
}
```

Figure 1: The `MultiDistributor` *implementation of* `setMinimumDistribution`

The documentation indicates that this method should be "*ONLY be used by trusted participants*" but that requirement is not implemented, so any user could add distributions into the `tokens` list.

Exploit Scenario
Bob calls the `setMinimumDistribution` a large number of times to extend the `tokens` list and increase the gas cost of related operations.

Recommendation
In the short term, `MultiDistributor` should be `ownable`, `controllable` or `whitelistable` in order to implement proper user validation for `setMinimumDistribution`. Select one of these alternatives and protect the function so only trusted participants can use it.

In the long term, carefully review all public contract methods to ensure that they are only callable by authorized users. Implement unit tests to validate the implementation.

21. <removed after discussion with Basis>

Note: this issue was related to minting and burning operations not appropriately updating Basis token distributions. Upon further investigation and a discussion with Basis, it was determined that this was expected functionality.

22. Basis owner is the only mint-able address

Severity: Low
Type: Access Controls
Target: Basis.sol (v1)

Difficulty: High
Finding ID: TOB-Basis-022

Description

The Basis contract has an `owner`, but has no way to recover or reset it if the private key is compromised or lost.

The Basis contract's `owner` is set during initialization:

```
contract Basis is Ownable, HasECVotable, Controllable, SafeTransferToken,
ECVotePausable {
```

Figure 1: The `Basis` contract definition

The `owner` is only used to receive a proposal-specified amount of minted tokens from the reserve when a proposal invokes `transferFromReserve`:

```
function transferFromReserve(uint256 _amount) public onlyByECVote
whenNotPaused {
  require(reserveRemaining >= _amount);
  reserveRemaining -= _amount;
  super.mint(owner, _amount);
}
```

Figure 2: the `transferFromReserve` function of `Basis`

If the private key of the `owner` account is compromised or lost, the call to `transferFromReserve` will be ineffective. The election commision cannot pass a proposal to change the owner, thus its tokens will be trapped in the reserve.

Exploit Scenario

Bob obtains the private key from the owner account of a Basis token contract. This intrusion is disclosed to the delegates, but they are unable to change the owner even by passing a proposal. As a result, the Basis tokens are trapped in the reserve.

Recommendation

In the short term, include a new parameter in `transferFromReserve` in order to allow the specification of the address to transfer the minted tokens.

In the long term, carefully specify every user and role in the contracts. Document how the keys are created and managed, and how to proceed in case of a compromise.

23. The Whitelist contract is not controlled by the election commission

Severity: Low
Difficulty: High
Type: Access Controls
Finding ID: TOB-Basis-023
Target: SimpleShares.sol (v1)

Description
The election commission can set the pointer to the `Whitelist` contracts in `SimpleShares`, but does not control the contracts themselves.

The election commission votes on proposals to control the whitelists within the `SimpleShares` contract. These pointers are updated using the `setWhitelist` and `setDistributionWhiteList` functions:

```
function setWhitelist(address _whitelist) public onlyByECVote {
  whitelist = WhitelistInterface(_whitelist);
}

function setDistributionWhiteList(address _whitelist) public onlyByECVote {
  distributionWhitelist = WhitelistInterface(_whitelist);
}
```

Figure 1: whitelist-setting functions in `SimpleShares`

However, the election commission does not control the Whitelist contracts themselves. These are controlled by their owners:

```
contract WhiteList is WhiteListInterface, Ownable {
…
function add(address addr, bytes32 hash, uint expTime) public onlyOwner
returns (bool success) { … }
function remove(address addr) public onlyOwner returns (bool success) { …
}
function update(address addr, bytes32 hash, uint expTime) public onlyOwner
returns (bool success) { … }
…
```

Figure 2: Relevant `WhiteList` *function definitions*

Exploit Scenario
Bob obtains the private key from the owner account of a `Whitelist` contract. Even if this intrusion is immediately disclosed, Bob has plenty of time to manipulate the whitelist, since a proposal to set a new `Whitelist` contract must be passed, voted, then executed.

Recommendation

In the short term, ensure that the `Whitelist` contracts can be controlled exclusively by the election commission.

In the long term, carefully specify every user and role in the contracts. Document how the keys are created and managed, and how to proceed in case of a compromise.

24. pendingDistribution and minDistribution arrays are not properly used

Severity: High
Difficulty: Low
Type: Data Validation
Finding ID: TOB-Basis-024
Target: MultiDistributor.sol (v1)

Description

The `pendingDistribution` and `minDistribution` dynamic arrays in `MultiDistributor` are not properly used and will lock the contract in its initial state.

The `pendingDistribution` dynamic array keeps track of the amount to be withdrawn per token. The `minDistribution` dynamic array keeps track of the minimum required distribution per token:

```
// amount yet to be withdrawn per token
uint256[] public pendingDistribution;
// minimum distribution required per token
uint256[] public minDistribution;
```

Figure 1: `MultiDistributor` *definition of* `pendingDistribution` *and* `minDistribution`

Solidity dynamic arrays must be extended before accessing them, either using the `push` method or modifying their `length` attribute. However, these dynamic arrays are used in the `MultiDistributor` contract without extension. Therefore, any access to these arrays will be out-of-bounds, reverting the transaction.

Exploit Scenario

Alice deploys the `SimpleShare` tokens. Any transaction accessing the `pendingDistribution` and `minDistribution` dynamic arrays will revert, locking the contract in its initial state.

Recommendation

In the short term, avoid the use of all dynamic arrays in `MultiDistributor` to ensure they are properly extended and no out-of-bounds access is possible.

In the long term, avoid the use of dynamic arrays. Their use is prone to out-of-bounds access and incurs comparatively expensive gas consumption. Additionally, `MultiDistributor` has no unit tests. Ensure all contracts have unit tests to cover the expected functionality.

A. Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Numerics	Related to numeric calculations
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General Recommendation

- Thoroughly document your code. Documentation helps reviewers to understand each function's purpose and helps with review overall.
- Follow the [Solidity naming convention guide](). Following a standard naming convention helps the review of the code. For example, uppercase variables are expected to be constant. These are not:
  - `MAX_INACTIVE_PERIOD` (ElectionCommission.sol#L96)
  - `MIN_PERCENT_CHANGE` (ElectionCommission.sol#L102)
  - `MAX_PERCENT_CHANGE` (ElectionCommission.sol#L103)
- Ensure consistent use of SafeMath operations. Mixing native math and SafeMath without care could cause unexpected results.

ElectionCommision.sol

- Emit events for voting operations, such as when someone votes, or when a proposal is executed. Events help to follow the correct execution of the contract.
- Remove `import "../traits/Controllable.sol";` (ElectionCommission.sol#L7). The import is not used.
- Consider using boolean for `proposalDelegateVote. proposalDelegateVote` considers a value greater than 0 as a yes, and less or equal to 0 as a no. While 0 can also indicate that a delegate did not vote, the information is not used. Using a boolean would simplify the code.
- Document the expected behavior of the different finite state machines. In particular, the variables that should be reinitialized, and the assumptions on the state transitions.
- Use constant variables instead of constant numbers. The use of constant variables clarifies the code and prevents mistakes caused by a constant that isn't updated. For example, use a constant variable for 10,000 in ElectionCommission.sol#L204 and ElectionCommission.sol#L208.

ERC20.sol

- Emit events for adding and revoking escrow, regardless of the number of tokens. Events help to follow the correct execution of the contract.

WhiteListToken.sol

- This contract is incorrectly imported as `WhitelistToken.sol in:`

    - `LimitedDistributionToken.sol`
    - `ShareToken.sol`
    - `SimpleShares.sol`

  This will make the compilation fail in case-sensitive file systems such as Linux.
- A proper constructor and `setWhitelist` function should be carefully implemented in the `WhitelistToken` contract. Otherwise, it could be incorrectly used by a developer.
- Documentation is incorrect: the contract is not abstract. All its functions have definitions.

WhiteList.sol

- The use of a mapping and a dynamic array to implement a whitelist is inefficient, and could be implemented as a mapping from address to structs. This change could drastically reduce the contract's complexity and optimize its gas use.
- The `getDetails` function returns a value even if the address is not in the whitelist. This could cause an issue in the future. Ensure that it reverts if `index` is invalid (Whitelist.sol#83).

WhiteListInterface.sol

- This contract is incorrectly imported as `WhitelistInterface.sol` in:
    - `MultiOwnerWhiteList.sol`
    - `WhiteList.sol`
    - `WhiteListToken.sol`

  This will make the compilation fail in case-sensitive file systems such as Linux.

BondTokenManager.sol

- Ensure the `distributionState` is properly used and updated. In the current implementation the variable is compared in an expression that always evaluates to `true` (BondTokenManager.sol#L129).

Basis.sol

- Ensure all the methods that modify the state of the contract are marked with `whenNotPaused`.
- The Basis contract inherits from `SafeTransferToken`, which implements the public functions `addEscrow`, `revokeAllEscrow`, `revokeEscrow`, and `refuseEscrow`. These functions invoke internal function definitions, passing a blank--and useless--string (`""`) as the `_logData` on every invocation. Remove these unused parameters.

SimpleShares.sol

- The `SimpleShares` contract is not initialized with a `whitelist` or `distributionWhitelist`. This results in improper restriction of whitelist-protected functions after deployment, and before whitelists are voted in.

MultiOwnerWhitelist.sol

- If a whitelister is added through `addWhitelister`, paused by `pauseWhitelister` then removed by `revokeWhitelister` and then added again, they are still paused. This might be a bit confusing because whitelister isn't paused when they're added for the first time. Consider holding the same invariants for non-existent whitelisters as for the removed ones and adding tests for this behavior.

C. Manticore formal verification

We reviewed the feasibility of formally verifying the contract with [Manticore](), our open-source dynamic EVM analysis tool that takes advantage of symbolic execution. Symbolic execution allows us to explore program behavior in a broader way than classical testing methods, such as fuzzing.

During this assessment we used Manticore to determine if certain invalid contract states were feasible. When applied to a simplified `BondToken` contract (Figure 2), Manticore identified a scenario in which the `faceValueRemaining` function which reverts when certain parameters are provided. The [TOB-Basis-013]() finding details the parameters leading to the revert, and the contract properties affected.

```
from manticore.ethereum import ManticoreEVM, evm, Operators

m = ManticoreEVM()
m.verbosity(2)

user_account = m.create_account(balance=1000, name='user_account')
print("[+] Creating a user account", user_account.name)

contract_account = m.solidity_create_contract(open("BondTokenInst.sol","r"),
owner=user_account, name='contract_account', contract_name='BondToken', args=None)
print("[+] Creating a contract account", contract_account.name)

contract_account.faceValueRemaining()

m.finalize()
print("[+] Look for results in %s" % m.workspace)
```

Figure 1: Manticore testing script which symbolically executes the `faceValueRemaining` *function*

```
pragma solidity ^0.4.23;

library SafeMath {

   function add(uint256 a, uint256 b) internal pure returns (uint256 c) {
      c = a + b;
      require(c >= a);
   }
```

```
    function sub(uint256 a, uint256 b) internal pure returns (uint256 c) {
        c = a - b;
        require(c <= a);
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
        if (a == 0) return 0;
        c = a * b;
        require(b == c / a);
    }
}

contract BondToken {

    using SafeMath for uint256;

    string private name = "Basis Bond ";
    string public symbol = "BB.";

    uint8 constant numDecimals = 12;

    bool public expires;

    // The date this bond expires worthless. Unit is an election commission date.
    uint256 public expireDate;

    // The amount of time before the expireDate we start to decay our paymentFactor
(straight line). Unit is number of
    // days as defined by election commission.
    uint256 public decayTerm;

    // This is used to compute the decay.
    uint256 public originalFaceValue;

    // This is used by the policy to limit payout. It is between (inclusively) 1 and 100.
    uint256 public payoutPercent;

    //ElectionCommissionInterface electionCommission;

    // From Distributor
    uint256 public distributed;
```

```
  // From LimitedDistributor
  uint256 private _faceValueRemaining;

  // now instrumentation
  uint256 private inst_now;

  // from ElectionCommisionInterface

  uint256 private tmVoteTallyStartTm;
  uint256 private constant tmVoteTallyPeriod = 1 days;

constructor(//SafeTransferToken _issueToken,
          //ElectionCommissionInterface _eci,
          uint _idx,
          uint256 _faceValue,
          uint256 _term,
          uint256 _decayTerm,
          uint256 _payoutPercent,
          uint256 _tmVoteTallyStartTm,
          uint256 _now) public
    //LimitedDistributionToken(numDecimals, _issueToken, _faceValue)
  {
    require(_payoutPercent > 0 && _payoutPercent <= 100);
    tmVoteTallyStartTm = _tmVoteTallyStartTm;
    inst_now = _now;

    payoutPercent = _payoutPercent;

    //name = appendIdx(name, _idx);
    //symbol = appendIdx(symbol, _idx);

    expires = _term > 0;
    uint256 date = currentDate();
    expireDate = date.add(_term);

    require(_decayTerm <= _term);
    decayTerm  = _decayTerm;

    originalFaceValue = _faceValue;
  }
```

```
  function currentDate() public view returns (uint256) {
      return inst_now.sub(tmVoteTallyStartTm) / tmVoteTallyPeriod;
   }

   function faceValueRemaining() public view returns (uint256) {
      uint256 date = currentDate(); //electionCommission.currentDate();
      if (!expires || date < expireDate.sub(decayTerm)) return _faceValueRemaining;
      if (date >= expireDate) return 0;
      // multiplying by frac via: a * (n/d) = a / d * n + (a % d) * n / d
      uint256 tm = expireDate.sub(date);
      uint256 maxPayout = (originalFaceValue / decayTerm).mul(tm) + (originalFaceValue
% decayTerm).mul(tm) / decayTerm;
      if (maxPayout <= distributed) return 0;
      return maxPayout.sub(distributed);
   }
}
```

Figure 2: `BondTokenInst.sol`

D. Echidna property-based testing

Trail of Bits used Echidna, our property-based testing framework, to find logic errors in the Solidity components of Basis.

During the engagement, Trail of Bits produced a custom Echidna testing harness for Basis' `SimpleShares` ERC20 token. This harness initializes the token and mints an appropriate amount of shares for two users. It then executes a random sequence of API calls from the two minted addresses in an attempt to cause anomalous behavior.

The harness includes tests of ERC20 invariants (*e.g.*, token burn, `balanceOf` correctness, *&c.*), and ERC20 edge cases (*e.g.*, transferring tokens to one's self and transferring zero tokens). Testing resulted in the identification of TOB-Basis-015, with the relevant harness and settings included below in Figure 1 and Figure 2.

```yaml
testLimit: 100
epochs: 1
range: 2
printCoverage: false
solcArgs: "--allow-paths ."
addrList: [0x0, 0x00a329c0648769a73afac7f9381e08fb43dbea70,
0x67518339e369ab3d591d3569ab0a0d83b2ff5198]
returnType: Success
```

Figure 1: The `SimpleShares_test.yaml` configuration file which defines `0x0` and two minted addresses within the test harness to be used as addresses within test execution.

```solidity
import "./contracts/tokens/SimpleShares.sol";

contract TEST is SimpleShares {
  address testerAddr = 0x00a329c0648769a73afac7f9381e08fb43dbea70;
  address otherAddr = 0x67518339e369ab3d591d3569ab0a0d83b2ff5198;
  uint256 initial_totalSupply;

  function TEST()
    SimpleShares()
  {
     uint256 initialShares = uint256(10)**9 * uint256(10)**numDecimals;
     super.burn(initialShares);
     initial_totalSupply = initialShares;
     super.mint(testerAddr, initial_totalSupply/2);
     super.mint(otherAddr, initial_totalSupply/2);
```

```
    require(balanceOf(testerAddr) == initial_totalSupply/2);
    require(balanceOf(otherAddr) == initial_totalSupply/2);

  }

  function echidna_max_balance() returns (bool) {
    return ((balanceOf(testerAddr) <= totalSupply/2) && balanceOf(otherAddr) >=
totalSupply/2);
  }
}
```

Figure 2: `SimpleShares_test.sol`*, which defines the* `SimpleShares` *contract test harness, including the* `max_balance` *property test.*

```
$ echidna-test SimpleShares_test.sol --config SimpleShares_test.yaml
…
━━━━━ SimpleShares.sol ━━━━━
  ✗ "echidna_max_balance" failed after 84 tests and 215 shrinks.

    | Call sequence: transfer(a329c0648769a73afac7f9381e08fb43dbea70,1);

  ✗ 1 failed.
```

Figure 3: An example run of Echidna with the `SimpleShares_test.sol` *test harness, including test results.*

E. Fix Log

Basis addressed issues TOB-Basis-001 to TOB-Basis-024 in their codebase as a result of the assessment. Each of the fixes was verified by the audit team. The reviewed code is available in git revision `951a656701535b69f3ce881b2d2ed18cb4556367`.

ID	Title	Severity	Status
01	Incorrect allowance check permits an attacker to perform undesired transactions to an allowed account	Medium	Fixed
02	BasisPolicy.executeOracleVote always returns false	Low	Fixed
03	Incorrect event information emission in Ownable.acceptOwnership	Low	Fixed
04	Incorrect authorization schema between BondTokenManager and the policies will prevent calls to createBondToken	High	Fixed
05	Re-use of a delegate address allows invalid proposals to execute	High	Fixed
06	Previously executed proposals can be re-executed	High	Fixed
07	Non-reinitialization of daily pledges will block the tally	High	Fixed
08	A multiplication overflow allows an attacker to block the tally	High	Fixed
09	block.timestamp() can be manipulated	Low	Will not fix
10	The approve function can be called on paused contracts	Low	In progress
11	A race condition in the Whitelist contract's functions makes them ineffective	Medium	Will not fix
12	An attacker can block the bond distribution	High	In progress
13	Creating bonds with certain parameters can block the bond distribution	Medium	In progress
14	ERC20 Incorrect return values and documentation	Informational	Fixed
15	SimpleShares allows infinite token creation	High	Fixed

16	ShareToken whitelist cannot be initialized	High	Fixed
17	User can be penalized even if no tokens are transferred	Informational	Fixed
18	Tokens can be moved without penalties using escrow functions	Informational	False positive
19	SimpleShares contract allows execution during pause	High	Fixed
20	setMinimumDistribution is not protected	High	Fixed
21	Minting and burning tokens will not update Basis tokens distributions	Low	False positive
22	Basis owner is the only mint-able address	Low	Fixed
23	The Whitelist contract is not controlled by the election commission	Low	Fixed
24	pendingDistribution and minDistribution arrays are not properly used	High	Fixed

Detailed Fix Log

Finding 1: Incorrect allowance check permits an attacker to perform undesired transactions to an allowed account
This appears to be resolved through changing `allowance[_from][_to]` to `allowance[_from][msg.sender]`.

Finding 2: BasisPolicy.executeOracleVote always returns false
This appears to be resolved through changing the `executeOracleVote` function name to `executePolicy`, and removing the `finished` variable, opting for explicit return paths.

Finding 3: Incorrect event information emission in Ownable.acceptOwnership
This appears to be resolved through the emission of the `OwnershipTransferred` event before `owner` reassignment.

Finding 4: Incorrect authorization schema between BondTokenManager and the policies will prevent calls to createBondToken
This appears to be resolved through the protection of the `createBondToken` with the `onlyModifiablePolicy` modifier, however, further testing is still needed to ensure complete resolution.

Finding 5: Re-use of a delegate address allows invalid proposals to execute
The checks in `tallyProposalResult` have been modified to ensure that delegates must be unique.

Finding 6: Previously executed proposals can be re-executed
This issue has been resolved by adding a check at the beginning of `tallyProposalResult` to ensure that the proposal has not been executed.

Finding 7: Non-reinitialization of daily pledges will block the tally
A new check has been added at the beginning of `tallyOracleSupplyChange` that resets `tallyPledgesToday` to 0 if it is no longer the same day as `lastTallyDate`.

Finding 8: A multiplication overflow allows an attacker to block the tally
This has been resolved by adding an overflow check in the `convertSupplyChangeToPercentileChange` function.

Finding 14: ERC20 incorrect return values and documentation
This issue has been resolved by changing the inline documents of `doTransferFrom` and `doTransferTo` functions.

Finding 15: SimpleShares allows infinite token creation
This has been resolved by changing `super.doTransferTo` to `super.doTransferFrom`.

Finding 16: ShareToken whitelist cannot be initialized
This issue has been resolved by adding a `setWhitelist` function into `ShareToken`. Note that a similar issue might appear in the future. We recommend creating an abstract `setWhitelist` function in `WhitelistToken`. This cannot be done currently because different tokens use different interfaces to set a whitelist:

- `BondToken:        function setWhitelist(WhitelistInterface newWhitelist)`
- `ShareToken:       function setWhitelist(address _whitelist)`
- `SimpleShares:     function setWhitelist(address _whitelist)`

Finding 17: User can be penalized even if no tokens are transferred
This issue has been resolved by adding documentation to `ShareToken` contract.

Finding 18: Tokens can be moved without penalties using escrow functions
This issue has been confirmed as a false positive.

Finding 19: SimpleShares contract allows execution during pause
This issue has been resolved by making the `MultiDistributor` `Pausable` and applying the `whenNotPaused` modifier to its functions. The modifier has also been applied to `SimpleShares's distribution` function, which is fine.

Finding 20: `setMinimumDistribution` is not protected
This issue has been resolved by renaming and making the `setMinimumDistribution_` function internal (the underscore has been appended to its name).

Finding 21: Minting and burning tokens will not update Basis tokens distributions
This issue has been confirmed as a false positive.

Finding 22: Basis owner is the only mint-able address
This issue has been resolved by removing the functionality.

Finding 23: The whitelist contract is not controlled by the election commission
This issue has been resolved by replacing the `Whitelist` contract with the `MultiOwnerWhitelist` contract which doesn't have the pointed issue.

Finding 24: `pendingDistribution` and `minDistribution` arrays are not properly used
This issue has been resolved by setting initial `minDistribution` and `pendingDistribution` lengths and pushing values in `setMinimumDistribution_`.