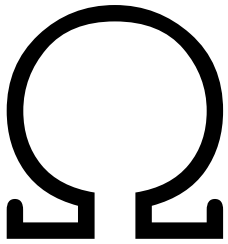
Delphia Token Contracts Audit

Final Report

November 29, 2021



Team Omega

Summary

Delphia has asked Team Omega to audit the contracts that define the behavior of the Delphia Token Contracts.

We found 1 critical issue.

We classified 4 issues as “medium” - issues we strongly believe you should address but that do not lead to loss of funds. In addition, 16 issues were classified as “low”.

An additional 19 issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Methods Used

Code Review

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

The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided.

Automatic analysis

We have used several automated analysis tools, including MythX and Remix to detect common potential vulnerabilities. No (true) high severity issues were identified with the automated processes. Some low severity issues, concerning mostly the state variables and functions visibility, were found and we have included them below in the appropriate parts of the report.

Scope of the Audit

The audit concerns the contracts committed here:
https://github.com/Delphia/token-contracts/commit/ec3f2d3e0e077db3629aef1381e1cedc3551c357

And specifically the following contracts that we have reviewed closely:

```
contracts
├── CoordinationToken.sol
├── DelphiaEscrow.sol
├── SecurityToken.sol
```

Disclaimer

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

Resolution

After delivering an intermediate report, the developers addressed a number of issues in https://github.com/Delphia/token-contracts/commit/220d2ac2dbedd3b4e8495ad6fb1e0320928e233e . We audited the changes and have incorporated them in this report.

Findings

General

G1. Licensing and attribution [low] [resolved]

The repository does not contain any licensing information. The code for the BancorFormula and BondingCurveToken contracts has been copied (almost) verbatim from the commons-stack repository, but the original Apache license has been removed on some of the files. When these contracts are published on the blockchain, Delphia will likely want to make the source code public. At that point, Delphia will need to assert copyright and provide a license that is compatible with the software dependencies.
Recommendation: Maintain the Apache license information in the files in utils. Choose a compatible license for the other files, and create a LICENSE file in the root directory and add SPDX license identifiers in the source files.
Severity: Low
Resolution: This issue was resolved.

G2. Incomplete Specifications [info] [partially resolved]

The specifications for the software are in this document https://hackmd.io/@mzargham/rkuuWa3o_ The specifications are incomplete and outdated (there are a number of "Outstanding Decisions for Delphia", and many details are implemented differently).
It did not seem useful for us to note all the differences, but we mention some specific divergences in the text below where that seemed useful.
Recommendation: Update the specifications to make sure the software matches expectations, as it is currently hard to verify if the software works as intended.
Severity: Info
Resolution: This issue was partially resolved.

G3. Compiler warnings [low] [resolved]

The compiler issues a number of warnings. Most of them pertain to missing SPDX license identifiers (see G1). In addition, the compiler issues the following warnings:

```
Warning: Visibility for constructor is ignored. If you want the contract to be
non-deployable, making it "abstract" is sufficient.
   --> contracts/utils/Power.sol:214:5:
    |
214 |     constructor() public {
    |     ^ (Relevant source part starts here and spans across multiple lines).
```

```
Warning: Unused function parameter. Remove or comment out the variable name to
silence this warning.
   --> contracts/SecurityToken.sol:147:67:
    |
147 |     function detectTransferRestriction (address from, address to, uint256
value)
  |                           ^^^^^^^^^^^^
```

Recommendation: Fix these warnings as described below

Severity: Low

Resolution: This issue was resolved.

G4. Linter Warnings [info] [resolved]

Running the linter gives 36 warnings:

```
./contracts/CoordinationToken.sol
   8:5  warning  Explicitly mark visibility of state    state-visibility
   9:5  warning  Explicitly mark visibility of state    state-visibility
  10:5  warning  Explicitly mark visibility of state    state-visibility
  11:5  warning  Explicitly mark visibility of state    state-visibility
  12:5  warning  Explicitly mark visibility of state    state-visibility
  22:9  warning  Error message for require is too long  reason-string
  40:9  warning  Error message for require is too long  reason-string
  47:9  warning  Error message for require is too long  reason-string

./contracts/DelphiaEscrow.sol
   9:5   warning  Explicitly mark visibility of state
state-visibility
  28:9   warning  Error message for require is too long
reason-string
  51:9   warning  Error message for require is too long
reason-string
  62:9   warning  Error message for require is too long
reason-string
  72:9   warning  Error message for require is too long
reason-string
  76:9   warning  Possible reentrancy vulnerabilities. Avoid state changes after
transfer  reentrancy
  84:9   warning  Error message for require is too long
reason-string
  87:13  warning  Error message for require is too long
reason-string

./contracts/SecurityToken.sol
   16:5   warning  Explicitly mark visibility of state    state-visibility
   17:5   warning  Explicitly mark visibility of state    state-visibility
   18:5   warning  Explicitly mark visibility of state    state-visibility
   19:5   warning  Explicitly mark visibility of state    state-visibility
```

```
  20:5   warning  Explicitly mark visibility of state    state-visibility
  21:5   warning  Explicitly mark visibility of state    state-visibility
  22:5   warning  Explicitly mark visibility of state    state-visibility
  23:5   warning  Explicitly mark visibility of state    state-visibility
  43:9   warning  Error message for require is too long  reason-string
  50:9   warning  Error message for require is too long  reason-string
  90:9   warning  Error message for require is too long  reason-string
 100:9   warning  Error message for require is too long  reason-string
 111:9   warning  Error message for require is too long  reason-string
 135:9   warning  Error message for require is too long  reason-string
 137:9   warning  Error message for require is too long  reason-string
 147:67  warning  Variable "value" is unused             no-unused-vars
 213:9   warning  Error message for require is too long  reason-string
 215:9   warning  Error message for require is too long  reason-string
 235:9   warning  Error message for require is too long  reason-string
 240:9   warning  Error message for require is too long  reason-string

✖ 36 problems (0 errors, 36 warnings)
```

Recommendation:
Change the linter configuration and instruct the linter to ignore the length of the error message (or consider rewriting the messages). Explicitly mark the visibility of the state, which can save some gas costs for callers in some cases. Other warnings are discussed below
Severity: Info
Resolution: This issue was resolved.

G5. Upgrade solidity version [info] [resolved]

The contracts use solidity 0.8.4. The latest version at the moment of writing is 0.8.9. This new version contains only minor fixes that do not seem directly relevant, but there is no reason to not upgrade
Recommendation: Upgrade solidity to 0.8.9.
Severity: Info
Resolution: This issue was resolved, the solidity version has been updated to version 0.8.9.

CoordinationToken.sol

The CoordinationToken contract allows users to deposit security tokens in exchange for coordination tokens. The exchange rate is set by a bonding curve.

C1. bondToMint mints wrong amount of tokens on first mint [critical] [not resolved]

The bondToMint implementation uses the code from BancorFormula.sol to calculate how many coordination tokens a user receives, given a certain amount of security tokens to bond.
However, this logic is overridden in lines 30-36 for the case that totalSupply() (which is the amount of already minted coordination tokens) is equal to 0. (This edge case is not handled by the original Bancor code. Note that there is *another* edge case not handled by the Bancor contracts - when the amount of collateral is 0. The bondToMint call will fail if this is the case)

This logic does *not* give the same results as the BancorFormula does, in the sense that when starting with a totalSupply of 0 tokens, depositing first 1 token (with a return calculated by the custom logic) and then 99 tokens (with a return calculated with the Bancor code) does not give the same output as depositing 100 tokens in a single transaction: the latter case will mint more Coordination tokens than the former.
The custom logic is applied only on deposits, but not on withdrawals (where the bancor formula is used). This has as a consequence that a user that deposits X tokens to get N coordination tokens, and then immediately returns N - she will get less than the expected (X minus the fee) tokens back: some of that value will flow to the first depositor.
Recommendation: Remove the code on lines 30-36 completely, and exclusively rely on the code in the BancorFormula contract to calculate the amount of coordination tokens to mint.
To avoid the edge cases where totalSupply is equal to 0, set the initial supply of the coordination token to a non-zero value (cf also issue C3). For the other edge case, where the poolSize is 0 (not handled in the current code), pre-seed the poolSize to a non-zero value (for example by sending tokens directly to the contract, or otherwise adding a virtual amount of security tokens to the actual balance).
Severity: Critical. Although it is an edge case, later depositors will lose (some) funds that will go to the first depositor.
Resolution: The custom calculation in the first minting has been replaced with a 1 to 1 minting for this first minting call. This simplifies the logic, but does not resolve the issue. This has been a conscious decision of the Delphia team after being made aware of the consequences detailed above.

C2. Gas price is not settable [medium] [resolved]

There is no public setter function for the gasPrice variable, which means that it will stay hard coded at 50Gwei. This means that users will not be able to withdraw their security tokens from the contract if the gas price is higher than 50Gwei
Recommendation: Create an external setGasPrice function, that calls a function _setGasPrice, and which is callable only by a specific address. This requires an admin role to limit access to the setter function: probably the best approach is probably to make the CoordinationToken contract inherit from Ownable. Alternatively, as we recommend in C5, remove the gas price logic altogether and replace it by a more effective front-running method.
Severity: Medium
Resolution: This issue was resolved, maximum gas price is now settable.

C3. Some bonding curve parameters are hardcoded [low] [not resolved]

The token price in the Bancor formula - i.e. the shape of the bonding curve - depends on a number of factors: the reserveRatio, the supply of coordination tokens, the amount of security tokens, and the fee.

Currently:

- the reserveRatio is hard-coded as 0,3333333 in the constructor on l. 20
- the fee is set as a parameter in the constructor, and not changeable later
- the initial supply of coordination tokens is hardcoded, and set to 0

- the amount of security tokens that are counted as collateral is equal to the balance of the coordination token contract (and can therefore be changed at any time by any address that has minting and burning permissions on the security token - see issue C6).

There is no description in the specifications of what the desired behavior of the bonding curve for Delphia's use case would be, but it seems unlikely that the current settings are desired. With the current settings (ignoring the issues C1, C4 and C6), depositing the initial 1e18 security tokens (i.e. one token) will give the depositor 1e6 coordination token in return (i.e. the price will be 1e12 security tokens for one coordination token), with the price rising quickly to 1e14 after 100e18 security tokens are deposited). This seems way too steep of a price curve.
Recommendation: Make the reserveRatio, and the initial supply of coordination tokens, parameters in the constructor, just as is already done with fee.
Severity: Low
Resolution: The developers explained that these settings are as intended, and the issue was not further addressed.

C4. The burn function is not distributive [medium] [resolved]

On l. 50ff, the withdrawal fee of the user is left in the contract. Effectively, this means that the fee is distributed over the remaining token holders. The result is that the function _curvedMint, that calculates the amount of tokens that a user receives on withdrawal, is not distributive: the amount of tokens received does not just depend on the amount of tokens that are burned, but also on whether these tokens are burnt in a single transaction or several ones (i.e. a user that calls _curvedBurn(A) and then _curvedBurn(B) will receive more tokens than if she calls _curvedBurn(A+B) - because in the former case, she will also receive a proportion of the fee from burning A. This provides skewed incentives for users interacting with the contract, and makes it very difficult to predict price changes - for example, it is not possible to represent the bonding curve as a function from supply to price.
Recommendation: Have fees flow to a separate account, not to the pool of security tokens
Severity: Medium
Resolution: Fees are now taken in coordination token and are transferred to the token contract itself as the owner.

C5. bondToMint and burnToWithdraw functions are vulnerable to front-running [medium] [resolved]

The bondToMint function is subject to a sandwich attack in which the attacker precedes the victims' bondToMint transaction with a bondToMint transaction of her own (which raises the price of the coordination token), then executing the victims transaction at this inflated price (which raises the price of the coordination tokens even more), and finally burns her own tokens at the inflated price.

Similarly, in a sandwich attack on burnToWithdraw , an attacker wraps the victim's transaction by preceding it with a call to burnToWithdraw (which will lower the price of the coordination token), then executing the victims transaction at this lower price (which will lower the price even more), and finally

bonding the tokens she received in the first burn transaction to receive more coordination tokens than she started with.

The attack surface is mitigated by the fee - the attack is only attractive if the gain of the sandwich attack is higher than the fee paid on burning the tokens.

In addition, the BondingCurveToken.sol contract implements some protection against front-running by specifying a maximum gas price that can be used: a bondToMint transaction that is submitted with a gas price that is higher than that price will fail. We do not believe that this is a very good solution: first of all, even if the attacker is limited to offering the maximum gas price, this will at most the probability that the attack succeeds. In addition, there are services that allow attackers to pay miners directly for prioritizing transactions, and these payments are not checked by the gas price check. Lastly, relying on the maximum gas price requires constant (i.e. hourly) adjustments of the maximal gas price in the contract by the administrators of that contract, and, if it is set too low, effectively makes it impossible to bond or withdraw the tokens.
Recommendation: Add front-running protection logic, for example, by allowing users that call bondToMint to specify a maximum price they are willing to pay (typically by adding an argument such as minAmountOfOutputTokens to the function)
Severity: Medium. Although this can lead to loss of funds, the attack can only be executed by security token holders, which are whitelisted and so, presumably, trusted and identified, and is only viable if the sums involved are large enough for the profit to larger than the fee paid on withdrawal
Resolution: This issue was resolved. bondToMint and burnToWithdraw now allow users to specify a minimal amount that they wish to receive, and the transaction will fail if they receive less.

C6. SCO operators control the price of the coordination tokens [low] [resolved]

The comment on line 55 states that the poolBalance() function returns the overall amount of bonded Security tokens. This is not true.

First of all, any user can send security tokens directly to the CoordinationToken contract without bonding these tokens.

In addition, any address with “SCO” rights can freely mint and burn the tokens that are deposited in the bondingCurve contract. This effectively gives any SCO operator from the security token contract control over the price of the coordination token at any time.

Recommendation: It is not clear if this is as intended. If it is, please fix the comment. If this is not intentional, then limit the access that other accounts have to the price balance - for example, by keeping a separate balance of tokens that are deposited via the mint function, and use that as the basis of the price calculation.
Severity: Low

Resolution: This issue was resolved - a new state variable to track the poolBalance was introduced.

C7. bondToMint, burnToWithdraw and getSecurityToken are marked public instead of external [low] [resolved]

Recommendation: Mark these functions external instead of public to save some gas costs
Severity: Low
Resolution: This issue was resolved.

C8. Return value is calculated twice [low] [resolved]

On l. 38, calculateCurvedMintReturn is called. This call is unnecessary, as the value is already returned by the _curvedMint call on l. 37
Recommendation: Save some gas by storing the return value of _curvedMint in a variable
Severity: Low
Resolution: This issue was resolved.

C9. Ambiguity in specifications regarding whitelisting owners of coordination tokens [info] [resolved]

Coordination token holders can transfer coordination tokens to any address, as with any standard ERC20 token.

The specifications say that “Accredited investors may bond security tokens to mint coordination tokens” and that “Coordination token holders may transfer coordination tokens between them”. It is not clear if the intention is that coordination tokens can be transferred to any user at all, or if that user must be whitelisted - as is done for the security token.
Recommendation: Either implement a limitation on transfer to whitelisted addresses, as the specs suggest, or change the specs
Severity: Info
Resolution: The issue was resolved: the specifications now read “Coordination token holders may transfer coordination tokens to anyone”.

C10. GAS_LIMIT constant is not needed and confusing [info] [resolved]

In l. 8: The GAS_LIMIT constant is used to set the initial maximal gas price that is accepted for a token withdrawal transaction. This value is already stored in a state variable gasPricethat is defined in BondingCurveToken.sol - there is no need to store this value in a new variable.
Recommendation: Do not store the initial gas price in a constant, but set it in the constructor so the deployer can choose a suitable value
Severity: Info
Resolution: This issue was resolved. Gas limit is now settable in the constructor.

C11. Mark immutable state variables as such [info] [resolved]

In l. 11-12: as there is no way of changing the values of phi and of securityToken after setting these in the constructor, they should be explicitly marked as immutable
Recommendation: Mark these variables as immutable
Severity: Info
Resolution: This issue was resolved.

C12. Explicitly mark the visibility of state variables [info] [resolved]

The state variables on lines 8-12 do not have explicit visibility markers - it is best practice to add these.
Recommendation: Mark these variables is public
Severity: Info
Resolution: This issue was resolved.

C13. Check return value of transfer call [info] [resolved]

On l. 51, the transfer function on an external ERC20 contract is called without checking the return value. Although in this case, the contract is known and trusted, it is still good to follow best practices and wrap the call in a require statement
Recommendation: Wrap the transfer call in a require statement
Severity: Info
Resolution: This issue was resolved.

C14. Use default getter for securityToken getter is superfluous [info] [resolved]

The getSecurityToken function can be omitted by declaring the securityToken constant public, which will expose a function securityToken(). This slightly simplifies the contract.
Severity: Info
Resolution: This issue was resolved.

DelphiaEscrow.sol

D1. Distribute function overrides any existing withdrawBalances [medium] [resolved]

The distribute function overrides any previously distributed but not yet redeemed tokens. This might lead to a loss of distributed tokens if a user is being distributed rewards of a game before redeeming from a previous one. This would be recoverable by “manual intervention” with the distribute function, but does not seem like the intended behaviour of the contract.
Recommendation: Change line 89 of DelphiaEscrow.sol from setting a value (with `=`) to adding to existing value (with `+=`).
Severity: Medium
Resolution: This issue was resolved.

D2. Mark addRDOperator and removeRDOperator as external [low] [resolved]

Marking addRDOOperator and removeRDOOperator as external instead of public will save some gas costs.
Recommendation: As described
Severity: Low
Resolution: This issue was resolved.

D3. Do state changes before external calls in the deposit function [low] [resolved]

On line 62, transferFrom is called on the coordinationToken contract before the balance is updated on line 64. This leaves the deposit function open to re-entrancy attacks. Although coordinationToken is a trusted contract, it is still recommended to do state changes before external calls
Recommendation: Exchange line 62-63 with line 64
Severity: Low
Resolution: This issue was resolved.

D4. Copy balance value to memory to save some gas costs [low] [resolved]

On l. 72ff, the value of withdrawBalances[msg.sender] is read 3 times. Some gas can be saved by copying the value to memory instead of reading it thrice
Recommendation: Copy the withdrawBalances[msg.sender] to a memory variable to save gas.
Severity: Low
Resolution: This issue was resolved.

D5. Make state changes before external calls in withdraw function [low] [resolved]

On line 74, transfer is called on the coordinationToken contract before the balance is updated on line 76. This leaves the withdraw function open to re-entrancy attacks. Although coordinationToken is a trusted contract, it is still recommended to do state changes before external calls
Recommendation: Exchange line 74 with line 76
Severity: Low
Resolution: This issue was resolved.

D6. Superfluous `require` check in distribute loop [low] [resolved]

The distribute function runs a loop which performs a `require` check on each iteration to check there are sufficient funds to distribute. This check is superfluous, as the subtraction at line 90 would revert if there are not enough funds. Removing this check might save a notable amount of gas since it is being performed up to 200 times in a call.
Recommendation: Remove the require statement at line 88, and move the subtraction action from line 90 to line 88 instead.
Severity: Low

Resolution: This issue was resolved.

D7. Declare variables that have explicit getters to be public [low] [resolved]

On lines 16-18, the values for operators, withdrawBalances and activeBalance are declared private, while later on in the contract, explicit getters for these variables are defined.
Recommendation: declare the variables public, so that the compiler can create default getter functions, and to remove the getter functions from the contract
Severity: Low
Resolution: This issue was resolved.

D8. Unused import of SafeMath [info] [resolved]

SafeMath is being imported at line 4 but never used, since it has become unnecessary with Solidity version 8 or higher.
Recommendation: Remove the unused SafeMath import at line 4.
Severity: Info
Resolution: This issue was resolved.

D9. Declare visibility and mutability of coordinationToken variable [info] [resolved]

The coordinationToken variable could be declared public and immutable
Severity: Info
Resolution: This issue was resolved.

D10. Inconsistent checks for state in add and remove operator functions [info] [resolved]

In removeRDOperator, on l. 51, the require statement checks whether the given operator address is already registered, in which case the call will fail. There is no similar check in addRDOperator.
Recommendation: Remove the require statement in l. 51, or add a similar check in the add function
Severity: Info
Resolution: This issue was resolved.

D11. Withdraw function signature is different from the specifications [info] [resolved]

The specifications strongly suggest that it should be possible to call the withdraw function on behalf of other users, i.e. that the function should have as a signature withdraw(address payee) instead of withdraw()
Recommendation: Implement the function as in the specifications
Severity: Info
Resolution: This issue was resolved.

D12. Check the return value of an external ERC20.transfer call [info] [resolved]

On l. 74: even if the coordinationToken is a trusted contract that will throw an error if the transfer fails, it would still be best practice to require that the return value be true - or even better to use safeTransfer.

Recommendation: Wrap the transfer call on line 74 in a require statement
Severity: Info
Resolution: This issue was resolved.

SecurityToken.sol

S1. Mark functions that are not called internally external instead of public [low] [resolved]

The functions addAWOperator, addSCOperator, setupBondingCurve, etc are not called internally, and so it would be more clear, and save a minimum amount of gas, by marking these functions external.
Recommendation: Mark these functions external instead of public.
Severity: Low
Resolution: This issue was resolved.

S2. Restrictions on approve function are not effective [low] [resolved]

On l. 209ff the approve function is overridden: the intention here seems to be that only whitelisted addresses can set allowances, and that only whitelist addresses can be added as spenders.
The implementation does not reach this goal, for different reasons.
First of all, the ERC20 contract that the securityToken contract inherits defines increaseAllowance and decreaseAllowance functions. These functions are overridden, and so allowances can be set at will without using the securityToken's implementation of approve.
In any case, restricting the token approval to whitelisted accounts seems superfluous, as the ownership of tokens is already restricted to whitelisted accounts.
The implementation also requires that the spender argument be a whitelisted address. This seems to us a category mistake: the spender typically is an external contract that can interact with the token contract on the user's behalf - for example, the bondingCurve contract, while the whitelist is a list of "accredited investors"
Recommendation: Consider removing the code altogether and to rely on the restrictions on ownership and transfer that are already implemented
Severity: Low
Resolution: This issue was resolved. The custom approve implementation was removed.

S3. setupBondingCurve will not remove a previous bondingCurve from the whitelist [low] [resolved]

setUpbondingCurve calls addToWhitelist on line 253 without first removing any previously set bondingCurve from the whitelist.
Recommendation: Remove the previous bondingCurve from the whitelist when calling setupBondingCurve
Severity: Low
Resolution: This issue was resolved.

S4. Set explicit visibility markers on state variables [info] [resolved]

The constants on lines 16-23 have no visibility markers.
Recommendation: Add visibility markers to these constants - they probably can all be private.
Severity: Info
Resolution: This issue was resolved.

S5. Make state variables that have getters public [info]

The state variables operators, whitelist and bondingCurve as marked private, but then there are getter functions defined to read the values of these variables. A more straightforward implementation would be to mark these variables public, and rely on the compiler-added default getter function to read the values.
Recommendation: Mark these variables public and remove the existing getters.
Severity: Info
Resolution: This issue was resolved.

S6. Lack of clarity about intended transfer limits [info] [resolved]

The specifications suggest that security tokens can be sent freely between whitelisted addresses,
The actual implementation allows transfers only to or from the `bondingCurve` contract.
Note that it is still possible to transfer equity tokens between addresses A and B: A can deposit her tokens in the bondingCurve contract, transfer the coordination tokens to B, who can then redeem these coordination tokens for the original security tokens (minus the fee)
Recommendation: Allow transfers between whitelisted addresses (as the specification says). Or, if value transfer between users must be forbidden, then restrict the transfer of coordination tokens as well.
Severity: Info.
Resolution: The developers report that the implemented behavior is as intended.

S7. Inconsistent use of _msgSender() function [info] [resolved]

_msgSender() is just an abbreviation for msg.sender - the code will be clearer if these two usages are not mixed (as, for example, on lines 215 and 217)
Recommendation: Use msg.sender consistently
Severity: Info
Resolution: This issue was resolved.

S8. Spurious use of whitelist on transferFrom [info] [resolved]

On line 235, it is required that the caller of transferFrom is whitelisted. This seems to be a category mistake: typically, the caller of a transferFrom call will be another contract, while the stated use of the whitelist is that it contains a list of accredited investors (see also S2).
Recommendation: It seems that the intended use case here is that only the bondingCurve contract can spend tokens on behalf of users. If that is true, the code can be simplified considerably by simply requiring in the transferFrom code that msg.sender == bondingCurve, and not using the allowances mechanism at all. This will also improve the UX for the users, which will now not need to approve the bondingCurve contract for spending, and can bond their tokens in a single transaction instead of two.
Severity: Info
Resolution: This issue was resolved.

S9. Unused function parameter [info] [resolved]

```
Warning: Unused function parameter. Remove or comment out the variable
name to silence this warning.
   --> contracts/SecurityToken.sol:147:67:
    |
147 |     function detectTransferRestriction (address from, address to,
uint256 value
```

To adhere to the ERC1404 standard, it is correct to keep this value in the signature
Recommendation: Add a linter-ignore statement so the linter will skip this
Severity: Info
Resolution: This issue was resolved. Value is now used in a check that it is greater than 0.

utils/BancorFormula.sol, utils/BondingCurveToken.sol and utils/Power.sol

These files are not part of the audit, and we did not review them in depth.

These files are copied (with some changes) from
https://github.com/commons-stack/genesis-contracts/tree/master/contracts/bondingcurve

The contracts from commons-stack were adapted in the following ways:
- the syntax of solidity 0.8.*
- the dependency on the obsolete SafeMath package was removed
- the Apache License statement was removed

We mentioned some issues related to this code above - the issues with front-running (C5), the fact that the gasPrice is not settable (C2), and the licensing issue (G2)

We do feel an additional warning is in order: these contracts have not been updated since May 2019, and the genesis-contracts repository has not seen any activity since January 2020: they are not actively maintained.

Severity definitions

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