metronome

Metronome Contracts Audit

version 0.9.0

—

Presented by

Manuel Araoz
CTO, Zeppelin

June 1st, 2018

zeppelin audits

01. Introduction

This document includes the results of the two audits performed by the Zeppelin team on the Metronome project, at the request of the Metronome team. The audited code can be found in the MetronomeToken/metronome Github repository.

The version used for the first report is commit

`753841b8e9a6ed355f5c9bad4b61342d91498601`.

The goal of this audit is to review Metronome's smart contracts (token, auctions, and autonomous converter), study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

The version used for the second report is commit:

`35a6cd37d962f136031c0fe648870f61abddf91a`

— Disclaimer

Note that as of the date of publishing, the contents of this document reflect the current understanding of known security patterns and state of the art regarding smart contract security. Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and auditing is recommended after the issues covered are fixed.

— Methodology

Metronome's Owner's Manual and FAQ were analyzed and synthesized into a series of specifications and expected behaviours, and the codebase was studied in detail in order to acquire a clear impression of how such specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The

problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

— Structure of the document

This report contains a list of issues and comments on the project, sorted by severity. Each issue is assigned a severity level based on the potential impact of the issue, as well as a small example to reproduce it and recommendations to fix it, if applicable. An index by severity is provided at the beginning of the report.

— Documentation

For this audit, we used the following sources of truth about how the Metronome system should work:

- Metronome Owner's Manual version 0.955 (last updated 03.13.2018)
- Metronome FAQ commit 6d10b7e16fc371e8b85a73c4a0b5b6ea34b779ca

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Metronome team or reported an issue.

02. About Zeppelin

Zeppelin is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Zeppelin Solutions has developed industry security standards for designing and deploying smart contract systems.

Zeppelin is the creator, maintainer, and major contributor of OpenZeppelin, the standard framework for secure smart contract development, maintained by a community of 3000+ developers distributed around the globe.

Over $600 million have been raised with Zeppelin's audited smart contracts. Clients include Golem, Brave, Augur, Blockchain Capital, Status, Cosmos, and Storj, among others.

More info at: https://zeppelin.solutions

03. Severity Level Reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

04. List of Issues by Severity

05. Issue Descriptions and Recommendations

Critical Severity

No critical severity issues were found.

High Severity

Mismatch of pricing with specification

The Owner's Guide states: "Descending price auction of all tokens in the Daily Supply Lot begins at a maximum price of twice the previous auction closing price (i.e., the price of the last token sold if the auction sold out, or the price when the auction timed out)". This is contradicted by the FAQ: "All tokens in the DSL start at a maximum price set by the contract at the previous auction's minimum price multiplied by two".

The behavior implemented by the contract is the one specified in the FAQ: it always uses the double of the last purchase price, leaving out the situation when the auction times out instead of selling out. The starting price is then set to this number plus one, which deviates further from both alternative specifications.

Additionally, a special case is made for when an auction does not sell any tokens. In this case, the price for the next auction is set as one hundredth of the last purchase price. We could not find this special case anywhere in the specification.

Change the implementation to conform to the specification, or vice versa.

Unclear supply curve specification and implementation

The available documentation states with respect to the supply curve: "new tokens are added [...] at the rate which is the greater of (i) 2,880 MTN per day, or (ii) an annual rate equal to 2% of the then-outstanding supply per year". We do not find this description to be precise or unambiguous enough. In particular, it's not clear exactly what is referred to as "then-outstanding".

The implementation only adds to the confusion, as it does not seem to conform to any possible interpretation of the above specification. There is a function `globalDailySupply` which purportedly returns, at any moment in time, the global (i.e. across blockchains) supply for the current auction. It is calculated as 1/365th of 2% of the supply that would be in circulation if 2,880 tokens had been minted daily. But this ignores the compounding effect that the 2% yearly inflation rate should have.

Additionally, there is a magic constant being used, which we assume is the point at which the rate changes from (i) to (ii). Avoid magic constants, as they make code much harder to understand.

As a last point, there is a public function `globalMtnSupply` which completely ignores the specification and assumes 2,880 tokens are minted every day.

Review the specification of the supply curve to make it precise and unambiguous, and make sure all parts of the implementation are correct with respect to it. Keep the compounding effect in mind along the way.

Duplicate implementations of pricing, refunding and supply curves

The logic that implements pricing, refunding, and supply caps is duplicated and spread across many places. The contract needs a thorough review and deduplication of all of this logic. Duplicate code can conceal bugs due to the different parts getting out of sync during development.

The places where this logic is repeated are: `nextAuction`, `whatWouldPurchaseDo`, `calcPurchase`, `calcPriceAt`. To greater or lesser degree, almost all of the functions in the contract repeat calculations that are also done elsewhere. It has already been mentioned in this report, for example, how the supply curves are implemented differently by different functions. It is very hard to grasp all of the places where this is happening, and we consider it a very high severity risk for the contract's correctness.

For an example, see the function `whatWouldPurchaseDo`. It is supposed to return the results of a prospective purchase, and it is exposed as a public function, presumably for users to use the information and assess whether they want to purchase or not. It has the same signature as `calcPurchase`, which is the function that is actually used in the fallback function. These two functions are implemented in a similar, but not exactly the same, way. This seems to result in `whatWouldPurchaseDo` returning inaccurate values,

that do not reflect the actual results of a purchase. To solve this, consider implementing `whatWouldPurchaseDo` in terms of `calcPurchase`.

Consider performing a review of the concepts that are needed for this contract's functioning: price curve, supply curve (local and global), refunds (due to supply or investment limits), ticks, etcetera. Implement each of these in one place only.

Accumulated subscription balance can be manipulated by subscriber

The owner of a subscription can create a new subscription for the same recipient via `subscribe`, overwriting the previous one, at any point in time. This resets the subscription start time, effectively cutting down to zero the accumulated balance to be withdrawn by the beneficiary. Furthermore, this method can be invoked front-running any attempts to extract the accumulated balance by the beneficiary. It can also be used to update an existing subscription to set a weekly payment of a single wei, effectively deleting it for all practical purposes.

Consider preventing the owner of the subscription from deleting any accumulated balance. This could be implemented by withdrawing the subscription balance before overwriting it. Alternatively, subscriptions could have a nonce, to allow for more than one subscription to exist between the same subscriber and recipient pair.

Medium severity

Misleading information about next auction

The `nextAuction` function returns information about the next auction at any point in time: the start time, starting price, and token supply. It is exposed as `public`, probably so that clients can easily query this information. The starting price, however, cannot really be known until the previous auction is finished, because it depends on the time and price of the last purchase during the entire auction period.

The value that is returned by the current implementation assumes that `lastPurchasePrice` corresponds to the last purchase in the auction period, but it naturally refers only to the last one up to the current point in time. Thus, the value is correct when `nextAuction` is called by `restartAuction`, but incorrect when called in the middle of an ongoing auction.

Since the next starting price cannot be known, we suggest removing this getter from the public interface, as it can be misleading.

Inaccurate reporting of funds entered through auction

The `Auctions` contract has an event `LogAuctionFundsIn` which is used to log whose and how much ether enters the auction contract through the auctions. The event is used in the fallback function where the amount parameter is set to `msg.value`, even though some of the money may have been returned due to exceeding the purchase limits. This will result in inaccurate reporting of the funds entered. The correct value at the end of the function would be `msg.value - refund`.

Invalid values in Approval events in approveMore/approveLess functions

Functions `approveMore` and `approveLess` of `Token` accept the delta by which the current approval is to be modified. However, the `Approval` event fired within those functions reports not the new allowance, but the delta parameter. According to ERC20, the `Approval` event should be emitted with the new allowance when changed via `approve`, and this semantics should be preserved in extensions to the standard such as `approveMore`.

Consider using the final approval value as the `Approval` event parameter, instead of the delta, so clients monitoring the `Approval` event can correctly identify the allowance set.

MTN disabled transfers can be bypassed via subscriptions

`MTNToken` has a `transferAllowed` flag, which blocks all transfers unless it is set. The flag is initially false, and can only be set once the initial auction is finished. However, the withdrawal methods, such as `subWithdraw`, do not check this flag, allowing any user to transfer funds (albeit with a 7-day delay) regardless of the value of `transferAllowed`.

Note that the fact that the initial auction lasts at most 7 days does not mitigate this issue, as any user may set up a subscription before the initial auction starts, accumulating a balance in it that can be withdrawn before the initial auction ends.

Consider adding a `transferable` modifier to all public withdrawal functions.

Non-standard ERC20 approval behaviour

The function `approve` of the `Token` contract requires that, in order to change the approval, it is first reduced to zero before updating it, in order to prevent front-running attacks. However, this is non-compliant and breaks the ERC20 standard behaviour, and is the reason why the `approveMore`/`approveLess` functions are preferred (which are already present in the contract).

Consider removing the extra condition in L497 for compliance.

Low severity

Pricer tests do not check for error bounds

`Pricer`'s `priceAt` function calculates the price given the number of minutes elapsed since the start of an auction, which should be at most 1440 (1 day). Its tests check only 4 different values, and include rounding errors as part of the expected results.

Consider running the tests in a loop to check all possible 1440 values, and enforce a cap on the numeric error of the computation.

Note that a test run performed with an initial price of 2 ETH yielded a maximum relative error of 1e-11, which is not made evident by the current test suite, and may not be acceptable.

No error analysis in Formula

The functions `returnForMint` and `returnForRedemption` are subject to rounding errors, but have no error analysis whatsoever, or any tests that check their error bounds.

Consider setting up a more comprehensive test suite, or performing analytical error propagation analysis, to ensure the errors generated by these formulas are within acceptable bounds.

Also, note that alternative expressions of the same formula may yield results with better precision. As an example for `returnForMint`, the expression `sqrt((R * S^2 + E * S^2)/R) - S` yields a result with 21 significant digits with the values used in test 3, versus the 18 significant digits in the current implementation (note that the example may

overflow if the values of S, E and R are not bounded, and as such may not be suitable to use as is).

Unsafe math operations in Formula

The functions `returnForMint` and `returnForRedemption` perform an unsafe multiplication by `MTNDECMULT`, which could silently overflow. Consider using `SafeMath` or, given that the result is immediately divided by the same value, remove the multiplication and division altogether (assuming that `DECMULT` and `MTNDECMULT` are equal).

Formula *priceAtInitialAuction* may return zero

The function `priceAtInitialAuction` from the `Formula` contract returns the price for the initial auction, by decreasing linearly from a last purchase price, and returning a floor value. However, the check for floor value is incorrect: if `lastPurchasePrice` is equal to `MULTIPLIER.mul(numTicks)`, then the price returned for each token will be zero.

Change the function so it checks whether the resulting value is less than the minimum price, and in that case returns that value instead.

Note that this issue is mitigated by the fact that, whenever this function is called, the caller manually performs this check. However, this should not be the callers' responsibility, especially given this is a public function.

Exchange values in *AutonomousAutoConverter* are not tested

The tests in the file *metronome.js* check the functions `getEthForMtnResult` and `convertEthToMtn` for exchanging ETH for MTN, where the former returns the expected value to be actually converted by the latter. The same applies to functions `getMtnForEthResult` and `convertMtnToEth` for the inverse conversion.

However, these tests only check that the converted values are equal between them, but never check that they are actually correct, by comparing them to an expected value in the tests.

Consider adding assertions in these tests to check that the actual value transferred by these functions is correct.

Token balance assertions should revert the transaction

Transfer methods in tokens rely on `SafeMath#sub` assertions to throw if the user does not have enough balance to perform the transfer (see L435, L466). This causes the transaction to fail consuming all gas. It is preferred to use `require` to revert the transaction when checking user input (see here for more info). Consider adding manual checks with `require` clauses on each transfer method to `revert` if the user lacks enough balance for a transaction, as was done in the OpenZeppelin implementations.

Unused *roots* mapping in the *MTNToken* contract

The `MTNToken` contract has a public `roots` mapping, which is exposed via a getter (even though there is an autogenerated getter as the field is marked as public), a setter, and a method for comparing if values match.

This field is not used in the `MTNToken` contract, or anywhere else in the codebase, and is untested. There are also no mentions to roots in the documentation, or any comments in the code that explain its usage. Consider removing this code altogether, or move it to a separate contract if it does not belong to the `MTNToken` itself.

Subscriptions cannot be cancelled until first withdrawal

The function `cancelSubscription` of `MTNToken` allows a user to cancel a previously set subscription. However, one of its preconditions prevents the subscription from being cancelled until the beneficiary has performed at least one withdrawal. This allows a user to abstain from withdrawing any balance, in order to keep an uncancellable subscription with an arbitrarily large amount. Consider dropping this requirement.

Note that this issue is mitigated by the fact that the subscriber can overwrite an existing subscription with an empty one at any time, and that anyone can trigger a withdrawal for any other user.

Inconsistent visibility for withdrawal functions

The function `subWithdraw` in MTNToken performs a withdrawal for the caller, and relies on the internal function `subWithdrawFor`, thus preventing a user from triggering a withdrawal for someone else. On the other hand, the function `multiSubWithdrawFor`, used similarly by `multiSubWithdraw`, is flagged as public, thus allowing anyone to trigger a withdrawal.

Consider documenting the expected behaviour with regard to executing a withdrawal for a different address, and fix the visibility for the corresponding method.

Duplicated code in *multiSubWithdrawFor*

Most of the logic for executing a withdrawal in the function `multiSubWithdrawFor` of `MTNToken` is repeated from `subWithdrawFor`. Consider extracting the shared logic to a separate function to avoid code duplication.

Off-by-one error in Auctions fallback

Buyers have a daily purchase limit in the daily auctions, which is enforced by L1037 and L1042-L1045. The former has an off-by-one error: a buyer cannot purchase exactly the daily limit. Consider changing the strict inequality operator < by its non-strict variant <=.

Partial refunding is only implemented for returning buyers

Purchases through the `Auctions` contract have available a partial refunding feature. Buyers can submit a purchase with a given amount of ether, and they will be returned the excess ether if they reach their daily limit or if there is not enough supply of tokens. The latter case is implemented in `calcPurchase`. The other one is implemented in lines 1042 to 1045, but it only applies to the case when the buyer has already participated in the auction. When the buyer participates in an auction for the first time, this functionality is not available, and instead the purchase will be rejected. This does not seem correct.

Consider fixing this by only resetting `purchaseInTheAuction` and `lastPurchaseAuction` inside the conditional, and moving the refunding logic outside of it, so it applies to both scenarios.

Issues with minimum auction price

Auctions have a minimum price at which tokens can be sold. According to the Owner's Manual, this minimum price is 0.0000033 during the initial auction. There is no mention of a minimum price during the daily auctions.

The `Pricer` contract implements the correct minimum price for the initial auction. Additionally, `Auctions` it implements a minimum price for the daily auctions, which is by default 0.0000033, but can be set to a different non-zero number during initialization of the contract.

Our first concern is that the daily minimum price deviates from the specification. Secondly, it's not clear whether the possibility to set a minimum different from 0.0000033 should also extend to the initial auction. Consider defining the state variable `minimumPrice` in the `Pricer` contract, and using it instead of the constant, should that minimum also be configurable.

Public *multiSubWithdraw* function accepts integers instead of addresses

The function `multiSubWithdraw` in `MTNToken` accepts addresses as `uint256`s, which are then casted, in order to "keep client code optimized" according to a comment. We do not know what optimization this refers to. Additionally, if the client code is written in JavaScript, there could be unexpected issues due to the native integer type being floating point.

Consider adding further information on why the code is optimized this way, or change the parameter type to `address` to avoid potential issues.

Monolithic codebase

All of the project's contracts are found in a single file called monolithic.sol. This makes reading and following the code harder, and is generally considered bad programming practice.

Consider moving each contract into their own .sol file, and using a tool such as truffle-flattener in order to generate a single file to be verified via Etherscan.

Use OpenZeppelin

The `Owned`, `Mintable`, `Token` and `TokenLocker` contacts are very similar to `Ownable`, `MintableToken`, `BurnableToken` and `TokenTimelock` from the OpenZeppelin library.

Consider making those contracts inherit from the OpenZeppelin implementations to minimize their custom logic, and following the recommended way to use OpenZeppelin contracts, which is via the zeppelin-solidity NPM package, allowing for any bugfixes and new features to be easily integrated into the codebase.

Notes

- The calculation of the square root in `fSqrt` uses bisection, which converges linearly to the solution. Consider switching to Newton-Raphson method for reduced gas consumption, as it has quadratic convergence.
- The algorithm for calculating `priceAt` runs one iteration per each digit in the units, tens, hundreds, and thousands of the number (worst case being 27 iterations in 999, assuming that n is below 1440), and requires storing three precomputed values. Consider using the exponentiation by squaring algorithm instead, which runs in logarithmic complexity in the input, requires no precomputed values, and involves less code.
- Lines 104-105 in `fSqrt` perform a squaring operation on the input, which is not used at all. The overflow check is also not needed, since any overflows would be caught in the algorithm via the usage of safe math. Consider removing both lines.
- The `Pricer` constructor calculates several intermediate values to be used for later operations. These values do not depend on any injected values in the constructor. Consider hardcoding them in the contract, instead of calculating them on-chain, in order to save gas.
- The functions `convertEthToMtn` and `convertMtnToEth` in `AutonomousConverter` accept an argument to determine the minimum acceptable return value, which is named `minReturn` in the internal function `convert`. However, in the public functions it is named `mintReturn` instead. Consider changing the argument name for clarity.

- The internal `mint` function of `AutonomousConverter` accepts a `_minReturn` argument, which is set to the constant value 1 in the only place where this function is called. Consider removing this argument for clarity, and simply check that the return value for `mint` is non-zero.

- The `SmartToken` constructor accepts instances of a minter and an autonomous converter contracts. However, due to the mechanics of the autonomous converter, the smart contract minter must be the autonomous converter itself (which is set in the deployment scripts). Consider enforcing it at the contract constructor level to prevent potential mistakes, by accepting only an autonomous converter contract instance, and using it as the minter of the `Mintable` superclass.

- The function `redeem` in `AutonomousConverter` performs a check that the contract balance is greater than or equal to the value to be transferred, and that the amount of smart token to be burnt is less than the total supply. Both checks are unneeded, as they are enforced by the `transfer` call or the `destroy` call respectively. Consider removing them.

- The contract `Owned` allows the ownership of the contract to be transferred only once. Consider documenting the rationale behind this design decision, or remove this restriction altogether.

- The `AutonomousConverter` contract has a reference to the `Auctions` contract in its state variables which is not used. Consider removing it.

- The `mintable` state variable in Auctions is initialized in the declaration of the variable as well as in the constructor of the contract, although with different values. Since it is a value known statically, it would be sufficient to initialize it in the declaration. Consider changing the first definition to read the same value that is set to `mintable` in the constructor.

- The `isRunning` helper checks the condition `genesisTime > 0` to see whether the auction is initialized, although there is a specific variable `initialized` that could be used in an equivalent way. The same can be said about the condition `initialAuctionEndTime != 0` in `isInitialAuctionEnded`, and the first precondition in `lockTokenLocker`. In all these cases, consider checking if `initialized` is `true` instead.

- The documentation for `whichAuction` states that the parameter `t` is a timestamp (which is in seconds), when it is actually a tick (which is in minutes). Besides fixing the

documentation, consider using different variable names for timestamp and tick values.

- The constants `DAY_IN_SECONDS` and `DAY_IN_MINUTES` are defined in `Auctions` by using magic constants. Although they are currently correct, this would be more immediately apparent if they were written using the Solidity time units as `1 day / 1 second` and `1 day / 1 minute` respectively.
- The `closeAuction` function in `Proceeds` only emits the `LogClosedAuction` event when the value to be transferred to the converter contract is non-zero. Similarly, only then is the value of the `latestAuctionClosed` state variable updated. Given that the function call succeeds, we would have expected the event to be logged and the variable to be updated, regardless of whether there is ether to transfer to the converter.
- There is a typo in the state variable `lastPurcahseAuction`.
- The function `isInitialAuctionEnded` returns true if the block timestamp is greater than `initialAuctionEndTime`, but also if the token supply is greater than the `INITIAL_SUPPLY`. This second condition is not needed, because if the initial supply is sold out, the fallback function updates `initialAuctionEndTime` to be the moment when it does. Furthermore, if a token is burnt, the supply may drop back below the `INITIAL_SUPPLY`, making this condition false even after the initial auction has ended. Consider removing the second, redundant, condition.
- Consider renaming `TokenLocker#lockTokenLocker` to `lock`.
- The beneficiary parameter of the function `TokenLocker#deposit` is not necessary. Consider removing it.
- The `deposited` state variable of the `TokenLocker` contract seems unneeded, as it always corresponds to the contract's token MTN balance. Consider removing it and using the token's balance instead. Additionally, this makes the whole `TokenLocker#deposit()` function unneeded too.
- There's an unchecked token transfer call in L1543 of `TokenLocker`. Consider adding a check for the boolean return value.
- Consider adding a comment explaining the calculation in L1455 of TokenLocker.

- `uint` is an alias for `uint256`. It is better to be explicit and consistent. It is safer to be 100% explicit about types. Although `uint` is an alias for `uint256`, it makes the size implicit, for the only benefit of having a few characters less. Consider changing all occurrences of `uint` to `uint256`.
- Consider changing the `Mintable` constructor so it accepts the total initial supply expressed in the minimal unit (such as wei for ether), instead of a value and a number of decimals as a multiplier. This allows for finer-grained values to be set, and is consistent with most public functions that work directly in wei. The same applies to the `SmartToken` constructor.
- The `Mintable` contract depends on `ITokenPorter`, but it does not use any of its public functions, as it is just used for access checks. Consider changing the field to a plain `address`, to improve the separation of concerns.
- Consider rewriting the `if-revert` clauses in L525, L1108, and L1271 using a `require` statement, to improve clarity.
- The `Token` method `multiTransfer` lacks enough test coverage. The only test in the suite checks the same value being transferred to all recipients, and uses a poor sentinel value (`000000000000000000000001`), especially taking into account that bit manipulation is involved in this function. Consider using different values for each recipient, and try other cases (zero, a value ending with zeros, a value represented in bits as all 1s, a randomly-generated value, etc).
- Consider adding the `payPerWeek` value of the subscription to the event `LogSubscription`, so that any client monitoring the event can retrieve all the subscription information directly from the event.
- Consider emitting an event in `MTNToken`'s `enableMTNTransfers`, to signal to any clients when the token transfers are enabled.
- The constant `MTNDECIMALS` is repeated in `FixedMath`, `Pricer`, and `SmartToken`. Consider defining this value in a single place and reusing it as needed.
- The calculations in `Pricer` L203, L212, L231, L241, L251 and L263 repeat the same logic as `FixedMath`'s `fMul`. Consider reusing the `fMul` function for clarity and robustness.

06. Audit Revision

General comments

- Code is extremely complex, which makes auditing it difficult. More detailed documentation could help improve understandability.
- Purpose of the ChainLedger contract is unclear.

Concerns over trust model

- It comes to our attention that the owner can mint or destroy anyone's tokens arbitrarily, because it has permission to modify the tokenPorter address. It is somewhat concerning the degree of manual intervention required in the system since the last time we looked at it (Validator, ChainLedger, TokenPorter). This is especially of note given the currency's asserted "self-governance" property. We may be missing the analysis of the safety of this part of the system; we couldn't find it in the documentation.

Unadvertised fee parameter

- There is a fee parameter in exports which we could not find advertised in the documentation. It is an additional amount of tokens that are burnt when exporting from a chain. This parameter is also found in the functions used for importing, yet in those it is not used for anything at all. Neither does it affect the "burn hash" of an export, and it is included but unused in the function called isReceiptClaimable. We are not sure if this omission is by mistake.

Confusing contract naming

- The Owned contract was replaced by new Owned and Ownable contracts, which are rather confusingly named. The semantics of the new Owned are completely different from the previous Owned, even though they share the same name. It used to be that the owner could be changed only once; now it can be changed any number of times. The new Owned, additionally, requires a confirmation transaction by the receiver in a

transfer of ownership. We would have suggested different names for these contracts, but otherwise be advised that the semantics were changed.

Deployment strategy

- It wasn't clear to us why constructors were changed into functions to be called in an additional post-deployment step. In this change, initPricer was left public with no authorization checks; this is usually not desirable, but doesn't seem to be a problem in this case, since initialization is not dependent on any parameters.