

Zer0 - zBanc

Date	May 2021
Lead Auditor	David Oz Kashi
Co-auditors	Martin Ortner

1 Executive Summary

This report is part of a series of reports presenting the results of our engagement with **zer0** to review **zNS**, **zAuction**, and **zBanc**, **zDAO Token**.

The review was conducted over four weeks, from **19 April 2021** to **21 May 2021**. A total of 2x4 person-weeks were spent.

1.1 Layout

It was requested to present the results for the four code-bases under review in individual reports. Links to the individual reports can be found below.

The Executive Summary and Scope sections are shared amongst the individual reports. They provide a general overview of the engagement and summarize scope changes and insights into how time was spent during the audit. The section [Recommendations](#) and [Findings](#) list the respective findings for the component under review.

The following reports were delivered:

- [zNS](#)
- [zAuction](#)
- [zBanc](#)
- [zDAO-Token](#)

1.2 Assessment Log

In the first week, the assessment team focussed its work on the `zNS` and `zAuction` systems. Details on the scope for the components was set by the client and can be found in the next section. A walkthrough session for the systems in scope was requested, to understand the fundamental design decisions of the system as some details were not found in the specification/documentation. Initial security findings were also shared with the client during this session. It was agreed to deliver a preliminary report sharing details of the findings during the end-of-week sync-up. This sync-up is also used to set the focus/scope for the next week.

In the second week, the assessment team focussed its work on `zBanc` a modification of the bancor protocol solidity contracts. The initial code revision under audit (`zBanc` `48da0ac1eebbe31a74742f1ae4281b156f03a4bc`) was updated half-way into the week on Wednesday to `zBanc` (`3d6943e82c167c1ae90fb437f9e3ed1a7a7a94c4`). Preliminary findings were shared during a sync-up discussing the changing codebase under review. Thursday morning the client reported that work on the `zDAO Token` finished and it was requested to put it in scope for this week as the token is meant to be used soon. The assessment team agreed to have a brief look at the codebase, reporting any obvious security issues at best effort until the end-of-week sync-up meeting (1day). Due to the very limited left until the weekly sync-up meeting, it was recommended to extend the review into next week as. Finally it was agreed to update and deliver the preliminary report sharing details of the findings during the end-of-week sync-up. This sync-up is also used to set the focus/scope for the next week.

In the third week, the assessment team continued working on `zDAO Token` on Monday. We provided a heads-up that the snapshot

functionality of zDAO Token was not working the same day. On Tuesday focus shifted towards reviewing changes to [zAuction](#) (`135b2aaddcf70775fd1916518c2cc05106621ec` , [remarks](#)). On the same day the client provided an updated review commit for [zDAO Token](#) (`81946d451e8a9962b0c0d6fc8222313ec115cd53`) addressing the issue we reported on Monday. The client provided an updated review commit for [zNS](#) (`ab7d62a7b8d51b04abea895e241245674a640fc1`) on Wednesday and [zNS](#) (`bc5fea725f84ae4025f5fb1a9f03fb7e9926859a`) on Thursday.

As can be inferred from this timeline various parts of the codebases were undergoing changes while the review was performed which introduces inefficiencies and may have an impact on the review quality (reviewing frozen codebase vs. moving target). As discussed with the client we highly recommend to plan ahead for security activities, create a dedicated role that coordinates security on the team, and optimize the software development lifecycle to explicitly include security activities and key milestones, ensuring that code is frozen, quality tested, and security review readiness is established ahead of any security activities. It should also be noted that code-style and quality varies a lot for the different repositories under review which might suggest that there is a need to better anchor secure development practices in the development lifecycle.

After a one-week hiatus the assessment team continued reviewing the changes for [zAuction](#) and [zBanc](#) . The findings were initially provided with one combined report and per client request split into four individual reports.

2 Scope

Our review focused on the following components and code revisions:

2.1 Objectives

Together with the zer0 team, we identified the following priorities for our review:

1. Ensure that the system is implemented consistently with the intended functionality, and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our [Smart Contract Best Practices](#), and the [Smart Contract Weakness Classification Registry](#).

2.2 Week - 1

- [zNS](#) (`b05e503ea1ee87dbe2b1d58426aaa510068e395`) ([scope doc](#)) (1, 2)
- [zAuction](#) (`50d3b6ce6d7ee00e7181d5b2a9a2eedcdd3fdb72`) ([scope doc](#)) (1, 2)

[Original Scope overview document](#)

2.3 Week - 2

- [zBanc](#) (`48da0ac1eebbe31a74742f1ae4281b156f03a4bc`) initial commit under review
- [zBanc](#) (`3d6943e82c167c1ae90fb437f9e3ed1a7a7a94c4`) updated commit under review (mid of week) ([scope doc](#)) (1)
 - Files in Scope:
 - `contracts/converter/types/dynamic-liquid-token/DynamicLiquidTokenConverter`
 - `contracts/converter/types/dynamic-liquid-token/DynamicLiquidTokenConverterFactory`
 - `contracts/converter/ConverterUpgrader.sol` (added handling new converterType 3)
- [zDAO token](#) provided on thursday ([scope doc](#)) (1)
 - Files in Scope:
 - `ZeroDAOToken.sol`
 - `MerkleTokenAirdrop.sol`
 - `MerkleTokenVesting.sol`
 - `MerkleDistributor.sol`
 - `TokenVesting.sol`
 - And any relevant Interfaces / base contracts

The [zDAO](#) review in week two was performed best effort from Thursday to Friday attempting to surface any obvious issues until the end-of-week sync-up meeting.

2.4 Week - 3

Continuing on [zDAO token](#) (`1b678cb3fc4a8d2ff3ef2d9c5625dff91f6054f6`)

- Updated review commit for [zAuction](#) (`135b2aaddcfc70775fd1916518c2cc05106621ec` , [1](#)) on Monday
- Updated review commit for [zDAO Token](#) (`81946d451e8a9962b0c0d6fc8222313ec115cd53`) on Tuesday
- Updated review commit for [zNS](#) (`ab7d62a7b8d51b04abea895e241245674a640fc1`) on Wednesday
- Updated review commit for [zNS](#) (`bc5fea725f84ae4025f5fb1a9f03fb7e9926859a`) on Thursday

2.5 Hiatus - 1 Week


The assessment continues for a final week after a one-week long hiatus.

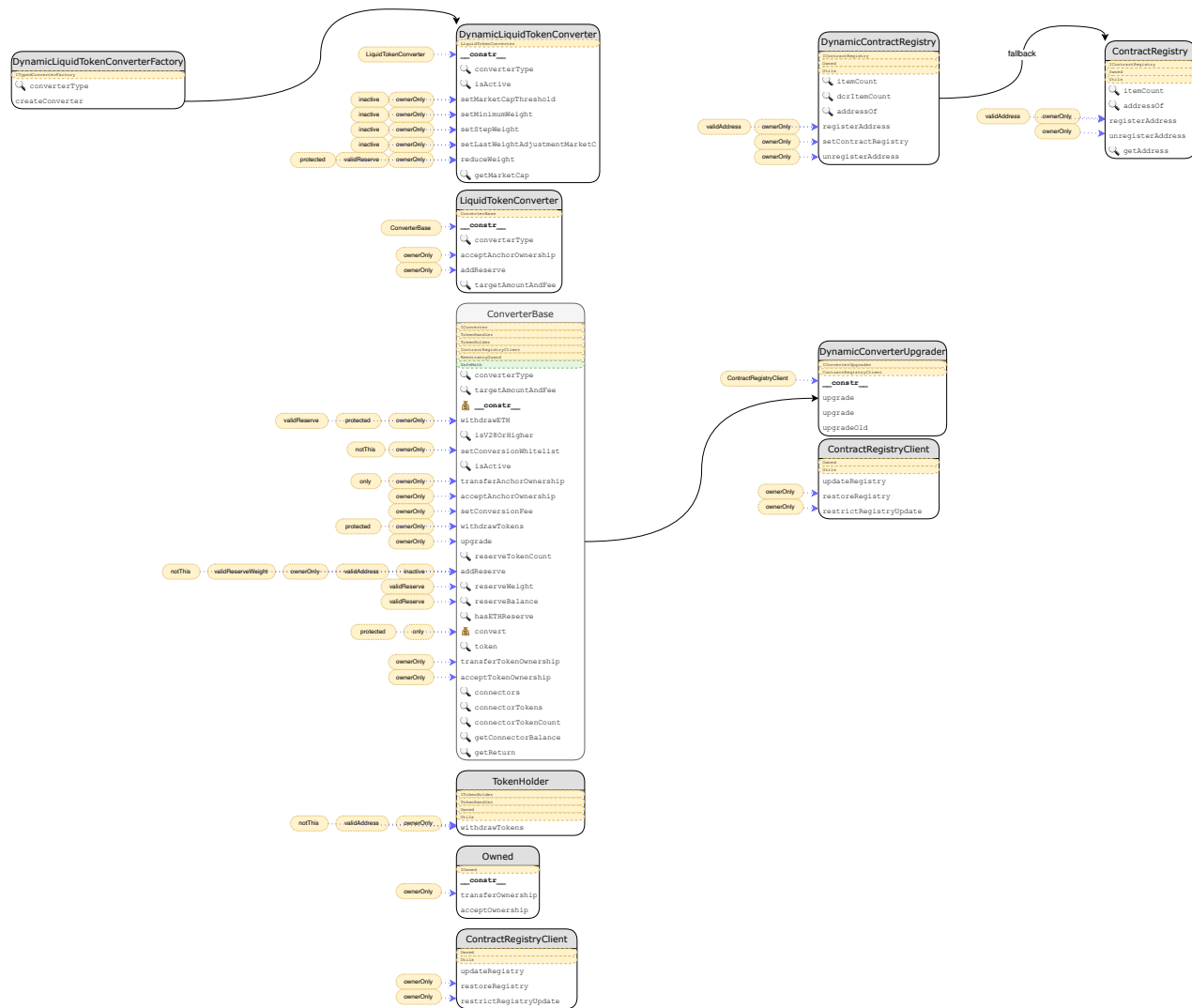
2.6 Week - 4

- Updated review commit for [zAuction](#) (`2f92aa1c9cd0c53ec046340d35152460a5fe7dd0` , [1](#))
- Updated review commit for [zAuction](#) addressing our remarks
- Updated review commit for [zBanc](#) (`ff3d91390099a4f729fe50c846485589de4f8173` , [1](#))

3 System Overview

This section describes the top-level/deployable contracts, their inheritance structure and interfaces, actors, permissions and important contract interactions of the initial [system](#) under review. This section does not take any fundamental changes into account that were introduced during or after the review was conducted.

Contracts are depicted as boxes. Public reachable interface methods are outlined as rows in the box. The  icon indicates that a method is declared as non-state-changing (view/pure) while other methods may change state. A yellow dashed row at the top of the contract shows inherited contracts. A green dashed row at the top of the contract indicates that that contract is used in a `usingFor` declaration. Modifiers used as ACL are connected as yellow bubbles in front of methods.



zBanc

zBanc is a fork from the bancor-protocol adding a new type of liquid token that allows an owner to change the reserve weights at specific milestones to pay out an amount of the tokens while the contract is active. Note that `withdrawETH` can only be called by the owner if the contract is inactive or upgrading. The same is true for `withdrawTokens` for reserve tokens. For this, a new converter type `DynamicLiquidTokenConverter` was created, extending the existing `LiquidTokenConverter`. The new converter requires a custom migration path for upgrades which is implemented in `DynamicConverterUpgrader` and registered in a shadow-registry `DynamicContractRegistry` that allows to override any Bancor registry settings and falls back to retrieving the data from the linked registry otherwise. This gives significant control to whoever is managing the registry.

4 Recommendations

4.1 zBanc - Potential gas optimizations

Description

DynamicLiquidTokenConverter.reduceWeight

1. Calling `reserveBalance` to fetch the reserve balance for a given `reserveToken` might be redundant, as the value has already been fetched, and resides in the `reserve` local variable.
2. Function visibility can be changed to `external` instead of `public`.

-Danc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L130-L150

```
function reduceWeight(IERC20Token _reserveToken)
    public
    validReserve(_reserveToken)
    ownerOnly
{
    _protected();
    uint256 currentMarketCap = getMarketCap(_reserveToken);
    require(currentMarketCap > (lastWeightAdjustmentMarketCap.add(marketCapThreshold)), "ERR_MARKET_CAP_BELOW_THRESHOLD");

    Reserve storage reserve = reserves[_reserveToken];
    uint256 newWeight = uint256(reserve.weight).sub(stepWeight);
    uint32 oldWeight = reserve.weight;
    require(newWeight >= minimumWeight, "ERR_INVALID_RESERVE_WEIGHT");

    uint256 percentage = uint256(PPM_RESOLUTION).sub(newWeight.mul(PPM_RESOLUTION).div(reserve.weight));

    uint32 weight = uint32(newWeight);
    reserve.weight = weight;
    reserveRatio = weight;

    uint256 balance = reserveBalance(_reserveToken).mul(percentage).div(PPM_RESOLUTION);
}
```

- ConverterUpgrader.upgradeOld - Redundant casting of _converter .

zBanc/solidity/contracts/converter/ConverterUpgrader.sol:L96-L99

```
function upgradeOld(DynamicLiquidTokenConverter _converter, bytes32 _version) public {
    _version;
    DynamicLiquidTokenConverter converter = DynamicLiquidTokenConverter(_converter);
    address prevOwner = converter.owner();
}
```

4.2 Where possible, a specific contract type should be used rather than address

Acknowledged

Description

Consider using the best type available in the function arguments and declarations instead of accepting `address` and later casting it to the correct type.

Examples

This is only one of many examples.

zAuction/contracts/zAuction.sol:L22-L26

```
function init(address accountantaddress) external {
    require(!initialized);
    initialized = true;
    accountant = zAuctionAccountant(accountantaddress);
}
```

zAuction/contracts/zAuction.sol:L52-L54

```
IERC721 nftcontract = IERC721(nftaddress);
weth.transferFrom(bidder, msg.sender, bid);
nftcontract.transferFrom(msg.sender, bidder, tokenId);
```

zAuction/contracts/zAuction.sol:L40-L42

```
IERC721 nftcontract = IERC721(nftaddress);
accountant.Exchange(bidder, msg.sender, bid);
nftcontract.transferFrom(msg.sender, bidder, tokenId);
```

zAuction/contracts/zAuctionAccountant.sol:L60-L63

```
function SetZauction(address zauctionaddress) external onlyAdmin{
    zauction = zauctionaddress;
    emit ZauctionSet(zauctionaddress);
}
```

5 Findings

Each issue has an assigned severity:

- **Minor** issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- **Medium** issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- **Major** issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- **Critical** issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 zBanc - DynamicLiquidTokenConverter ineffective reentrancy protection Major ✓ Fixed

Resolution
Fixed with zer0-os/zBanc@ ff3d913 by following the recommendation.

Description

`reduceWeight` calls `_protected()` in an attempt to protect from reentrant calls but this check is insufficient as it will only check for the `locked` statevar but never set it. A potential for direct reentrancy might be present when an `erc-777` token is used as reserve.

It is assumed that the developer actually wanted to use the `protected` modifier that sets the lock before continuing with the method.

Examples

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L123-L128

```
function reduceWeight(IERC20Token _reserveToken)
    public
    validReserve(_reserveToken)
    ownerOnly
{
    _protected();
}
```

```
contract ReentrancyGuard {
    // true while protected code is being executed, false otherwise
    bool private locked = false;

    /**
     * @dev ensures instantiation only by sub-contracts
     */
    constructor() internal {}

    // protects a function against reentrancy attacks
    modifier protected() {
        _protected();
        locked = true;
        _;
        locked = false;
    }

    // error message binary size optimization
    function _protected() internal view {
        require(!locked, "ERR_REENTRANCY");
    }
}
```

Recommendation

To mitigate potential attack vectors from reentrant calls remove the call to `_protected()` and decorate the function with `protected` instead. This will properly set the lock before executing the function body rejecting reentrant calls.

5.2 zBanc - DynamicLiquidTokenConverter input validation Medium ✓ Fixed

Resolution
fixed with zer0-os/zBanc@ ff3d913 by checking that the provided values are at least $0\% < p \leq 100\%$.

Description

Check that the value in `PPM` is within expected bounds before updating system settings that may lead to functionality not working correctly. For example, setting out-of-bounds values for `stepWeight` OR `setMinimumWeight` may make calls to `reduceWeight` fail. These values are usually set in the beginning of the lifecycle of the contract and misconfiguration may stay unnoticed until trying to reduce the weights. The settings can be fixed, however, by setting the contract inactive and updating it with valid settings. Setting the contract to inactive may temporarily interrupt the normal operation of the contract which may be unfavorable.

Examples

Both functions allow the full `uint32` range to be used, which, interpreted as `PPM` would range from `0%` to `4.294,967295%`

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L75-L84

```
function setMinimumWeight(uint32 _minimumWeight)
    public
    ownerOnly
    inactive
{
    //require(_minimumWeight > 0, "Min weight 0");
    //_validReserveWeight(_minimumWeight);
    minimumWeight = _minimumWeight;
    emit MinimumWeightUpdated(_minimumWeight);
}
```

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L92-L101

```
function setStepWeight(uint32 _stepWeight)
    public
    ownerOnly
    inactive
{
    //require(_stepWeight > 0, "Step weight 0");
    //_validReserveWeight(_stepWeight);
    stepWeight = _stepWeight;
    emit StepWeightUpdated(_stepWeight);
}
```

Recommendation

Reintroduce the checks for `_validReserveWeight` to check that a percent value denoted in `PPM` is within valid bounds

`_weight > 0 && _weight <= PPM_RESOLUTION`. There is no need to separately check for the value to be `>0` as this is already ensured by `_validReserveWeight`.

Note that there is still room for misconfiguration (step size too high, min-step too high), however, this would at least allow to catch obviously wrong and often erroneously passed parameters early.

5.3 zBanc - DynamicLiquidTokenConverter introduces breaking changes to the underlying bancorprotocol base Medium ✓ Fixed

Resolution

Addressed with [zer0-os/zBanc@ ff3d913](#) by removing the modifications in favor of surgical and more simple changes, keeping the factory and upgrade components as close as possible to the forked bancor contracts.

Additionally, the client provided the following statement:

5.14 Removed excess functionality from factory and restored the bancor factory pattern.

Description

Introducing major changes to the complex underlying smart contract system that zBanc was forked from (bancorprotocol) may result in unnecessary complexity to be added. Complexity usually increases the attack surface and potentially introduces software misbehavior. Therefore, it is recommended to focus on reducing the changes to the base system as much as possible and comply with the interfaces and processes of the system instead of introducing diverging behavior.

For example, `DynamicLiquidTokenConverterFactory` does not implement the `ITypedConverterFactory` while other converters do. Furthermore, this interface and the behavior may be expected to only perform certain tasks e.g. when called during an upgrade process. Not adhering to the base systems expectations may result in parts of the system failing to function for the new converter type. Changes introduced to accommodate the custom behavior/interfaces may result in parts of the system failing to operate with existing converters. This risk is best to be avoided.

In the case of `DynamicLiquidTokenConverterFactory` the interface is imported but not implemented at all (unused import). The reason for this is likely because the function `createConverter` in `DynamicLiquidTokenConverterFactory` does not adhere to the bancor-provided interface anymore as it is doing way more than “just” creating and returning a new converter. This can create problems when trying to upgrade the converter as the upgraded expected the shared interface to be exposed unless the update mechanisms are modified as well.

In general, the factories `createConverter` method appears to perform more tasks than comparable type factories. It is questionable if this is needed but may be required by the design of the system. We would, however, highly recommend to not diverge from how other converters are instantiated unless it is required to provide additional security guarantees (i.e. the token was instantiated by the factory and is therefore trusted).

`ConverterUpgrader` changed in a way that it now can only work with the `DynamicLiquidTokenconverter` instead of the more generalized `IConverter` interface. This probably breaks the update for all other converter types in the system.

The severity is estimated to be medium based on the fact that the development team seems to be aware of the breaking changes but the direction of the design of the system was not yet decided.

Examples

- unused import

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverterFactory.sol:L6-L6

```
import "../../../../interfaces/ITypedConverterFactory.sol";
```

- converterType should be external as it is not called from within the same or inherited contracts

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverterFactory.sol:L144-L146

```
function converterType() public pure returns (uint16) {  
    return 3;  
}
```

- createToken can be external and is actually creating a token and converter that is using that token (the converter is not returned)(consider renaming to createTokenAndConverter)

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverterFactory.sol:L54-L74

```
{  
    DSToken token = new DSToken(_name, _symbol, _decimals);  
  
    token.issue(msg.sender, _initialSupply);  
  
    emit NewToken(token);  
  
    createConverter(  
        token,  
        _reserveToken,  
        _reserveWeight,  
        _reserveBalance,  
        _registry,  
        _maxConversionFee,  
        _minimumWeight,  
        _stepWeight,  
        _marketCapThreshold  
    );  
  
    return token;  
}
```

- the upgrade interface changed and now requires the converter to be a `DynamicLiquidTokenConverter`. Other converters may potentially fail to upgrade unless they implement the called interfaces.

zBanc/solidity/contracts/converter/ConverterUpgrader.sol:L96-L122

```
function upgradeOld(DynamicLiquidTokenConverter _converter, bytes32 _version) public {
    _version;
    DynamicLiquidTokenConverter converter = DynamicLiquidTokenConverter(_converter);
    address prevOwner = converter.owner();
    acceptConverterOwnership(converter);
    DynamicLiquidTokenConverter newConverter = createConverter(converter);

    copyReserves(converter, newConverter);
    copyConversionFee(converter, newConverter);
    transferReserveBalances(converter, newConverter);
    IConverterAnchor anchor = converter.token();

    // get the activation status before it's being invalidated
    bool activate = isV280rHigherConverter(converter) && converter.isActive();

    if (anchor.owner() == address(converter)) {
        converter.transferTokenOwnership(address(newConverter));
        newConverter.acceptAnchorOwnership();
    }

    handleTypeSpecificData(converter, newConverter, activate);
    converter.transferOwnership(prevOwner);

    newConverter.transferOwnership(prevOwner);

    emit ConverterUpgrade(address(converter), address(newConverter));
}
```

solidity/contracts/converter/ConverterUpgrader.sol:L95-L101

```
function upgradeOld(
    IConverter _converter,
    bytes32 /* _version */
) public {
    // the upgrader doesn't require the version for older converters
    upgrade(_converter, 0);
}
```

Recommendation

It is a fundamental design decision to either follow the bancorsystems converter API or diverge into a more customized system with a different design, functionality, or even security assumptions. From the current documentation, it is unclear which way the development team wants to go.

However, we highly recommend re-evaluating whether the newly introduced type and components should comply with the bancor API (recommended; avoid unnecessary changes to the underlying system,) instead of changing the API for the new components. Decide if the new factory should adhere to the usually commonly shared `ITypedConverterFactory` (recommended) and if not, remove the import and provide a new custom shared interface. It is highly recommended to comply and use the bancor systems extensibility mechanisms as intended, keeping the previously audited bancor code in-tact and voiding unnecessary re-assessments of the security impact of changes.

5.4 zBanc - DynamicLiquidTokenConverter isActive should only be returned if converter is fully configured and converter parameters should only be updateable while converter is inactive Medium

✓ Fixed

Resolution

Addressed with [zer0-os/zBanc@ ff3d913](#) by removing the custom ACL modifier falling back to checking whether the contract is configured (`isActive` , `inactive` modifiers). When a new contract is deployed it will be inactive until the main vars are set by the owner (upgrade contract). The upgrade path is now aligned with how the `LiquidityPoolV2Converter` performs upgrades.

Additionally, the client provided the following statement:

5.13 - upgrade path resolved - inactive modifier back on the setters, and upgrade path now mirrors lpv2 path. An important note here is that `lastWeightAdjustmentMarketCap` setting isn't included in the `isActive()` override, since it has a valid state of 0. So it must be set before the others settings, or it will revert as inactive

Description

By default, a converter is `active` once the anchor ownership was transferred. This is true for converters that do not require to be properly set up with additional parameters before they can be used.

zBanc/solidity/contracts/converter/ConverterBase.sol:L272-L279

```
/**
 * @dev returns true if the converter is active, false otherwise
 *
 * @return true if the converter is active, false otherwise
 */
function isActive() public view virtual override returns (bool) {
    return anchor.owner() == address(this);
}
```

For a simple converter, this might be sufficient. If a converter requires additional setup steps (e.g. setting certain internal variables, an oracle, limits, etc.) it should return `inactive` until the setup completes. This is to avoid that users are interacting with (or even pot. frontrunning) a partially configured converter as this may have unexpected outcomes.

For example, the `LiquidityPoolV2Converter` overrides the `isActive` method to require additional variables be set (`oracle`) to actually be in `active` state.

zBanc/solidity/contracts/converter/types/liquidity-pool-v2/LiquidityPoolV2Converter.sol:L79-L85

```
* @dev returns true if the converter is active, false otherwise
*
* @return true if the converter is active, false otherwise
*/
function isActive() public view override returns (bool) {
    return super.isActive() && address(priceOracle) != address(0);
}
```

Additionally, settings can only be updated while the contract is `inactive` which will be the case during an upgrade. This ensures that the `owner` cannot adjust settings at will for an active contract.

zBanc/solidity/contracts/converter/types/liquidity-pool-v2/LiquidityPoolV2Converter.sol:L97-L109

```
function activate(
    IERC20Token _primaryReserveToken,
    IChainlinkPriceOracle _primaryReserveOracle,
    IChainlinkPriceOracle _secondaryReserveOracle)
    public
    inactive
    ownerOnly
    validReserve(_primaryReserveToken)
    notThis(address(_primaryReserveOracle))
    notThis(address(_secondaryReserveOracle))
    validAddress(address(_primaryReserveOracle))
    validAddress(address(_secondaryReserveOracle))
{
```

The `DynamicLiquidTokenConverter` is following a different approach. It inherits the default `isActive` which sets the contract active right after anchor ownership is transferred. This kind of breaks the upgrade process for `DynamicLiquidTokenConverter` as settings cannot be updated while the contract is active (as anchor ownership might be transferred before updating values). To unbreak this behavior a `now` authentication modifier was added, that allows updates for the upgrade contradict **while the contract is active**. Now this is a behavior that should be avoided as settings should be predictable while a contract is active. Instead it would make more sense

initially set all the custom settings of the converter to zero (uninitialized) and require them to be set and only the return the contract as active. The behavior basically mirrors the upgrade process of `LiquidityPoolV2Converter`.

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L44-L50

```
modifier ifActiveOnlyUpgrader(){
    if(isActive()){
        require(owner == addressOf(CONVERTER_UPGRADER), "ERR_ACTIVE_NOTUPGRADER");
    }
    -;
}
```

Pre initialized variables should be avoided. The marketcap threshold can only be set by the calling entity as it may be very different depending on the type of reserve (eth, token).

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L17-L20

```
uint32 public minimumWeight = 30000;
uint32 public stepWeight = 10000;
uint256 public marketCapThreshold = 10000 ether;
uint256 public lastWeightAdjustmentMarketCap = 0;
```

Here's one of the setter functions that can be called while the contract is active (only by the upgrader contract but changing the ACL commonly followed with other converters).

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L67-L74

```
function setMarketCapThreshold(uint256 _marketCapThreshold)
    public
    ownerOnly
    ifActiveOnlyUpgrader
{
    marketCapThreshold = _marketCapThreshold;
    emit MarketCapThresholdUpdated(_marketCapThreshold);
}
```

Recommendation

Align the upgrade process as much as possible to how `LiquidityPoolV2Converter` performs it. Comply with the bancor API.

- override `isActive` and require the contracts main variables to be set.
- do not pre initialize the contracts settings to "some" values. Require them to be set by the caller (and perform input validation)
- mirror the upgrade process of `LiquidityPoolV2Converter` and instead of `activate` call the setter functions that set the variables. After setting the last var and anchor ownership been transferred, the contract should return active.

5.5 zBanc - DynamicLiquidTokenConverter frontrunner can grief owner when calling reduceWeight

Medium Acknowledged

Resolution

The client acknowledged this issue by providing the following statement:

5.12 - admin by a DAO will mitigate the owner risks here

Description

The owner of the converter is allowed to reduce the converters weights once the marketcap surpasses a configured threshold.

The threshold is configured on first deployment. The marketcap at the beginning of the call is calculated as `reserveBalance / reserve.weight` and stored as `lastWeightAdjustmentMarketCap` after reducing the weight.

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L130-L138

```
function reduceWeight(IERC20Token _reserveToken)
    public
    validReserve(_reserveToken)
    ownerOnly
{
    _protected();
    uint256 currentMarketCap = getMarketCap(_reserveToken);
    require(currentMarketCap > (lastWeightAdjustmentMarketCap.add(marketCapThreshold)), "ERR_MARKET_CAP_BELOW_THRESHOLD");
}
```

The reserveBalance can be manipulated by buying (adding reserve token) or selling liquidity tokens (removing reserve token). The success of a call to `reduceWeight` is highly dependant on the marketcap. A malicious actor may, therefore, attempt to grief calls made by the owner by sandwiching them with `buy` and `sell` calls in an attempt to (a) raise the barrier for the next valid payout marketcap or (b) temporarily lower the marketcap if they are a major token holder in an attempt to fail the `reduceWeights` call.

In both cases the griever may incur some losses due to conversion errors, bancor fees if they are set, and gas spent. It is, therefore, unlikely that a third party may spend funds on these kinds of activities. However, the owner as a potential major liquid token holder may use this to their own benefit by artificially lowering the marketcap to the absolute minimum (old+threshold) by selling liquidity and buying it back right after reducing weights.

5.6 zBanc - outdated fork Medium Acknowledged

Description

According to the client the system was forked off bancor [v0.6.18 \(Oct 2020\)](#). The current version 0.6.x is [v0.6.36 \(Apr 2021\)](#).

Recommendation

It is recommended to check if relevant security fixes were released after v0.6.18 and it should be considered to rebase with the current stable release.

5.7 zBanc - inconsistent DynamicContractRegistry, admin risks Medium ✓ Fixed

Resolution
<p>The client acknowledged the admin risk and addressed the <code>itemCount</code> concerns by exposing another method that only returns the overridden entries. The following statement was provided:</p> <p>5.10 - keeping this pattern which matches the bancor pattern, and noting the DCR should be owned by a DAO, which is our plan. solved itemCount issue - Added <code>dcrItemCount</code> and made <code>itemCount</code> call the bancor registry's <code>itemCount</code>, so unpredictable behavior due to the count should be eliminated.</p>

Description

`DynamicContractRegistry` is a wrapper registry that allows the zBanc to use the custom upgrader contract while still providing access to the normal bancor registry.

For this to work, the registry owner can add or override any registry setting. Settings that don't exist in this contract are attempted to be retrieved from an underlying registry (`contractRegistry`).

zBanc/solidity/contracts/utility/DynamicContractRegistry.sol:L66-L70

```
function registerAddress(bytes32 _contractName, address _contractAddress)
    public
    ownerOnly
    validAddress(_contractAddress)
{
}
```

If the item does not exist in the registry, the request is forwarded to the underlying registry.

zBanc/solidity/contracts/utility/DynamicContractRegistry.sol:L52-L58

```
function addressOf(bytes32 _contractName) public view override returns (address) {
    if(items[_contractName].contractAddress != address(0)){
        return items[_contractName].contractAddress;
    }else{
        return contractRegistry.addressOf(_contractName);
    }
}
```

According to the [documentation](#) this registry is owned by zer0 admins and this means users have to trust zer0 admins to play fair.

To handle this, we deploy our own ConverterUpgrader and ContractRegistry owned by zer0 admins who can register new addresses

The owner of the registry (zer0 admins) can change the underlying registry contract at will. The owner can also add new or override any settings that already exist in the underlying registry. This may for example allow a malicious owner to change the upgrader contract in an attempt to potentially steal funds from a token converter or upgrade to a new malicious contract. The owner can also front-run registry calls changing registry settings and thus influencing the outcome. Such an event will not go unnoticed as events are emitted.

It should also be noted that `itemCount` will return only the number of items in the wrapper registry but not the number of items in the underlying registry. This may have an unpredictable effect on components consuming this information.

zBanc/solidity/contracts/utility/DynamicContractRegistry.sol:L36-L43

```
/**
 * @dev returns the number of items in the registry
 *
 * @return number of items
 */
function itemCount() public view returns (uint256) {
    return contractNames.length;
}
```

Recommendation

Require the owner/zer0 admins to be a DAO or multisig and enforce 2-step (notify->wait->upgrade) registry updates (e.g. by requiring voting or timelocks in the admin contract). Provide transparency about who is the owner of the registry as this may not be clear for everyone. Evaluate the impact of `itemCount` only returning the number of settings in the wrapper not taking into account entries in the subcontract (including pot. overlaps).

5.8 zBanc - DynamicLiquidTokenConverter consider using PPM_RESOLUTION instead of hardcoding integer literals Minor ✓ Fixed

Resolution
This issue was present in the initial commit under review (zer0-os/zBanc@ 48da0ac) but has since been addressed with zer0-os/zBanc@ 3d6943e .

scription

`getMarketCap` calculates the reserve’s market capitalization as `reserveBalance * 1e6 / weight` where `1e6` should be expressed as the constant `PPM_RESOLUTION` .

Examples

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L157-L164

```
function getMarketCap(IERC20Token _reserveToken)
    public
    view
    returns(uint256)
{
    Reserve storage reserve = reserves[_reserveToken];
    return reserveBalance(_reserveToken).mul(1e6).div(reserve.weight);
}
```

Recommendation

Avoid hardcoding integer literals directly into source code when there is a better expression available. In this case `1e6` is used because weights are denoted in percent to base `PPM_RESOLUTION` (`=100%`).

5.9 zBanc - DynamicLiquidTokenConverter avoid potential converter type overlap with bancor Minor

Acknowledged

Resolution
<p>Acknowledged by providing the following statement:</p> <p>5.24 the <code>converterType</code> relates to an array selector in the test helpers, so would be inconvenient to make a higher value. we will have to maintain the value when rebasing in <code>DynamicLiquidTokenConverter</code> & <code>Factory</code>, <code>ConverterUpgrader</code>, and the <code>ConverterUpgrader.js</code> test file and <code>Converter.js</code> test helper file.</p>

Description

The system is forked from [bancorprotocol/contracts-solidity](#). As such, it is very likely that security vulnerabilities reported to bancorprotocol upstream need to be merged into the zer0/zBanc fork if they also affect this codebase. There is also a chance that security fixes will only be available with feature releases or that the zer0 development team wants to merge upstream features into the zBanc codebase.

zBanc introduced `converterType=3` for the `DynamicLiquidTokenConverter` as `converterType=1` and `converterType=2` already exist in the bancorprotocol codebase. Now, since it is unclear if `DynamicLiquidTokenConverter` will be merged into bancorprotocol there is a chance that bancor introduces new types that overlap with the `DynamicLiquidTokenConverter` converter type (`3`). It is therefore suggested to map the `DynamicLiquidTokenConverter` to a converterType that is unlikely to create an overlap with the system it was forked from. E.g. use converter type id `1001` instead of `3` (Note: `converterType` is an `uint16`).

Note that the current master of the bancorprotocol already appears to defined converterType 3 and 4: <https://github.com/bancorprotocol/contracts-solidity/blob/5f4c53ebda784751c3a90b06aa2c85e9fdb36295/solidity/test/helpers/Converter.js#L51-L54>

Examples

- The new custom converter

zBanc/solidity/contracts/converter/types/liquid-token/DynamicLiquidTokenConverter.sol:L50-L52

```
function converterType() public pure override returns (uint16) {
    return 3;
```

- ConverterTypes from the bancor base system

zBanc/solidity/contracts/converter/types/liquidity-pool-v1/LiquidityPoolV1Converter.sol:L71-L73

```
function converterType() public pure override returns (uint16) {
    return 1;
}
```

zBanc/solidity/contracts/converter/types/liquidity-pool-v2/LiquidityPoolV2Converter.sol:L73-L76

```
*/
function converterType() public pure override returns (uint16) {
    return 2;
}
```

Recommendation

Request a Security Review Today

Get in touch with our team to request a quote for a smart contract audit.

Choose a converterType id for this custom implementation that does not overlap with the codebase the system was forked from.

e.g. uint16(-1) or 1001 instead of 3 which might already

CONTACT US

5.10 zBanc - unnecessary contract duplication

Minor ✓ Fixed

Resolution

fixed with [zer0-os/zBanc@ ff3d913](#) by removing the duplicate contract.

SCRIBBLE

the world of blockchain security.

Description

BLOG

Email*

TOOLS

DynamicContractRegistryClient is an exact copy of ContractRegistryClient . Avoid unnecessary code duplication.

RESEARCH

```
< contract DynamicContractRegistryClient is Owned, Utils {
---
> contract ContractRegistryClient is Owned, Utils {
```

PRIVACY POLICY

Appendix 1 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas and specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments

depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.