bZx Network Smart Contracts Audit by ZK Labs

MATTHEW DI FERRANTE

2018-09-01

Introduction

ZK Labs was contracted to perform an audit of the bZx Network smart contracts. Our findings are detailed below.

Neither ZK Labs nor Matthew Di Ferrante have any stake or vested interest in bZx Network. This audit was performed under a contracted rate with no other compensation.

Authenticity

This document should have an attached cryptographic signature to ensure it has not been tampered with. The signature can be verified using the public key from http://keybase.io/mattdf

Audit Goals and Focus

Smart Contract Best Practices

This audit will evaluate whether the codebase follows the current established best practices for smart contract development.

Code Correctness

This audit will evaluate whether the code does what it is intended to do.

Code Quality

This audit will evaluate whether the code has been written in a way that ensures readability and maintainability.

Security

This audit will look for any exploitable security vulnerabilities, or other potential threats to either the operators of bZx or its users.

Testing and testability

This audit will examine how easily tested the code is, and review how thoroughly tested the code is.

About bZx Network

The bZx network project is a decentralized margin lending protocol & liquidation oracle marketplace on the Ethereum blockchain.

Terminology

This audit uses the following terminology.

Likelihood

How likely a bug is to be encountered or exploited in the wild, as specified by the OWASP risk rating methodology.

Impact

The impact a bug would have if exploited, as specified by the OWASP risk rating methodology.

Severity

How serious the issue is, derived from Likelihood and Impact as specified by the OWASP risk rating methodology.

Overview

Source Code

The protocol smart contract source code was made available in the “audit” branch of the https://github.com/b0xNetwork/protocol_contracts/ Github repository.

The snapshot of the codebase at the time of the audit can be found below:

```
./modules/BZxLoanHealth.sol
f9f57fa157d1e815ca3ab95ed2dcf59eaa5856c7e5c44cc37ebdd588716f87f7

./modules/BZxTradePlacing0xV2.sol
92d4789b6e6bd98bb98e18cf2578ad8b5f9eb6c2816d99d18be384422fbaef5e

./modules/BZxOrderHistory.sol
ca26464dff5113495a694093c81eb3eba24a03e99f5ddc8816d424cf63ea86a2

./modules/BZxProxyContracts.sol
bbc6794ae0bb66b62bb79fd08530d836c5b0a6e122f02f7019e5ff57b89f2a6d

./modules/BZxStorage.sol
920f9f4daeba22536895ad0d60ce7bf1baee5974f6fad8b366034abf563a17a1

./modules/BZxTradePlacing.sol
3726abc7c5aa1cebae1ea72b220e0e53e2464dd74ab6300da5e462e70829dbf2

./modules/BZxOrderTaking.sol
2adad6e9c1d0a15ff8d61c070832409ccd3ecc25efd69386ec45a525f3744b59

./modules/BZxLoanMaintenance.sol
ba3895f93841d797ddffcb327cb650f7deeb2fcd933bff3f2f20e22f195d88de

./tokens/BZRxToken.sol
97e79c4ef188eee0fb223d588d59fae1ccede8d2e1cb34060deb58e9faa21535

./tokens/EIP20.sol
4e5fcba2465cc22e4685f60cc83d23873f795c7ae8425966f849d9dedd16551b

./tokens/TokenRegistry.sol
3a648548d3084925f5c04d10c7e3b1bf88b5fb3d60a575dcbeddb8498cd1feff

./tokens/EIP20Wrapper.sol
```

```
847d03fa54037836eb74ee4a781d9559ecf8d4cde2635da228ccdbcb59a0ae89

./tokens/BaseToken.sol
2a986d3dae51530628abc20acf8c2f7981b20bd737b7f3f0d7e83ac8c6840054

./tokens/BZRXFakeFaucet.sol
b0f49db7a82c19b1679b5d5d5e8aaaba195cf378cc73930be5796db066dd1e3c

./tokens/UnlimitedAllowanceToken.sol
97d7901ec3c6c5c5dff32e682a7f2bb6007920e059a8ef53a092a4e519690ecf

./BZx.sol
1fee5cdb1433a131d2e64dc3416636dbfe7865dae2a57e5870624456900c7980

./BZxVault.sol
b3a86f1730634b01a036e72655595efbdc94f594e17016f13db1d75689a91657

./MultiSigWallet/MultiSigWallet.sol
387b274da5e94b6fe3d480143b61c06286ddc1177423848e90cbd0aed08634bd

./MultiSigWallet/MultiSigWalletWithCustomTimeLocks.sol
50d7e27130f8cd9c886ed3d053579ac4a6df7d55cee2a1fb6d61f6bc77a203da

./shared/Debugger.sol
ced37922ac936e78ea0219995fa8911e16343ea650884edda04a96d28aeaeab0

./shared/InternalFunctions.sol
353229b419671ca8afcbcc362cf3f6e265e7ddf74f435c0c72887416c1173264

./ZeroEx/ExchangeInterface.sol
8ffafa2b8d191ee2629b7e15177f772a662fc05f240ce900d990f88ddeb26d3d

./ZeroEx/ExchangeV2InterfaceWithEvents.sol
d55afb55e24e23fbb19b84b28232e58253e7b3e4185097e2762c821fb47edf4f

./ZeroEx/ZeroExV2Helper.sol
bb39fe0b3be5a16a0d5d52bca91f31e904dc98045b2474801856797c7b2e8253

./ZeroEx/BZxTo0x.sol
561fb395e9d0f11eb750a74d6c3ef717e20d0dffaf1b488c6546ab80300d7e93

./ZeroEx/BZxTo0xV2.sol
1246f99d2d7f39cb48f022716a924624fd02d68ed36451a3e0e78f5dda261752
```

```
./ZeroEx/ExchangeV2Interface.sol
ab50684b4d82a8cf56b997690031118d1ce2d4929f464933931268ea489be53b

./ZeroEx/LibEIP712.sol
be742d6bfccbd1cb135bccdce886f21364d3373ff458c6263a7b1257d9c5e226

./oracle/OracleInterface.sol
025254a55a625be6c9f7bcba552cff50ed08c6b72b1470e4c9a62fb6bf7065c9

./oracle/OracleRegistry.sol
c7e7f1e2c927511fa2524d975b638240f2b773472c50f912b611093a70938f57

./oracle/BZxOracle.sol
15fd8259410a20d19f54933715f884b9c3fc0cacb8a0dae5b3841d40b0522a2b

./oracle/OracleTest.sol
8a67f41721cde1815267788b037e2cfac7388d985b7d5cbd5c581a4f05c956be

./modifiers/GasRefunder.sol
5ad39de3967be056fd1f52ff17b4912c06704e57bcde473e3bc3758af024c728

./modifiers/GasTracker.sol
112a200663cce01991f9a55de78ff8d2743cc429735d21a30eee3ab5fe0c898d

./modifiers/BZxOwnable.sol
c62ef444fc90401fd6433058947525026d0183ed6d5b2ebf66b04cd9dfdd571b

./modifiers/EMACollector.sol
3f812f48b3f5c8f5fce6315d87cd7abc2054006e455bff4a0671ce8cdaec70d7

./migrations/Migrations.sol
00bec8074c72a22393cf64d31cb0f9d8f153acc58a72e4268577e49dc38886bf
```

The code makes extensive use of OpenZeppelin library code, which was *not* audited as part of this audit.

General Notes

The code is generally well structured and properly compartmentalized. It makes extensive use of the OpenZeppelin smart contracts, which reduces the count of lines that need to be independently audited and the risk of bugs.

Most of the platform contracts are easily upgradeable through the Proxy, and the Oracle contract upgradeable via a registry. Only the Proxy contract cannot be upgraded after deployment.

The main risk comes from the Proxy contract storage being able to be written freely to by any of the delegatecall contracts, which makes it such that an upgrade to any component can affect the state or balances in the entire protocol. This is a fragile setup that requires an extremely careful upgrade and maintainance process. See the issues section for further comments on this.

The code makes use of revert with error messages which also helps a lot with testing and debugging.

Through the contract proxy, individuals functions can be paused via the multisig, which allows the team to limit or mitigate damage an issue is found while the protocol is live.

As part of the audit, a call and state graph was generated via in-house tooling, and can be used as a guide when reading the audit. The graph is at the end of this PDF.

Top-level Contracts

BZx.sol

This contract is an interface contract, and has no executable code. The proxy address will be typecast to this interface.

BZxVault.sol

The `bZxVault` contract implements an ERC20 and ETH Vault that is controlled the the `bZx` contract address. It supports depositing and withdrawing ether and tokens, and calling ERC20 transferFrom through the EIP20Wrapper interface.

It is ownable, and only maintains the two storage variables inherited by the Ownable contract.

BZxTo0x.sol

The `bZxTo0x` contract allows the taking/filling of 0x v1 trades from the ZRX contract. It has only 1 non-owner state affecting public function, `take0xTrade`, which only the `bZx` contract address can call, and fills a ZRX trade through the vault.

The contract owner has the ability to call the following functions:

- set0xExchange - changes the address of the ZRX contract
- setZRXToken - changes the address of the ZRX token
- set0xTokenProxy - changes the address of the ZRX token proxy
- approveFor - allows the owner to approve amounts for any spender

BZxTo0xV2.sol

The `bZxTo0xV2` contract is similar to the `bZxTo0x` contract, but handles V2 of the 0x protocol instead. It contains the same set of internal state and owner functions, with the with the only public function that modifies state being `take0xV2Trade`, which like before can only be called by the `bZx` contract address.

Module Contracts

bZxStorage.sol

The `bZxStorage` contract is mainly composed of struct declarations from the `bZxObjects` defined in the same file. This contract is what all state-affecting contracts inherit, and variables for all modules are defined here. Due to the proxy and delegatecall, all these variables are actually held in the proxy contract's storage. It is important that the ordering and representation of these variables is kept the same across deployments and upgrades.

bZxLoanMaintenance.sol

The `bZxLoanMaintenance` contract is a `Proxiable` contract that allows participants to "maintain" their loans by managing collateral, and withdrawing profit, if any. It calls to `Oracle_Interface` for price information for calculating profit and loss, and hence also calls the Kyber contracts indirectly.

The only `bZxStorage` variable this contract manages is `loanPositions`. The four interface functions that modify this variable when called are:

- changeCollateral
- depositCollateral
- withdrawProfit
- withdrawExcessCollateral

All four functions also call out to BZxVault and hence affect token balances.

As a `bZxStorage` contract, it uses the proxy's storage and no storage of its own.

bZxLoanHealth.sol

The `bZxLoanHealth` contract is a `Proxiable` contract that allows triggering constraints for loans, such as margin calls or closing of loans. All functions are callable by any msg.sender such that triggers can be executed by addresses not part of the trade, the only exception being `forceCloanLoan`

This contract is also what calls the oracle's `didCloseLoan` and hence can trigger the gas refund. However it can only do so inside of `forceCloanLoan` or `_finalizeLoan`, and the EMA will be registered only if the closing of the loan is valid. With the outlier detection in EMA this provides some protection against artificially inflating the EMA to trigger high refunds. A byzantine actor could still issue many loans and fill them by themselves over and over to inflate the gas price, however this would likely be at a loss to them overall. The only reason to do this would be to slowly grief the refund pool, but the cost involved means it is less likely to happen for long periods.

As a `bZxStorage` contract, it uses the proxy's storage and no storage of its own. The BZxStorage that this contract affects is:

- loanList
- interestPaid
- loanPositions

The functions that affect these states directly or indirectly are:

- payInterest
- liquidatePosition
- closeLoan
- forceCloanLoan

The functions that affect the state are also capable of affecting token balances.

bZxProxyContracts.sol

The `bZxProxyContracts` contains the `Proxiable` definition contract and the main `bZxProxy` endpoint that manages the forwarding of calls to all subcontracts. `bZxProxy` also inherits from `bZxStorage` and hence keeps all the storage maps for all subcontracts.

The `Proxiable` interface requires subcontracts have an `initialize` method defined, which is through by `bZxProxy`"s" `replaceContract` when upgrading a contract. It registers which contract is to be the target of a certain method call.

`BZxProxy`'s contract owner has the ability to change the addresses/state for:

- `bZRxTokenContract`
- `bZxTo0xV2Contract`
- `oracleRegistryContract`
- `vaultContract`
- `bZxTo0xContract`
- `targetIsPaused`
- `targets`
- `oracleAddresses`

bZxOrderTaking.sol

The `bZxOrderTaking` contract is a `Proxiable` contract that implements the main lending functionality. It is the second-most stateful contract with the largest interface, beaten only by `BZxOracle`.

It allows for taking loands as a trader or lender, cancelling unfilled loans, and retrieving information about loans, orders and their respective states.

All public functions affect can state and balances, with the exception of `getUnavailableLoanTokenAmount`.

The BZxStorage that can be modified by this contract is:

- loanList
- orderLender
- loanPositions
- orderFilledAmounts
- orderList
- orderIndexes
- orderCancelledAmounts
- orderTraders
- orders
- orderFees

The `_fillLoanOrder` function, reachable by:

- takeLoanOrderAsTrader
- takeLoanOrderOnChainAsTrader
- takeLoanOrerAsLender
- takeLoanOrderOnChainAsLender

Is able to modify balances directly via calls to BZxVault.

This contract depends heavily on the behaviour of the Vault and Oracle contracts, and is heavily coupled to those interfaces.

As with all other `Proxiable` contracts, `bZxOrderTaking` does not have any storage of its own.

bZxOrderHistory.sol

This contract is not called by any other state affecting contracts. Its only state is inherited from Ownable and Proxiable.

bZxTradePlacing.sol

The `bZxTradePlacing` contract is a `Proxiable` contract which allows executing trades either via 0x or via the Oracle/Kyber using loaned funds. It either uses the bZxTO0X contract address or the oracleAddress defined in the order for the trade.

The only storage affected is `loanPositions`, via both of the public functions, `tradePositionWith0x` and `tradePositionWithOracle`.

As a `bZxStorage` contract, it uses no storage of its own.

bZxTradePlacing0xV2.sol

This contract is the same as `bZxTradePlacing`, but for V2 of the 0x protocol. It has only one function, `tradePositionWith0xV2`, which affects `loanPositions` and token balances via BZxTo0xV2, BZxOracle and BZxVault.

Oracle Contracts

OracleRegistry.sol

The `OracleRegistry` contract is a modification of `TokenRegistry` from ZRX, but extended to keep track of oracle entities instead of tokens. It supports the same set of getter and setter functions.

bZxOracle.sol

The `bZxOracle` contract provides price feed data and records events that happen in the protocol. It is also not upgradeable as easily as the rest of the contracts, it requires entries in the oracle registry and loans are always associated to a specific oracle. It is the most stateful contract in the protocol, with the highest number of state affecting functions, and the highest coupling to all other modules.

The majority of the functions are either only-owner setters, or in the case of the `did` prefix functions, just record return true with a gas price EMA modifier attached. The owner is also able to transfer ether and tokens out of the oracle.

The main functionality of the Oracle is to provide margin, profit/loss and price information, and act as an interface to the Kyber Network's trading platform.

The functions that interact with Kyber are:

- doTrade
- doManualTrade
- getTradeRate
- verifyAndLiquidate
- getProfitOrLoss
- getCurrentMarginAmount
- processCollateral

Other functions that transfer tokens or ETH:

- didPayInterest
- didCloseLoan
- processCollateral
- transferToken

This contract contains a mechanism that gives a gas refund for calling a liquidation. This is only ever called by `BZxLoanHealth`, see that section for more analysis.

Note: It is best practice to use active only verb prefixes for functions that affect state. Functions like "didX" have passive tense and "sound" like getters. The codebase is more robust when there is clear

separation between accessors and modifiers. Replacing the “magic number” `0xcb3c28c7` for the KyberNetwork function call to a keccak would also be preferable, since it is resolved at compile time and does not impose a gas penalty.

Modifier/support Contracts

InternalFunctions.sol

This contract contains various support functions used by many of the module contracts. It does not contain any functions that directly affect state, and it does not contain any state itself.

GasRefunder.sol

The `GasRefunder` contract implements a modifier that refunds the caller the gas cost of execution in ETH. The modifiers defined in this contract are not used.

The only function in this contract that is used is `sendRefund`, which is called by BZxOracle.

GasTracker.sol

The `GasTracker` contract defines a modifier that records the amount of gas left when the function is entered. The value is stored inside an internal `gasUsed` variable, which is zeroed out at the end of the function's execution.

EMACollector.sol

The `EmaCollector` contract defines a modifier that updates an estimated moving average counter according to the following equation:

```
NewEmaValue = ((value / (emaPeriods + 1)* 2)+ emaValue)- (emaValue / (
emaPeriods + 1)* 2)
```

The actual contract uses safemath to perform this operation.

A mechanism to prevent outliers from being recorded is also used, with the formula being as follows:

```
if gasprice is >= 2x + 5 gewi above the current EMA then don't register
```

Hence a byzantine actor cannot artificially increase the gas price EMA very sharply in a short amount of time, they must do it over many valid transactions which is far more costly.

bZxOwnable.sol

This contract is an extension of Zeppelin's `Ownable` contract, adding a `onlybZx` modifier that throws if the caller is not the `bZx` contract address.

Token Contracts

EIP20.sol

This contract only defines ERC20 variable names for `name`, `decimals` and `symbol` accessors.

TokenRegistry.sol

This contract implements a Token Registry. It has been sourced from the 0x repository.

EIP20Wrapper.sol

The `EIP20Wrapper` contract is a wrapper around ERC20 token interface functions. It supports 3 ERC20 constructs, `transfer`, `approve`, and `transferFrom`, and takes a token as an argument for the target of the function calls.

BaseToken.sol

This contract is an instantation of `UnlimitedAllowanceToken` with `BurnableToken` support. Upon the creation of the contract, the entire balance is assigned to the contract creator address.

UnlimitedAllowanceToken.sol

`UnlimitedAllowanceToken` implements an ERC827 token which is an ERC20 with the ability for setting unlimited allowance for an address by calling `approve` with `UINT_MAX` as an argument.

Testing

Test coverage is adequate, with end to end tests and examples with specific focus on token interaction. The main recommendation in regards to testing would be to split up the truffle file in the future so that it isn't one monolithic test file.

Findings

We found the following issues to be aware of.

Main Proxy uses DelegateCall and a single storage point, which makes upgrades fragile if not done correctly

- Likelyhood: medium
- Impact: high

Depending on compiler versions, or upgrades to contract functionality, it is possible that the storage contained inside the proxy gets mangled due to differing storage representations. We advise the bZx team to be extremely careful and test any upgrades on a fork of the mainnet using the same compiler versions when possible, to avoid surprises or destruction of state on the live contracts.

DelegateCall + Proxy combination gives bZx team high level of control over the protocol

- Likelyhood: low
- Impact: high

A contract upgrade can change the state in Proxy arbitrarily, hence can change the state for any protocol contracts arbitrarily - this includes all balances since the proxy has the ability to call the vault. At the time of writing, the bZx team have made the owner of the protocol contracts a multisig with enforced 28 day delay for critical functions, and 14 day delay for the rest. Since the platform relies on 0x and the 0x network itself has the same mechanic (time delay multisig that can change protocol parameters), the threat model is effectively the same as 0x itself.