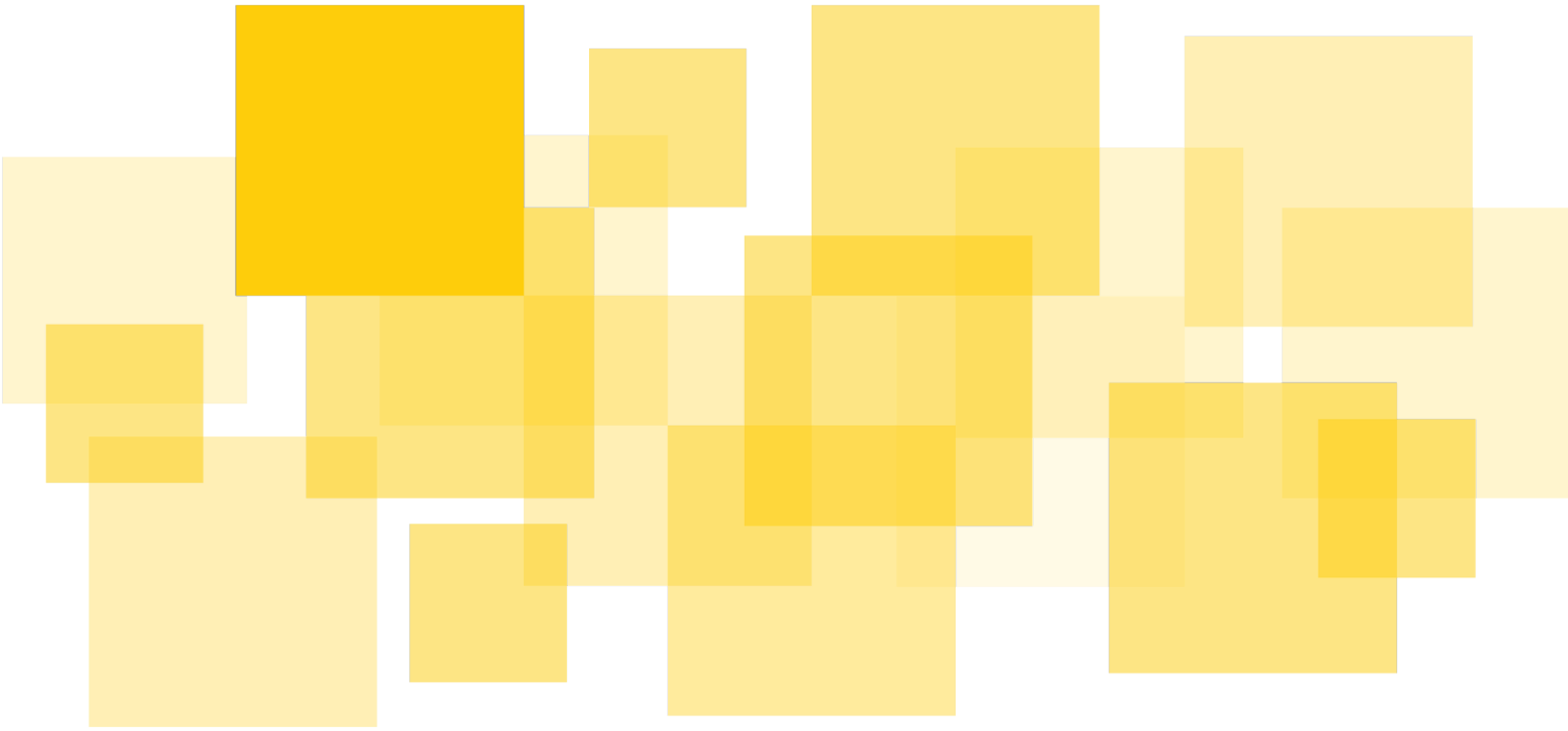
# Security Audit Report

## xBacked Smart Contracts

**May 27th, 2022**



**Prepared for xBacked by**

runtime verification

# Table of Contents

# Executive Summary

xBacked engaged Runtime Verification Inc. to conduct a security audit of their smart contracts. The objective was to review the contracts' business logic and implementations in Reach language and identify any issues that could potentially cause the system to malfunction or be exploited.

The audit was conducted in two separate rounds over the course of 8 calendar weeks:

1. Round 1 (6 weeks, over a period spanning April , 2022 through May 14, 2022) considered an early version of the contracts (commit ID **df0b3f1c242ccced2f5ea8aace5b4dea6a639886** for the branch **audit/q2-2022**).
2. Round 2 (2 weeks, May 15, 2022 through May 27, 2022) focused on analyzing the security implications of the changes/additions the development team made to the contracts (commit ID **a5471c0d9f7233aaf838847ae6c7a0bde7dc3eeb** for the branch **main**) based on the outcome of the first round of the audit.

The audit led to identifying a set of issues, which can be found in corresponding sections of this report.

# Goal

The audit has two main goals:

- first goal is to check high-level properties of the protocol and analyze behavior of participants that could potentially render the system vulnerable to attacks or cause it to malfunction;
- second one - is to check arithmetic properties, precision issues, etc of financial formulae used in the protocol for potential exploit possibility.

Furthermore, the audit highlights informative findings that could be used to improve safety and efficiency of the implementation.

# Platform and Contract Features Description

xBacked is a stablecoin implementation for Algorand blockchain. It consist of two main smart contracts:

1. Master vault smart contract which holds stable-coins, user vaults (CDP - collateralized debt position) and expose API for minting, redeeming, liquidating, etc.
2. Liquidation and stake smart contract which is used to keep user stablecoins to accrue some interest from liquidations.

# Scope

The scope of the audit is limited to the artifacts available at git-commit-id **df0b3f1c242ccced2f5ea8aace5b4dea6a639886** for the branch **audit/q2-2022** at the first round and **a5471c0d9f7233aaf838847ae6c7a0bde7dc3eeb** for the branch **main** at the second round of a repository provided by the client.

Smart contracts are implemented in Reach language. Source code is well annotated, separate overview documentation was provided by the client. Also there is publicly available documentation for users of the protocol.

Repository of the source code has a good structure, instructions for build and test are provided, almost all development is done using containers, so all development and testing activities are simple, well-defined, and repeatable.

It should be noted that the overall code quality is high and the development process of xBacked smart-contracts is well-established.

Reach compiler does a good job of static verification of various code properties (like assets balances, etc) and inserting checks into the low-level Teal code.

This audit omits analysis of the resulting Teal code. It is assumed that the compiler keeps all of the semantic of the high-level Reach code in the generated Teal code.

The exact scope of this audit is limited to the following artifacts:

For the first round:

- **master_vault.rsh** - the main smart-contract for a stable-coin backed by algos, supports local user states and a global one;

- **master_vault_asa.rsh** - the same as above, the difference is only in backed currency - it may be another asset;
- **liquidate_and_stake.rsh** - smart-contract for support of crowd-based liquidation activities for algos;
- **liquidate_and_stake_algo.rsh** - the same as above, but for assets.

For the second round:

- **stability_pool.rsh** - is a renamed version of **liquidate_and_stake_algo.rsh**
- **stability_pool_asa.rsh** - is a renamed version of **liquidate_and_stake.rsh**
- vault.rsh - is a renamed version of **master_vault.rsh**
- **vault_asa.rsh** - is a renamed version of **master_vault_asa.rsh**

All scripts (deployment, supporting, upgrade, and so on), off-chain code, etc are out of the scope of this engagement.

# Assumptions

Generic assumptions are:

- **Admin** behaves honestly with respect to all stakeholders' interests and will do his best to mitigate all possible issues in case of exploits and compromise;
- **Oracle** behaves honestly and does not manipulate collateral/asset price.
- **Reach compiler** is considered trusted
- All basic constants in protocol are carefully chosen by xBacked to fulfill their requirements
- All financial formulae describe desired properties of the protocol and are correct (fees ratios, etc)

# Methodology

Although the audit cannot guarantee to find all possible security vulnerabilities as mentioned in our Disclaimer, we have followed the approaches described below to make our audit as thorough as possible.

First, an abstract formal structural model specification in Alloy is used to model state structure and operations. Using a model-finder allows us to find interesting (possibly malicious or dangerous) operation sequences and then check them on a more detailed financial python model.

Second, an RV inhouse tool, named **tstmodel**, parameterized by python financial model was used to search among state traces, generated by patterns (that were found on Alloy structural specification).

Third, a manual code review was conducted to detect arithmetic precision issues, possible overflows/underflows (that eventually should be eliminated by Reach compiler on static analysis and verification stage of compilation process), redundant code, etc. that could lead to undesired or unexpected (and possible exploitable) behaviors.

Fourth, a thorough analysis of attack vectors was performed, taking into account known information about previous exploits and attacks in the past.

# Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

# First round

## List of Findings

### High

1. Oracle dysfunction
2. Immutability of Oracle and Admin addresses
3. Manipulation of vault proposals
4. Arithmetics precision, overflows/underflows
5. Errors, typos, etc

### Medium

6. Absence of clear state handlers

### Informative

7. Refactoring recommendations

## Oracle dysfunction

For proper work the smart-contract relies on correct and timely price updates by **Oracle**.

**Oracle** for some reason may be in a dysfunctional state (for instance in case of some sort of DDoS attack).

In this case divergence of real collateral price and last recorded one in the smart-contract global state may open the door for an attack.

### Recommendation

Implement a mechanism in smart contract for:

1. Temporal suspension of some smart contract operations, like minting, liquidation, etc;
2. Changing **Oracle** address in smart-contract global state by **Admin** for promptly switching to a backup oracle service.
3. An additional approach is to record the last time of price update, and suspend some functions of the smart contract in case the last timestamp is older than some threshold value (or, as alternative to full suspension, limit some values (minting, liquidating, etc) of operations to reduce possible harm)

### Status

This issue was fully resolved by introducing a **paused** state of the smart contract. So, in case of **Oracle** dysfunction **Admin** can temporarily suspend smart contracts until resolution issues with **Oracle**.

## Immutability of Oracle and Admin addresses

In case of compromising **Oracle** or **Admin** addresses (leakage of private keys, for instance), there is no way to prevent an exploit of the smart-contract, because in the current version of the smart-contract these addresses are immutable and stored in global state.

### Recommendation

As possible mitigation an **Admin** API for changing aforementioned addresses may be introduced.

It will not prevent an exploit for 100% (in case of private keys leakage), but at least will give a chance to save users funds by promptly changing compromised addresses.

A spare pair of addresses (private keys) may be stored separately in a secure place as hot backup and changing old ones for new versions may be implemented in an automatic push-button way.

### Status

This issue was fully addressed. All addresses can now be updated by **Admin**.

## Manipulation of vault proposals

Current implementation of the protocol has no measures for preventing manipulation of proposed vaults.

### Attack scenario

An attacker prepares a set of controlled vaults with a small amount of collateral and debt.

Then the attacker needs to find some normal vault with a large amount of collateral.

Using controlled vaults the attacker can manipulate collateral ratio and push out any other vaults from an array of proposed risky vaults.

Taking into account the fact that the fee for proposing is a fraction of the proposed vault collateral (currently 0.1%), putting in and out a normal vault with a large amount of collateral, the attacker can drain a lot of collateral.

So using controlled vaults and a normal vault the attacker can repeat this algorithm many times until all paid fees for controlled vaults manipulations and proposal transactions are less than received fees.

### Recommendation

There are several viable alternatives:

1. Dynamically adjustable fee.
   A fee for proposing a vault may be adjustable and special service can keep it under the corresponding level, so malicious user payments for the manipulation of proposed vaults will be higher than possible profit.

This service can track attack attempts and change the fee on the fly accordingly to render the attack useless.

2. Fixed fee.
   The fee may be set at such levels that make manipulation of proposed vaults useless.
3. Cooldown time interval.
   To render such sort of attacks impractical a frequency of vault proposals should be lowered.
   This may be achieved using some sort of time window for prohibiting vault proposals.
   This constraint may be placed at various levels:
   a. Cooldown interval per user.
      Cons: larger local stale (a timestamp is needed), can be overcome by creation of additional controlled accounts (parallelize the attack using controlled accounts)
   b. Cooldown interval per vault (eventually attacked one).
      Cons: larger local state, can be overcome by using a set of attacked vaults. I.e. the attack is parallelized using a set of vaults with a high collateral amount.
   c. Cooldown interval per operation (proposing vault) call.
      Pros: local states are untouched, cannot be overcomed by parallelization.
      Cons: larger global state, can influence properties of the protocol a lot (users eventually will be less motivated to make a fair proposals of risky vaults, because of hard limit for possible fees flow, and overall shift will be not proposing a more risky vaults but proposing a vaults with a large collateral amount, so users will be naturally forced to use some sort of unfair manipulations to gain their fees)

**Status**

This attack scenario is mitigated by xBacked via introduction cooldown period per operation.

## Absence of clear state handlers

No smart-contract has a handler of forced clearing of local state by an user. It is so due to the lack of corresponding functionality in Reach language for Algorand specifics.

It cannot be assumed that no users will ever perform forced clearing of their local states.

Consequences will be distortion of financial properties of the protocol because of loss of tracking of some part of collateral and stablecoin.

**Recommendation**

While waiting for implementation of corresponding functionality in Reach language, next options are possible:

- A possible solution may be creation of multisig-account per user, so the user alone cannot execute a clear state call.
- Implementation of some sort of bot that will scan block-chain, detect clear-state calls and update global state of the smart contract accordingly. This scanner service does not need to be fast, etc, so it will require not much resources.

**Status**

xBacked plans to implement a multisig/rekeying solution.

## Errors, etc

### master_vault.rsh:361, in canCreateVault

Possible division by zero.

```
355       const canCreateVault = (who, initialCollateral,
initialVaultDebt) => {
356         const initialCollateralValue = tokenDollarValue(
357           initialCollateral,
358           collateralPrice
359         );
360
361         const vaultCollateralRatio = calcCollateralRatio(
362           initialCollateralValue,
363           initialVaultDebt
364         );
365
366         const vault = fromSome(userVaults[who],
EMPTY_VAULT);
367         check(
```

```
368           initialCollateral > 0 && initialVaultDebt > 0,
369           'Cannot use zero values'
370         );
371         // Must deposit?
```

To fix it, the green check may be moved to the top of the function:

```
355       const canCreateVault = (who, initialCollateral,
initialVaultDebt) => {
367         check(
368           initialCollateral > 0 && initialVaultDebt > 0,
369           'Cannot use zero values'
370         );
356         const initialCollateralValue = tokenDollarValue(
357           initialCollateral,
358           collateralPrice
359         );
360
361         const vaultCollateralRatio = calcCollateralRatio(
362           initialCollateralValue,
363           initialVaultDebt
364         );
365
366         const vault = fromSome(userVaults[who],
EMPTY_VAULT);
371         // Must deposit?
```

The same is for **master_vault_asa.rsh:370**

### master_vault.rsh:1008, in VaultOwner.returnVaultDebt

interestAccrued is added for the second time in updating local user state.

This issue was found by another auditor too, and was confirmed on models by RV.

```
994         if (!close) {
995           const vaultDebt = vault.vaultDebt -
amountToReturn + interestAccrued;
996           const vaultCollateralValue = tokenDollarValue(
997             vault.collateral,
998             collateralPrice
999           );
1000           const cRatioAfterReturn = calcCollateralRatio(
1001             vaultCollateralValue,
1002             vaultDebt
1003           );
1004
1005           const updatedVault = UserVault.fromObject({
1006             collateral: vault.collateral,
1007             collateralRatio: cRatioAfterReturn,
1008             vaultDebt: vaultDebt + interestAccrued,
1009             liquidating: checkVaultLiquidating(vault,
cRatioAfterReturn),
1010             lastAccruedInterestTime: lastTime
1011           });
```

Red expression has to be removed.

The same is for **master_vault_asa.rsh:1016**

### liquidation_and_stake.rsh:167, in canDeprecate

```
165   const canDeprecate = (deprecateTime) => {
166     check(
167       this === Admin,
168       'Only contract initializer can deprecate the
```

```
contract.'
169     );
```

In **AdminAPI.deprecateContract** any caller may be evaluated to Admin, so the check in red won't work as supposed. I suggest using the approach from **master_vault.rsh** and check **Admin** address (already stored as **deployer**) explicitly.

### liquidation_and_stake.rsh:422, in LiquidationVaultAPI.stakeAsset

```
419         userLocalState[this] = UserState.fromObject({
420           ...Struct.toObject(userState),
421           stakedAmount: userState.stakedAmount +
amountToStake,
422           liquidationRewards: userState.rewardPoints +
pendingPoints,
423           lastRewardedSecs: lastConsensusSecs()
424         });
```

Typo in field name, that is marked in red. Should be **rewardPoints** instead **liquidationRewards**.

## Arithmetics precision, overflows/underflows

All these findings were double-checked by a parallel audit process and confirmed on models by RV.

## liquidation_and_stake.rsh:282, in calculatePendingRewardPoints

To increase arithmetic precision, I suggest to rewrite next code:

```
   282       const rewardChunks = timeDelta /
INITIAL_REWARD_INTERVAL_SECONDS;
   283       // What is the rate per chunk this user will receive?
   284       const rewardPerChunk =
   285         userState.stakedAmount / PERCENTAGE_REWARD_PER_CHUNK;
   286       return rewardChunks * rewardPerChunk;
```

Into:

```
   22 const  REWARD_PERCENTAGE_PER_SECONDS =
PERCENTAGE_REWARD_PER_CHUNK *
INITIAL_REWARD_INTERVAL_SECONDS;



            ….

   282       const reward = muldiv(timeDelta, userState.stakedAmount,
   283           REWARD_PERCENTAGE_PER_SECONDS);
   284       return reward
```

The same is for **liquidation_and_stake_algo.rsh:265**

## master_vault.rsh:338, in getInterestAccrued

To avoid possible overflows, I suggest to rewrite this code:

```
const interestAccrued =

    (interestRateOverTimePassed * vault.vaultDebt) /

    INTEREST_RATE_DENOMINATOR;
```

Into:

```
const interestAccrued =

    muldiv(interestRateOverTimePassed, vault.vaultDebt,
INTEREST_RATE_DENOMINATOR);
```

The same is for **master_vault_asa.rsh:347**

### master_vault.rsh:748, in checkRedeemVault

To avoid possible overflows, I suggest to rewrite this code:

```
    const collateralAmountToReceive =

        (amountOfXusdToRedeem * MICRO_UNITS) / collateralPrice;
```

Into:

```
const collateralAmountToReceive =

    muldiv(amountOfXusdToRedeem,  MICRO_UNITS, collateralPrice);
```

The same is for **master_vault_asa.rsh:757**

### master_vault.rsh:846, in proposeNewRedeemableVault

To get more precision, I suggest to rewrite this code:

```
  const proposerFee = (fee * 3) / 4;
```

Into:

```
  const proposerFee = muldiv(proposedVaultData.collateral, 3,
4000);
```

The same is for **master_vault_asa.rsh:854**

# Second round

## Addressed and fixed issues from the first round

- All issues with arithmetic were fixed
- Redundant checks were removed
- Possible attacks were mitigated
- Some refactoring were done

## List of Findings in the second round

### Medium

1. Possible issues in protocol termination stage
2. Pause state in stability_pool.rsh

### Informative

3. Refactoring recommendations

## Possible issues in protocol termination stage

Both smart-contracts depend on rest balances upon termination. But these balances depend on the user's behavior. So a single user, who refuses to withdraw collateral or staking assets, may block deprecation.

After discussion with xBacked the following solution was suggested for both smart-contracts: after starting a deprecation process, participants can only withdraw their collateral or staking assets during so-called ‘grace period’, and then smart-contracts will be terminated with transferring all assets and algos to an Admin. Period duration, notification scheme, etc are to be defined.

## Pause state in **stability_pool.rsh**

There is no pause state in the smart-contract. It may be useful to have an ability to temporarily pause operations. For example in case of abnormal operation or in case of exploit.

xBacked is going to implement a similar scheme as in **vault.rsh.**

## Refactoring recommendations

### Checks for the current state of the contract

It is better to move them from helper functions to the API entries. For the following reasons:

- Better visibility of checks, far less chances to overlook check presence
- Less possibility of redundant execution (functions may call each other and checks will be executed more than once)