Security Assessment for xBacked

Report by Ulam Labs

Security Assessment for xBacked

Findings and Recommendations Report Presented to:

xBacked Team

July 25, 2022 Version: 0.1

Presented by:

Ulam Labs

Grabiszynska 163/502,

53-332 Wroclaw, POLAND

Executive Summary

Overview

xBacked engaged Ulam Labs to perform a Security Assessment for xBacked smart contracts.

The assessment was conducted remotely by the Ulam Labs Security Team. Testing took place on June 20 - July 22, 2022, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks discovered within the environment during the engagement.
- Provide a professional opinion on the maturity, adequacy, and efficiency of the security measures.
- Identify potential issues and include improvement recommendations based on the result of our tests.
- Confirmation of remediation for all reported issues.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Ulam Labs Security Teams took to identify and validate each issue, and any applicable recommendations for remediation.

Scope

The audit has been conducted on the commit **8035c73**, **54c867c** and **df0679b** of xBacked private GitHub Repository.

Files included in the audit

```
xbacked-contracts
└── src
    ├── vault.rsh
    ├── utils.rsh
```

Key findings

During the Security Assessment for xBacked, we discovered the following findings.

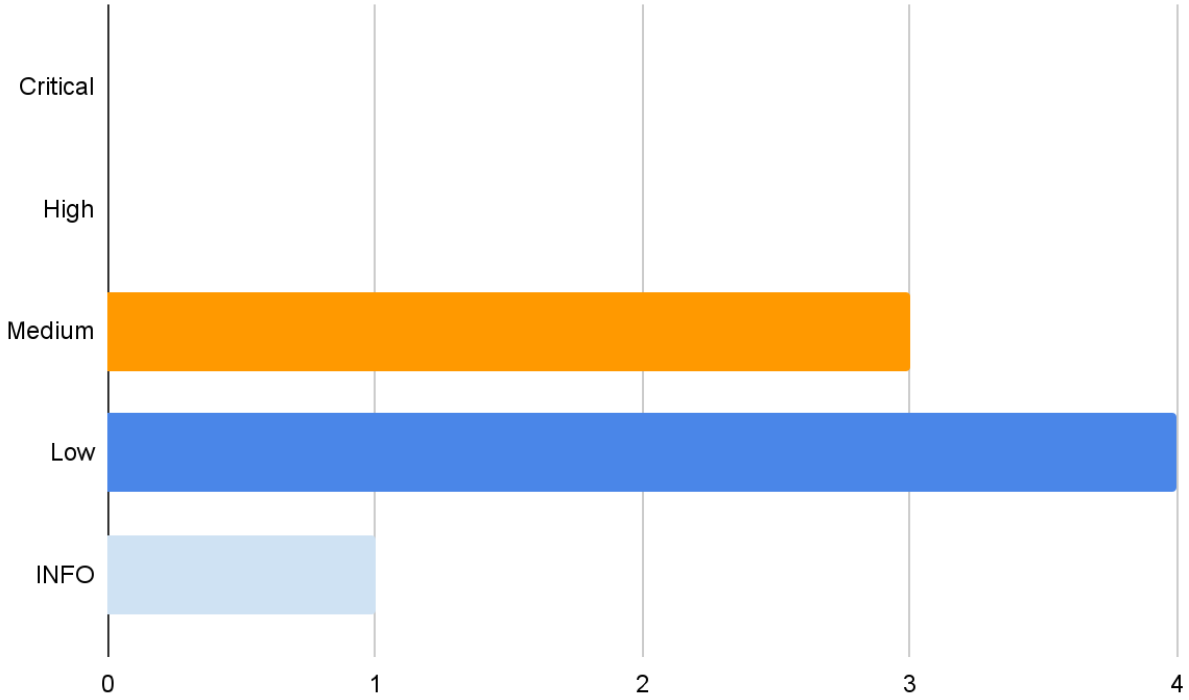

Chart 1: Findings by severity.

During the Security Assessment for xBacked, we discovered:

- 0 findings with a CRITICAL severity rating,
- 0 findings with a HIGH severity rating,
- 3 findings with a MEDIUM severity rating,
- 4 findings with a LOW severity rating,
- 1 findings with a INFO severity rating

All findings have been acknowledged and fixed by the xBacked team.

Issues reported in the previous report: **XB-L15** is still unresolved, but the next Reach update should solve those issues without any change in code. Issue **XB-H7** is only partially solved right now (and will be solved after Reach update), because the user can still clear state, making returning the debt by

the DAO unavailable. Final correction for **XB-H6** was delivered at **df0679b** which was also reviewed in this report.

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 the 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.

Technical analysis & findings

Invalid calculation order causing integer overflow

Finding ID: **XBV-2**
Contract: vault.rsh
Severity: Low
Status: Fixed

Description

It is common operation to subtract some amount from vault debt:

$$updatedVault.vaultDebt = vault.vaultDebt + interestAccrued - amount$$

It is crucial to keep calculation order like in the equation above. Other ways like this:

$$updatedVault.vaultDebt = vault.vaultDebt - amount + interestAccrued$$

would be fine if AVM didn't panic on overflow.

Impact

Integer overflow is always unexpected and in this case also limiting maximum amount which is:

$$0 = vault.vaultDebt + interestAccrued - amount_{max}$$
$$amount_{max} = vault.vaultDebt + interestAccrued$$

Invalid calculation order make maximum amount invalid:

$$0 = vault.vaultDebt - amount_{max} + interestAccrued$$
$$amount_{max} = vault.vaultDebt$$

If the interest accrued is significant, it can be a problem.

Problem is present at lines: **1321**, **1336** and **1654**.

Solution

Calculation order should be fixed as suggested above.

Status

Addressed by the xBacked team. The fix was applied to the source code with commit **54c867c** and reviewed by Ulam Labs Security Team.

Returning more debt than needed causing integer overflow

Finding ID: **XBV-3**
Contract: vault.rsh
Severity: Low
Status: Fixed??

Description

Subtraction parameters should always be checked if one of them is provided by the user.

$$a \geq b \Rightarrow c = a - b$$

Such a check is present in most cases, but is missed in one pay expression.

Impact

Unexpected integer overflow can be emitted at line **1145**.

Solution

Add the suggested check before subtraction.

Status

Addressed by the xBacked team. The fix was applied to the source code with commit **?????????** and reviewed by Ulam Labs Security Team.

Long compilation process discourages from improving the code

Finding ID: **XBV-4**
Contract: vault.rsh
Severity: Info
Status: Accepted

Description

It can take more than 10 minutes to compile the vault contract:

./reach compile src/vault.rsh 0.57s user 0.28s system 0% cpu 13:26.28 total

Compilation time can be even two times longer, when intermediate files are generated.

Impact

If compilation takes so long, it is harder to stay focused. Some ideas will probably never be tried, because it takes too much time to check them.

Solution

Each API call makes the compilation process longer. Contract is complex, so it takes time to compile and verify it...

Status

Accepted by the xBacked team. There is no solution.

Interest rate calculated using last accrued interest time

Finding ID: **XBV-5**
Contract: vault.rsh
Severity: Low
Status: Fixed

Description

Interest is calculated using the formula:

$$amountOfTimePassed = lastTime - vault.lastAccruedInterestTime$$
$$accruedRate = VAULT\_INTEREST\_RATE * amountOfTimePassed$$
$$denominator = INTEREST\_RATE\_DENOMINATOR * AMOUNT\_OF\_SECONDS\_IN\_YEAR$$
$$interestAccrued = muldiv(accruedRate, vault.vaultDebt, denominator)$$

If last time is the same as last accrued interest time, then interest accrued will be zero.

Impact

At line **505**, last accrued interest time is used instead of last consensus time to calculate the interest accrued. Later it is used to validate the amount to return and collateral value (look at **XBV-6** for more details).

$$amount_{max_0} \leq vault.vaultDebt + interestAccrued_0$$
$$interestAccrued_0 = 0 \Rightarrow amount_{max} = amount_{max_0} - interestAccrued$$

However this is not a serious issue, because maximum amount is also limited by minimum balance, which is expected to be greater than any interest accrued.

Solution

User proper formula described above.

Status

Addressed by the xBacked team. The fix was applied to the source code with commit **54c867c** and reviewed by Ulam Labs Security Team.

Accrued interest omitted while checking collateral value

Finding ID: **XBV-6**
Contract: vault.rsh
Severity: Low
Status: Fixed

Description

As described above (**XBV-5**), interest accrued calculated in return debt API call is always zero. It is used to validate the amount to return and also to decide if vault debt can be repaid by anyone.

$$who \neq address \Rightarrow vaultCollateralValue < vault.vaultDebt + interestAccrued$$

Impact

If interest accrued is more than zero, then the request to repay the debt can be rejected, because preconditions are more strict.

However there is a workaround for this bug. If the vault is updated lately, all the debt is stored and interest accrued is zero.

Solution

Add the interest accrued at line **529**.

Status

Addressed by the xBacked team. The fix was applied to the source code with commit **54c867c** and reviewed by Ulam Labs Security Team.

Vault debt can be zero

Finding ID: **XBV-7**
Contract: vault.rsh
Severity: Medium
Status: Fixed

Description

Collateral ratio is calculated using formula:

$$vault.vaultDebt == 0 \Rightarrow cRatio = 0$$
$$vault.vaultDebt \neq 0 \Rightarrow cRatio = muldiv(collateralValue, MICRO\_UNITS, vault.vaultDebt)$$

Vault health depends on collateral ratio.

$$cRatio_0 < cRatio_1 \Leftrightarrow vaultHealth_0 < vaultHealth_1$$

However if the collateral ratio is zero, the health formula is not working then. That's why vault should never have zero debt.

Impact

There are three cases, when vault debt can be zero.

In the return debt API call, final debt is compared to minimum debt. Minimum debt however can also be zero making zero debt possible.

$$vault.vaultDebt + interestAccrued - amountToReturn \geq minimumDebtAmount$$
$$minimumDebtAmount == 0 \Leftrightarrow vault.vaultDebt + interestAccrued - amountToReturn == 0$$

The vault zero debt is also possible in the redeem API call.

$$(amountToRedeem == vault.vaultDebt + interestAccrued \Leftrightarrow$$
$$vault.vaultDebt + interestAccrued - amountToRedeem == 0) \Rightarrow$$
$$amountToRedeem \leq vault.vaultDebt + interestAccrued$$

Solution

Minimum debt amount should be always above zero.

Amount to return should be less than vault debt plus interest, but not equal.

Collateral ratio should be max integer value if vault has no debt.

Status

Addressed by the xBacked team. The fix was applied to the source code with commit **54c867c** and reviewed by Ulam Labs Security Team.

Invalid fee structure is making all the fees locked forever

Finding ID: **XBV-8**
Contract: vault.rsh
Severity: Medium
Status: Fixed

Description

Fees are calculated using formula:

$$fees = muldiv(feeStructure_{type}, accruedFees, 1000)$$

$$feeStructure_{type} \leq 1000 \Rightarrow fees \leq accruedFees$$

If the statement above is true it is safe to calculate remaining fees

$$remainingFees = accruedFees - fees$$

However in the code, there is no guarantee that fee structure stores values less or equal than 1000.

Impact

If it is not guaranteed that fee structure contains values less than 1000, integer overflow will be emitted, when fees are distributed.

Fee distribution is required, everytime fee structure is changed, so it makes fees locked forever.

Solution

Fee structure should be checked and should contain values only below or equal to 1000.

Status

Addressed by the xBacked team. The fix was applied to the source code with commit **54c867c** and reviewed by Ulam Labs Security Team.

Cooldown period is too static

Finding ID: **XBV-9**
Contract: vault.rsh
Severity: Medium
Status: Fixed??

Description

The Cooldown period was introduced to make it impossible to propose a sorted list of vaults and earn a fee for each proposal.

However, when one of the proposed vaults is closed or its collateral is below the limit, the new proposal should be possible.

Impact

Right now, if all of the proposed vaults are closed, users must wait for a cool down period to make the redemption.

The cooldown period is now 12 seconds, so it is not a lot of time, but it is annoying to wait.

Solution

Abort proposal, because of cooldown period, only if all the proposed vaults meet the requirements.

Status

Addressed by the xBacked team. The fix was applied to the source code with commit **?????????** and reviewed by Ulam Labs Security Team.

Recommended tests

A. Create vault

1. **Initial collateral not paid**
    a. **Risk analysis**
    If collateral is not paid and a vault is created, stablecoins are transferred to the user for free. Later it would be also possible to withdraw all the contract collateral assets using withdraw collateral API call.
    b. **Protection**
    Contract is protected very well using a payment expression. Initial collateral is required to be paid.
    c. **Tests**

Reach API does not allow to test the case of missing payment. Creating test in TEAL is the only option, but it is hard to maintain. The best option seems to be using static assertions.

d. **Static asserts**
$assert(balance() >= initialCollateral)$;

2. Collateral price slippage

a. **Risk analysis**
Collateral price can change anytime, but in this case, the worst thing that can happen is transaction rejection, because no side effects are affected by collateral price.

b. **Protection**
Contract is protected using slippage checks.

c. **Tests**
Following cases should be tested:
- $minPrice = collateralPrice + 1$
- $maxPrice = collateralPrice - 1$
- $minPrice = minPrice + 1$

d. **Static asserts**
$assert(coldState.collateralPrice >= minPrice)$;
$assert(coldState.collateralPrice <= maxPrice)$;
$assert(minPrice <= maxPrice)$;

3. Initial vault debt below minimum vault debt

a. **Risk analysis**
Allowing creating a vault with debt below minimum vault debt is not a problem from a contract point of view, but it allows the creation of a lot of vaults cheaply, which can affect other parts of the system.

b. **Protection**
Contract is checking if initial vault debt is greater or equal than minimum vault debt.

c. **Tests**
Following cases should be tested:
- $initialVaultDebt = minVaultDebt - 1$

d. **Static asserts**
$assert(initialVaultDebt >= coldState.minimumDebtAmount)$;

4. Initial vault debt greater than contract stablecoin balance

a. **Risk analysis**
If initial vault debt is greater than contract stablecoin balance and such situation is not handled, contract will panic generating crypting error message.

b. **Protection**
Contract is checking if the stablecoin balance is big enough to transfer the debt.

c. **Tests**
Following cases should be tested:
- $initialVaultDebt = balance(stablecoin) + 1$

5. Collateral ratio below liquidation collateral ratio

a. **Risk analysis**
It is crucial to check the collateral ratio, because without such a check it is possible to withdraw any amount of stablecoin for a small amount of collateral.

b. **Static asserts**
- $assert(vaultCollateralRatio >= LIQUIDATION\_COLLATERAL\_RATIO);$

c. **Protection**
Contract is checking if the vault collateral ratio is greater or equal than liquidation collateral ratio.

d. **Tests**
Following cases should be tested:
- $vaultCollateralRatio = LIQUIDATION\_COLLATERAL\_RATIO - 1$
- **Hint:** Following assertion placed before check helps to find parameters to generate this case. For the best performance compile with **--verify-fail-once** flag.
  $assert(vaultCollateralRatio != LIQUIDATION\_COLLATERAL\_RATIO - 1);$

6. Contract not in normal state

a. **Risk analysis**
If the contract is paused, collateral price may be invalid. If the collateral price is invalid, the collateral ratio is not calculated properly. See **A.5** for details.
If the contract is deprecated, the admin is expecting to close the contract soon. Creating a new vault does not help in that.

b. **Protection**
Contract is checking if the state is normal.

c. **Tests**
Following cases should be tested:
- $contractState = CONTRACT\_STATE\_PAUSED$
- $contractState = CONTRACT\_STATE\_DEPRECATED$

d. **Static asserts**
- $assert(coldState.contractState == CONTRACT\_STATE\_NORMAL);$

7. Vault initialized twice

a. **Risk analysis**
If a vault could be initialized twice for the same address, then all the collateral (if any) for such a vault would be lost and the contract could not be closed.

b. **Protection**
Contract is checking if the last accrued interest time is zero. It is possible only for vault with no collateral.

c. **Tests**

Following cases should be tested:

- $vault.lastAccruedInterestTime != 0$

d. **Static asserts**

- $assert(vault.lastAccruedInterestTime == 0)$;
  Hint: To make it work, the vault should be returned from the canCreateVault function.

8. Vault initialized, when last consensus time is out of date

a. **Risk analysis**
Contract is not using current time to handle API calls, but last consensus time. If the last consensus was long ago, the time is out of date. For example, if the last consensus was one year ago, if the user wants to close the vault after one round, interest for the whole year (instead for one round) must be paid.

b. **Protection**
The only protection against that is to use the contract frequently, because last consensus time is updated by any API call.

c. **Tests**
Following cases should be tested:

- $thisConsensusTime != lastConsensusTime$

B. Mint token

1. Collateral price slippage

a. **Risk analysis**
Collateral price can change anytime, but in this case, the worst thing that can happen is transaction rejection, because no side effects are affected by collateral price.

b. **Protection**
Contract is protected using slippage checks.

c. **Tests**
Following cases should be tested:

- $minPrice = collateralPrice + 1$
- $maxPrice = collateralPrice - 1$
- $minPrice = minPrice + 1$

d. **Static asserts**

$assert(coldState.collateralPrice >= minPrice)$;
$assert(coldState.collateralPrice <= maxPrice)$;
$assert(minPrice <= maxPrice)$;

2. Collateral ratio after mint below liquidation collateral ratio

a. **Risk analysis**
   It is crucial to check the collateral ratio after mint, because without such a check it is possible to withdraw any amount of stablecoin.
b. **Static asserts**
   - $assert(cRatioAfterMint >= LIQUIDATION\_COLLATERAL\_RATIO)$;
c. **Protection**
   Contract is checking if the vault collateral ratio after mint is greater or equal than liquidation collateral ratio.Vault debt used in calculations is increased by new debt and accrued interest.
d. **Tests**
   Following cases should be tested:
   - $cRatioAfterMint = LIQUIDATION\_COLLATERAL\_RATIO - 1$
   - **Hint:** Following assertion placed before check helps to find parameters to generate this case. For the best performance compile with **--verify-fail-once** flag.
     $assert(cRatioAfterMint != LIQUIDATION\_COLLATERAL\_RATIO - 1)$;

3. Contract not in normal state

a. **Risk analysis**
   If the contract is paused, collateral price may be invalid. If the collateral price is invalid, the collateral ratio is not calculated properly. See **B.2** for details.
   If the contract is deprecated, the admin is expecting to close the contract soon. Minting new debt does not help in that.
b. **Protection**
   Contract is checking if the state is normal.
c. **Tests**
   Following cases should be tested:
   - $contractState = CONTRACT\_STATE\_PAUSED$
   - $contractState = CONTRACT\_STATE\_DEPRECATED$

d. **Static asserts**
   - $assert(coldState.contractState == CONTRACT\_STATE\_NORMAL)$;

4. Mint more than contract stablecoin balance

a. **Risk analysis**
   If new vault debt is greater than contract stablecoin balance and such situation is not handled, contract will panic generating crypting error message.
b. **Protection**
   Contract is checking if the stablecoin balance is big enough to transfer new debt.
c. **Tests**
   Following cases should be tested:
   - $newVaultDebt = balance(stablecoin) + 1$

5. Mint on uninitialized vault

a. **Risk analysis**

It is impossible to mint a token on an uninitialized vault, because vault debt is required to be greater than zero and then collateral ratio will be zero (because of no collateral).

b. **Protection**
Contract is checking if the last accrued interest time is not zero.

c. **Tests**
Following cases should be tested:
- $vault.lastAccruedInterestTime == 0$

d. **Static asserts**
- $assert(vault.lastAccruedInterestTime != 0);$

6. Mint less than min debt

a. **Risk analysis**
Allowing minting a token for a vault with debt below minimum vault debt is not a problem from a contract point of view, but it allows the creation of a lot of vaults cheaply, which can affect other parts of the system. Such a situation can only occur if admin increases min vault debt, after the vault is created. Minting a new token can make the debt only greater, so protection is not necessary.

b. **Protection**
There is no protection.

7. Token minted, when last consensus time is out of date

a. **Risk analysis**
Contract is not using current time to handle API calls, but last consensus time. If the last consensus was long ago, the time is out of date. For example, if the last consensus was one year ago, interest for minted stablecoins after one round will be the same as for the whole year.

b. **Protection**
The only protection against that is to use the contract frequently, because last consensus time is updated by any API call.

c. **Tests**
Following cases should be tested:
- $thisConsensusTime != lastConsensusTime$

C. Return debt

1. Insufficient amount to return paid

a. **Risk analysis**
If vault debt is not paid and a vault is closed, collateral is transferred to the user for free. Malicious user could create the vault and close it multiple times and steal all the stablecoins.

b. **Protection**
Contract is protected very well using a payment expression. Vault debt including interest is required to be paid.

c. **Tests**
Reach API does not allow to test the case of missing payment. Creating test in TEAL is the only option, but it is hard to maintain. The best option seems to be using static assertions.

d. **Static asserts**
$assert(implies(!\,close,\ balance()\ >=\ amountToReturn))$;
$assert(implies(close,\ balance()\ >=\ vaultWithInterest))$;

2. Vault debt after return below minimum vault debt

a. **Risk analysis**
Allowing vault debt after return below minimum vault debt is not a problem from a contract point of view, but it allows the creation of a lot of vaults cheaply, which can affect other parts of the system.

b. **Protection**
Contract is checking if initial vault debt is greater or equal than minimum vault debt, if not the vault is closed.

c. **Tests**
Following cases should be tested:
- $amountToReturn\ =\ vaultWithInterest\ -\ minVaultDebt\ +\ 1$
  $close\ =\ false$

d. **Static asserts**

$assert(implies(!\,close,\ vaultWithInterest\ -\ amountToReturn\ >=\ minVaultDebt))$;

3. Contract in paused state

a. **Risk analysis**
If the contract is paused, collateral price may be invalid. If the collateral price is invalid, wrong collateral value is calculated. See **C.4** what can go wrong then.

b. **Protection**
Contract is checking if the state is not paused.

c. **Tests**
Following cases should be tested:
- $contractState\ =\ CONTRACT\_STATE\_PAUSED$

d. **Static asserts**
- $assert(coldState.contractState\ !=\ CONTRACT\_STATE\_PAUSED)$;

4. Close other user healthy vault

a. **Risk analysis**
Closing another user's healthy vault is very dangerous functionality. Healthy vault has more collateral than debt, so attacker gain is collateral ratio minus one.

b. **Protection**
Contract is checking if the specified address is not owned by the sender, then collateral value must be less than vault debt with interest.

c. **Tests**
Following cases should be tested:
- $vaultCollateralValue = vault.vaultDebt + interestAccrued$
  $address != this$
  $close = true$
  $interestAccrued > 0$
  Hint: add following asserts before check to calculate parameters:
  $assert(vaultCollateralValue != vault.vaultDebt + interestAccrued);$
  $assert(interestAccrued > 0);$

d. **Static asserts**
$assert(implies(address != this, vaultCollateralValue < vaultWithInterest));$

5. Return whole debt without closing vault

a. **Risk analysis**
If vault debt is zero there could be some problems, because the debt is used as a denominator while calculating collateral rate.

b. **Protection**
If a vault is not closed debt after returning must be greater than min debt. Minimum debt is guaranteed to be greater than zero.

c. **Tests**
Following case should be tested:
- $close = false$
  $amountToReturn = vault.vaultDebt$
  $interestAccrued = 0$

d. **Static asserts**
$assert(!close, updatedVault.vaultDebt > 0));$

6. Return more debt than needed

a. **Risk analysis**
the user should not be allowed to return more than needed and if it happened, a nice error should be emitted.

b. **Protection**
Algorand AVM is taking care of integer overflows, but it is better to check it to avoid cryptic error messages. Contract is checking if final debt will be above minimum debt and in case of vault closure, user input is ignored.

c. **Tests**
Following case should be tested:
- $amountToReturn = vaultWithInterest + 1$
  $close = false$

7. Return to uninitialized vault

a. **Risk analysis**

It is impossible to return a debt on an uninitialized vault, because the amount to return is required to be greater than zero.

b. **Protection**
Contract is checking if the last accrued interest time is not zero.

c. **Tests**
Following cases should be tested:
- $vault.lastAccruedInterestTime == 0$

d. **Static asserts**
- $assert(vault.lastAccruedInterestTime != 0);$

8. Debt returned, when last consensus time is out of date

a. **Risk analysis**
Contract is not using current time to handle API calls, but last consensus time. If the last consensus was long ago, the time is out of date. For example, if the last consensus was one year ago, interest for returned stablecoins after one round will be nearly zero.

b. **Protection**
The only protection against that is to use the contract frequently, because last consensus time is updated by any API call.

c. **Tests**
Following cases should be tested:
- $thisConsensusTime != lastConsensusTime$

D. Withdraw collateral

1. Collateral price slippage

a. **Risk analysis**
Collateral price can change anytime, but in this case, the worst thing that can happen is transaction rejection, because no side effects are affected by collateral price.

b. **Protection**
Contract is protected using slippage checks.

c. **Tests**
Following cases should be tested:
- $minPrice = collateralPrice + 1$
- $maxPrice = collateralPrice - 1$
- $minPrice = minPrice + 1$

d. **Static asserts**
$assert(coldState.collateralPrice >= minPrice);$
$assert(coldState.collateralPrice <= maxPrice);$
$assert(minPrice <= maxPrice);$

2. Collateral ratio after withdrawal below liquidation collateral ratio

a. **Risk analysis**

It is crucial to check the collateral ratio after withdrawing collateral, because without such a check it is possible to withdraw any amount of stablecoin.

b. **Static asserts**
   - $assert(collateralRatioAfterWithdrawal >= LIQUIDATION\_COLLATERAL\_RATIO);$

c. **Protection**
   Contract is checking if the vault collateral ratio after withdrawal is greater or equal than liquidation collateral ratio.Vault debt used in calculations is increased by new debt and accrued interest.

d. **Tests**
   Following cases should be tested:
   - $collateralRatioAfterWithdrawal = LIQUIDATION\_COLLATERAL\_RATIO - 1$
   - **Hint:** Following assertion placed before check helps to find parameters to generate this case. For the best performance compile with **--verify-fail-once** flag.
     $assert(collateralRatioAfterWithdrawal != LIQUIDATION\_COLLATERAL\_RATIO - 1);$

3. Contract in paused state

a. **Risk analysis**
   If the contract is paused, collateral price may be invalid. If the collateral price is invalid, wrong collateral value is calculated. See **C.2** what can go wrong then.

b. **Protection**
   Contract is checking if the state is not paused.

c. **Tests**
   Following cases should be tested:
   - $contractState = CONTRACT\_STATE\_PAUSED$

d. **Static asserts**
   - $assert(coldState.contractState != CONTRACT\_STATE\_PAUSED);$

4. Withdraw from uninitialized vault

a. **Risk analysis**
   It is impossible to withdraw from an uninitialized vault, because nothing is stored. Nice error message should be emitted.

b. **Protection**
   Contract is checking if the last accrued interest time is not zero.

c. **Tests**
   Following cases should be tested:
   - $vault.lastAccruedInterestTime == 0$

d. **Static asserts**
   - $assert(vault.lastAccruedInterestTime != 0);$

5. Withdraw more collateral than allowed

a. **Risk analysis**

the user should not be allowed to withdraw more than needed and if it happened, a nice error should be emitted. It could happen only if collateral is not updated in the user's local state.

b. **Protection**
Algorand AVM is taking care of integer overflows, but it is better to check it to avoid cryptic error messages. Contract is checking if final collateral will be above zero. Even if such a check is not present, if collateral is zero, collateral ratio is zero and if **D.2** is working correctly everything should be fine.

c. **Tests**
Following case should be tested:
   - $amountToWithdraw = vault.collateral + 1$

E. Deposit collateral

1. Deposited collateral not paid

a. **Risk analysis**
If a deposit is not paid, but the user's local state is updated, any amount of collateral can be transferred to the attacker.

b. **Protection**
Contract is protected very well using a payment expression. Amount to deposit is required to be paid.

c. **Tests**
Reach API does not allow to test the case of missing payment. Creating test in TEAL is the only option, but it is hard to maintain. The best option seems to be using static assertions.

d. **Static asserts**
$assert(balance() >= collateralDeposit)$;

2. Contract in not normal state

a. **Risk analysis**
If the contract is paused, only the admin can call the contract. Other
If the contract is deprecated, the admin is expecting to close the contract soon. Depositing the new collateral does not help in that.

b. **Protection**
Contract is checking if the state is normal.

c. **Tests**
Following cases should be tested:
   - $contractState = CONTRACT\_STATE\_PAUSED$
   - $contractState = CONTRACT\_STATE\_DEPRECATED$

d. **Static asserts**
   - $assert(coldState.contractState == CONTRACT\_STATE\_NORMAL)$;

3. Deposit collateral to uninitialized vault

a. **Risk analysis**
   It is important to check if the vault is already initialized, because it would be easy to create a vault having less than min debt.
b. **Protection**
   Contract is checking if the last accrued interest time is not zero.
c. **Tests**
   Following cases should be tested:
   - $vault.lastAccruedInterestTime == 0$
d. **Static asserts**
   - $assert(vault.lastAccruedInterestTime != 0);$

F. Liquidate vault

1. Debt not paid

a. **Risk analysis**
   If a debt is not paid, it would be possible to get free collateral from a liquidated vault.
b. **Protection**
   Contract is protected very well using a payment expression. Debt is required to be paid.
c. **Tests**
   Reach API does not allow to test the case of missing payment. Creating test in TEAL is the only option, but it is hard to maintain. The best option seems to be using static assertions.
d. **Static asserts**
   $assert(balance() >= debtAmount);$

2. Collateral price slippage

a. **Risk analysis**
   Collateral price can change anytime and in this case the sender wants to be sure that price is within a given range, because the amount of collateral transferred to sender depends on collateral price.
b. **Protection**
   Contract is protected using slippage checks.
c. **Tests**
   Following cases should be tested:
   - $minPrice = collateralPrice + 1$
   - $maxPrice = collateralPrice - 1$
   - $minPrice = minPrice + 1$
d. **Static asserts**
   $assert(coldState.collateralPrice >= minPrice);$
   $assert(coldState.collateralPrice <= maxPrice);$
   $assert(minPrice <= maxPrice);$

3. Contract in paused state

a. **Risk analysis**
If the contract is paused, collateral price may be invalid. If the collateral price is invalid, wrong collateral value is calculated. If collateral value is not correct, healthy vault can be liquidated.

b. **Protection**
Contract is checking if the state is not paused.

c. **Tests**
Following cases should be tested:

- $contractState = CONTRACT\_STATE\_PAUSED$

d. **Static asserts**

- $assert(coldState.contractState != CONTRACT\_STATE\_PAUSED);$

4. Liquidate healthy vault

a. **Risk analysis**
If the healthy vault is liquidated, its owner is losing collateral. The more healthy the vault is, the more the attacker is gaining.

b. **Protection**
Contract is checking if the liquidating flag is not set, then if collateral ratio is below liquidation level, the flag is set and liquidation is possible. If the flag is not set, liquidation is possible if the collateral ratio is below minimum level.

c. **Tests**
Following test cases should be tested:

- $vault.liquidating = false$
  $currentVaultCollateralRatio = LIQUIDATION\_COLLATERAL\_RATIO$
  $interestAccrued > 0$
  $collateralRatioAfterLiquidation = MINIMUM\_COLLATERAL\_RATIO$
- $vault.liquidating = false$
  $currentVaultCollateralRatio = LIQUIDATION\_COLLATERAL\_RATIO - 1$
  $interestAccrued > 0$
  $collateralRatioAfterLiquidation = MINIMUM\_COLLATERAL\_RATIO + 1$
- $vault.liquidating = true$
  $currentVaultCollateralRatio = MINIMUM\_COLLATERAL\_RATIO$
  $interestAccrued > 0$
  $collateralRatioAfterLiquidation = MINIMUM\_COLLATERAL\_RATIO$
- $vault.liquidating = true$
  $currentVaultCollateralRatio = MINIMUM\_COLLATERAL\_RATIO - 1$
  $interestAccrued > 0$
  $collateralRatioAfterLiquidation = MINIMUM\_COLLATERAL\_RATIO + 1$

5. Return more debt than vault has

a. **Risk analysis**
   It is possible if the user's local state is not updated, because AVM will panic if subtraction overflows. However, a nice error should be emitted.
b. **Protection**
   There is a check validating debt.
c. **Tests**
   Following test cases should be tested:
   - $debtAmount = vault.vaultDebt + interestAccrued + 1$
     $interestAccrued > 0$

6. Force vault debt under minimum specified by admin

a. **Risk analysis**
   Minimum vault debt has been introduced to protect against worthless vaults. If the vault is liquidated, it is not worthless anymore, because fees are collected from that vault.

7. Liquidate uninitialized vault

a. **Risk analysis**
   It is impossible to liquidate an uninitialized vault, because nothing is stored. Nice error message should be emitted.
b. **Protection**
   Contract is checking if the last accrued interest time is not zero.
c. **Tests**
   Following cases should be tested:
   - $vault.lastAccruedInterestTime == 0$
d. **Static asserts**
   - $assert(vault.lastAccruedInterestTime != 0);$

G. Update price

1. Send as not oracle

a. **Risk analysis**
   If anyone can update the price, anyone can withdraw all the assets from the contract.
b. **Protection**
   Contract is checking if the sender updating the price is the oracle specified by the admin.
c. **Tests**
   Following cases should be tested:
   - $this != addresses.oracleAddress$

2. Real price not sent (oracle offline)

a. **Risk analysis**

If the oracle is offline, the contract should be paused or another oracle should be chosen.

H. Replenish supply

1. Supply not paid

a. **Risk analysis**
If a supply amount is not paid, the worst what can happened is unexpected overflow when handling user requests.

b. **Protection**
Contract is protected very well using a payment expression. Supply amount is required to be paid.

c. **Tests**
Reach API does not allow to test the case of missing payment. Creating test in TEAL is the only option, but it is hard to maintain. The best option seems to be using static assertions.

d. **Static asserts**
$assert(balance() >= supplyAmt)$;

2. Sent as not admin

a. **Risk analysis**
It is not risky to call this API call by anyone, but it is good practice to block calls making user assets lost permanently.

b. **Protection**
Contract is checking if the sender updating the price is the admin.

c. **Tests**
Following cases should be tested:

- $this != addresses.adminAddress$

I. Set admin properties

1. Sent as not admin

a. **Risk analysis**
If anyone can set admin properties, anyone can set the oracle address. If anyone can set the oracle address, then any collateral price can be set and all the assets could be stolen.

b. **Protection**
Contract is checking if the sender updating the properties is the admin.

c. **Tests**
Following cases should be tested:

- $this != addresses.adminAddress$

2. Set invalid state

a. **Risk analysis**
   If a invalid state is set, API calls will see it as deprecated, but the contract could not be closed.
b. **Protection**
   Contract is checking if the given state is valid.
c. **Tests**
   Following cases should be tested:
   - $contractState = CONTRACT\_STATE\_PAUSED + 1$

3. Set minimum debt as 0

a. **Risk analysis**
   If a minimum debt is zero, then vault debt can be zero. It can be problematic, because vault debt is used as a denominator, while calculating collateral ratio.
b. **Protection**
   Contract is checking if a new minimum debt is greater than zero. Initial minimum debt is by default greater than zero.
c. **Tests**
   Following test cases should be tested:
   - $minimumDebtAmount = 0$

J. Update admin address

1. Sent as not admin

a. **Risk analysis**
   If anyone can set a new admin, anyone can steal anything from the contract.
b. **Protection**
   Contract is checking if the sender updating the properties is the admin.
c. **Tests**
   Following cases should be tested:
   - $this != addresses.adminAddress$

K. Drip interest

1. Contract in paused state

a. **Risk analysis**
   If the contract is paused, collateral price may be invalid. If the collateral price is invalid, wrong collateral value is calculated. If collateral value is not correct, then the liquidation flag can be cleared.
b. **Protection**
   Contract is checking if the state is not paused.
c. **Tests**
   Following cases should be tested:
   - $contractState = CONTRACT\_STATE\_PAUSED$

d. **Static asserts**
   - $assert(coldState.contractState != CONTRACT\_STATE\_PAUSED);$

2. Vault updated long, long time ago

a. **Risk analysis**
   If the vault is not refreshed for a long time, it can be locked forever, because of integer overflow, while calculating accrued interest.

b. **Protection**
   There is no protection, but it takes ~300 years without refreshing to observe the overflow.

c. **Tests**
   Following cases should be tested:
   - $amountOfTimePassed = 1 + \frac{2^{64}}{VAULT\_INTEREST\_RATE}$

L. Redeem vault

1. Amount not paid

a. **Risk analysis**
   If an amount to redeem is not paid, it would be possible to get free collateral from a redeemed vault.

b. **Protection**
   Contract is protected very well using a payment expression. Supply amount is required to be paid.

c. **Tests**
   Reach API does not allow to test the case of missing payment. Creating test in TEAL is the only option, but it is hard to maintain. The best option seems to be using static assertions.

d. **Static asserts**
   $assert(balance() >= amountToRedeem);$

e. add static assert balance(stablecoin) >= amountToRedeem at the end of API call

2. Collateral price slippage

a. **Risk analysis**
   Collateral price can change anytime and in this case the sender wants to be sure that price is within a given range, because the amount of collateral transferred to sender depends on collateral price.

b. **Protection**
   Contract is protected using slippage checks.

c. **Tests**
   Following cases should be tested:
   - $minPrice = collateralPrice + 1$
   - $maxPrice = collateralPrice - 1$
   - $minPrice = minPrice + 1$

d. **Static asserts**
$assert(coldState.collateralPrice >= minPrice)$;
$assert(coldState.collateralPrice <= maxPrice)$;
$assert(minPrice <= maxPrice)$;

3. Contract in paused state

a. **Risk analysis**
If the contract is paused, collateral price may be invalid. If the collateral price is invalid, wrong collateral value is calculated. If collateral value is not correct, more collateral can be received than expected.
b. **Protection**
Contract is checking if the state is not paused.
c. **Tests**
Following cases should be tested:
- $contractState = CONTRACT\_STATE\_PAUSED$

d. **Static asserts**
- $assert(coldState.contractState != CONTRACT\_STATE\_PAUSED)$;

4. Redeem below min balance

a. **Risk analysis**
Minimum vault debt has been introduced to protect against worthless vaults. If the vault is redeemed, it is not worthless anymore, because fees are collected from that vault.

M. Propose vault

1. Propose vault without required minimum collateral

a. **Risk analysis**
If a vault without minimum collateral can be proposed, an attacker could keep proposing the smallest vaults and make redemption unavailable.
b. **Protection**
Only the vault having minimum collateral can be proposed.
c. **Tests**
Following cases should be tested:
- $vault.collateral = MINIMUM\_REDEMPTION\_COLLATERAL\_VALUE - 1$

2. Withdraw collateral from proposed vault (below min collateral limit) and try to replace it

a. **Risk analysis**
If the owner of an already proposed vault withdraws collateral, such vault should be replaced by another proposed vault as soon as possible. If it is not possible, the redeem slot will be blocked until the vault is closed.
b. **Protection**

Vault can be proposed, even if its collateral ratio is higher, but one of the proposed vaults must have less collateral than required.

c. **Tests**
   - Propose the vault
   - Withdraw collateral from that vault:

$$vault.collateral = MINIMUM\_REDEMPTION\_COLLATERAL\_VALUE - 1$$

   - Propose healthier vault

3. Propose vault during cooldown period

a. **Risk analysis**
If a cooldown period is not implemented, it is possible to propose all the vaults, just to get a fee if one of earlier proposed vaults is closed. Attacker can propose his own vault at the end, close it and repeat the attack gaining a lot of collateral

b. **Protection**
Contract waits 12 seconds, before a new vault can be proposed.

c. **Tests**
Propose vault after 11 seconds.

4. Propose vault during cooldown period if some slots are empty or any vault has collateral below limit

a. **Risk analysis**
If proposed vaults are closed right after the last proposal, redemption is not available for 12 seconds.

b. **Protection**
If one of the proposed vaults does not meet the requirements, cooldown period should be ignored.

c. **Tests**
   - Propose the vault
   - Close proposed vault
   - Propose the new vault immediately

5. Propose super healthy vault and get a fee

a. **Risk analysis**
If any vault can be proposed, proposers will always choose the biggest vault to get the highest possible fee, but not the riskiest one.

b. **Protection**
If the vault is super healthy and has a collateral ratio above 150%, the fee should not be transferred to the proposer.

c. **Tests**
   - Propose vault with collateral ratio **151%**
   - Check if fee was transferred

6. Contract is paused state

a. **Risk analysis**

If the contract is paused, collateral price may be invalid. If the collateral price is invalid, wrong collateral value is calculated. If collateral value is not correct, then even a super healthy vault can be proposed and the sender will get a fee.

b. **Protection**
Contract is checking if the state is not paused.

c. **Tests**
Following cases should be tested:
   - $contractState = CONTRACT\_STATE\_PAUSED$

d. **Static asserts**
   - $assert(coldState.contractState != CONTRACT\_STATE\_PAUSED);$

7. **Propose healthier vault**

a. **Risk analysis**
If it is possible to propose a healthier vault, it is possible to get **0.1%** collateral of any vault.

b. **Protection**
Before a proposed vault is accepted, its collateral ratio is always checked.

c. **Tests**
   - Propose the vault with collateral ratio **120%**
   - Propose the other vault with collateral ratio **130%**

Other

Severity classification

We have adopted a severity classification inspired by the **Immunefi Vulnerability Severity Classification System - v2**. It can be found here.