Rivus Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, 0xbepresent, peanuts

May 13th 2024 - May 17th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **rivus-comai-contracts** and **rivusDAO-contract** repositories was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Rivus

Rivus DAO offers a liquid staking solution through rsTAO, in which users can earn staking rewards while staying on the Ethereum Network. LSTs can be used as collateral in DeFi activities and at the same time earn stakes for it.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- fe259d9
- 15f8444

fixes review commit hash - 49967a9402ecfd761bbb94bb17886d1e291814b6

Scope

The following smart contracts were in scope of the audit:

- `rsCOMAI`
- `rsTAO`
- `RivusCOMAI`
- `RivusTAO`

7. Executive Summary

Over the course of the security review, 0xunforgiven, 0xbepresent, peanuts engaged with Rivus to review Rivus. In this period of time a total of **14** issues were uncovered.

Protocol Summary

Protocol Name	Rivus
Repository	https://github.com/cryptoLteam/rivus-contract-audit
Date	May 13th 2024 - May 17th 2024
Protocol Type	Liquid Staking Derivatives Protocol

Findings Count

Severity	Amount
Critical	2
High	1
Medium	2
Low	9
Total Findings	**14**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Unlimited token transfer	Critical	Resolved
[C-02]	Function rebase() apply wrong APR in RivusTAO	Critical	Resolved
[H-01]	Slashing isn't supported in the rebasing mechanism	High	Resolved
[M-01]	The fee calculation is inconsistent	Medium	Acknowledged
[M-02]	APR won't be applied precisely	Medium	Resolved
[L-01]	Use payable.call instead of payable.send	Low	Resolved
[L-02]	Excess msg.value is not refunded to the user	Low	Resolved
[L-03]	Fee calculation is different when converting token to rsToken and back	Low	Resolved
[L-04]	The wrap() and requestUnstake() functions do not specify the minimum expected amount	Low	Acknowledged
[L-05]	The verification of minStakingAmt is performed before fees are deducted	Low	Resolved
[L-06]	User role hasTokenSafePullRole can withdraw wrappedToken	Low	Acknowledged
[L-07]	totalRsTAOMinted may exceed cap	Low	Resolved
[L-08]	Wrong formula in function getWTAOByrsTAOAfterFee()	Low	Resolved

[L-09]	maxUnstakeRequests is not validated when unstakeRequests are reused	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Unlimited token transfer

Severity

Impact: High

Likelihood: High

Description

Function `getSharesByMintedRsTAO()` has been used to calculate amount of shares that corresponds to the RsTAO amount and it's been used in multiple functions like `_transfer()` and `_mintRsTAO()`. The issue is that the function `getSharesByMintedRsTAO()` returns 0 when the total minted amount is 0 while it should have returned the amount itself. This has multiple impacts like transferring unlimited tokens while the total mint is 0, this is the POC:

1. Suppose RivusTAO is recently deployed or for other reasons (upgrade or ...) the total amount is zero.
2. Now attacker can transfer unlimited tokens to 3rd party contracts like DEX or lending platforms and credit tokens for himself.
3. This is possible because when RivusTAO wants to transfer tokens in the `_transfer()` it would call `getSharesByMintedRsTAO()` to calculate the share amount and the share amount would be 0, so the code would have no problem transferring 0 shares.
4. In the end the 3rd party contract would charge the user account while in reality, it didn't receive any tokens. The `transferFrom()` call would return true and won't revert.

There are other impacts. Function `_mintRsTAO()` uses `getSharesByMintedRsTAO()` too and when the return amount is 0 then the code uses `amount` to mint shares. The issue is that the return amount of `getSharesByMintedRsTAO()` can be 0 because of the rounding error (small values of `amount`) and the code should have minted 0 shares while it would mint >0 shares. This can be used to extract value from contracts with small deposit amounts while the share price is high.

This issue exists for rsCOMAI and RivusCOMAI contracts too.

Recommendations

When the total minted tokens are 0 then the code should return `amount` in `getSharesByMintedRsTAO()`. The function `getMintedRsTAOByShares()` should be fixed too. Those fixes should be applied to COMAI contracts too.

[C-02] Function `rebase()` apply wrong APR in RivusTAO

Severity

Impact: High

Likelihood: High

Description

The function `rebase()` is supposed to apply daily APR to the share price by decreasing the total share amount. The issue is that the code uses `totalSharesAmount * apr` to calculate `burnAmount` so the burned amount would be bigger than what it should be. This is the POC:

1. Suppose there are 100 shares and 100 tokens.
2. Admin wants to apply a 20% increase for one day.
3. Code would calculate the burn amount as `100 * 20% = 20` and the new total share would be 80.
4. Now the token to share price would be `100/80 = 1.25` and as you can see the ratio increases by 25%.
5. This would cause a wrong reward rate and those who withdraw sooner would steal others tokens.

Recommendations

Code should use `totalSharesAmount * apr / (1 +apr)` to calculate the burn amount.

8.2. High Findings

[H-01] Slashing isn't supported in the rebasing mechanism

Severity

Impact: High

Likelihood: Medium

Description

The function `rebase()` has been used to increase the share price and apply staking rewards. The issue is that staking has its own risks and stake amounts can be slashed (In Commume or Bittensor network) and current logic doesn't support decreasing the share price. Also, some tokens can be stolen or lost when bridging so it would be necessary to have the ability to decrease the share price to adjust the share price according to those events.

Recommendations

Add the ability to decrease the share price too.

8.3. Medium Findings

[M-01] The fee calculation is inconsistent

Severity

Impact: Medium

Likelihood: Medium

Description

The `RivusCOMAIN::approveMultipleUnstakes` function allows users with the `hasApproveWithdrawalRole` role to approve `requestUnstake` requests and send the requested `wCOMAIN` assets. Users can then call the `RivusCOMAIN::unstake` function to receive the corresponding `wCOMAIN` tokens.

The issue arises because fees can change between the approval of `requestUnstake` and when users perform `unstake`, affecting the actual amount required. Consider the following scenario:

1. A `requestUnstake` for `1 wCOMAI` in exchange for `1 rsCOMAI` is made.
2. The manager approves the request and calculates the `wCOMAI` amount required to be sent to the contract using the `getWCOMAIByrsCOMAIAfterFee` function in line `RivusCOMAI#L724`:

```
File: RivusCOMAI.sol
  function approveMultipleUnstakes(UserRequest[] calldata requests)
    public
    hasApproveWithdrawalRole
    nonReentrant
    checkPaused
  {
...
...
    // Loop through each request to unstake and check if the request is
// valid
    for (uint256 i = 0; i < requests.length; i++) {
...
...
        (
  uint256wcomaiAmt,
  ,
  uint256unstakingFeeAmt
) = getWCOMAIByrsCOMAIAfterFee(unstakeRequests[request.user][request.requestIndex].com

      totalRequiredComaiAmt = totalRequiredComaiAmt + wcomaiAmt + unstakingFeeAmt;
    }
...
...
    // Transfer the COMAI from the withdrawal manager to this contract
    require(
      IERC20(commonWrappedToken).transferFrom(
        msg.sender,
        address(this),
        totalRequiredComaiAmt
      ),
      "comaiAmt transfer failed"
    );
...
...
  }
```

```
File: RivusCOMAI.sol
  function getWCOMAIByrsCOMAIAfterFee(uint256 rsCOMAIAmount)
    public
    view
    returns (uint256, uint256, uint256)
  {
    uint256 unstakingFeeAmt = rsCOMAIAmount * unstakingFee / 1000;
    uint256 bridgingFeeAmt = rsCOMAIAmount * bridgingFee / 1000;
    if(bridgingFeeAmt < 1 * (10 ** decimals())) bridgingFeeAmt = 1 *
  (10 ** decimals());

  uint256 unstakingAmt = rsCOMAIAmount - bridgingFeeAmt - unstakingFeeAmt;
    return (unstakingAmt, bridgingFeeAmt, unstakingFeeAmt);
  }
```

3. Fees are then changed using the `RivusCOMAI::setUnstakingFee` and `RivusCOMAI::setBridgingFee` functions.
4. The user with the approved `requestUnstake` calls the `RivusCOMAI::unstake` function, which again calls the `getWCOMAIByrsCOMAIAfterFee` function in line `RivusCOMAI#L781`, resulting in different fees due to the changes made in `step 3`:

```
File: RivusCOMAI.sol
function unstake(uint256 requestIndex) public nonReentrant checkPaused {

  require(
    requestIndex < unstakeRequests[msg.sender].length,
    "Invalid request index"
  );

UnstakeRequest memory request = unstakeRequests[msg.sender][requestIndex];
  require(request.amount > 0, "No unstake request found");
  require(request.isReadyForUnstake, "Unstake not approved yet");

  // Transfer wCOMAI tokens back to the user
  (
uint256amountToTransfer,
,
uint256unstakingFeeAmt
) = getWCOMAIByrsCOMAIAfterFee(request.comaiAmt
  _transferToVault(address(this), unstakingFeeAmt);

  // Update state to false
  delete unstakeRequests[msg.sender][requestIndex];

  // Perform ERC20 transfer
  bool transferSuccessful = IERC20(request.wrappedToken).transfer(
    msg.sender,
    amountToTransfer
  );
  require(transferSuccessful, "wCOMAI transfer failed");

  // Process the unstake event
  emit UserUnstake(msg.sender, requestIndex, block.timestamp);
}
```

5. The amount transferred to the user may be incorrect due to the fee changes, leading to insufficient `wCOMAI` deposited by the manager in `step 2`.

This inconsistency can cause some unstake operations to fail because the correct amount of `wCOMAI` was not deposited into the `RivusCOMAI` contract via `approveMultipleUnstakes()`.

Recommendations

It is recommended to calculate the fees and wrapped token amounts during the `RivusCOMAI::requestUnstake` function, similar to how it is done in `RivusTAO::requestUnstake#L572`. This way, the same `wCOMAI` and `unstakingFees` values are used consistently in both the `approveMultipleUnstakes()` and `unstake()` functions.

[M-02] APR won't be applied precisely

Severity

Impact: Medium

Likelihood: Medium

Description

The function `rebase()` handles the share price increase to apply the desired APR. The issue is that the code doesn't use `lastRebaseTime` when calculating rate increase and it assumes 24 hours have passed from the last time which couldn't not be the case. This would wrong APR for some of the time. For example if `rebase()` is called 1 hour sooner or later then for those 1 hour the share price rate would be wrong.

Recommendations

Use `duration = block.timestamp - lastRebaseTime` to calculate the real rate increase that accrued from the last rebase call.

8.4. Low Findings

[L-01] Use `payable.call` instead of `payable.send`

In `RivusCOMAI.sol`, `payable.send` is used when sending the serviceFee to the withdrawalManager.

```
// in the guard condition at start of function
    bool success = payable(withdrawalManager).send(serviceFee);
```

Any smart contract that uses send() is taking a hard dependency on gas costs by forwarding a fixed amount of gas: 2300. If gas costs are subject to change, then smart contracts can't depend on any particular gas costs.

Use payable(withdrawalManager){value: serviceFee}.call("") instead.

[L-02] Excess msg.value is not refunded to the user

When the user calls `requestUnstake()`, `msg.value` is required as part of the payment for serviceFee. However, if the user sends more `msg.value` than required, then the excess value will be stuck in the contract.

```
// Ensure that the fee amount is sufficient
    require(msg.value >= serviceFee, "Fee amount is not sufficient");
```

Recommend having a strict equality check instead.

[L-03] Fee calculation is different when converting token to rsToken and back

When converting wToken to rsToken, the fee is calculated as such:

```
function calculateAmtAfterFee(uint256 wcomaiAmount)
    ...
    uint256 _bridgeFee = wcomaiAmount * bridgingFee / 1000;
    if(_bridgeFee < 1 * (10 ** decimals())) _bridgeFee = 1 * (10 ** decimals());
    uint256 amountAfterBridgingFee = wcomaiAmount - _bridgeFee;
    uint256 feeAmount = 0;
    if(stakingFee > 0) {
      feeAmount = (amountAfterBridgingFee * stakingFee) / 1000;
    }
    uint256 amountAfterTotalFees = amountAfterBridgingFee - feeAmount;
  }
```

The total amount deducts the bridging fee and then the new amount deducts the staking fee.

When unstaking, the bridging fee and unstaking fee are calculated on the total amount.

```
function getWCOMAIByrsCOMAIAfterFee(uint256 rsCOMAIAmount)
    public
    view
    returns (uint256, uint256, uint256)
  {
    uint256 unstakingFeeAmt = rsCOMAIAmount * unstakingFee / 1000;
    uint256 bridgingFeeAmt = rsCOMAIAmount * bridgingFee / 1000;
    if(bridgingFeeAmt < 1 * (10 ** decimals())) bridgingFeeAmt = 1 *
      (10 ** decimals());
    uint256 unstakingAmt = rsCOMAIAmount - bridgingFeeAmt - unstakingFeeAmt;
    return (unstakingAmt, bridgingFeeAmt, unstakingFeeAmt);
  }
```

This means that the fee calculation is different when staking and unstaking. Users will pay a little less fee when staking given staking and unstaking percentage is the same.

For example, if both staking and unstaking are 1% and the bridge is 1%, a user with 1000 tokens will pay 10 tokens for unstaking and 10 tokens for the bridge fee, whereas the user will pay 10 tokens for the bridge fee and 9.9 tokens for staking.

Standardize the way the fee calculation is.

[L-04] The `wrap()` and `requestUnstake()` functions do not specify the minimum expected amount

The `exchangeRate`, `stakingFees`, and `bridgingFees` can change before the transactions for the `wrap()` and `requestUnstake()` functions are executed, due to the nature of blockchain and transaction order under incentives. This can result in users receiving fewer assets than expected.

It is necessary to include parameters in the `wrap()` and `requestUnstake()` functions of the `RivusTAO.sol` and `RivusCOMAI.sol` contracts to specify the minimum expected amount in the exchange.

[L-05] The verification of `minStakingAmt` is performed before fees are deducted

The `RivusTAO::wrap` function checks if the `wTAO` amount exceeds the `minStakingAmt` before fees are subtracted:

```
File: RivusTAO.sol
    function wrap(uint256 wtaoAmount) public nonReentrant checkPaused returns (uint256) {
...
...
        // Ensure that at least 0.125 TAO is being bridged
        // based on the smart contract
        require(wtaoAmount > minStakingAmt, "Does not meet minimum staking amount");


        // Ensure that the wrap amount after free is more than 0
        (uint256 wrapAmountAfterFee, uint256 feeAmt) = calculateAmtAfterFee(wtaoAmount);

        uint256 rsTAOAmount = getRsTAObyWTAO(wrapAmountAfterFee);
...
...
    }
```

This approach is flawed as it should ensure that the net amount after fees, which is actually being staked, meets the `minStakingAmt`. Performing the check prior to fee deductions can lead to scenarios where the actual staked amount is less than the intended minimum due to the fees subtracted afterward.

Adjust the validation process to check `minStakingAmt` after the fees have been calculated and deducted from the `wTAO` amount. This ensures that the amount effectively being staked still meets the minimum requirements stipulated by the protocol, preventing users from staking less than the minimum due to fees.

```
function wrap(uint256 wtaoAmount) public nonReentrant checkPaused returns
    (uint256) {
    ...
    ...

--  // Ensure that at least 0.125 TAO is being bridged
--  // based on the smart contract
--  require(wtaoAmount > minStakingAmt, "Does not meet minimum staking amount");

    // Ensure that the wrap amount after free is more than 0
    (uint256 wrapAmountAfterFee, uint256 feeAmt) = calculateAmtAfterFee
      (wtaoAmount);

++  // Ensure that at least 0.125 TAO is being bridged
++  // based on the smart contract
++  require
+ (wrapAmountAfterFee > minStakingAmt, "Does not meet minimum staking amount");

    uint256 rsTAOAmount = getRsTAObyWTAO(wrapAmountAfterFee);
    ...
    ...
  }
```

[L-06] User role `hasTokenSafePullRole` can withdraw `wrappedToken`

The `safePullERC20()` function is designed to allow the withdrawal of tokens from the contract by authorized users:

```
File: RivusTAO.sol

  function safePullERC20(
    address tokenAddress,
    address to,
    uint256 amount
  ) public hasTokenSafePullRole checkPaused {
    _requireNonZeroAddress(to, "Recipient address cannot be null address");

    require(amount > 0, "Amount must be greater than 0");

    IERC20 token = IERC20(tokenAddress);
    uint256 balance = token.balanceOf(address(this));
    require(balance >= amount, "Not enough tokens in contract");

    // "to" have been checked to be a non zero address
    bool success = token.transfer(to, amount);
    require(success, "Token transfer failed");
    emit ERC20TokenPulled(tokenAddress, to, amount);
  }
```

However, this function also allows the withdrawal of `wrappedTokens` (e.g., `wTAO` or `wCOMAI`), which should remain in the contract as they are assets deposited via the `approveMultipleUnstakes()` function and are intended to be available for users when they execute `unstake()`. Allowing the `safePullERC20` function to withdraw `wrappedTokens` can lead to misuse of the protocol's assets, as these tokens are essential for fulfilling user `unstake` requests.

To prevent unauthorized withdrawal of essential assets, the `safePullERC20` function should be restricted to exclude `wrappedTokens`. This ensures that the tokens meant for user `unstake` requests remain available. Alternatively, ensure that the approved amounts for `unstake` are not eligible for withdrawal through the `safePullERC20` function, thus maintaining the integrity of the assets needed for user operations.

[L-07] `totalRsTAOMinted` may exceed `cap`

The `cap` variable is intended to limit the total supply of `rsTAO` and `rsCOMAI`, and is checked in the `wrap()` function to ensure that the total minted does not exceed this cap:

```
File: RivusTAO.sol
  function wrap
  (uint256 wtaoAmount) public nonReentrant checkPaused returns (uint256) {
...
...
    require(
      cap >= totalRsTAOMinted,
      "Deposit amount exceeds maximum"
    );
...
...
```

However, the current implementation allows the `cap` to be exceeded under certain conditions. Consider the following scenario:

1. The `cap` is set to `10` and `totalRsTAOMinted` is `9`.
2. A user performs a `wrap()` operation that would mint `30 rsTAO`, resulting in `totalRsTAOMinted` becoming `39`, thereby exceeding the `cap` of `10`.

This can happen because the `cap` check is performed before the new `rsTAO` is minted and added to the `totalRsTAOMinted`.

It is advisable to adjust the validation logic to include the amount of `rsTAO` to be minted in the cap check. This can be done by moving the cap check to after the calculation of `rsTAO` to be minted. This ensures the cap is not exceeded after new tokens are minted:

```
function wrap(uint256 wtaoAmount) public nonReentrant checkPaused returns
    (uint256) {
    ...
    ...
--  require(
--    cap >= totalRsTAOMinted,
--    "Deposit amount exceeds maximum"
--  );
    ...
    ...
    uint256 rsTAOAmount = getRsTAObyWTAO(wrapAmountAfterFee);

    // Perform token transfers
    _mintRsTAO(msg.sender, rsTAOAmount);
++  require(
++    cap >= totalRsTAOMinted,
++    "Deposit amount exceeds maximum"
++  );
    ...
    ...
  }
```

[L-08] Wrong formula in function `getWTAOByrsTAOAfterFee()`

In the function `getWTAOByrsTAOAfterFee()` code should calculate wTAO amount and then subtract the `unstakingFee` from it. This won't cause an issue in the current code because the ratio is 1 but in general the formula is wrong. The issue exists in COMAI contracts too.

[L-09] `maxUnstakeRequests` is not validated when `unstakeRequests` are reused

The `maxUnstakeRequests` variable helps limit the number of `unstakeRequests` a user can have. This variable is validated in the `requestUnstake()` function in lines `RivusTAO#L607-609`:

```
File: RivusTAO.sol
  function requestUnstake
  (uint256 rsTAOAmt) public payable nonReentrant checkPaused {
...
...
    if (!added) {
      require(
        unstakeRequests[msg.sender].length < maxUnstakeRequests,
        "Maximum unstake requests exceeded"
      );
      unstakeRequests[msg.sender].push(
        UnstakeRequest({
          amount: rsTAOAmt,
          taoAmt: outWTaoAmt,
          isReadyForUnstake: false,
          timestamp: block.timestamp,
          wrappedToken: wrappedToken
        })
      );
...
    }
...
...
  }
```

The problem is that `maxUnstakeRequests` can be modified via the `setMaxUnstakeRequest()` function, causing this variable validation to not work as expected. Consider the following scenario:

1. The owner calls `setMaxUnstakeRequest()` with a value of `2`.
2. `Account1` deposits `10e9 wTAO tokens`, obtaining `10e9 rsTAO tokens`.
3. For some reason, `Account1` performs an `unstaking` of `1e9 rsTAO` and another of `2e9 rsTAO`, obtaining `3e9 wTAO - fees`.
4. The owner modifies `setMaxUnstakeRequest()` and decreases it to a value of `1`.
5. `Account1` performs another `unstaking` of `3e9 rsTAO` and `4e9 rsTAO` in separate transactions, resulting in `Account1` having 2 `requestStaking`. **This is incorrect** as `Account1` should be limited to only 1 `requestUnstaking` since `maxUnstakeRequests` was decreased in `step 4`.

The following test demonstrates the previous scenario:

```
it(
      "maxUnstakeRequestsisnotusedwhenuserhavealreadyunstakeRequests",
      asyncfunction
    ) {
      // setup

      await wTAO.setBridge(owner.address)
      const froms = ["from0", "from1", "from1", "from1"]

             const tos = [owner.address, account1.address, account2.address, account3.
      const amounts = [
        ethers.parseUnits("100", 9),
        ethers.parseUnits("10", 9),
        ethers.parseUnits("10", 9),
        ethers.parseUnits("10", 9)]
      await wTAO.bridgedTo(froms, tos, amounts)
      console.log("\nAccount1 wTAO balance: ", ethers.formatUnits
        (await wTAO.balanceOf(account1.address), 9))
      console.log("Account1 rsTAO balance: ", ethers.formatUnits
        (await rsTAO.balanceOf(account1.address), 9))
      //
      // 1. Owner set max unstakeRequest to 2 just for the testing purposes
      await rsTAO.setMaxUnstakeRequest(2);
      //
      // 2. Account1 stakes wTAO
      let amountToStake = ethers.parseUnits("10", 9);
      console.log("\nStaking", ethers.formatUnits(amountToStake, 9), "wTAO...")
      await wTAO.connect(account1).approve(rsTAO.target, amountToStake)
      await rsTAO.connect(account1).wrap(amountToStake)
      console.log("Account1 wTAO balance: ", ethers.formatUnits
        (await wTAO.balanceOf(account1.address), 9))
      console.log("Account1 rsTAO balance: ", ethers.formatUnits
        (await rsTAO.balanceOf(account1.address), 9))
      //
      // 3. Account1 request unstake 1 wTAO
      console.log("\nRequest unstake 1 wTAO...")
      await rsTAO.connect(account1).requestUnstake(ethers.parseUnits
        ("1", 9), {value: ethers.parseEther("0.003")});
      //
      // 4. Account1 request unstake 2 wTAO
      console.log("\nRequest unstake 2 wTAO...")
      await rsTAO.connect(account1).requestUnstake(ethers.parseUnits
        ("2", 9), {value: ethers.parseEther("0.003")});
      //
      // 5. Account1 unstake both `requestUnstakes`
      console.log("\nOwner approves all the `Account1` requestUnstake")
      const userRequests = [
        { user: account1.address, requestIndex: 0 },
        { user: account1.address, requestIndex: 1 }
      ];
      await wTAO.approve(rsTAO.target, ethers.parseUnits("10", 9))
      await rsTAO.approveMultipleUnstakes(userRequests);
      console.log("\nAccount1 unstake both requests")
      await rsTAO.connect(account1).unstake(0);
      await rsTAO.connect(account1).unstake(1);
      console.log("Account1 wTAO balance: ", ethers.formatUnits
        (await wTAO.balanceOf(account1.address), 9))
      console.log("Account1 rsTAO balance: ", ethers.formatUnits
        (await rsTAO.balanceOf(account1.address), 9))
      //
      // 6. Owner set max unstakeRequest to 1 but the account1 can still use 2
      // request unstakes
      await rsTAO.setMaxUnstakeRequest(1);
      console.log("\nRequest unstake 3 wTAO...")
      await rsTAO.connect(account1).requestUnstake(ethers.parseUnits
        ("1", 9), {value: ethers.parseEther("0.003")});
      console.log("\nRequest unstake 4 wTAO...")
```

```
        await rsTAO.connect(account1).requestUnstake(ethers.parseUnits
          ("2", 9), {value: ethers.parseEther("0.003")});
        let getUserRequests = await rsTAO.getUnstakeRequestByUser
          (account1.address);
        console.log(getUserRequests.length);
      });
```

It is recommended to ensure that `maxUnstakeRequests` is not exceeded within the section where empty `unstakeRequests` are reused in lines `RivusTAO#L577-L602`.

```
File: RivusTAO.sol
555:   function requestUnstake
  (uint256 rsTAOAmt) public payable nonReentrant checkPaused {
...
...
576:     // Loop throught the list of existing unstake requests
577:     for (uint256 i = 0; i < length; i++) {
578:       uint256 currAmt = unstakeRequests[msg.sender][i].amount;
579:       if (currAmt > 0) {
580:         continue;
581:       } else {
582:         // If the curr amt is zero, it means
583:         // we can add the unstake request in this index
584:         unstakeRequests[msg.sender][i] = UnstakeRequest({
585:           amount: rsTAOAmt,
586:           taoAmt: outWTaoAmt,
587:           isReadyForUnstake: false,
588:           timestamp: block.timestamp,
589:           wrappedToken: wrappedToken
590:         });
591:         added = true;
592:         emit UserUnstakeRequested(
593:           msg.sender,
594:           i,
595:           block.timestamp,
596:           rsTAOAmt,
597:           outWTaoAmt,
598:           wrappedToken
599:         );
600:         break;
601:       }
602:     }
...
...
```