Resolv Security Review

Pashov Audit Group

Conducted by: ast3ros, btk, Said

July 27th 2024 - July 29th 2024

Contents

1. About Pashov Audit Group 2
2. Disclaimer 2
3. Introduction 2
4. About wstUSR 2
5. Risk Classification 3
   5.1. Impact 3
   5.2. Likelihood 3
   5.3. Action required for severity levels 4
6. Security Assessment Summary 4
7. Executive Summary 5
8. Findings 7
   8.1. Medium Findings 7
      [M-01] The returned stUSRAmount from the unwrap operation may be inaccurate due to rounding 7
      [M-02] mintWithPermit provides the wrong amount to the usrPermit.permit 9
      [M-03] Incorrect rounding in previewWithdraw function leads to residual wstUSR balance 10
      [M-04] wstUSR.previewWithdraw could return 0 and result in free withdraw 12
   8.2. Low Findings 15
      [L-01] Whales can frontrun rewards distribution 15
      [L-02] WstUSR is not EIP-4626 compliant 15
      [L-03] Potential accumulation of dust amounts of stUSR in the contract due to rounding 16

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 **resolv-im/resolv-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About wstUSR

`WstUSR` is a wrapper for the staked USR token (`stUSR`) that allows users to convert between the underlying token and its wrapped form. `stUSR` has the same value as `USR`, and its amount increases over time with staking rewards. `USR` is a stablecoin backed by ETH and pegged to the US Dollar, ensuring stability through hedging and an insurance liquidity pool.

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 hash - 4f5498fc548eff4db9e5463f9385d27df8a23296

fixes review commit hash - c131133ae1413bcafeb217d6eac46ac0b94de53f

Scope

The following smart contract were in scope of the audit:

- `WstUSR`

7. Executive Summary

Over the course of the security review, ast3ros, btk, Said engaged with Resolv to review wstUSR. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	wstUSR
Repository	https://github.com/resolv-im/resolv-contracts
Date	July 27th 2024 - July 29th 2024
Protocol Type	Stablecoin protocol

Findings Count

Severity	**Amount**
Medium	4
Low	3
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[M-01]	The returned stUSRAmount from the unwrap operation may be inaccurate due to rounding	Medium	Resolved
[M-02]	mintWithPermit provides the wrong amount to the usrPermit.permit	Medium	Resolved
[M-03]	Incorrect rounding in previewWithdraw function leads to residual wstUSR balance	Medium	Resolved
[M-04]	wstUSR.previewWithdraw could return 0 and result in free withdraw	Medium	Resolved
[L-01]	Whales can frontrun rewards distribution	Low	Acknowledged
[L-02]	WstUSR is not EIP-4626 compliant	Low	Resolved
[L-03]	Potential accumulation of dust amounts of stUSR in the contract due to rounding	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] The returned `stUSRAmount` from the `unwrap` operation may be inaccurate due to rounding

Severity

Impact: Low

Likelihood: High

Description

When users `unwrap` their wstUSR, the underlying `stUSRAmount` is returned from the operation :

```
function unwrap(uint256 _wstUSRAmount, address _receiver) public returns
      (uint256 stUSRAmount) {
        _assertNonZero(_wstUSRAmount);

        IERC20Rebasing stUSR = IERC20Rebasing(stUSRAddress);

>>>     stUSRAmount = stUSR.convertToUnderlyingToken(_wstUSRAmount);
        _burn(msg.sender, _wstUSRAmount);
        IERC20(stUSRAddress).safeTransfer(_receiver, stUSRAmount);

        emit Unwrap(msg.sender, _receiver, stUSRAmount, _wstUSRAmount);

>>>     return stUSRAmount;
    }
```

However, the returned `stUSRAmount` value is inaccurate. This issue occurs because of additional rounding down that happens when transferring `stUSR` from the `wstUSR` contract, and when calculating the user's underlying balances based on the transferred shares.

Now, when users call `stUSR.withdraw` using the returned value, the call will revert due to insufficient balance.

```
function withdraw(uint256 _usrAmount, address _receiver) public {
        // @audit - if providing the returned stUSRAmount, the call will revert
        // due to insufficient balance
        uint256 shares = previewWithdraw(_usrAmount);
>>>     super._burn(msg.sender, shares);

        IERC20Metadata usr = super.underlyingToken();
        usr.safeTransfer(_receiver, _usrAmount);
        emit Withdraw(msg.sender, _receiver, _usrAmount, shares);
    }
```

PoC :

```
it("PoC - unwrap and withdraw revert", async () => {
      // given
      const {stUSR, wstUSR, USR, otherAccount2} = await loadFixture
        (deployFixture);
      // Initial deposit
      await USR["mint(address,uint256)"](otherAccount2.getAddress(), parseUSR
        (100));
      const wstUSRFromAnotherSigner2 = wstUSR.connect(otherAccount2);
      const usrFromAnotherSigner2 = USR.connect(otherAccount2);
      const stUSRFromAnotherSigner2 = stUSR.connect(otherAccount2);
      await usrFromAnotherSigner2.approve(wstUSR.getAddress(), parseUSR(100));
      await wstUSRFromAnotherSigner2["deposit(uint256)"](parseUSR(100));
      // Mint USR to StUSR as a reward
      await USR["mint(address,uint256)"](stUSR.getAddress(), parseUSR(100));

      const withdrawAmount = parseUSR(100)
      const expectedWstUSR = await wstUSR.previewWithdraw(withdrawAmount);
      const expectedUsrInShares = await stUSR.convertToUnderlyingToken
        (expectedWstUSR);
      const balancewstBefore = await wstUSR.balanceOf(otherAccount2);
      console.log("balance wstUSR before withdraw : ");
      console.log(balancewstBefore);
      const sharestUSRBefore = await stUSR.sharesOf(otherAccount2);
      console.log("share stUSR before unwrap : ");
      console.log(sharestUSRBefore);
      // when
      const tx = await wstUSRFromAnotherSigner2["unwrap(uint256,address)"]
        (expectedWstUSR, otherAccount2);
      const sharestUSRAfter = await stUSR.sharesOf(otherAccount2);
      console.log("share stUSR after unwrap : ");
      console.log(sharestUSRAfter);
      const tx2 = stUSRFromAnotherSigner2["withdraw(uint256)"]
        (expectedUsrInShares);
      // then
      await expect(tx2)
      .to.revertedWithCustomError(stUSR, "ERC20InsufficientBalance")
      .withArgs(otherAccount2, expectedWstUSR - BigInt(1), expectedWstUSR);

    })
```

Recommendations

Update `stUSRAmount` by querying the user's balance from `stUSR`.

```
function unwrap(uint256 _wstUSRAmount, address _receiver) public returns
      (uint256 stUSRAmount) {
        _assertNonZero(_wstUSRAmount);

        IERC20Rebasing stUSR = IERC20Rebasing(stUSRAddress);

        stUSRAmount = stUSR.convertToUnderlyingToken(_wstUSRAmount);
+       uint256 balanceBefore = stUSR.balanceOf(_receiver);
        _burn(msg.sender, _wstUSRAmount);
        IERC20(stUSRAddress).safeTransfer(_receiver, stUSRAmount);

+       stUSRAmount = stUSR.balanceOf(_receiver) - balanceBefore;

        emit Unwrap(msg.sender, _receiver, stUSRAmount, _wstUSRAmount);

        return stUSRAmount;
    }
```

[M-02] `mintWithPermit` provides the wrong amount to the `usrPermit.permit`

Severity

Impact: Medium

Likelihood: Medium

Description

`WstUSR` allows users to mint wstUSR directly from the contract by providing the required USR amount. If a user wants to mint using permit, they can utilize the `mintWithPermit` function.

```
function mintWithPermit(
        uint256 _wstUSRAmount,
        address _receiver,
        uint256 _deadline,
        uint8 _v,
        bytes32 _r,
        bytes32 _s
    ) external returns (uint256 usrAmount) {
        IERC20Permit usrPermit = IERC20Permit(usrAddress);
        // the use of `try/catch` allows the permit to fail and makes the code
        // tolerant to frontrunning.
        // solhint-disable-next-line no-empty-blocks
        // @audit - this should provide usrAmount
        try usrPermit.permit(msg.sender, address
          (this), _wstUSRAmount, _deadline, _v, _r, _s) {} catch {}
        usrAmount = previewMint(_wstUSRAmount);
        _deposit(msg.sender, _receiver, usrAmount, _wstUSRAmount);

        return usrAmount;
    }
```

However, it provides `_wstUSRAmount` to `usrPermit.permit` instead of the required `usrAmount`. Users operation who sign the permit and provide the required USR amount using the result of `previewMint` will always revert.

Recommendations

provide `usrAmount` instead of `_wstUSRAmount` :

```
function mintWithPermit(
        uint256 _wstUSRAmount,
        address _receiver,
        uint256 _deadline,
        uint8 _v,
        bytes32 _r,
        bytes32 _s
    ) external returns (uint256 usrAmount) {
        IERC20Permit usrPermit = IERC20Permit(usrAddress);
        // the use of `try/catch` allows the permit to fail and makes the code
        // tolerant to frontrunning.
        // solhint-disable-next-line no-empty-blocks
-        try usrPermit.permit(msg.sender, address
- (this), _wstUSRAmount, _deadline, _v, _r, _s) {} catch {}
        usrAmount = previewMint(_wstUSRAmount);
+        try usrPermit.permit(msg.sender, address
+ (this), usrAmount, _deadline, _v, _r, _s) {} catch {}
        _deposit(msg.sender, _receiver, usrAmount, _wstUSRAmount);

        return usrAmount;
    }
```

[M-03] Incorrect rounding in previewWithdraw function leads to residual

wstUSR balance

Severity

Impact: Low

Likelihood: High

Description

The `previewWithdraw` function in the `WstUSR` contract uses incorrect rounding when calculating the amount of wstUSR required to withdraw a certain amount of USR. The function rounds down instead of rounding up, which favors the user rather than the protocol. This leads to a situation where users can withdraw their full USR balance while leaving a small residual wstUSR balance that cannot be withdrawn.

```
function previewWithdraw(uint256 _usrAmount) public view returns
      (uint256 wstUSRAmount) {
        return IStUSR(stUSRAddress).previewWithdraw
        //(_usrAmount) / ST_USR_SHARES_OFFSET; // @audit round down number of wstUSR r
    }
```

POC: link

This test case shows that after depositing 1e18 USR and then withdrawing the maximum amount possible, there is still a residual wstUSR balance left in the account.

- Add foundry to hardhat: link
- Put the file in `test/RoundingErrorPOC.t.sol` and
- Run `forge test -vvvvv --match-path test/RoundingErrorPOC.t.t.sol --match-test testPOC`

Recommendations

Modify the previewWithdraw function to round up the required wstUSR amount:

```
function previewWithdraw(uint256 _usrAmount) public view returns
      (uint256 wstUSRAmount) {
-       return IStUSR(stUSRAddress).previewWithdraw
- (_usrAmount) / ST_USR_SHARES_OFFSET;
+       return IStUSR(stUSRAddress).previewWithdraw(_usrAmount).ceilDiv
+ (ST_USR_SHARES_OFFSET);
    }
```

[M-04] `wstUSR.previewWithdraw` could return 0 and result in free withdraw

Severity

Impact: Medium

Likelihood: Medium

Description

`previewWithdraw` is a function that accepts `_usrAmount` and returns the required `wstUSRAmount` that should be burned to withdraw the `_usrAmount`.This calculation is generally implemented to favor the vault by rounding up the required share amount to prevent issues for the vault. While `stUSR.previewWithdraw` already implements virtual shares and rounding up the calculation, `wstUSR.previewWithdraw` has the following implementation.

```
function previewWithdraw(uint256 _usrAmount) public view returns
      (uint256 wstUSRAmount) {
        return IStUSR(stUSRAddress).previewWithdraw
          (_usrAmount) / ST_USR_SHARES_OFFSET;
    }
```

If the returned value from `IStUSR(stUSRAddress).previewWithdraw(_usrAmount)` is less than `ST_USR_SHARES_OFFSET`, the required `wstUSRAmount` will be 0. This situation could occur if an attacker or griefer donates `USR` to the `stUSR` to manipulate the `stUSR` vault share price. Although it is difficult to make this profitable, the required donation and gas costs for a small `withdraw` operation could potentially make the profit even smaller. However, the natural growth of `USR` inside the `stUSR` vault could make the desired share price for the attack reach more quickly. Overall, the combination of these factors could make the attack cheaper and more profitable.

PoC :

```
it("PoC - previewWithdraw return 0", async () => {
      // given

      const wstUSRFromAnotherSigner = wstUSR.connect(otherAccount);

      await USR["mint(address,uint256)"](otherAccount, parseUSR(3000));
      const usrFromAnotherSigner = USR.connect(otherAccount);
      await usrFromAnotherSigner.approve(wstUSR, parseUSR(3000));

      // when
      const tx = await wstUSRFromAnotherSigner["deposit(uint256)"](parseUSR
        (3000));
      const balancewstAfter = await wstUSR.balanceOf(otherAccount);
      console.log("balance wstUSR after deposit : ");
      console.log(balancewstAfter);
      // test
      const totalSupplyBefore = await stUSR.totalSupply();
      console.log("total supply before : ");
      console.log(totalSupplyBefore);
      const totalShares = await stUSR.totalShares();
      console.log("total shares : ");
      console.log(totalShares);
      await USR["mint(address,uint256)"](otherAccount1, parseUSR(300000));
      const usrFromAnotherSigner1 = USR.connect(otherAccount1);
      await usrFromAnotherSigner1.transfer(stUSR, parseUSR(300000));
      const previewWIthdraw = await wstUSR.previewWithdraw(100);
      console.log("previewWithdrawResult : ");
      console.log(previewWIthdraw);
    });
```

Log output :

```
balance wstUSR after deposit :
3000000000000000000000n
total supply before :
3000000000000000000000n
total shares :
3000000000000000000000000n
previewWithdrawResult :
0n
```

Recommendation :

if the result of `wstUSR.previewWithdraw` is 0, revert the `withdraw` operations :

```
function withdraw(
      uint256_usrAmount,
      address_receiver,
      address_owner
    ) public returns (uint256 wstUSRAmount
        uint256 maxUsrAmount = maxWithdraw(_owner);
        if (_usrAmount > maxUsrAmount) revert ExceededMaxWithdraw
          (_owner, _usrAmount, maxUsrAmount);
        wstUSRAmount = previewWithdraw(_usrAmount);
+       _assertNonZero(wstUSRAmount);
        _withdraw(msg.sender, _receiver, _owner, _usrAmount, wstUSRAmount);

        return wstUSRAmount;
    }
```

8.2. Low Findings

[L-01] Whales can frontrun rewards distribution

The WstUSR vault allows users to deposit and withdraw within the same block. This enables malicious users to monitor the mempool and front-run the reward distribution by making a large deposit, and then withdrawing within the same block, which allows them to siphon off a portion of the rewards. To mitigate this vulnerability, consider enforcing a delay between deposits and withdrawals.

[L-02] WstUSR is not EIP-4626 compliant

The WstUSR implementation returns `stUSRAddress.totalSupply()` as its `totalManagedUsrAmount`:

```
function totalAssets() external view returns
      (uint256 totalManagedUsrAmount) {
        return IERC20Rebasing(stUSRAddress).totalSupply();
    }
```

According to EIP-4626:

> totalAssets: Total amount of the underlying asset that is “managed” by Vault.

- The purpose of the `totalAssets()` function is to provide the total amount of the underlying assets currently held in the vault. This means the contract needs to track the assets that are deposited (or transferred) only through this contract.
- However, `WstUSR.totalAssets()` returns `stUSR.totalSupply()`, which makes it incompatible with the EIP-4626 standard since not all `stUSR` underlying assets are managed by `WstUSR`.
- This implementation can lead to unexpected behavior and integration issues in the future.

Consider following the EIP-4626 standard or properly document this behavior.

[L-03] Potential accumulation of dust amounts of stUSR in the contract due to rounding

The WstUSR contract's wrap and unwrap functions may lead to small amounts of stUSR (dust) accumulating in the contract over time due to rounding in the conversion process between stUSR and wstUSR.

In the `wrap` function, users transfer `_stUSRAmount` of stUSR, but receive `wstUSRAmount` of wstUSR, which is reduced by a factor of `ST_USR_SHARES_OFFSET` (1000):

```
function wrap(uint256 _stUSRAmount, address _receiver) public returns
      (uint256 wstUSRAmount) {
        ...

        wstUSRAmount = stUSR.convertToShares
          (_stUSRAmount) / ST_USR_SHARES_OFFSET;

        IERC20(stUSRAddress).safeTransferFrom(msg.sender, address
          (this), _stUSRAmount);
        ...
    }
```

Conversely, in the `unwrap` function, users receive `_wstUSRAmount * ST_USR_SHARES_OFFSET` amount of stUSR shares:

```
function unwrap(uint256 _wstUSRAmount, address _receiver) public returns
      (uint256 stUSRAmount) {
        _assertNonZero(_wstUSRAmount);

        IERC20Rebasing stUSR = IERC20Rebasing(stUSRAddress);

        uint256 stUSRSharesAmount = _wstUSRAmount * ST_USR_SHARES_OFFSET;
        stUSRAmount = stUSR.convertToUnderlyingToken(stUSRSharesAmount);
        _burn(msg.sender, _wstUSRAmount);
        // slither-disable-next-line unused-return
        stUSR.transferShares(_receiver, stUSRSharesAmount);
        ...
    }
```

The combination of these operations can be represented as `_stUSRAmount / ST_USR_SHARES_OFFSET * ST_USR_SHARES_OFFSET`. Due to integer division, this may result in a value slightly less than the original `_stUSRAmount`, leaving a small amount of stUSR in the contract.