Ethena Security Review

Pashov Audit Group

Conducted by: T1MOH, btk, peanuts

October 17th - October 20th

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

4. About Ethena

Ethena USDB smart contract allows minting and redeeming stablecoins backed by tokenized assets, with strict limits and role-based access control.

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 - ae1856b5f23db2436ffc811f648194328bbbe58c

fixes review commit hash - c2256464c586b984cf97a0e9e41c9fcd4a3ef554

Scope

The following smart contracts were in scope of the audit:

- `SingleAdminAccessControl`
- `SingleAdminAccesscontrolUpgradeable`
- `UStb`
- `UStbMinting`

7. Executive Summary

Over the course of the security review, T1MOH, btk, peanuts engaged with Ethena to review Ethena. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Ethena
Repository	https://github.com/ethena-labs/ethena-ustb-audit
Date	October 17th - October 20th
Protocol Type	Synthetic Dollar Protocol

Findings Count

Severity	**Amount**
Low	7
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[L-01]	Some variables are never used	Low	Resolved
[L-02]	Blacklisted tokens can be transferred during WHITELIST_ENABLED	Low	Resolved
[L-03]	Incorrect variable's type used to calculate ORDER_TYPE	Low	Resolved
[L-04]	_beforeTokenTransfer() should check that the WHITELISTED_ROLE does not have a BLACKLISTED_ROLE	Low	Resolved
[L-05]	Functions should check whether tokenAsset is active	Low	Resolved
[L-06]	Blacklisted users can bypass restriction through approvals	Low	Resolved
[L-07]	Blacklisted users can front-run redistributeLockedAmount and burn their tokens	Low	Acknowledged

8. Findings

8.1. Low Findings

[L-01] Some variables are never used

There are 2 such variables in UStbMinting.sol:

```
bytes32 private constant EIP712_DOMAIN_TYPEHASH = keccak256(abi.encodePacked
    (EIP712_DOMAIN));
  bytes32 private constant ROUTE_TYPE = keccak256("Route
    (address[] addresses,uint128[] ratios)");
```

Either remove or add missing functionality.

[L-02] Blacklisted tokens can be transferred during `WHITELIST_ENABLED`

In `FULLY_ENABLED` it disallows transferring blacklisted tokens. However, during `WHITELIST_ENABLED` blacklisted logic is missing:

```
function _beforeTokenTransfer
    (address from, address to, uint256) internal virtual override {
    // State 2 - Transfers fully enabled except for blacklisted addresses
    if (transferState == TransferState.FULLY_ENABLED) {
      if (hasRole(BLACKLISTED_ROLE, msg.sender) || hasRole
        (BLACKLISTED_ROLE, to)){
          revert OperationNotAllowed();
      }
      if (hasRole(BLACKLISTED_ROLE, from) && to != address(0)) {
        revert OperationNotAllowed();
      }
    // State 1 - Transfers only enabled between whitelisted addresses
    } else if (transferState == TransferState.WHITELIST_ENABLED) {
      if (!hasRole(WHITELISTED_ROLE, msg.sender) || !hasRole
        (WHITELISTED_ROLE, to)){
        revert OperationNotAllowed();
      }
    // State 0 - Fully disabled transfers
    } else if (transferState == TransferState.FULLY_DISABLED) {
      revert OperationNotAllowed();
    }
  }
```

Whitelisted addresses can transfer tokens from the blacklisted owner. Add an additional check to `WHITELIST_ENABLED` state:

```
} else if (transferState == TransferState.WHITELIST_ENABLED) {
      if (!hasRole(WHITELISTED_ROLE, msg.sender) || !hasRole
        (WHITELISTED_ROLE, to)){
        revert OperationNotAllowed();
      }
+     if (hasRole(BLACKLISTED_ROLE, from) && to != address(0)) {
+       revert OperationNotAllowed();
+     }
```

[L-03] Incorrect variable's type used to calculate `ORDER_TYPE`

Here you can see `uint128 expiry,uint120 nonce`:

```
bytes32 private constant ORDER_TYPE = keccak256(
    "Order(
      stringorder_id,
      uint8order_type,
      uint128expiry,
      uint120nonce,
      addressbenefactor,
      addressbeneficiary,
      addresscollateral_asset,
      uint128collateral_amount,
      uint128ustb_amount
    )"
  );
```

However actual types are slightly different:

```
struct Order {
    string order_id;
    OrderType order_type;
@>  uint120 expiry;
@>  uint128 nonce;
    ...
  }
```

Using EIP712 as it is will produce an incorrect signature, though easily mitigateable. Update `ORDER_TYPE` to contain correct types.

[L-04] `_beforeTokenTransfer()` should check that the WHITELISTED_ROLE does not

have a BLACKLISTED_ROLE

An address can have both the white and blacklisted role in UStb.sol.

Consider the scenario where the `BLACKLIST_MANAGER_ROLE` and `WHITELIST_MANAGER_ROLE` are not the same person. If a whitelisted address turns malicious and the whitelist manager role is not available to remove the whitelist, the blacklist manager can still blacklist the address.

Ensure that in `transferState == TransferState.WHITELIST_ENABLED`, the `WHITELISTED_ROLE` does not have the `BLACKLISTED_ROLE` as well.

```
// State 1 - Transfers only enabled between whitelisted addresses
    } else if (transferState == TransferState.WHITELIST_ENABLED) {
+     if (!hasRole(WHITELISTED_ROLE, msg.sender) || !hasRole
+ (WHITELISTED_ROLE, to) || hasRole(BLACKLISTED_ROLE, msg.sender) || hasRole(BLACKLIST
        revert OperationNotAllowed();
      }
```

[L-05] Functions should check whether tokenAsset is active

`setMaxMintPerBlock()` is called by DEFAULT_ADMIN_ROLE to set the new max mint of the asset per block. The function should have additional checks, like checking `tokenConfig[asset].isActive`.

```
function setMaxMintPerBlock(
  uint128_maxMintPerBlock,
  addressasset
) external onlyRole(DEFAULT_ADMIN_ROLE
    _setMaxMintPerBlock(_maxMintPerBlock, asset);
  }

  function _setMaxMintPerBlock
    (uint128 _maxMintPerBlock, address asset) internal {
    //@audit - like other functions, should check `tokenConfig[asset].isActive`
    uint128 oldMaxMintPerBlock = tokenConfig[asset].maxMintPerBlock;
    tokenConfig[asset].maxMintPerBlock = _maxMintPerBlock;
    emit MaxMintPerBlockChanged(oldMaxMintPerBlock, _maxMintPerBlock, asset);
  }
```

Also good to add a zero amount check since it is done in other functions as well.

[L-06] Blacklisted users can bypass restriction through approvals

The UStb token is an upgradeable ERC20 contract that includes mint and burn functionality, as well as multiple transfer states:

- FULLY_DISABLED
- WHITELIST_ENABLED
- FULLY_ENABLED

According to the documentation, blacklisted users should be restricted from sending or receiving tokens:

> "In any case blacklisted addresses cannot send or receive tokens."

However, the OpenZeppelin ERC20 contract allows approved addresses to transfer tokens on behalf of another address. This creates a loophole: blacklisted users can still transfer their tokens if the whitelist is enabled, as the `_beforeTokenTransfer()` function does not check the from address for blacklisting.

Here is a coded PoC to demonstrate the issue:

```
function testBypassBlacklistRole() public {
        address user1 = makeAddr("user1");
        address user2 = makeAddr("user2");

        vm.startPrank(newOwner);
        UStbContract.updateTransferState
          (IUStbDefinitions.TransferState.WHITELIST_ENABLED);
        UStbContract.grantRole(WHITELISTED_ROLE, user1);
        UStbContract.grantRole(WHITELISTED_ROLE, user2);
        vm.stopPrank();

        vm.prank(minter_contract);
        UStbContract.mint(user1, _amount);

        vm.prank(newOwner);
        UStbContract.revokeRole(WHITELISTED_ROLE, user1);

        vm.prank(newOwner);
        UStbContract.grantRole(BLACKLISTED_ROLE, user1);

        vm.prank(user1);
        UStbContract.approve(user2, _amount);

        vm.prank(user2);
        UStbContract.transferFrom(user1, user2, _amount);

        assert((UStbContract.balanceOf(user2)) == _amount);
    }
```

Test Setup:

- Incorporate the tests in `UStbTest`
- Execute: `forge test --mc UStbTest --mt testBypassBlacklist`

Adding a blacklist check in the `_beforeTokenTransfer()` function to block transfers from a blacklisted address may cause issues, such as preventing the admin from calling `redistributeLockedAmount()`. A better approach would be to block blacklisted users from using the approve function. You can achieve this by adding a check like this:

```
function _approve
    (address owner, address spender, uint256 value) internal virtual override {
    if (hasRole(BLACKLISTED_ROLE, owner)) {
      revert OperationNotAllowed();
    }
    super._approve(owner, spender, value);
  }
```

[L-07] Blacklisted users can front-run `redistributeLockedAmount` and burn their

tokens

The UStb token contract includes a blacklist mechanism that restricts certain addresses from transferring tokens. It also provides an admin function to forcibly transfer tokens from blacklisted addresses to non-blacklisted ones.

However, there is a vulnerability where blacklisted users can front-run the redistributeLockedAmount function and burn their tokens. This allows them to prevent the admin from redistributing their tokens to another address. They achieve this by approving a burner account to spend and burn their tokens before the admin's redistribution takes place.

Here is a coded PoC to demonstrate the issue:

```
function testFrontrunRedistributeLockedAmount() public {
        address alex = makeAddr("Alex");
        address alexBurner = makeAddr("AlexBurner");

        vm.prank(minter_contract);
        UStbContract.mint(alex, _amount);

        vm.startPrank(newOwner);
        UStbContract.updateTransferState
          (IUStbDefinitions.TransferState.FULLY_ENABLED);
        UStbContract.grantRole(BLACKLISTED_ROLE, alex);
        vm.stopPrank();

        // Admin attempt to redistribute alex tokens
        // Alex
        //(The blacklisted user) approve his burner account to spend his tokens
        // alexBurner calls burnFrom to burn Alex tokens

        vm.prank(alex);
        UStbContract.approve(alexBurner, _amount);

        vm.prank(alexBurner);
        UStbContract.burnFrom(alex, _amount);

        uint256 balanceBef = UStbContract.balanceOf(newOwner);

        vm.prank(newOwner);
        UStbContract.redistributeLockedAmount(alex, newOwner);

        uint256 balanceAft = UStbContract.balanceOf(newOwner);

        // newOwner received 0 tokens as they were all burned
        assert(balanceBef == balanceAft);
    }
```

- Incorporate the tests in `UStbTest`
- Execute: **`forge test --mc UStbTest --mt testFrontrunRedistributeLockedAmount`**

To mitigate this issue, consider adding access controls to the `burnFrom()` function to prevent blacklisted users from burning their tokens.