USDV

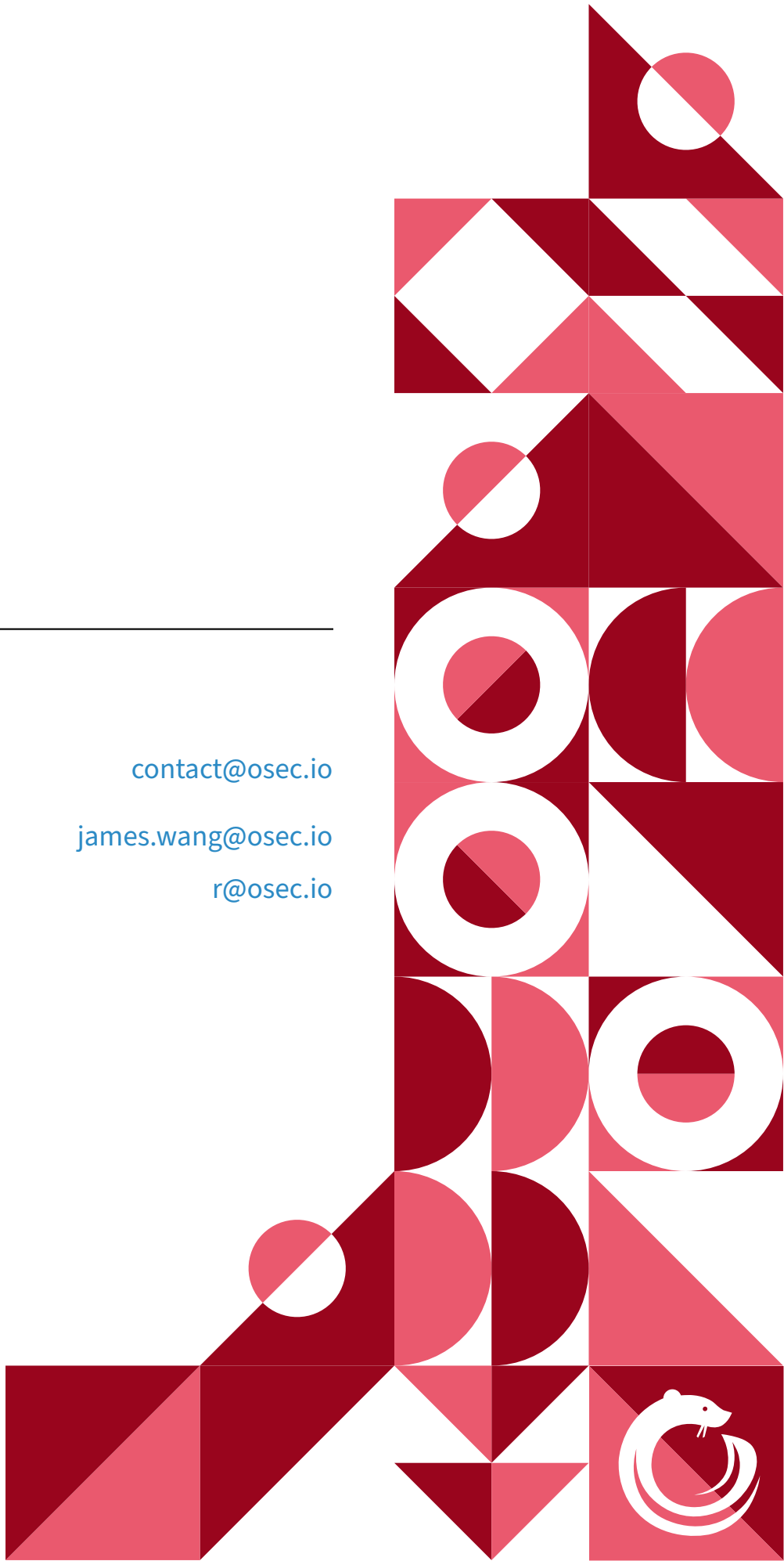
Audit

Presented by:

OtterSec contact@osec.io

James Wang james.wang@osec.io

Robert Chen r@osec.io



Contents

01 | Executive Summary

Overview

USDV engaged OtterSec to perform an assessment of the `usdv` program. This assessment was conducted between October 2nd and October 13th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 7 findings in total.

In particular, we discovered that incorrect function overrides prevent the retry of failed messages (OS-USDV-ADV-00), mistakes in message length checks hinder the proper receipt of `sendAndCall` messages (OS-USDV-ADV-01), and incorrect reward capping may cause a temporary denial of service on reward distribution until admin intervention (OS-USDV-ADV-02).

We also recommended updates to the rate limiter setting to ensure rate limits are fully respected (OS-USDV-SUG-01). We further suggested implementing checks against the sum of multiple fees to limit the impact of operator configuration mistakes on users (OS-USDV-SUG-02).

02 | Scope

The source code was delivered to us in a git repository at github.com/LayerZero-Labs/usdv. This audit was performed up to commit e20bf33.

All contracts in `packages/usdv/evm/contracts/contracts/usdv` and `packages/usdv/evm/contracts/contracts/Vault`, excluding `MessagingV2.sol` are within audit scope.

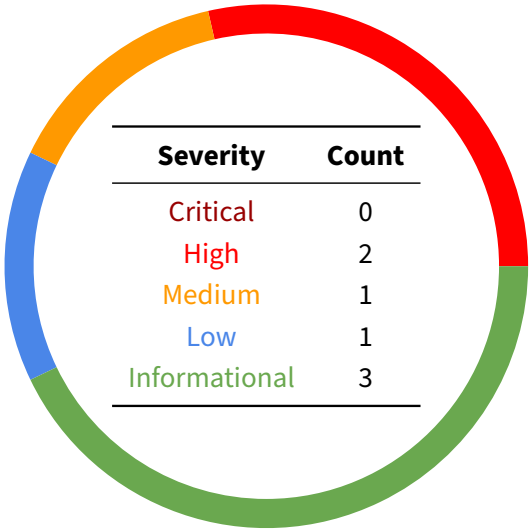
A brief description of the programs are as follows:

Name	Description
usdv	`usdv` includes the implementation of a custom ERC20 that realizes the token coloring algorithm proposed by USDV. It also includes messaging components responsible for sending messages through USDV.
vault	`Vault` handles `usdv` minting based on whitelisted collateral tokens. It is also responsible for pro-rata reward distribution with respect to the amount of `usdv` each minter color holds.

03 | Findings

Overall, we reported 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	0
High	2
Medium	1
Low	1
Informational	3

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-USDV-ADV-00	High	Resolved	Failed messages in `MessagingV1` cannot be retried due to overriding incorrect function.
OS-USDV-ADV-01	High	Resolved	Incorrect message length check in `MsgCodec` decode logic makes it impossible to receive `sendAndCall` message.
OS-USDV-ADV-02	Medium	Resolved	`VaultManager.distributeReward` does not properly utilize capped `rateLimitedRewardInUSDV`, resulting in reverts when result exceeds mint rate limit.
OS-USDV-ADV-03	Low	Resolved	`VaultManager.clearPendginRemint` does not update `lastColor` in for loop, rendering color duplication check dysfunctional.

OS-USDV-ADV-00 [high] | Broke Retry Of Failed Message

Description

In `MessagingV1.sol`, the processing of incoming messages is handled by `nonblockingLzReceive`, while `_nonblockingLzReceive` has been left unimplemented. However, since `retryMessage` only invokes `_nonblockingLzReceive`, leaving it unimplemented means that the retry function won't execute the actual message reception logic when called.

packages/usdv/evm/contracts/contracts/usdv/MessagingV1.sol SOLIDITY

```
// @notice overrides the parent function to use _message as calldata
function nonblockingLzReceive(
    uint16 /*_srcChainId*/,
    bytes calldata /*_srcAddress*/,
    uint64 /*_nonce*/,
    bytes calldata _message
) public override {
    // only internal transaction
    require(msg.sender == address(this), "MessagingV1: only self");

    uint8 msgType = _message.msgType();

    if (msgType == MsgCodec.MSG_TYPE_SEND) {
        [...]
    }
}

//@notice do nothing
function _nonblockingLzReceive(
    uint16 _srcChainId,
    bytes memory _srcAddress,
    uint64 _nonce,
    bytes memory _payload
) internal override {}
```

Remediation

Move the message handling logic into `_nonblockingLzReceive` so `retryMessage` can also utilize it.

Patch

Resolved in a58fe68.

OS-USDV-ADV-01 [high] | Incorrect Message Length Check

Description

`MsgCodec` is responsible for data serialization and deserialization before pushing messages to LayerZero. The `decodeSendAndCallMsg` function attempts to perform a sanity check on the length of deserialized data but performs the opposite of the required comparison. Since all messages for `SendAndCallMsg` will have at least 53 bytes, this incorrect check renders the entire API unusable.

usdv2/packages/usdv/evm/contracts/contracts/messaging/libs/MsgCodec.sol SOLIDITY

```
function decodeSendAndCallMsg(bytes calldata _message) internal pure returns
    ↪  (SendAndCallMsg memory) {
    if (_message.length >= 53) revert InvalidSize();
    [...]
}
```

Remediation

Assert against the correct condition `_message.length < 53`.

Patch

Resolved in e216fb0.

OS-USDV-ADV-02 [med]| Incorrect Reward Capping

Description

`VaultManager` uses `mintRateLimiter` to limit the amount of USDV that can be minted for each token over time. Since rewards are also distributed in the form of minting USDV, they should also be subject to rate limits.

To allow distribution when the total reward exceeds the cap, `VaultManager` calculates the minting `limit` and uses it to determine how much reward to process when `distributeReward` is called. However, a mistake is made, and the calculated and capped `rateLimitedRewardInUSDV` is not used in later calculations, nullifying the attempt.

packages/usdv/evm/contracts/contracts/vault/VaultManager.sol SOLIDITY

```
function distributeReward(address[] calldata _tokens) external nonReentrant
    ↪ whenNotPaused {
    [...]
    for (uint i = 0; i < _tokens.length; i++) {
        address token = _tokens[i];
        Asset.Info storage asset = assetInfos[token];

        uint rewardInUSDV = asset.distributable();

        asset.mintRateLimiter.refill();
        uint limit = asset.mintRateLimiter.tokens;

        uint rateLimitedRewardInUSDV = rewardInUSDV > limit ? limit : rewardInUSDV;
        if (rateLimitedRewardInUSDV == 0) continue; // skip if no reward

        _mint(token, address(this), rewardInUSDV.toUint64(), color, 0x0, false);
        [...]
    }
    [...]
}
```

In extreme conditions where the total unclaimed reward exceeds the maximum minting capacity of the target token, it becomes impossible to distribute any reward until `Operators` intervene and temporarily raise the minting capacity.

Remediation

Use the calculated `rateLimitedRewardInUSDV` when minting reward.

Patch

Resolved in 68b82c7.

OS-USDV-ADV-03 [low] | Incorrect Color Duplicate Check

Description

In `clearPendingRemint`, the for loop checks the current color against the local variable `lastColor` to ensure that user passed `_deltas` are sorted and do not contain duplicate colors. However, since `lastColor` is never updated within the for loop, this check does not work as intended.

packages/usdv/evm/contracts/contracts/vault/VaultManager.sol SOLIDITY

```
function clearPendingRemint(Delta[] calldata _deltas) external nonReentrant
    ↪ whenNotPaused {
    int64 totalDelta;

    Delta calldata delta;
    uint32 lastColor = 0;
    for (uint i = 1; i < _deltas.length; i++) {
        delta = _deltas[i];
        if (delta.color <= lastColor) revert InvalidColor(delta.color); //
            ↪ duplicated

        _clampPendingRemint(delta);
        _burnVST(delta.color, delta.amount, false);
        totalDelta += delta.amount;
    }
    [...]
}
```

Although other checks within `clearPendingRemint` and `_burnVST` protect the state and render this coding mistake unexploitable, it is still imperative to address this bug and remediate appropriately.

Remediation

Update `lastcolor` within for loop.

Patch

Resolved in f9f5497.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-USDV-SUG-00	USDV`Base` does not contain gap slots to guard against potential storage collisions in future upgrades.
OS-USDV-SUG-01	Refill `RateLimiter` before setting new `rate` so that the previous rate will be fully respected.
OS-USDV-SUG-02	Sum of `_minterRemintFeeBps` and `_operatorRemintFeeBps` are not checked to not exceed 100%.

OS-USDV-SUG-00 | Missing Gap Storage Slot

Description

By default, Solidity utilizes a linear storage layout, meaning that the storage for child contracts follows directly after inherited parent contracts. Consequently, if a contract undergoes an upgrade and the parent contracts increase their storage usage, it will overlap with the storage that was originally assigned for child contract usage.

The typical approach to prevent such overlapping is to pre-allocate spare slots in the parent contract. In the event of an upgrade where a parent contract requires more storage, these slots are then utilized. This ensures independence between parent and child contract storage.

`USDVBase` is missing these gap slots. Although the current versions of `USDVMain` and `USDVSide` don't use any storage, and the immediate risk of storage collisions is minimal, it is advised to adhere to best coding practices, as bugs of this nature can result in severe consequences.

Remediation

Add gap slots at the end of `USDVBase` contract.

packages/usdv/evm/contracts/contracts/usdv/MessagingV1.sol DIFF

```
 abstract contract USDVBase is IUSDV, ERC20PermitUpgradeable {
     [...]
+    uint256[49] private __gap;
 }
```

Patch

Resolved in 80b9b21.

OS-USDV-SUG-01 | Refill RateLimiter Before Setting Rate

Description

`RateLimiters` are used to limit the speed at which USDV can be minted or burned with respect to certain collateral tokens. In the current implementation, when `operators` attempt to update the refill rate of `RateLimiters`, the rate is immediately modified, without any preceding refill. As a result, the time between the most recent refill and the configuration of the new rate will utilize the updated `rate` instead of the initial `rate`, which is not optimal.

packages/usdv/evm/contracts/contracts/vault/VaultManager.sol SOLIDITY

```
function setRate(address _token, uint64 _rate) external onlyRole(Role.OPERATOR) {
    assetInfos[_token].mintRateLimiter.setRate(_rate);
}
```

packages/usdv/evm/contracts/contracts/vault/libs/RateLimiter.sol SOLIDITY

```
function setRate(Info storage _self, uint64 _rate) internal {
    _self.rate = _rate;
}
```

Remediation

Perform a `refill` before setting a new `rate`, so that at any moment, the current `RateLimiter.rate` is fully respected.

Patch

Resolved in 6527986.

OS-USDV-SUG-02 | Unchecked Fee Sum

Description

`Operators` are permitted to set the fees collected by `operator` and `minter` during remints. The current implementation checks that each individual fee does not exceed 100% of the reminted token amount, but fails to verify the total sum of these fees.

packages/usdv/evm/contracts/contracts/usdv/Operator.sol SOLIDITY

```
function setOperatorRemintFeeBps(uint16 _operatorRemintFeeBps) external onlyOwner {
    require(_operatorRemintFeeBps <= 10000, "Operator: invalid
        ↪  operatorRemintFeeBps");
    operatorRemintFeeBps = _operatorRemintFeeBps;
    emit OperatorRemintFeeBpsChanged(_operatorRemintFeeBps);
}

function setMinterRemintFeeBps(uint16 _minterRemintFeeBps) external onlyOwner {
    require(_minterRemintFeeBps <= 10000, "Operator: invalid minterRemintFeeBps");
    minterRemintFeeBps = _minterRemintFeeBps;
    emit MinterRemintFeeBpsChanged(_minterRemintFeeBps);
}
```

Remediation

Implement checks to ensure the sum of fees does not exceed 100%

Patch

Resolved in 34a5c5d.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings section.

Critical	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
High	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
Medium	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.