USDV

Smart Contract Security Assessment

October 26, 2023

Prepared for:

USDV

Prepared by:

Jasraj Bedi, Katerina Belotskaia, and Aaron Esau

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for LayerZero Labs from September 25th to October 10th, 2023. During this engagement, Zellic reviewed USDV's code for security vulnerabilities, design issues, and general weaknesses in security posture.

The follow up patches were reviewed upto the commit e20bf331

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is it possible for a minter to abuse the design to reap more rewards than they should be entitled to?
- Is it possible for a minter to be in a non-remintable state when they are in a deficit?
- Can flash loans lead to unfair coloring for the distributors / TVL aggregators?
- Is the cross-chain messaging secure and sound?
- Is the instant finality guarantee always maintained?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- MM components

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

The focus of this assessment was the general design and architechture of USDV and its asynchronous cross-chain nature. The review of the implementation was secondary, with majority of the time being spent on the potential abuses of the unique design of the project.

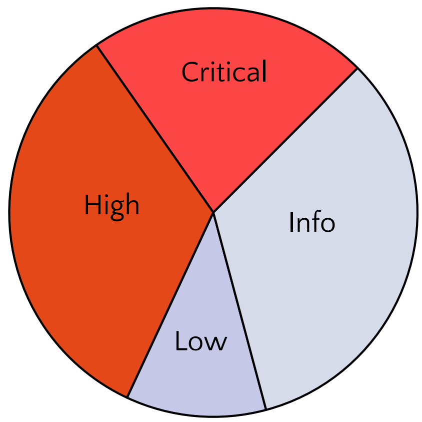
1.3 Results

During our assessment on the scoped USDV contracts, we discovered nine findings. Two critical issues were found. Three were of high impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for LayerZero Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	3
Medium	0
Low	1
Informational	3



2 Introduction

2.1 About USDV

USDV is an omnichain stablecoin backed by a basket of whitelisted, highly secure assets such as T-Bills tokens. USDV is fully compatible with the ERC-20 standard and built with a novel coloring algorithm that attributes minters in circulations.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

USDV Contracts

Repository	https://github.com/layerZero-Labs/usdv/
Version	usdv: `2c4196c1c0c1020f1de52d605e837672b6328645`
Program	packages/usdv/evm/contracts/contract/**.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of four person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi, Co-founder/Engineer
jazzy@zellic.io

Katerina Belotskaia, Engineer
kate@zellic.io

Aaron Esau, Engineer
aaron@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 25, 2023	Start of primary review period
October 10, 2023	End of primary review period

3 Detailed Findings

3.1 Inverted authentication logic in `VaultManager.setRole`

- **Target**: VaultManager
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

VaultManager's `setRole` function configures whether the caller is authorized to call certain functions in the contract.

The `setRole` function itself performs a check to ensure that the configurer is authorized:

```
function setRole(Role _role, address _addr) external {
    // both owner and self are valid for all roles config
    bool validCaller = msg.sender == govInfo.roles[_role] || msg.sender
    == govInfo.roles[Role.OWNER];

    // foundation can only change the operator if
    // (a) the operator is address(0x0) or
    // (b) the operator has not interacted with the contract for 30 day
    if (!validCaller && _role == Role.OPERATOR) {
        if (govInfo.roles[Role.OPERATOR] == address(0) || block.timestamp
    - govInfo.operatorLastPing > 30 days) {
            validCaller = msg.sender == govInfo.roles[Role.FOUNDATION];
        }
    }

    if (validCaller) revert Unauthorized();
    govInfo.roles[_role] = _addr;
}
```

Note that execution reverts if the caller is valid — not if the caller is invalid.

Impact

Any caller — provided they are not authorized to configure roles — may configure roles in the VaultManager.

An attacker could potentially configure a malicious owner address that registers a new asset that mints USDV arbitrarily.

The following test demonstrates the ability to change any role:

```
function test_Zellic_setRole() public {
    VaultManager vault = fixtureMain.vaultManager;

    // show that real owner cannot change owner to 0x1338
    vm.expectRevert();
    vm.prank(address(this));
    vault.setRole(Role.OWNER, address(0x1338));

    // show that 0xdeadbeef can change owner to 0x1337
    vm.prank(address(0xdeadbeef));
    vault.setRole(Role.OWNER, address(0x1337));
}
```

Recommendations

Invert the `validCaller` condition before reverting:

```
// [...]

if (validCaller) revert Unauthorized();
if (!validCaller) revert Unauthorized();
govInfo.roles[_role] = _addr;
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 01d62e74.

3.2 Ability to cause reversion on destination chain

- **Target**: USDVBase
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

It is possible to use the `send` function to send funds from the source chain to the address `0` on the destination chain. The message will send (i.e., will not revert) on the source side.

On the destination chain, however, when receiving the message — in the `_sendAck` function after calling `_mintBalance`, which calls `_credit` — the execution will revert:

```
function _credit(address _to, uint32 _inboundColor, uint64 _amount)
    internal notBlacklisted(_to) {
    // following OZ's ERC20
    if (_to == address(0)) revert InvalidUser();
    if (_amount == 0) return; // transfer 0 is allowed

    // [...]

    // increment the balance
    state.balance += _amount;

    userStates[_to] = state;
}
```

Impact

As of the assessment version, only support for Endpoint V1 has been implemented. In LayerZero's Endpoint V1, message execution is blocking by default. So, a reversion on the destination chain would mean all future message execution would be blocked.

Regardless of the Endpoint version, a reversion on the destination would break the global delta zero invariant since deltas cannot be finalized on the destination chain.

The following proof of concept demonstrates this behavior:

```
event PayloadStored(uint16 srcChainId, bytes srcAddress,
```

```
    address dstAddress, uint64 nonce, bytes payload, bytes reason);
function test_Zellic_sendAddressZero() public {
    uint32 color1 = fixtureMain.mintColors[0];
    uint64 amount = 100;
    mint(fixtureMain, color1, amount, ALICE);

    //send(fixtureMain.usdv, CHAINID_SIDE_1, ALICE, address(0), amount);
    //assertColorState(fixtureMain.usdv, color1, 0, 0); // no surplus,
    delta = 0

    address usdv = fixtureMain.usdv;
    uint16 toChainId = CHAINID_SIDE_1;
    address sender = ALICE;
    address receiver = address(0);

    bytes memory options = abi.encodePacked(uint16(1), uint(200000)); //
    type1, gasLimit
    IOFT.SendParam memory param = IOFT.SendParam({
        to: MsgCodec.addressToBytes32(receiver),
        dstEid: toChainId,
        amountLD: amount,
        minAmountLD: amount
    });
    (uint nativeFee, uint lzTokenFee) = IOFT(usdv).quoteSendFee(param,
    options, false, "");
    MessagingFee memory msgFee = MessagingFee({nativeFee: nativeFee,
    lzTokenFee: lzTokenFee});

    // expect that it's gonna get stored (because it failed to execute)
    // can't catch InvalidUser() :(
    vm.expectEmit(false, false, false, false);
    emit PayloadStored(0, "0x", address(0), 1, "0x", "0x");
    vm.expectEmit(true, true, true, true);
    emit SendOFT(bytes32(0), sender, amount, "");

    hoax(sender);
    IOFT(usdv).send{value: nativeFee}(param, options, msgFee,
    payable(sender), "");
}
```

Recommendations

Check that the `SendParams`'s `to` address is nonzero in the `send` function:

```
function send(
    SendParam calldata _param,
    bytes calldata _extraOptions,
    MessagingFee calldata _msgFee,
    address payable _refundAddress,
    bytes calldata _composeMsg
) external payable whenNotPaused returns (MessagingReceipt memory
    msgReceipt)
    {
    if (_param.to == address(0)) revert InvalidUser();

    uint64 amount = _param.amountLD.toUint64();
    (uint32 color, uint64 theta) = _send(amount);

    msgReceipt
    = IMessaging(getRole(Role.MESSAGING)).send{value: _msgFee.nativeFee}(
        _param,
        _extraOptions,
        _msgFee,
        _refundAddress,
        _composeMsg,
        color,
        amount,
        theta
    );

    emit SendOFT(msgReceipt.guid, msg.sender, amount, _composeMsg);
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 54f20163.

3.3 Inverted authentication logic in `VaultManager.rotateMinter`

- **Target**: VaultManager
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The VaultManager contract's `rotateMinter` function changes the minter address for a given color. Its logic to determine whether the caller is authorized is inverted:

```
/// @dev unregistered color will have addr as 0x0, don't need to check
    color here
function rotateMinter(uint32 _color, address _newAddr) external {
    if (registry.colorToMinter[_color].addr == msg.sender) revert
        Unauthorized();
    registry.colorToMinter[_color].addr = _newAddr;
}
```

Any caller — except the currently configured minter — can change the minter address.

Impact

An attacker can change the minter address for any color and then withdraw its rewards.

The following test demonstrates the ability to change any color's configured minter address:

```
function test_Zellic_rotateMinterAuthLogicInverted() public {
    uint32 color_1 = fixtureMain.mintColors[0];
    mint(fixtureMain, color_1, 100, ALICE);

    address minter = fixtureMain.vaultManager.minterInfoOf(color_1).addr;

    // let's try rotating it as the minter
    vm.expectRevert(IVaultManager.Unauthorized.selector);
    vm.prank(minter);
    fixtureMain.vaultManager.rotateMinter(color_1, address(0x1337));

    // now show that a random address can
```

```
    vm.prank(address(0xdeadbeef));
    fixtureMain.vaultManager.rotateMinter(color_1, address(0x1337));
}
```

Additionally, an attacker could change the address from `0` and trick the `onlyRegisteredColor` check into allowing an invalid color.

Recommendations

Invert the following condition:

```
/// @dev unregistered color will have addr as 0x0, don't need to check
    color here
function rotateMinter(uint32 _color, address _newAddr) external {
    if (registry.colorToMinter[_color].addr == msg.sender) revert
        Unauthorized();
    if (registry.colorToMinter[_color].addr != msg.sender) revert
        Unauthorized();
    registry.colorToMinter[_color].addr = _newAddr;
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit a4c78050.

3.4 Redemption with positive delta fails to update shares

- **Target**: VaultManager
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

The `redeem` function allows a user to receive collateral tokens in exchange for USDV.

Judging by the documentation, the USDV tokens of the color that were redeemed, and the vault shares of the color, `_self.colorToMinter[_color].shares`, should be burned. However, the function will burn the shares of the color for which the value of `delta.amount` from `used` will be negative.

If `_self.colorStates[_color].delta` of the redeemed color is negative or zero, the `usdv.redeem` function returns an array `used`, which will contain only one element, `Delta(_redeemedColor, amountInt64)`, where `_redeemedColor` is the color of tokens owned by the user and a negative value `amountInt64` for redeem.

But if `_self.colorStates[_color].delta` is positive, the first element of the `used` array will contain `Delta` with zero amount for the color of tokens owned by the user but a negative amount for the colors from the `_deficits` array, for which the shares will be burned instead.

```
function redeem(
    address _token,
    address _receiver,
    uint64 _amount,
    uint32[] calldata _deficits
) external nonReentrant whenNotPaused notZeroAmount(_amount)
    returns (uint amountAfterFee) {
    // burn USDV from msg.sender
    // usdv.burn will burn all surplus then minted
    // only returns negative delta
    Delta[] memory used = usdv.redeem(msg.sender, _amount, _deficits);

    int64 pending = int64(_amount);
    for (uint i = 0; i < used.length; i++) {
        Delta memory delta = used[i];

        if (delta.amount > 0) revert InvalidAmount();
        if (delta.amount < 0) {
```

```
            // burn surplus
            _burnVST(delta.color, uint64(-delta.amount), false);
            pending += delta.amount;
        }
    }
    if (pending != 0) revert InvalidAmount();

    // transfer collateral to receiver
    Asset.Info storage asset = assetInfos[_token];
    amountAfterFee = asset.redeem(govInfo, _receiver, _amount);

    emit Redeem(msg.sender, _amount);
}
```

Impact

This behavior contradicts the documentation.

Recommendations

Burn shares of the same color as the user owns.

Remediation

This issue has been acknowledged, and no fix is needed as it aligns with the intended behavior.

3.5 The `pendingRemint[delta.color]` is mistakenly reduced

- **Target**: VaultManager
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The `pendingRemint` is used by the `remind` function to cache the number of shares that should have been burned or minted, but only in cases where `delta` changes lead to a negative result of the number of vault shares.

The function `clearPendingRemint` allows to reset the `pendingRemint` amount when the number of shares is sufficient. The function accepts the `_delta` array of Delta structures consisting of values `color` and signed `amount`.

For each color from the `_delta` array, the function will check that the `pendingRemint` is not zero, which means it can be cleaned.

The `delta.amount` is how much the `pendingRemint` value should be changed for the corresponding `delta.color`. The `delta.amount` and `pendingRemint[delta.color]` can be a positive or negative amount, but it must be of the same sign.

So if `delta.amount` is positive, the `pendingRemint[delta.color]` will be decreased by `delta.amount` and a corresponding amount of shares will be minted. But if the `delta.amount` is negative, the negative `pendingRemint[delta.color]` amount will be reduced by it, resulting in the addition of the negative numbers.

For example, if `pendingRemint[delta.color] = -100` and `delta.amount = -100`, the result will be -200 instead of 0.

```
function clearPendingRemint(Delta[] calldata _delta)
    external whenNotPaused {
        int64 totalDelta;
        for (uint i = 0; i < _delta.length; i++) {
            Delta calldata delta = _delta[i];

            int64 pendingQuota = pendingRemint[delta.color];
            if (pendingQuota == 0) revert InvalidAmount();
            if (pendingQuota ^ delta.amount < 0) revert WrongSign();

            // converges pendingRemint to zero
            if (delta.amount > 0) {
                // surplus delta more than surplus quota, i.e. revert if
```

```
quota(10) < amount(11)
            if (pendingQuota < delta.amount) revert InvalidAmount();
            _mintVST(delta.color, uint64(delta.amount), false);
            pendingRemint[delta.color] -= delta.amount;
        } else {
            // deficit delta more than deficit quota, i.e. revert if
quota(-10) > amount(-11)
            if (pendingQuota > delta.amount) revert InvalidAmount();
//1. pendingQuota = -100 delta.amount = -100
pendingRemint[delta.color] = -100 + (-100) = -200
            _burnVST(delta.color, uint64(-delta.amount), false); //
2.
            pendingRemint[delta.color] += delta.amount; // -100 +10 =
-110
        }
        totalDelta += delta.amount;
    }
    if (totalDelta != 0) revert NotDeltaZero();
}
```

Impact

The `pendingRemint[delta.color]` amount will be incorrectly reduced for each `clearPendingRemint` call, and if shares of this color have to be saved for minting in the future, they will be lost until this value is increased to 0.

Recommendations

Rather than increasning the stored `pendingRemint`, decrease it to account for the negative delta:

```diff
- pendingRemint[delta.color] += delta.amount
+ pendingRemint[delta.color] -= delta.amount
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit da8a4201.

3.6 Use of blocking LzApp

- **Target**: MessagingV1
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: High
- **Impact**: High

Description

The MessagingV1 contract inherits the LzApp contract to interact with LayerZero to send and receive cross-chain messages.

MessagingV1 overrides the `lzReceive` function from LzApp to add the necessary functionality of handling receiving messages.

```
function lzReceive(
    uint16 _srcChainId,
    bytes calldata _srcAddress,
    uint64 _nonce,
    bytes calldata _message
) public virtual override {
    // lzReceive must be called by the endpoint for security
    require(_msgSender() == address(lzEndpoint), "LzApp: invalid endpoint
    caller");

    bytes memory trustedRemote = trustedRemoteLookup[_srcChainId];
    // if will still block the message pathway from (srcChainId,
    srcAddress). should not receive message from untrusted remote.
    require(
        _srcAddress.length == trustedRemote.length &&
            trustedRemote.length > 0 &&
            keccak256(_srcAddress) == keccak256(trustedRemote),
        "LzApp: invalid source sending contract"
    );

    _handleLzReceive(_srcChainId, _srcAddress, _nonce, _message);
}
```

Impact

Overriding this function provides no additional functionality; that is, MessagingV1 might as well rename `_handleLzReceive` to `_blockingLzReceive` as the security checks of `lzReceive` will still be performed.

More importantly, LzApp is blocking by default. Any reversion on the destination chain will result in the entire path being blocked — at least, until the OApp owner has the opportunity to intervene. This can happen when USDV is sent to a blacklisted address.

Recommendations

To minimize errors and potential security issues, it is advisable for the application to inherit from NonblockingLzApp and override the `_nonblockingLzReceive` function instead of `_blockingLzReceive` or `lzReceive`.

Remediation

The issue with inheriting from LzApp was fixed in commit 03f7e66d.

The issue with overriding the `nonblockingLzReceive` function was fixed in commit a58fe689.

3.7 Arithmetic error in `redeem`

- **Target**: USDVBase
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `_burnBalance` function subtracts from `_totalSupply` before the call to `_debit` has a chance to check the amount for validity:

```
function _burnBalance(address _from, uint64 _targetAmount)
    internal returns (uint32 color) {
    // change balance
    totalSupply_ -= _targetAmount;
    color = _debit(_from, _targetAmount);
    emit Transfer(_from, address(0), _targetAmount);
}

// [...]

function _debit(address _from, uint64 _amount)
    internal notBlacklisted(_from) returns (uint32 color) {
    if (_from == address(0)) revert InvalidUser();
    uint64 balance = userStates[_from].balance;
    if (balance < _amount) revert InsufficientBalance();
    userStates[_from].balance = balance - _amount;
    return userStates[_from].color;
}
```

Impact

Attempting to redeem an amount greater than the supply results in an arithmetic error instead of an `InsufficientBalance()` error.

Recommendations

Subtract from `totalSupply_` after debiting the account so the `balance < _amount` check executes first.

```
function _burnBalance(address _from, uint64 _targetAmount)
    internal returns (uint32 color) {
    // change balance
    totalSupply_ -= _targetAmount;
    color = _debit(_from, _targetAmount);
    totalSupply_ -= _targetAmount;
    emit Transfer(_from, address(0), _targetAmount);
}
```

Remediation

This issue has been acknowledged by LayerZero Labs.

3.8 Bypassable `minterRemintFee` and `operatorRemintFee`

- **Target**: Operator
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Note that the fees round down:

```
function getRemintFees(
    address /*_caller*/,
    Delta[] calldata /*deltas*/,
    uint64 _amount
) external view returns (uint64 minterRemintFee,
    uint64 operatorRemintFee) {
    minterRemintFee = (_amount * minterRemintFeeBps) / 10000;
    operatorRemintFee = (_amount * operatorRemintFeeBps) / 10000;
}
```

Impact

If the `_amount` is low enough, the fee calculation will round down to zero, resulting in no fees being taken when reminting.

In practice, it is unlikely this would be exploited because gas fees can be prohibitively expensive. However, a user may make several smaller transactions (as opposed to one large transaction) to exploit the rounding-down behavior to minimize their fees.

Recommendations

Round up the division and/or consider requiring that both fees are nonzero if the multipliers are also nonzero.

Remediation

This issue has been acknowledged by LayerZero Labs.

3.9 Unimplemented `ping` function

- **Target**: VaultManager
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The following function only executes the code in the `onlyRole` modifier:

```
function ping() external onlyRole(Role.OPERATOR) {
    // ping operation is done in the onlyOperator modifier
}
```

However, the comment seems incorrect; the modifier does not contain any code relating to pinging:

```
modifier onlyRole(Role _role) {
    if (msg.sender != govInfo.roles[_role]) revert Unauthorized();
    _;
}
```

Impact

The operator would not be able to ping without using `setFeeBps`.

Recommendations

Implement this functionality:

```
function ping() external onlyRole(Role.OPERATOR) {
    // ping operation is done in the onlyOperator modifier
    govInfo.ping();
}
```

Remediation

This issue has been acknowledged by LayerZero Labs, and a fix was implemented in commit 3723f8f0.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Ability to `rotateMinter` to 0 address

It is possible to `rotateMinter` to 0, which would disable minting a certain color indefinitely and prevent rewards from being claimable.

This behavior may or may not be intended; however, if it is intended, note that USDV may still be reminted to the color.

4.2 Enforced color is per chain

The `setEnforceColor` function enforces a color on the chain it is called on only.

5 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped USDV contracts, we discovered nine findings. Two critical issues were found. Three were of high impact, one was of low impact, and the remaining findings were informational in nature. LayerZero Labs acknowledged all findings and implemented fixes.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.