PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For USDV

27 October 2023

paladinsec.co

info@paladinsec.co

Table of Contents

Table of Contents 2

Disclaimer 4

1 Overview 5

1.1 Summary 5

1.2 Contracts Assessed 6

1.3 Findings Summary 7

1.3.1 Global Issues 8

1.3.2 VaultManager 8

1.3.3 Asset 9

1.3.4 Vault 9

1.3.5 Governance 9

1.3.6 USDVBase 10

1.3.7 USDVMain 10

1.3.8 USDVSide 10

1.3.9 Colors 10

1.3.10 Operator 11

1.3.11 Messaging 11

1.3.12 MessagingV1 11

1.3.13 MsgCodec 11

2 Findings 12

2.1 Global Issues 12

2.1.1 Issues & Recommendations 13

2.2 vault/VaultManager 20

2.2.1 Privileged Functions 24

2.2.2 Issues & Recommendations 25

2.3 vault/Asset 43

2.3.1 Issues & Recommendations 44

2.4 vault/Vault 46
2.4.1 Issues & Recommendations 46
2.5 vault/Governance 47
2.5.1 Issues & Recommendations 48
2.6 USDV/USDVBase 51
2.6.1 Privileged Functions 53
2.6.2 Issues & Recommendations 54
2.7 USDV/USDVMain 59
2.7.1 Privileged Functions 59
2.7.2 Issues & Recommendations 60
2.8 USDV/USDVSide 61
2.8.1 Privileged Functions 61
2.8.2 Issues & Recommendations 61
2.9 USDV/Colors 62
2.9.1 Issues & Recommendations 63
2.10 USDV/Operator 65
2.10.1 Privileged Functions 65
2.10.2 Issues & Recommendations 66
2.11 USDV/Messaging 67
2.11.1 Privileged Functions 67
2.11.2 Issues & Recommendations 68
2.12 USDV/MessagingV1 69
2.12.1 Privileged Functions 70
2.12.2 Issues & Recommendations 71
2.13 USDV/MsgCodec 74
2.13.1 Issues & Recommendations 75

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for the USDV contracts on the Ethereum, Arbitrum, Optimism, BNB Smart Chain and Avalanche networks. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	USDV
URL	TBC
Platform	Ethereum, Arbitrum, Optimism, BNB Smart Chain, Avalanche
Language	Solidity
Preliminary Contracts	https://github.com/LayerZero-Labs/usdv/tree/9715304f0ce7e4c2156e58b76937d80af4bdb8bd/packages/usdv/evm/contracts/contracts - usdv/ - vault/ Excluded contracts: - usdv/MessagingV2.sol
Resolution	https://github.com/LayerZero-Labs/usdv/tree/79dd57db6efa04b16fbd56276e5beac28ebbd6a1/packages/usdv/evm/contracts/contracts

1.2 Contracts Assessed

Name	Contract	Live Code Match
VaultManager	Proxy (ETH): 0x2A30E3C5c9DaF417663Dd3903144B394a82C999b Implementation (ETH): 0x903d58a8fa472eb671689d79d708841999703c0b	MATCH
Asset	Dependency	MATCH
Vault	Dependency	MATCH
Governance	Dependency	MATCH
USDVBase	Dependency	MATCH
USDVMain	Proxy (ETH): 0x0E573Ce2736Dd9637A0b21058352e1667925C7a8 Implementation (ETH): 0x0f4c265cfda2f0ba07537014687dbe6f22062785#code	MATCH
USDVSide	ARB/OP/BSC/AVAX Proxy: 0x323665443CEf804A3b5206103304BD4872EA4253 ARB/OP/BSC/AVAX Implementation: 0xc298e2a4e05d60e6495c0e8e445def88eaa23bee	MATCH
Colors	Dependency	MATCH
Operator	ETH : 0xE5feD5b0f777F3244D8523F7FC41EF61147cDf4c ARB : 0xE5feD5b0f777F3244D8523F7FC41EF61147cDf4c OP : 0x0fC841AbDa2AcF9c4c531D22A0cF1cF08aF1155e BSC : 0xE5feD5b0f777F3244D8523F7FC41EF61147cDf4c AVAX: 0xE5feD5b0f777F3244D8523F7FC41EF61147cDf4c	MATCH
Messaging	Dependency	MATCH
MessagingV1	ETH : 0x35E8d1DA73e927fA6E9B01892de0cAB468f647dF ARB : 0x35E8d1DA73e927fA6E9B01892de0cAB468f647dF OP : 0xE5feD5b0f777F3244D8523F7FC41EF61147cDf4c BSC : 0x35E8d1DA73e927fA6E9B01892de0cAB468f647dF AVAX: 0x35E8d1DA73e927fA6E9B01892de0cAB468f647dF	MATCH
MsgCodec	Dependency	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
Governance	1	-	1	-
High	2	2	-	-
Medium	8	6	1	1
Low	13	11	1	1
Informational	14	12	2	-
Total	**38**	**31**	**5**	**2**

Classification of Issues

Severity	Description
Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 Global Issues

ID	Severity	Summary	Status
01	GOV	Governance risk: The codebase is fully upgradeable and multiple roles have highly centralized degrees of control over the system	PARTIAL
02	MEDIUM	Risk Management: Contract lacks additional safeguards for "worst-case-scenarios"	✓ RESOLVED
03	INFO	Minter attribution will not be perfectly accurate due to the asynchronous nature of a cross-chain environment	✓ RESOLVED
04	INFO	Allowing users to freely define adapter options for cross-chain communication might be an excessive privilege	✓ RESOLVED

1.3.2 VaultManager

ID	Severity	Summary	Status
05	HIGH	`clearPendingRemint` erroneously reduces the pending remint further for negative deltas which get burned, allowing these to be re-burned	✓ RESOLVED
06	MEDIUM	Read-only reentrancy: `distribute()` contains an incorrect value when it is called during reentrancy within mint and redeem	ACKNOWLEDGED
07	MEDIUM	Minting remains possible even when the minter is marked as paused, even though this should not be possible	✓ RESOLVED
08	LOW	`clearPendingRemint` lacks a `nonReentrant` modifier	✓ RESOLVED
09	LOW	Frontend phishing risk: `_receiver` can be configured for mint and redeem even when the user calls these directly	ACKNOWLEDGED
10	LOW	Many operator interactions do not trigger a liveness ping	✓ RESOLVED
11	LOW	Lack of default value validation on various registration functions can be a configurational hazard	✓ RESOLVED
12	LOW	Adding non-standard tokens as a collateral asset would severely break the codebase	✓ RESOLVED
13	LOW	mint presently does not return whether the recipient's color was successfully re-assigned, making it more difficult than needed for integrations to validate this	✓ RESOLVED
14	LOW	Lack of appropriate caps on the fees	PARTIAL
15	LOW	`OPERATOR` can overwrite the enforced color directly causing a state discrepancy	✓ RESOLVED
16	INFO	Lack of safeCast usage within various sections of the contract	✓ RESOLVED

17	INFO	Token inputs should be explicitly validated to be registered within the contract's functions that interact with tokens	✓ RESOLVED
18	INFO	Typographical issues	✓ RESOLVED

1.3.3 Asset

ID	Severity	Summary	Status
19	LOW	credit does not adhere to checks-effects-interactions	✓ RESOLVED
20	INFO	Typographical issues	✓ RESOLVED

1.3.4 Vault

ID	Severity	Summary	Status
21	MEDIUM	Gas optimizations	PARTIAL

1.3.5 Governance

ID	Severity	Summary	Status
22	MEDIUM	`safeFeeTransfer` can be gas-griefed by an exploiter to avoid paying the redemption fee in certain theoretical instances	✓ RESOLVED
23	LOW	`safeFeeTransfer` should use something like `functionCall` as it will succeed even when calling an EOA	✓ RESOLVED
24	INFO	Typographical issues	✓ RESOLVED

1.3.6 USDVBase

ID	Severity	Summary	Status
25	MEDIUM	An exploiter is able to keep negative deltas for prolonged periods "in-flight", preventing them from being settled to the mainnet	✓ RESOLVED
26	LOW	Lack of denylist and to validation on send increases the likelihood of these tokens to be stuck in transit	✓ RESOLVED
27	LOW	Allowing operators to add colors manually could lead to configurational errors	✓ RESOLVED
28	INFO	Typographical issues	PARTIAL

1.3.7 USDVMain

ID	Severity	Summary	Status
29	MEDIUM	`remintAck` lacks a `whenNotPaused` modifier when the fee is zero	✓ RESOLVED
30	INFO	Colors appear to not be explicitly validated for the deltas	✓ RESOLVED

1.3.8 USDVSide

No issues other than the ones in USDVBase were found.

1.3.9 Colors

ID	Severity	Summary	Status
31	LOW	`NIL` color could accidentally be added if communication malfunctions or an operator adds it	✓ RESOLVED
32	INFO	Typographical issues	✓ RESOLVED

1.3.10 Operator

ID	Severity	Summary	Status
33	INFO	Typographical issues	✓ RESOLVED

1.3.11 Messaging

ID	Severity	Summary	Status
34	INFO	Typographical issues	✓ RESOLVED

1.3.12 MessagingV1

ID	Severity	Summary	Status
35	HIGH	The contract does not support retrying failed non-blocking messages due to incorrectly overriding `_nonblockingLzReceive`	✓ RESOLVED
36	MEDIUM	The contract attempts to support LayerZero token payment support but fails at doing so, bricking the contract if such a token is ever configured	✓ RESOLVED
37	INFO	Typographical issues	PARTIAL

1.3.13 MsgCodec

ID	Severity	Summary	Status
38	INFO	Typographical issues	✓ RESOLVED

2 Findings

2.1 Global Issues

The issues listed in this section apply to the protocol as a whole. Please read through them carefully and take care to apply the fixes across the relevant contracts.

2.1.1 Issues & Recommendations

Issue #01	**Governance risk: The codebase is fully upgradeable and multiple roles have highly centralized degrees of control over the system**
Severity	GOVERNANCE
Description	The USDV system is novel and innovative, thus the developing team determined that there should be a degree of control over it, as new requirements might arise when this goes into production, or limitations might get discovered. We understand and agree with the fact that this is a codebase where being able to address those concerns over time through contract upgrades makes sense. However, for users, this poses a governance risk. If any of the governance keys are ever compromised, all value within the system could be compromised. Furthermore, all approvals to the various contracts could be drained. Here is a list of various roles which are of most relevance users: - **Proxy admins [HIGH RISK]**: The admins of the `VaultManager` and the token contracts are able to adjust the code implementation freely. A malicious admin can drain all value in these contracts and upgrade them to drain approvals as well. - **VaultManager OWNER [HIGH RISK]**: Can set themselves as any of the roles within the `VaultManager`. Can register new fake collateral to drain the vault. - **VaultManager OPERATOR [MEDIUM RISK]**: Can set redemption fee to 100%. - **MessagingV1 owner [HIGH RISK]**: Can pass on the `MESSAGING` role to a malicious contract and can fully configure the LayerZero stack to allow fake messages to be accepted. - **USDV OWNER [HIGH RISK]**: Can change the `MESSAGING` role to do the above as well. - **USDV OPERATOR [MEDIUM RISK]**: Can fully pause all operation on the token. - **USDV FOUNDATION [MEDIUM RISK]**: Can blacklist specific addresses.

Recommendation	Consider only granting limited approvals with the frontend to avoid incentivizing users in having open allowances to the system. Consider locking down the critical roles within the system and documenting what these roles can do and how they can affect users. Strong, reputable and diverse multi-signature wallets should be used.
Resolution	PARTIALLY RESOLVED The client has indicated they will move these role to reputable multi-signature wallets.

Issue #02	Risk Management: Contract lacks additional safeguards for "worst-case-scenarios"
Severity	MEDIUM SEVERITY
Description	The USDV system is presently not robust against failure of either collateral tokens or chains. If a collateral token is hacked, malfunctions or depegs, this will likely drain all value within USDV. As such scenarios are more common than we would like, it is absolutely crucial for a multi-collateral system to add safeguards against this. Specifically, arbitrage will cause the de-pegged token to be deposited to redeem all non-depegged collaterals. This will turn the vault's whole collateral into a fully de-pegged version. If that token subsequently de-pegs to zero, the USDV system is valueless. Furthermore, as USDV gets deployed simultaneously on several chains, we again find ourselves in a "weakest link" scenario as the system trusts any individual chain's consensus fully. If any of these individual chains is compromised and can start creating fake messages, the whole system breaks. It should be noted that writing safeguards against these vectors that cannot be DoS'ed is challenging. We typically recommend adding time-weighted limits that slowly move up and down over time. We would like to finish the description of this issue with a reiteration of how strong and important sensible safeguards can be. Just a handful of safeguard lines around the TVL can literally reduce the impact of any unknown exploit vector to just 10% of that TVL. This is insanely valuable and we believe it should be considered in complex systems like this. Of course, the client needs to keep in mind that the impact is not limited to the TVL: If USDV over mints on any of the chains, this would cause all paired liquidity to get drained which could be significant value as well. This is unfortunately less easy to safeguard with a handful of lines (except for mint caps on all chains but this is difficult as it can block messaging).

Recommendation	Consider first and foremost safeguarding the collateral. This is the core value of the system and remaining (partially) collateralized in black-swan events is what we would heavily recommend as a primary design goal. This means that adding "simple" (non-DoS-able) safeguards that prevent the `VaultManager`'s collateral from being drained on short notice are probably ideal. One such an example could be a time-weighted TVL limit — e.g., TVL of the collateral must be within an upper and lower limit, and these limits shift up and down with the actual TVL as time passes (eg. at most 10% per day). Such time weighted caps would significantly reduce the maximum impact of any given exploit, and pretty much guarantee that at least some portion of the value within the system can be recovered after an exploit. Remember that exploiters can also specifically abuse the fact that caps exist by strategically causing the caps to revert at times that suit them. Specifically, frequency limits tend to be bad as exploiters can loop to trigger them (e.g. total number of redemptions). For cross-chain risks, it is less trivial to add sensible safeguards. Adding "routes" with limited "throughput" between all of the chains may make some sense but this is relatively difficult to implement within the current design and may make it bloated. Especially since transfers are free, this is a challenge. The easiest solution here seems to address this off-chain via LayerZero's innovative Precrime solution. We strongly recommend to add safeguards if the token gets deployed on many chains.

Resolution RESOLVED

The most important risk, token dependency, has been addressed to a degree which is acceptable to the client (though this degree is obviously subjective and might be insufficient in practice, depending on the circumstances).

Rate limits have been added for minting and redeeming with the vault. This means that the underlying collateral, given that all else works, can only be stolen at the given rate limits their rates, and this means that impact is limited to these rates for the collateral being drained through any given exploit.

The client also indicated that they have built precrime tools to mitigate the cross-chain risk to a large extent, but Paladin is not able to validate these due to the off-chain nature of them. Users should of course be careful as precrime is extremely novel at this point.

We remind the users that rate-limits only manage risk for the underlying collateral. In an exploit to the actual USDV token (and not the collateral), tokens paired with USDV and so forth might be at risk still. Lending protocols using USDV might be at risk still. This list is of course not limited by the examples mentioned above.

UPDATE: During deployment, a change was made to these rate limits that caused a bug if they are set from 0 to a positive number. In this case, the limits will instantly refill fully. We have pointed this out during the live match and the client is aware of this — they will update this the next time they need to redeploy the operator and rate limiter contracts.

Note that this sending rate limit which is introduced also is not extremely robust as it is enforced on the sending chain and not the receipt chain. If a sending chain is compromised, this limit can of course also be circumvented.

Issue #03	**Minter attribution will not be perfectly accurate due to the asynchronous nature of a cross-chain environment**
Severity	INFORMATIONAL
Description	`VaultManager` on Ethereum defines the central ledger where minters are attributed their portion of the collateral rebase rewards. As recolors happen on the sidechains through transfers for example, this ledger needs to be updated accordingly. However, due to the asynchronous nature of these updates within the system, these updates might only get updated within the central ledger hours later, causing a tracking error for the yield distribution. Though this is a fundamental and accepted property of the system, we thought it worthwhile to explicitly document this for the users.
Recommendation	This is a fundamental feature of the system and therefore cannot be resolved in the current design. One potential solution in a different design is to track average color supplies on all the sidechains and periodically (e.g. daily) sync these to mainnet, then those time-weighted average supplies are used to distribute the yield of that day. It should be noted that in flight supplies would not be trackable in this mechanism, though this is an acceptable trade-off and is not necessarily considered "inaccurate". It should be noted that this proposed "solution" has a different sort of tracking error, specifically when yield is very volatile over time.
Resolution	RESOLVED This is by design. This issue is marked as resolved as the client has indicated they have carefully simulated the maximum impact of this. No changes were made.

Issue #04	**Allowing users to freely define adapter options for cross-chain communication might be an excessive privilege**
Severity	INFORMATIONAL
Description	All functions that involve cross-chain communication allow for the user to freely define the adapter parameters. This appears to be by design within the LayerZero system (e.g. the user should be able to freely set these). However, this does mean that if a new adapter parameter can ever get misinterpreted by the off-chain components to cause the message to not be automatically delivered, an exploiter can abuse this to keep negative deltas in transit for longer. Furthermore, if any of the future adapter parameters can do anything unknown which might not be desired to be configurable by the user, this could pose a risk as well.
Recommendation	Consider this carefully — if it is possible to only allow for the specific parameters that are important to be provided, this might be safer.
Resolution	RESOLVED The client has indicated this is by design and has re-assured us that they will carefully validate any adapter parameters evolutions.

2.2 vault/VaultManager

`VaultManager` represents the primary contract for the USDV system. It allows users to mint USDV by supplying RWAs, which are subsequently stored within the `VaultManager`. Additionally, it acts as the primary source of truth to keep track of all minters and which portion of the supply is attributed to them. This is relevant since all rebasing rewards of the collateralized tokens (the RWA yield) is attributed pro-rata to these minter addresses according to the portion of the supply they represent. These minters will be referenced as "colors" throughout the codebase, where "color" is simply the identifier given to an individual minter. It should be noted that a color's minter address can be passed on over time, to allow for key updates.

From the last paragraph, it can be gathered that "the correct attribution of the RWA collateral yield to the minters" is the main requirement and design goal for the USDV token. This is the problem it aims to solve. It does so by keeping track of the total supply allocated to any individual minter/color within the `VaultManager`. Whenever RWA rewards accrue, these rewards are allocated on a pro-rata basis to these minters according to these supply distributions.

Although this is quite simple by itself, the system becomes more complex once we realize that these supplies are transferred between users, and if a user transfers tokens to a recipient which is actually related to a different minter, that supply should probably be relabeled (called "recolored" in the rest of the report) to the recipient's color. The system therefore adds a set of recoloring rules for transfers which will be further described in the `USDVBase` portion of this audit. The system becomes even more challenging when we consider a multi-chain context. As tokens get recolored on side-chains due to transfers (see the previous phrase), these recolorings must be propagated to the `VaultManager`, otherwise the `VaultManager` will continue to incorrectly attribute RWA yield to outdated minters/colors.

This propagation logic, which forwards the minter supply differentials (called "deltas" in the rest of the report) to the `VaultManager` is the primary source of complexity within the USDV system. Deltas can be sent from chain to chain using LayerZero, but will eventually be sent to Ethereum, where the `VaultManager` is deployed. These deltas can then be synced with the `VaultManager`, which essentially just means that it will update its records of which minter/color owns which portion of the supply.

It should therefore be very clear that at any point in time, the "source of truth", which is the `VaultManager`'s record keeping, is very likely to be outdated. This means that the reward distribution for RWA tokens will almost never exactly be correct. However, the system defines a propagation logic with the goal of converging to correct reward distribution over time.

That wraps up the high level overview of the `USDV` and `VaultManager` system. Below we will further explain the specifics of this contract:

The `OWNER`, which is going to be a multi-signature wallet, is responsible for registering collateral tokens using `registerAsset`. The vault exclusively supports positively rebasing tokens where 1 nominal token is always worth exactly $1. It does not support tokens which have a fee on transfer, negative rebase potential, or a likelihood of de-peg. The last bit means that the system collateralizes the USDV token exactly 1:1. If any of the collateral de-pegs, this will have a direct impact on the solvency of the system. Though this is a design choice, we will recommend several safeguards throughout this report to limit the impact of such an event. However, the client has also indicated to us that collateral will be extremely strictly whitelisted and not many tokens will be part of this collateral.

Tokens can be minted by anyone by calling the `mint` function. To do so, users provide a number of collateral tokens and the receiver receives an identical number of USDV tokens in return. During minting, the caller defines the desired "color" of these tokens (read: the desired minter these tokens should be attributed to). This color will be assigned to the tokens in case the recoloring rules of the receiver

permit it, otherwise these tokens are colored according to the current color of the receiver. More about these rules can be read within the USDVBase portion of the audit.

USDV can be redeemed back to the underlying collateral by calling the `redeem` function on the vault. In this case, a set of deficit colors can be provided. Accumulated negative deltas within the USDV token can be consumed directly this way, and are offset by a potential positive delta of the user's color, which was just actually burned. It should be noted that this deficit logic is effectively neutral in the actual supply of these colors: it burns a set of colors, but the pending negative delta of these colors is reduced proportionally. The redeemed amount might not be fully burned in the user's assigned color, but any unburned portion is taken from the pending positive delta of that color instead, which would've been minted at a later point.

A redemption fee initially configured at 0.1 applies. This redemption fee can be reconfigured to between 0.1% and 100% of the redeemed amount by the `OPERATOR`. Users should be careful to demand that all roles (operator, owner, foundation, proxy) are carefully safeguarded to avoid that this fee is set to an excessive value.

Anyone can trigger the distribution of accumulated rebases to the minters by calling `distributeReward` on the vault. This iterates over a provided set of rebase tokens and records the number of tokens that have been added to the vault since the last call (excluding tokens added via `mint` or removed via `redeem`). These rewards can be withdrawn by the configured minter, or the `OPERATOR` who is capable of withdrawing them by first marking the minter as paused, or forcing their address to rotate to a new one. An `OWNER`, `LIQUIDITY_PROVIDER` and `OPERATOR` fee are deducted from the rebase rewards, before they are granted to the respective minters. The `LIQUIDITY_PROVIDER` fee is initially set to 20% and the `OPERATOR` fee is initially set to 30%. The `OWNER` fee is initially set to 1% and can be set to at most 3%. The other fees can typically be adjusted freely, though the codebase incorporates a safeguard where a different role configures the limit of this fee.

An initial single asset is configured as collateral: STBT.

The OWNER is capable of pausing the contract, which effectively pauses all non-privileged interactions on the vault.

The OPERATOR can be re-assigned by the FOUNDATION role if said operator does not make a transaction with the vault manager for over 30 days. It should be noted that adjusting the OPERATOR address is currently seen as an interaction, meaning that the 30 day cooldown period sets in at that time. This is a design choice but does mean that the OPERATOR needs to wait the whole cooldown if they accidentally misconfigured the address. Since the OWNER can re-assign any role, they can of course still reset the OPERATOR to bypass this mistake.

The contract, alongside other key contracts, is upgradeable. This means that the proxy admin is capable of freely adjusting the logic of this contract. If this admin is compromised, or becomes malicious, this means that the contract can be upgraded to a malicious version to not only drain all collateral, but also all open approvals by users. We highly recommend the team and users to be diligent with validating the quality of this proxy admin. At the very least, this should be a reputable multi-signature set-up.

UPDATE: After the audit, a DONOR role was introduced. Mints from this role do not mint actual USDV tokens, though the accounting values are still adjusted. This means that those donated tokens are essentially stuck in the vault without being eligible to be distributed as yield or withdrawn. As this is rather odd behavior, we recommend that the client be careful with this post-audit feature.

2.2.1 Privileged Functions

- setPaused [OWNER]
- setRateLimiter [OWNER, introduced as a resolution]
- registerAsset [OWNER]
- setAssetEnabled [OWNER]
- setRole [OWNER or current role account, FOUNDATION can also assign OPERATOR if no operator tx occurred for 30 days]
- setFeeBpsCap [OWNER can set OPERATOR/LIQUIDITY_PROVIDER fee cap, FOUNDATION can set its own fee cap]
- setFeeBps [OPERATOR can set FOUNDATION and their own fee, LIQUIDITY_PROVIDER and OWNER can set their own fee]
- withdrawFees [OPERATOR can withdraw FOUNDATION and their own fee, LIQUIDITY_PROVIDER and OWNER can withdraw their own fee]
- registerMinter [OPERATOR]
- setUSDVVaultColor [OPERATOR]
- setColorPaused [OPERATOR]
- ping [OPERATOR]
- rotateMinter [OPERATOR]
- withdrawReward [minter while minter not paused, otherwise OPERATOR]

2.2.2 Issues & Recommendations

Issue #05	**`clearPendingRemint` erroneously reduces the pending remint further for negative deltas which get burned, allowing these to be re-burned**
Severity	HIGH SEVERITY
Description	`clearPendingRemint` allows pending remint deltas to be offset against each other and subsequently burned/minted to the actual color's supply. An example could be: `pendingRemint = {"blue": 10, "green": -5}`. In this example, `clearPendingRemint` would allow up to 5 blue supplies to be minted as long as an equal number of green supplies are burned. The `pendingRemint` values would subsequently be updated to for example `{"blue": 5, "green": 0}`. However, the codebase contains a critical error within the remint reduction (increase towards 0) for the negative remints: Line 325 `pendingRemint[delta.color] += delta.amount;` Since `delta.amount` is negative for these values, `pendingRemint` is in fact reduced for these values. This means that in the example, the final `pendingRemint` would equal `{"blue": 5, "green": -10}`. This would break essential properties of the system and effectively break the whole system due to the fundamental accounting being broken. Over time, all pending remints would become negative and positive deltas would no longer be incorporated in the supply, breaking the contract
Recommendation	Consider subtracting `delta.amount` instead as this value is negative and we want to move the `pendingRemint` towards zero within this function.
Resolution	RESOLVED The recommended change has been introduced.

Issue #06	**Read-only reentrancy: `distribute()` contains an incorrect value when it is called during reentrancy within `mint` and `redeem`**
Severity	MEDIUM SEVERITY
Description	The functions interacting with collateral tokens do not fully adhere to checks-effects-interactions. This is also impossible as the `distributable()` function uses the balance of the token directly, making it nearly impossible to organize the code in a fashion where this function always returns the exact correct value on any reentrancy. This is because the function which actually updates this value, the token transfers, could call a reentrancy before or after the actual balance update. This issue has been marked as medium severity as read-only reentrancies can be detrimental to derivative systems which expand upon this contract as they might call functions such as `distributable` and assume their value is correct. An exploiter is then able to call these via a reentrancy and cause the derivative application to incorrectly receive a different number of tokens that are distributable, potentially exploiting this application. This has famously occurred with multiple apps building on top of Curve in the past, which has a read-only-reentrancy vulnerability as well.
Recommendation	Consider enforcing a `nonReentrant` modifier with any `balanceOf` interaction as these balances are not correct in the intermediary reentrant states.
Resolution	ACKNOWLEDGED

Issue #07	**Minting remains possible even when the minter is marked as paused, even though this should not be possible**
Severity	MEDIUM SEVERITY
Location	Line 197 `// @dev set paused to true to disable minter from minting`
Description	The contract defines logic allowing the `OWNER` of the contract to disable specific minters from being allowed to be minted from. Once disabled, they should no longer be usable. However, presently this check is improperly implemented allowing for mint still be called. It should be noted that this issue is very difficult to resolve given that re-colorings will always remain possible on side-chains, effectively allowing minting to a paused minter.
Recommendation	Consider adding a requirement within the `_mint` function if desired. Consider documenting that this check is not effective against preventing minting with that minter color.
Resolution	RESOLVED The `mint` function is now prevented from being called within the `VaultManager` for that color.

Issue #08	`clearPendingRemint` lacks a `nonReentrant` modifier
Severity	LOW SEVERITY
Description	The contract has been extremely consistent with adding `nonReentrant` modifiers to any external function which can be called by regular users. However, this modifier was left out with `clearPendingRemint`, which reduces the overall security guarantees that the codebase can make with regards to reentrancy prevention. This issue is only marked as low severity as we could not find any way of exploiting a reentrancy by calling this function from any of the interactions within the contract (`mint`/`redeem`). However, this is by no means a reason not to guard this function as it is extremely tedious to make reentrancy security guarantees over a codebase that does not adhere to checks-effects-interactions. We strongly urge the client to add this guard as well.
Recommendation	Consider enforcing a `nonReentrant` modifier with any `balanceOf` interaction as these balances are not correct in the intermediary reentrant states.
Resolution	RESOLVED

Issue #09	**Frontend phishing risk: `_receiver` can be configured for `mint` and `redeem` even when the user calls these directly**
Severity	LOW SEVERITY
Description	The codebase supports minting and redeeming such that the resulting tokens are sent to a provided `_receiver` address that is different from the sender. This is very useful as the vault might be used by intermediary contracts like zappers, which can then directly transfer the assets to the user with these functions. However, these parameters are not so useful for users, and even worse, users might be misled into signing a transaction where they accidentally set the receiver as a different wallet than their own as many wallets do not decode these signatures properly for the user.

Recommendation	Consider adding a requirement that the receiver must be equal to `msg.sender` if `msg.sender` is the same as `tx.origin`. This effectively bans EOAs from minting and redeeming to a different wallet:
	`if (msg.sender == tx.origin && _receiver != msg.sender)`
	`    revert OnlyContract();`

Resolution	ACKNOWLEDGED

Issue #10	Many operator interactions do not trigger a liveness ping
Severity	LOW SEVERITY

Location	Line 48-51
	`modifier onlyRole(Role _role) {`
	`    if (msg.sender != govInfo.roles[_role]) revert Unauthorized();`
	`    _;`
	`}`

Description	The codebase allows the `FOUNDATION` to re-assign the `OPERATOR` if an `OPERATOR` has not made any transactions for more than 30 days. This is done by updating a ping timestamp whenever the operator interacts. However, in many interactions, this ping does not currently occur.
Recommendation	Consider adding an `if` statement to the `onlyRole` function which issues a ping whenever the role is equal to the `OPERATOR` role.
Resolution	RESOLVED

Issue #11	Lack of default value validation on various registration functions can be a configurational hazard
Severity	LOW SEVERITY
Description	The contract facilitates the registration of assets and minters. When registering an asset, a collateral token address is provided. When configuring a minter, the minter address is provided. Throughout the codebase, however, the `address(0)` value is seen as a special address as it represents the unset token and minter. Because this value is special, it should not be allowed to be configured. `rotateMinter` should also check that the new address is not zero and more importantly it must validate that the minter is already configured, as it can rotate a non-existent color at the moment! Several `view` functions do not revert for non-existent values. Consider whether it makes sense to more explicitly handle these cases as they may cause issues for integrations who incorrectly assume that non-existent values do exist. We are fine with this being unchanged as long as such integrations are careful. Finally, it might make sense to explicitly prevent certain colors such as 0 and THETA from being configured with `setUSDVVaultColor`. However, these colors are typically cannot be set within the current implementation of the USDV token so this might be considered less relevant.
Recommendation	Consider explicitly preventing the default value in all places of the codebase. Any "special" values should be explicitly prevented throughout the codebase. Consider only allowing `rotateMinter` to be called for registered colors and with a non-zero new address. Having special meanings for specific values is considered bad practice. Consider also refactoring the code to have a `bool` registered value within the relevant minter and asset `struct`. Since there is still room within the storage of the existing slots of these structs, this hardly comes at a gas cost penalty and improves the explicitness of the codebase.
Resolution	RESOLVED The client has taken steps to validate such values more explicitly.

Issue #12	**Adding non-standard tokens as a collateral asset would severely break the codebase**
Severity	LOW SEVERITY
Description	The contract does not support various non-standard tokens, and is only designed to work with rebasing stablecoins that are strictly pegged to $1. The following ERC20 types are explicitly not supported within the current design: - Fee-on-transfer: This breaks deposits as the `mint` function does not actually record how much tokens were received. - Negative rebase tokens: This causes an underflow exception in the distribution calculation, causing it to revert. - Reentrancy tokens: Though theoretically supported, the codebase does not fully adhere to checks-effects-interactions which has resulted in us finding at least one read-only reentrancy if a reentrancy token is added. Even if that vulnerability is patched, we strongly recommend against adding reentrancy tokens as it is easy to miss reentrancy exploits in a codebase which does not fully adhere to CEI. It should be noted that since the rebasing logic uses `balanceOf`, writing this codebase in CEI is not easily possible. - Tokens which can de-peg: If a collateral token becomes worth less than $1, users can arbitrage by mass depositing the token and redeeming it for other tokens. There is no automated liquidation or backstop system at this point. Any token with any form of de-peg risk should therefore be avoided, alongside the addition of safeguards which we recommend. - Tokens which return malicious balances (e.g. through a proxy compromised upgrade): This would cause an excessive upgrade. We will make recommendations to limit this in a separate issue. - Tokens that can be frozen: If the underlying collateral is frozen by its issuer, this could cause an issue for the USDV token as users would not be able to redeem these collateral tokens again.

	- USDV: Over time, the client may decide to add non-rebasing stablecoins as collateral, and manually supply yield for them by transferring it to the vault. One such special case could be when usdv is added as a recursive collateral. Though this makes very little sense in our mind, it would be detrimental for the vault contract which would effectively break as it holds and transfers usdv for its fee and distribution logic.
Recommendation	Consider explicitly disallowing the USDV address to be registered as a collateral token to explicitly prevent this case. Consider internally documenting that all above token types are not permitted as collateral. Consider adding safeguards as recommended in a separate issue to limit the impact of depeg and token compromise.
Resolution	RESOLVED - Fee on transfer tokens: Not planned to be supported - Reentrancy tokens: Client has taken steps to make the codebase mostly secure against it but will consult with auditors again before adding such tokens - De-peg risk: Client has added configurable mint/redeem rate limits on the vault. - Malicious balances: The above mint limit also applies to yield distribution - Tokens with a risk of being frozen: Not explicitly dealt with but the client indicated they will carefully vet this. - USDV: Not planned to be supported

Issue #13	**`mint` presently does not return whether the recipient's color was successfully re-assigned, making it more difficult than needed for integrations to validate this**
Severity	LOW SEVERITY
Description	As a recipient's color is not always overridden when tokens are sent to a recipient, there may be cases where the `_color` parameter within mint is effectively ignored, and where the `_receiver` just retains the `color` they already have. In this case, certain integration contracts might be reluctant to make such a mint succeed. It is difficult for these integrations to validate that the `_receiver` in fact did either recolor or was already in the correct color. This must be done in a subsequent call. It may make more sense to return this state as a boolean value.
Recommendation	Consider whether it makes sense to return a boolean within the mint function to indicate whether the `_receiver` is now colored appropriately in the provided `_color`.
Resolution	RESOLVED The client has indicated they do not need this.

Issue #14	Lack of appropriate caps on the fees
Severity	LOW SEVERITY
Description	Most of the fees can be set disproportionately. For example, the redemption fee could be set to 100%, resulting in users receiving exactly nothing after they redeem their tokens. Since the contract is upgradeable, this does not really reduce the security for users as they already need to trust the teams managing the contracts, but more realistic caps might make sense. Furthermore, the `LP + OPERATOR` fees can sum to a value greater than 100%. This does not appear to be a problem as it is a fee on top but does not make much economical sense given that there is no incentive to remint at that point. Finally, when the caps are in fact set, the fees are not accordingly reduced if they are greater than the caps.
Recommendation	Consider whether it makes sense to cap the absolute redemption value to a more realistic value. Consider at the very least capping the sum of the two aforementioned fees to 100% whenever these are configured (the caps can remain at 100%), as otherwise underflow would occur.
Resolution	PARTIALLY RESOLVED Redemption can still be set arbitrarily high but the value of the two fees now must sum to at most 100%.

Issue #15	**OPERATOR can overwrite the enforced color directly causing a state discrepancy**
Severity	LOW SEVERITY
Location	Line 342 `_mint(token, address(this), rewardInUSDV.toUint64(), usdvVaultColor, 0x0, false);`
Description	The distribution logic assumes that the enforced color of the vault was the one configured via `setUSDVVaultColor`. However, this color may have been overwritten directly within USDV by the operator, as the operator can re-assign the recolorer of any address. Consider whether it makes sense to instead get the vault's enforced color directly at the beginning of `distributeReward` and removing `usdvVaultColor` all together.
Recommendation	Consider adding a `bool` registered to the collateral struct and consistently checking it for any collateral interaction including the functions mentioned above.
Resolution	RESOLVED The client has made the operator setting the only way of configuring the vault color and fetches the color directly from the `USDV` contract.

Issue #16	Lack of `safeCast` usage within various sections of the contract
Severity	INFORMATIONAL

Location	Lines 283, 442, 451
	`int64 pending = int64(_amount);`
	`pendingRemint[_color] -= int64(_amount);`
	`pendingRemint[_color] += int64(_amount);`

Description	Throughout the codebase, a defensive approach is used and amounts are converted with `SafeCast` and math is done with checked operations. However, this is forgotten within the above locations. Although we do not believe these locations can be easily exploited to overflow, we strongly believe that it makes sense for this codebase to guarantee this through using `SafeCast` as this turns a "belief" into a "guarantee". The little amount of gas saved does not stack up against security guarantees in our view.
Recommendation	Consider using `SafeCast` for these locations, or at least explicitly documenting the requirements for these lines not to overflow and why these are never breached.
Resolution	RESOLVED

Issue #17	**Token inputs should be explicitly validated to be registered within the contract's functions that interact with tokens**
Severity	INFORMATIONAL
Description	There are various functions that interact with the collateral assets such as `mint`, `redeem`, `distributeReward`, `distributable`, `redeemOut` and `setAssetEnabled`. Many of these functions will break if a non-registered token is provided which is desired. However, with many of these, this breakage is implicit in, for example, an underlying function reverting due to the `calldata` being non-decodable for the zero address. This is less than ideal as it causes essential behavior to be safeguarded with implicit "lucky" checks. If this contract is upgraded over time with new features, such "lucky" checks might get refactored and omitted, causing these functions to suddenly become callable with tokens which were never registered.
Recommendation	Consider adding a `bool` registered to the collateral struct and consistently checking it for any collateral interaction including the functions mentioned above.
Resolution	RESOLVED The non-view functions are now checked with an explicit registration boolean.

Issue #18	Typographical issues
Severity	INFORMATIONAL

Description	Line 14
	`import "../mocks/STBT.sol";`
	
	This import appears unused.
	
	Line 28
	`uint8 internal constant usdvDecimals = 6;`
	
	Constants should be written in all caps.
	
	Line 30
	`IUSDVMain internal usdv;`
	
	This variable can be marked as `public` to allow for users to inspect it from within the browser.
	
	Line 36
	`mapping(Role role => uint64 amount) roleFees;`
	
	This mapping can be marked as `public` to allow for users to inspect it from within the browser.
	
	Line 41
	`address[] public assets;`
	
	A `getAssetsLength()` function or similar should be exposed to allow for querying this array effectively.
	
	Line 59
	`_safeGetMinterInfo(_color);`
	
	This check is rather implicit, and we do not see the advantage of this compared to actively checking the specific variable with a requirement. It appears like this code is used because a side-effect of it is that it reverts in case the color does not exist. Cleaner code would use the exact code that only does the reversion logic.

Line 64

```
address _usdv,
```

This variable can be directly provided as `IUSDVMain` to avoid casting it later on.

Line 140

```
govInfo.roles[_role] = _addr;
```

It may make sense to validate that `_role` is one of the relevant roles as the `VAULT/MESSAGING` roles are not used within this contract. This applies to other sections of the contract as well where these roles would not create a reversion. Leaving this unchanged is fine with us however.

Line 171 and 379

```
if (fees > 0) {
if (reward > 0) {
```

Inverting these to a reversion makes the state-space smaller and therefore more theoretically secure.

Line 178 and 380

```
usdv.transfer(_receiver, fees);
usdv.transfer(_receiver, reward);
```

The return value is not checked for this transfer. Although this is currently fine as `usdv` never returns `false`, this could become an issue if the `usdv` implementation is ever upgraded, which is a real possibility. Consider using `safeTransfer` instead.

Line 180

```
emit WithdrewFees(msg.sender, fees);
```

The _receiver should probably be included in this event.

Line 223

```
function remint(Delta[] calldata _deltas, uint64 _remintFee)
external nonReentrant whenNotPaused {
```

This array unnecessarily encodes multiple values. Consider splitting it up into `Delta calldata surplus`, `Delta[] calldata deficits`.

Line 240

```
uint64 remintFee = i == lastBurntIdx
```

It might suffice to simply check that the delta is the last element in the array, as an earlier check validates that all deltas are non-zero, which we believe means that the last one should simply receive the remainder.

Line 271

```
/// @dev if deficit colors are unknown, it will fail in
usdv.redeem, don't need to validate registered color here
```

This does not appear to explicitly fail and just ignores the color instead. Explicit validation would make sense given that it reduces the state spaces and therefore gives exploiters less freedom

Line 358

```
amounts[i] = rewardInUSDV;
```

This amount is emitted in a later event. However, the amount in question is neither the gross nor the net fee amount. Instead, it represents an intermediary calculation where a part of the fees are already subtracted, but the `LIQUIDITY_PROVIDER` and `OPERATOR` fee have not yet. We highlight this as we believe the client might have preferred to emit the net value within that event.

—

Various hardcoded integers occur within the initializer and distributeReward function. It may make sense to label these constants and re-use the constants within the governance contract.

	The `stbt` definition within the initializer should probably re-use an internal `_registerAsset` function instead of repeating code. — It should be noted that `OWNER` and `FOUNDATION` can call `ping()` on behalf of the operator using the same role and address. Exception logic might not be worth adding as this does not cause harm. — A sensible additional check in the `remint` / `validateDeltas` logic is that the deficit deltas should be unique (e.g. ordered). This is already guaranteed in other components of the system but if cross-chain communication breaks this property can of course break as well theoretically. Generally, validating more is not often bad. — A check should be in place to prevent a final color from being added, as that index is already identified as "THETA". — `registerMinter`, `mint` and `redeemOut` can be marked as `external`. — `registerAsset`, `setAssetEnabled`, `setRole`, `setFeebBpsCap`, `setFeeBps`, `registerMinter`, `setColorPaused`, `ping` and `rotateMinter` should emit an event.
Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED Many of these issues have been resolved; the client did not address some of them due to gas and contract side concerns.

2.3 vault/Asset

`Asset` is a library used by the `VaultManager` to store configurations and accounting for the collateral assets. For each asset, the ERC20 token can be configured as well as whether that asset is enabled. The library then keeps track of the number of assets deposited into the `VaultManager`.

The `Asset` library also contains the logic to pull the token from the user's wallet for deposits and send it to the recipient for withdrawals. It should be noted that this logic does not support tokens with a fee on transfer and also does not adhere to checks-effects-interactions (an anti-reentrancy pattern), making the reentrancy guards in the `VaultManager` vital.

The `Asset` library finally defines the calculation which calculates the number of tokens eligible for distribution, which is essentially the current balance in the `VaultManager` with the deposited tokens subtracted. We reiterate that the `VaultManager` cannot support negative rebases as this subtraction would underflow in that case.

2.3.1 Issues & Recommendations

Issue #19	`credit` does not adhere to checks-effects-interactions
Severity	LOW SEVERITY
Description	The credit function does not adhere to checks-effects-interactions, which makes certain sections of the codebase vulnerable to reentrancy. Most notably, the `distributable()` function within `VaultManager` is vulnerable to a read-only reentrancy. Other functions such as the actual distribute rewards functions do not appear to be vulnerable due to the reentrancy guards within the `VaultManager`, which is good.
Recommendation	Given that `balanceOf` is used for `distributable()` calculations, it is not easy to rewrite the Asset library to adhere to checks-effects-interactions. We recommend being extremely careful with reentrancy attacks throughout the codebase and triple-checking that everything is guarded behind reentrancy guards (such as the recommended extra functions to guard, see the `VaultManager` issues).
Resolution	RESOLVED `credit` adheres to checks-effects-interactions now to the extent where it is possible (given that `balanceOf` is still present in the contract).

Issue #20	Typographical issues
Severity	INFORMATIONAL

Description	Line 21
	`error TokenDecimalInvalid(uint provided, uint max);`
	
	This second parameter should likely be named "min" instead.
	
	Line 24
	`function initialize(Info storage _self, address _token, uint8 _shareDecimals) internal {`
	
	`_token` can be provided as `IERC20Metadata` to avoid casting it later on.
	
	Line 29
	`// set token2shareRate`
	
	This comment is outdated as the variable is now called `usdvToTokenRate`.

Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED

2.4 vault/Vault

`Vault` is a library that defines the logic for the rebase reward assignment to the individual minters/colors. It is considered the central accounting unit for the USDV token as a multi-chain token. Similar to the `Colors` library for the USDV tokens, this library stores the supply for each `color`. Alongside this supply, reward assignment variables such as the reward debt and the pending rewards are stored as well.

2.4.1 Issues & Recommendations

Issue #21	Gas optimizations
Severity	INFORMATIONAL
Description	`accumRewardPerShare` could fit within the same slot as `totalShares` as it does not appear to be possible to exceed the remaining bytes for this variable. This would save a storage slot for the Vault info. Within `addShares`, the storage value for the Info's `totalShares` can be cached to save gas. It is re-used in the `Overflow` check at this moment.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	PARTIALLY RESOLVED The first recommendation has been implemented.

2.5 vault/Governance

`Governance` is a library used within the `VaultManager` to define the `govInfo` data which includes the addresses which are assigned to roles like `OWNER`, `OPERATOR` and `FOUNDATION` and their related fee data (fee and cap). It also defines the logic for the operator liveliness ping and the logic to calculate and pay the redemption fee.

2.5.1 Issues & Recommendations

Issue #22	**`safeFeeTransfer` can be gas-griefed by an exploiter to avoid paying the redemption fee in certain theoretical instances**
Severity	MEDIUM SEVERITY
Description	`safeFeeTransfer` succeeds even if the fee transfer fails. It is theoretically possible in certain cases that the fee transfer fails due to an "out of gas" error while the subsequent code still succeeds with the remaining portion of the gas. This is called a "gas-griefing" and is most notably possible when the griefable code call consumes a lot of gas (as this increases the gas remaining for the final code). As this call occurs on an arbitrary token, this could therefore occur in practice and for certain tokens which waste a lot of gas on this call compared to the subsequent calls, could allow for malicious redeemers to bypass the fee.
Recommendation	Consider refactoring this code to either use a sensible minimum amount of gas, to keep these tokens in the vault instead as USDV or to keep these tokens in the vault as a special accounting value. The first recommendation feels the least intrusive while the second one might be most desirable from a perfectionist perspective, though it is much more intrusive.
Resolution	RESOLVED The `try` logic has been fully removed in favor of `safeTransfer`.

Issue #23	**`safeFeeTransfer` should use something like `functionCall` as it will succeed even when calling an EOA**
Severity	LOW SEVERITY
Description	`safeFeeTransfer` will emit a success event even when it is called to an EOA address. We instead recommend using something like `functionCall` within OpenZeppelin's `Address.sol` library which checks that the bytecode is greater than zero. It should be noted that Solidity always does this check as well when making high-level calls, hence we recommend being consistent here as well.
Recommendation	Consider checking the token's bytecode, e.g. with an `address(_token).code.length > 0` addition to the if statement.
Resolution	RESOLVED The `try` logic has been fully removed in favor of `safeTransfer`.

Issue #24	Typographical issues
Severity	INFORMATIONAL

Description	Line 27
	`error InvalidArgument();`
	
	It seems odd to define this error as it is unused within this library.
	
	Line 36
	`return (_amount * _self.fees[Role.FOUNDATION].bps) / 10000;`
	
	The `ONE_HUNDRED_PERCENT` literal can be re-used for `10000`.
	
	Lines 39-43
	`function payRedemptionFee(Info storage _self, IERC20 _token, uint _amount) internal returns (uint afterFeeAmount) { afterFeeAmount = _amount; uint totalFee = getRedemptionFee(_self, _amount); afterFeeAmount -= safeFeeTransfer(_token, totalFee, _self.roles[Role.OPERATOR]); }`
	
	This can be more neatly refactored as the current implementation can improve on semantics:
	`uint totalFee = getRedemptionFee(_self, _amount); totalFee = safeFeeTransfer(_token, totalFee, _self.roles[Role.OPERATOR]);`
	
	`return _amount - totalFee;`
	
	———
	
	`ping` lacks an event.

Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED Almost all issues were resolved, however `ping` still does not have an event.

2.6 USDV/USDVBase

`USDVBase` represents the core code for the USDV ERC20 deployments on all chains. It is an upgradable contract which extends the `ERC20PermitUpgradeable` contract for its ERC20 properties.

The contract defines the logic and conditions where a user's account balance will be recolored to a different minter than their current one. Specifically, recoloring of the balance solely occurs if the user has not set their enforced color yet and if the amount transferred to them exceeds their current balance. This means that an amount sent to that user gets recolored to the user's color even if it is greater than the user's balance, as long as that user has defined their enforced color. There is therefore no way to recolor individual users as soon as they set their enforced color, which is especially useful for smart contracts that will have a USDV balance, as these contracts will want to typically attribute that USDV to a specific minter.

The user can define an account which is permitted to set their enforced color. This is similar to granting that account "approval" but then specifically to set the enforced color and not to make transfers for the user. This allowed account is called the "colorer" for that user and can also be configured by the `OPERATOR` role. This means that the `OPERATOR` can override the enforced color of any account by reclaiming the colorer of that account and then setting their enforced color. Subsequently, `setEnforceColor` can be called by the colorer (or by default the user itself) to override the enforced color, which is NIL initially (e.g. undefined). If this new enforced color is different to the one that the user's balance is currently set to, that balance is immediately recolored.

Tokens can be sent cross-chain using the send function. This deducts the provided amount from the balance of the sender and will generate a message to send it to the balance of the provided recipient on a desired chain. Tokens can only be sent to chains configured in the `MESSAGING` contract, which are all chains USDV deploys to. When sending tokens, the balance of the color on the source chain is decreased

according to the amount. Subsequently, if there is any positive delta remaining, that delta is reduced on the source chain (and converted into `THETA`) and forwarded to the destination chain to be increased again there (taking from `THETA`). `THETA` is allowed to go negative and will always sum to zero once all bridge transactions arrive.

As the `send` function moves positive deltas to other chains together with a token balance, there should probably also be a function to move the accompanied negative delta to other chains (as both deltas eventually need to be reconsolidated to allow them to be re-minted on the Ethereum chain). The function to do that is called `syncDelta` which sends a negative delta to another chain. This can only be done by transferring a positive `THETA` and only if the source chain has such a positive `THETA` (e.g. it does not allow `THETA` to go negative). The goal of syncing negative deltas is to eventually sync them to a chain which has positive deltas, to subsequently remint them into the source chain.

The contract finally defines the internal `remint` calling logic, which is the logic which nets out a positive delta with negative deltas and forwards them to the Ethereum chain to incorporate the deltas into the supply allocations for the relevant minters. This is in fact the crucial step at which point the deltas finally get synced to the mainnet. It should be noted that to avoid economical abuse where minters try to abuse the system, a `remint` fee is introduced. A portion of the `remint` fee goes to the operator, while the remaining portion (presently planned as a larger portion) goes to the minters of the deltas which were burned on `remint`. This means that once you mint tokens as a minter and users do not redeem them again, it is guaranteed that you eventually receive a `remint` fee if users swap your tokens into another color as long as they actually sync this all the way back to the vault manager. As long as it is not synced, you will continue generating rewards which is fine as well.

The configured `FOUNDATION` role is able to add accounts to `denylist` which prevents them from making any transfers.

The `OPERATOR` role is able to pause and unpause the USDV token. While paused, several functions revert: transfers, sending USDV to other chains, receiving USDV from other chains, syncing delta to other chains, receiving delta sync from other chains.

It should be noted that even though there is a `remint` fee to incorporate deltas into the mainnet vault to update the accounting of which color/minter is attributed which portion of the supply, this `remint` fee is not levied on ordinary recolors/delta adjustments. That is, if a user's color is changed, this incurs a delta adjustment which might eventually get re-minted into mainnet's accounting, but the fee is not levied during the recolor and only gets levied on the actual `remint`.

UPDATE: We noticed that within the deployed contract, a sending rate limit has been introduced. We informed the client that due to the lack of send fees, this limit can be reached at no cost by a malicious actor, especially when set to a low value.

2.6.1 Privileged Functions

- `setRole [ OWNER or current role bearer ]`
- `blacklist [ FOUNDATION ]`
- `setPause [ OPERATOR ]`
- `addColor [ OPERATOR ]`
- `setColorer [ the user itself or the OPERATOR ]`
- `setEnforceColor [ colorer configured by the user or by the operator ]`

2.6.2 Issues & Recommendations

Issue #25	**An exploiter is able to keep negative deltas for prolonged periods "in-flight", preventing them from being settled to the mainnet**
Severity	MEDIUM SEVERITY
Description	`syncDelta` sends negative deltas from one chain to another. This can be done as long as the source chain has a positive `THETA` which means there is a net surplus outflow on that chain. Given that message transmission between chains is not instantaneous and that calling `syncDelta` is free, an exploiter is able to instantaneously call `syncDelta` repeatedly whenever the negative deltas arrive on the destination chain. This would effectively cause the negative deltas to permanently be in-flight, making it extremely tedious for minters to recapture these deltas for re-minting. Even worse, `syncDelta` provides an `_extraOptions` adapter parameter value, which can be freely set by the sender. It may be possible for the exploiter to figure out parameters that causes the message to not automatically execute on the receipt chain (e.g. an incorrectly formatted options which causes an exception on the off-chain components). This issue is rated as medium severity instead of high as it appears like there are mitigation techniques to force slow re-mints of these deltas by actively distributing positive deltas to all chains and then re-minting them as soon as they capture some of the negative in-flight deltas. However, given that the practical deployment will have many colors and many different stakeholders, it may be more complex to actively mitigate such an attack, hence still makes more sense to mitigate it with for example a sync fee.
Recommendation	Consider whether it makes sense to add another layer of defense and burden for the syncer such as a sync fee. By sending this fee to the `OPERATOR`, syncing remains free for the `OPERATOR`.
Resolution	RESOLVED A sync fee is now added, which goes to the operator.

Issue #26	**Lack of `denylist` and `to` validation on send increases the likelihood of these tokens to be stuck in transit**
Severity	LOW SEVERITY
Location	Line 207 `function send(`
Description	The function to send tokens cross-chain has less validation than the function to transfer tokens from one account to another. However, as this function also transfers tokens from one account to another, it should do the same validations. This is of course not always possible as the sending chain cannot know about all states of the receive chain, but it should strive to be as close as possible. Specifically, the source chain should validate that the destination address is not blacklisted at the source, and does not equal zero. Specifically, the blacklisted check is a heuristic check and not a perfect check given that the blacklisted set can have a divergence across chains.
Recommendation	Consider checking that the destination address is not zero or blacklisted.
Resolution	RESOLVED The client has added a non-zero validation but opted against `denylist` validation as there can be non-EVM chains with more than 20 bytes of addresses. We believe that the `denylist` could still be checked for the EVM compatible chains but the client has opted against this, likely favoring gas savings.

Issue #27	**Allowing operators to add colors manually could lead to configurational errors**
Severity	LOW SEVERITY
Location	Line 74 `function addColor(uint32 _color) external onlyRole(Role.OPERATOR) {`
Description	Presently the USDV token allows for its operator to register colors. However, the system already implements a color propagation mechanism which is aimed to allow new colors to automatically propagate to the side-chains once transfers are made from mainnet to these chains (either directly or indirectly). Given that this propagation methodology is relatively straightforward, we see no immediate need to manually register colors. Allowing for this to be easily done might lead to configurational error. If a nonexistent color is added to a side-chain, this can cause a lot of damage to the state of the system.
Recommendation	Consider whether there is any case for registering colors manually compared to propagating them through transfers. If the function is kept, consider explicitly preventing the NIL color (0) from ever being added.
Resolution	RESOLVED This function was removed.

Issue #28	Typographical issues
Severity	INFORMATIONAL

Description	Line 18
	`bool public paused;`
	
	In other contracts, OpenZeppelin's Pausable is used. It is unclear to us why it is implemented here internally. Perhaps it makes sense to be consistent across these various contracts.
	
	Line 109
	`emit ColorerSet(_colorer);`
	
	It might make sense to emit `_user` in this event as well.
	
	Line 116
	`uint64 amountU64 = _amount.toUint64();`
	
	It would be more consistent to revert with `InsufficientBalance` here as well, given that the user indeed always has insufficient balance here. This may make more sense to third party integrations who are inspecting the errors.
	
	Line 171
	`function _mintBalance(address _receiver, uint64 _amount, uint32 _color) internal {`
	
	This should probably return whether the receiver got the `color` assigned or not.
	
	Line 351
	`uint32 idx = i - _startIdx;`
	
	This appears slightly inconsistent with `getColors` as `idx` is not used there and this is inlined. We recommend adjusting either function to make the style more consistent.

	`getColors` and `getDeltas` should be completed with a function that returns the total number of colors/deltas to allow for more targeted iteration. — It may make sense to be more explicit with the return values for functions such as `send` and `syncDelta` as the caller cannot easily figure out which theta, surplus and deltas were actually used. Of course this does come at a gas cost as more data is returned.
Recommendation	Consider fixing the typographical issues.
Resolution	PARTIALLY RESOLVED Some of these informational recommendations were resolved.

2.7 USDV/USDVMain

`USDVMain` represents the main `USDV` `ERC20` deployment on the Ethereum mainnet (or in general, the chain where the `VaultManager` is deployed). It differs from the `USDVSide` implementation which is deployed on side-chains as the `USDVMain` implementation is the sole token that actually mints and redeems USDV into the `VaultManager`.

`USDVMain` extends `USDVBase` which is detailed in a previous section and therefore inherits all descriptions, issues and limitations of that contract. It allows the `VaultManager` to call `mint` on the contract to mint new USDV tokens. It also facilitates redemption via a `redeem` function to the `VaultManager` which burns the tokens from a user.

UPDATE: During the live match, we noticed the client moved `mint` and `redeem` into `USDVBase`, allowing the team to call these on side chains in emergencies. We recommend them to be careful with this as it may break crucial properties.

2.7.1 Privileged Functions

- `setRole [ OWNER or current role bearer ]`
- `blacklist [ FOUNDATION ]`
- `setPause [ OPERATOR ]`
- `addColor [ OPERATOR ]`
- `setColorer [ the user itself or the OPERATOR ]`
- `setEnforceColor [ colorer configured by the user or by the operator ]`

2.7.2 Issues & Recommendations

Issue #29	`remintAck` lacks a `whenNotPaused` modifier when the fee is zero
Severity	MEDIUM SEVERITY
Description	The `remintAck` will revert due to being paused if a fee was set. This is because the underlying `_sendAck` is called on that path and triggers a reversion on `pause`. However, when no fee is sent, this path is ignored and the remint can just go through.
Recommendation	Consider adding a `whenNotPaused` modifier to the `remintAck` function that catches all paths.
Resolution	RESOLVED

Issue #30	Colors appear to not be explicitly validated for the deltas
Severity	INFORMATIONAL
Description	The deltas their colors appear to not explicitly be validated to exist. This increases the attack surface of the contract for no reason.
Recommendation	Consider validating all deltas in the various functions (e.g. in the `VaultManager` contract and `Colors` library) consistently to reduce the state space.
Resolution	RESOLVED

2.8 USDV/USDVSide

`USDVSide` represents the `USDV` `ERC20` deployments on side-chains (or in general the chains other than where the `VaultManager` is deployed). They define the function which can be called to re-mint to the main chain's `VaultManager`.

`USDVSide` extends `USDVBase` which has been detailed in a previous section and therefore inherits all descriptions, issues and limitations of that contract.

2.8.1 Privileged Functions

- `setRole [ OWNER or current role bearer ]`
- `blacklist [ FOUNDATION ]`
- `setPause [ OPERATOR ]`
- `addColor [ OPERATOR ]`
- `setColorer [ the user itself or the OPERATOR ]`
- `setEnforceColor [ colorer configured by the user or by the operator ]`

2.8.2 Issues & Recommendations

No issues other than the ones in `USDVBase` were found.

2.9 USDV/Colors

`Colors` is a library used by all of the `USDV` token implementation as the accounting ledger for each individual chain's color supplies. It also accounts for each color's delta which represents the amount of tokens of that color that still need to propagate to the main chain's vault.

This library contains critical sections of code that deal with the evolution of these delta accountancy and the `THETA` counterpart.

2.9.1 Issues & Recommendations

Issue #31	**`NIL` color could accidentally be added if communication malfunctions or an operator adds it**
Severity	LOW SEVERITY
Description	The `Colors` library allows the `NIL` color to be added within the `addColor` function. This `color` has a special meaning which is "non existent" and should never be added. This can specifically happen via an operator or through malfunctioning communication. It may also make sense to validate that mint's color is not THETA explicitly alongside potential other functions. This makes the state space limitations more explicit, which is nice. Finally, adding an additional validation for colors to be correct within outflows might not hurt.
Recommendation	Consider adding a requirement (not the if-clause!) that causes a full revert as this state should never occur in our opinion. Consider carefully checking that there is no such valid scenario, however, we do not believe there is.
Resolution	RESOLVED A requirement has been introduced.

Issue #32	Typographical issues
Severity	INFORMATIONAL

Description	Lines 63 and 125
	`int64 amountInt64 = int64(_amount);`
	`int64 amountInt64 = -int64(_amount);`
	
	Although the fact that this cannot overflow is a property within `VaultManager`, it is probably worth the handful of gas to revalidate it here as the property could break if cross-chain communication malfunctions. We believe that in general, this codebase should swallow the handful of extra gas for any implicit casting to make them checked.
	
	Line 146
	`// not reverting 0 value because redeem can have 0 surplus (burning minted only)`
	
	This input amount seems to still mean the full redemption amount at this stage, making the comment potentially inaccurate.

Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED

2.10 USDV/Operator

`Operator` is managed by its `Owner` and is used as the `OPERATOR` role within the USDV tokens.

It allows the owner to call the `operator` functions of the USDV token, and more importantly defines the `getRemintFees` function that returns the `remint` minter and operator fees for any input amount. This function is called within the `remint` logic to figure out the fee to charge the `remint` caller. This means that the operator can freely define this fee and is allowed to configure it to a point where the sum of the two fees can even exceed the provided amount.

The initial `remint` fee sent to the operator is 0.01% while the `remint` fee sent to the minter's who have their supply reduced is 0.04%.

2.10.1 Privileged Functions

- `rotateOperator`
- `withdrawToken`
- `setPause`
- `addColor`
- `setOperatorRemintFeeBps`
- `setMinterRemintFeeBps`
- `setColorer`
- `transferOwnership`
- `renounceOwnership`

2.10.2 Issues & Recommendations

Issue #33	Typographical issues
Severity	INFORMATIONAL

Description	Line 18
	`constructor(address _usdv) Ownable() {`
	
	`_usdv` can be provided as `IUSDV`.
	
	Line 28
	`if (_token == address(0)) payable(_to).transfer(_amount);`
	
	Consider using `call` instead as it allows transferring to contracts which consume more gas on receipt such as custom vaults.
	
	Line 29
	`else IERC20(_token).transfer(_to, _amount);`
	
	Consider using `safeTransfer` instead as this might ignore failure of underlying tokens and malfunction for ill-specified ERC20 tokens which do not return a boolean.
	
	—
	
	From a gas perspective, it may make sense to make the fee percentages immutable if these are not planned to change often. This would save on gas and changing them could still happen through rotating the operator to a new address.

Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED Pretty much everything has been resolved, though the client has opted for mutable fees, meaning the last suggestion was not implemented, which is fine. The client has included a sync fee to resolve one of our other issues and has also added minimum absolute fees to `remint` and sync fees, which are freely configurable.

2.11 USDV/Messaging

`Messaging` is a dependency used by `MessagingV1` and the future `MessagingV2` (out-of-scope of this audit).

It defines some shared functionality such as the address of the USDV token on that chain, the `eid` of the main Ethereum chain, as long as whether this chain is that Ethereum chain or not.

Finally, it provides a configurable mapping of the extra gas required of any given message type.

2.11.1 Privileged Functions

- `setPerColorExtraGas`
- `transferOwnership`
- `renounceOwnership`

2.11.2 Issues & Recommendations

Issue #34	Typographical issues
Severity	INFORMATIONAL

Description	Lines 6 and 12
	`import "./libs/MsgCodec.sol";`
	`using MsgCodec for bytes;`
	
	This `import` is unused and can be removed in favor of importing it in the actual libraries. If desired, only the `using` clause can be removed as that one definitely serves no purpose here.
	
	Lines 14-15
	`uint32 internal immutable mainChainEid;`
	`bool internal immutable isMainChain;`
	
	These variables should be marked as `public` to allow them to be inspected from within the browser.
	
	—
	
	`setPerColorExtraGas` lacks an event.

Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED "extra gas" is now also segmented per destination chain, which makes sense.

2.12 USDV/MessagingV1

`MessagingV1` is the component used by the `USDV` deployments to send and receive messages over the LayerZero network. It defines functions that USDV can use to transmit a `send`, `syncDelta` and `remint` message to other changes. The first two can be sent to any registered chain while the latter can only be sent to the main Ethereum chain, as it is the message responsible for re-minting the color shares correctly in the `VaultManager` to update the reward allocations there.

The `MessagingV1` components extends `NonBlockingLzApp` (out of scope for this audit), a utility contract developed by LayerZero to write applications on top of the LayerZero endpoints. It should be noted that the `NonBlockingLzApp` is a governed application, meaning that the owner address of `MessagingV1` has full control over the configuration of the messaging component. This includes the ability to receive and by extension transmit false messages. It is therefore absolutely crucial that the owner of this component is properly safeguarded behind a reputable multi-signature wallet, alongside the other core governance roles and proxy admins within the system.

On message receipt of any of these three messages, `MessagingV1` decodes the messages into understandable structures, which are forwarded to the `USDV` token on that chain. The `USDV` token has `MessagingV1` set as its `MESSAGING` role which means that only this contract can forward messages to the USDV contract, unless that role is adjusted to another contract using `setRole`.

This component extends the `Messaging` dependency.

It should be noted that a `Buffer` library (also out of this audit's scope) is used to concatenate the deltas into a packed bytes array. We agree that this is not possible in Solidity and requires inline assembly which is desired to be done from a library shared by multiple users. However, as this library is out of scope for this audit, we

cannot speak for its validity. Users should carefully read the audits done on this library as it appears like it was audited in the past for Ethereum Name Service.

2.12.1 Privileged Functions

- `setPerColorExtraGas`
- `setConfig`
- `setSendVersion`
- `setReceiveVersion`
- `forceResumeReceive`
- `setTrustedRemote`
- `setTrustedRemoteAddress`
- `setPrecrime`
- `setMinDstGas`
- `setPayloadSizeLimit`
- `transferOwnership`
- `renounceOwnership`

2.12.2 Issues & Recommendations

Issue #35	**The contract does not support retrying failed non-blocking messages due to incorrectly overriding `_nonblockingLzReceive`**
Severity	HIGH SEVERITY

Location	Lines 196-201
	`function _nonblockingLzReceive(`
	`  uint16 _srcChainId,`
	`  bytes memory _srcAddress,`
	`  uint64 _nonce,`
	`  bytes memory _payload`
	`) internal override {}`

Description	`_nonblockingLzReceive` is not implemented as the client instead overrides `nonblockingLzReceive` to keep the payload out of the memory (this `public` function has it defined as `calldata`). However, this `public` function is not always used within the `NonBlockingLzApp` dependency. NonBlockingLzApp::L84 `_nonblockingLzReceive(_srcChainId, _srcAddress, _nonce, _payload);` Unfortunately, the `retry` logic of the dependency still calls the `internal` function. This means that failed messages cannot be retried within this system and such messages would require the team to intervene and redeploy a fixed Messaging contract that re-submits the failed messages to the USDV token.
Recommendation	Consider overriding the internal function instead.
Resolution	RESOLVED `_nonblockingLzReceive` is now overridden. Note that a custom implementation is now used for `NonBlockingLzApp`. As this is out-of-scope, changes compared to the trusted version should be carefully checked by the team and users. To our knowledge, only the memory field has been changed into `calldata` which is an innocent change.

Issue #36	**The contract attempts to support LayerZero token payment support but fails at doing so, bricking the contract if such a token is ever configured**
Severity	MEDIUM SEVERITY
Location	Lines 39, 66 and 94 `msg.sender,`
Description	`zroPaymentAddres`, which is the address intended to pay for the LayerZero token, is misconfigured to the `msg.sender` of the messaging transaction. `msg.sender` is in fact the USDV token and not the user. Due to a requirement in the deployed contracts, this payer must either be `tx.origin` or the app itself, causing the whole send function to revert in such a case. This would therefore likely brick the contract as soon as a LayerZero token is configured on the endpoint.
Recommendation	We do not necessarily like the usage of `tx.origin` here either as a solution, as it is very prone to phishing risk. Instead, either simply mark the address as `unset` to explicitly not support the token in this version, or come up with a forwarding scheme where the user ends up sending tokens to the `MessagingV1` contract first (ideally through an approval to the `USDV` token), which then forwards them to the Endpoint.
Resolution	RESOLVED For V1, the payment address is now unset as this version does not need to support such payments yet.

Issue #37	Typographical issues
Severity	INFORMATIONAL

Description	Line 22
	`) Messaging(_usdv, _mainChainEid, _isMainChain) NonblockingLzApp(_endpoint) {}`
	
	A simple validation can be added within the body of the constructor to check that the `_mainChainEid` equals the current `chainId` as reported by the `_endpoint` if `_isMainChain` is set. This would reduce configurational risk slightly.
	
	Even better, `_isMainChain` should be derived from this check directly. I.e., it should simply be set to the result of the comparison of the provided `_mainChainEid` and the actual `_endpoint` `eid`. This greatly reduces configurational risk and overhead.
	
	`Lines 41, 68 and 96`
	`_msgFee.nativeFee`
	
	It is unclear what the advantage of this is compared to `msg.value` — what happens if this value is lower than `msg.value`? Should equality not be checked to avoid such a case?

Recommendation	Consider fixing the typographical issues.
Resolution	PARTIALLY RESOLVED `msg.value` is now used throughout the contract.

2.13 USDV/MsgCodec

`MsgCodec` is a library used by `MessagingV1` to handle the translation of raw bytes to and from understandable Solidity types. It also does basic checking on the data format.

It should be noted that the `sendAndCall` logic is not used within the audited code.

2.13.1 Issues & Recommendations

Issue #38	Typographical issues
Severity	INFORMATIONAL

Description	Line 80
	`theta: uint64(bytes8(_message[THETA_OFFSET:]))`
	
	It would be more consistent to explicitly mark the up to portion of the slice.
	
	Line 111
	`composeMsg: _message[COMPOSED_MSG_OFFSET:]`
	
	It may make sense to validate that this is at least 32 bytes, as this is the required minimum length. Note that this can also be directly validated at the top of the function in the requirement mentioned at a later point in this aggregated issue.
	
	Line 194
	`return address(uint160(uint256(_b)));`
	
	Explicitly validate that there are no dirty bits on `_b` would help detect configuration errors where a non-160 bit address was sent into the system. Otherwise, this silently overflows which in our opinion is undesired behavior. A reversion would allow for detection and patching the mistake.
	
	—
	
	To be consistent, consider explicitly validating the lengths of `_message` for `decodeSendMsg` (exact validation) and `decodeSendAndCallMsg` (greater or equal validation).
	
	—
	
	Consider 1+4+8+8 as a `const` within `encodeRemintMsg` to potentially save some gas.

	There are still quite a few magic values (e.g. integers throughout the codebase). This is typically avoided due to by code quality but we leave it up to the client as to whether they want to extract these into constants or not.
Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED Most of these issues have been resolved. Length validation has been added.


PALADIN
BLOCKCHAIN SECURITY